# Finite Machine Word Library

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

March 19, 2025

**Abstract**

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

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

## Contents

# 1 Comprehension syntax for bit expressions

```
theory Bit_Comprehension
  imports
    "HOL-Library.Word"
begin

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

lemma set_bits_False_eq [simp]:
  ‹(BITS _. False) = 0›
  using set_bits_bit_eq [of 0] by (simp add: bot_fun_def)

end

instantiation word :: (len) bit_comprehension
begin

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

instance by standard
  (simp add: word_set_bits_def horner_sum_bit_eq_take_bit)

end

lemma bit_set_bits_word_iff [bit_simps]:
  ‹bit (set_bits P :: 'a::len word) n ⟷ n < LENGTH('a) ∧ P n›
  by (auto simp add: word_set_bits_def bit_horner_sum_bit_word_iff)

lemma word_of_int_conv_set_bits: "word_of_int i = (BITS n. bit i n)"
  by (rule bit_eqI) (auto simp add: bit_simps)

lemma set_bits_K_False:
  ‹set_bits (λ_. False) = (0 :: 'a :: len word)›
```

```
  by (fact set_bits_False_eq)

lemma word_test_bit_set_bits: "bit (BITS n. f n :: 'a :: len word) n
⟷ n < LENGTH('a) ∧ f n"
  by (fact bit_set_bits_word_iff)

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

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

lemma bit_set_bit_aux [bit_simps]:
  ‹bit (set_bits_aux n w) m ⟷ m < LENGTH('a) ∧
    (if m < n then f m else bit w (m - n))› for w :: ‹'a::len word›
  by (auto simp add: bit_simps set_bits_aux_def)

corollary set_bits_conv_set_bits_aux:
  ‹set_bits f = (set_bits_aux LENGTH('a) 0 :: 'a :: len word)›
  by (rule bit_word_eqI) (simp add: bit_simps)

lemma set_bits_aux_0 [simp]:
  ‹set_bits_aux 0 w = w›
  by (simp add: set_bits_aux_def)

lemma set_bits_aux_Suc [simp]:
  ‹set_bits_aux (Suc n) w = set_bits_aux n (push_bit 1 w OR (if f n then
1 else 0))›
  by (rule bit_word_eqI) (auto simp add: bit_simps le_less_Suc_eq mult.commute
[of _ 2])

lemma set_bits_aux_simps [code]:
  ‹set_bits_aux 0 w = w›
  ‹set_bits_aux (Suc n) w = set_bits_aux n (push_bit 1 w OR (if f n then
1 else 0))›
  by simp_all

lemma set_bits_aux_rec:
  ‹set_bits_aux n w =
  (if n = 0 then w
   else let n' = n - 1 in set_bits_aux n' (push_bit 1 w OR (if f n' then
1 else 0)))›
  by (cases n) simp_all

end

end
```

# 2 More on bitwise operations on integers

**theory** More_Int
  **imports** Main
**begin**

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

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

**lemma** bin_cat_eq_push_bit_add_take_bit:
  ‹concat_bit n l k = push_bit n k + take_bit n l›
  **by** (simp add: concat_bit_eq)

**lemma** bin_cat_assoc: "(λk n l. concat_bit n l k) ((λk n l. concat_bit
n l k) x m y) n z = (λk n l. concat_bit n l k) x (m + n) ((λk n l. concat_bit
n l k) y n z)"
  **by** (fact concat_bit_assoc)

**lemma** bin_cat_assoc_sym: "(λk n l. concat_bit n l k) x m ((λk n l. concat_bit
n l k) y n z) = (λk n l. concat_bit n l k) ((λk n l. concat_bit n l k)
x (m - n) y) (min m n) z"
  **by** (fact concat_bit_assoc_sym)

**lemma** bin_nth_cat:
  "(bit :: int ⇒ nat ⇒ bool) ((λk n l. concat_bit n l k) x k y) n =
    (if n < k then (bit :: int ⇒ nat ⇒ bool) y n else (bit :: int ⇒
nat ⇒ bool) x (n - k))"
  **by** (simp add: bit_concat_bit_iff)

**lemma** bin_nth_drop_bit_iff:
  ‹(bit :: int ⇒ nat ⇒ bool) (drop_bit n c) k ⟷ (bit :: int ⇒ nat
⇒ bool) c (n + k)›
  **by** (simp add: bit_drop_bit_eq)

**lemma** bin_nth_take_bit_iff:
  ‹(bit :: int ⇒ nat ⇒ bool) (take_bit n c) k ⟷ k < n ∧ (bit :: int
⇒ nat ⇒ bool) c k›
  **by** (fact bit_take_bit_iff)

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

```
  by (simp add: bin_cat_eq_push_bit_add_take_bit)

lemma bintr_cat1: "(take_bit :: nat ⇒ int ⇒ int) (k + n) ((λk n l.
concat_bit n l k) a n b) = (λk n l. concat_bit n l k) ((take_bit :: nat
⇒ int ⇒ int) k a) n b"
  by (metis bin_cat_assoc bin_cat_zero)

lemma bintr_cat: "(take_bit :: nat ⇒ int ⇒ int) m ((λk n l. concat_bit
n l k) a n b) =
    (λk n l. concat_bit n l k) ((take_bit :: nat ⇒ int ⇒ int) (m - n)
a) n ((take_bit :: nat ⇒ int ⇒ int) (min m n) b)"
  by (rule bit_eqI) (auto simp add: bit_simps)

lemma bintr_cat_same [simp]: "(take_bit :: nat ⇒ int ⇒ int) n ((λk
n l. concat_bit n l k) a n b) = (take_bit :: nat ⇒ int ⇒ int) n b"
  by (auto simp add : bintr_cat)

lemma cat_bintr [simp]: "(λk n l. concat_bit n l k) a n ((take_bit ::
nat ⇒ int ⇒ int) n b) = (λk n l. concat_bit n l k) a n b"
  by (simp add: bin_cat_eq_push_bit_add_take_bit)

lemma drop_bit_bin_cat_eq:
  ‹drop_bit n ((λk n l. concat_bit n l k) v n w) = v›
  by (rule bit_eqI) (simp add: bit_drop_bit_eq bit_concat_bit_iff)

lemma take_bit_bin_cat_eq:
  ‹take_bit n ((λk n l. concat_bit n l k) v n w) = take_bit n w›
  by (rule bit_eqI) (simp add: bit_concat_bit_iff)

lemma bin_cat_num: "(λk n l. concat_bit n l k) a n b = a * 2 ^ n + (take_bit
:: nat ⇒ int ⇒ int) n b"
  by (simp add: bin_cat_eq_push_bit_add_take_bit push_bit_eq_mult)

lemma bin_cat_cong: "concat_bit n b a = concat_bit m d c"
  if "n = m" "a = c" "take_bit m b = take_bit m d"
  using that(3) unfolding that(1,2)
  by (simp add: bin_cat_eq_push_bit_add_take_bit)

lemma bin_cat_eqD1: "concat_bit n b a = concat_bit n d c ⟹ a = c"
  by (metis drop_bit_bin_cat_eq)

lemma bin_cat_eqD2: "concat_bit n b a = concat_bit n d c ⟹ take_bit
n b = take_bit n d"
  by (metis take_bit_bin_cat_eq)

lemma bin_cat_inj: "(concat_bit n b a) = concat_bit n d c ⟷ a = c
∧ take_bit n b = take_bit n d"
  by (auto intro: bin_cat_cong bin_cat_eqD1 bin_cat_eqD2)
```

```
lemma bin_last_def:
  "(odd :: int ⇒ bool) w ⟷ w mod 2 = 1"
  by (fact odd_iff_mod_2_eq_one)

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)"
  by simp_all

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

lemma bin_rl_eqI: "⟦(λk::int. k div 2) x = (λk::int. k div 2) y; odd
x = odd y⟧ ⟹ x = y"
  by (auto elim: oddE)

lemma [simp]:
  shows bin_rest_lt0: "(λk::int. k div 2) i < 0 ⟷ i < 0"
  and  bin_rest_ge_0: "(λk::int. k div 2) i ≥ 0 ⟷ i ≥ 0"
  by auto

lemma bin_rest_gt_0 [simp]: "(λk::int. k div 2) x > 0 ⟷ x > 1"
  by auto

lemma bin_nth_eq_iff: "(bit :: int ⇒ nat ⇒ bool) x = (bit :: int ⇒
nat ⇒ bool) y ⟷ x = y"
  by (simp add: bit_eq_iff fun_eq_iff)

lemma bin_eqI:
  "x = y" if "⋀n. (bit :: int ⇒ nat ⇒ bool) x n ⟷ (bit :: int ⇒ nat
⇒ bool) y n"
  using that by (rule bit_eqI)

lemma bin_eq_iff: "x = y ⟷ (∀n. (bit :: int ⇒ nat ⇒ bool) x n =
(bit :: int ⇒ nat ⇒ bool) y n)"
```

```
  by (metis bit_eq_iff)

lemma bin_nth_zero [simp]: "¬ (bit :: int ⇒ nat ⇒ bool) 0 n"
  by simp

lemma bin_nth_1 [simp]: "(bit :: int ⇒ nat ⇒ bool) 1 n ⟷ n = 0"
  by (cases n) (simp_all add: bit_Suc)

lemma bin_nth_minus1 [simp]: "(bit :: int ⇒ nat ⇒ bool) (- 1) n"
  by simp

lemma bin_nth_numeral: "(λk::int. k div 2) x = y ⟹ (bit :: int ⇒ nat
⇒ bool) x (numeral n) = (bit :: int ⇒ nat ⇒ bool) y (pred_numeral n)"
  by (simp add: numeral_eq_Suc bit_Suc)

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

lemmas bin_nth_simps =
  bit_0 bit_Suc bin_nth_zero bin_nth_minus1
  bin_nth_numeral_simps

lemma nth_2p_bin: "(bit :: int ⇒ nat ⇒ bool) (2 ^ n) m = (m = n)" —
for use when simplifying with bin_nth_Bit
  by (auto simp add: bit_exp_iff)

lemma nth_rest_power_bin: "(bit :: int ⇒ nat ⇒ bool) (((λk::int. k
div 2) ^^ k) w) n = (bit :: int ⇒ nat ⇒ bool) w (n + k)"
  by (induct k arbitrary: n) (auto simp flip: bit_Suc)

lemma bin_nth_numeral_unfold:
  "(bit :: int ⇒ nat ⇒ bool) (numeral (num.Bit0 x)) n ⟷ n > 0 ∧ (bit
:: int ⇒ nat ⇒ bool) (numeral x) (n - 1)"
  "(bit :: int ⇒ nat ⇒ bool) (numeral (num.Bit1 x)) n ⟷ (n > 0 ⟶
(bit :: int ⇒ nat ⇒ bool) (numeral x) (n - 1))"
  by (cases n; simp)+

lemma bintrunc_mod2p: "(take_bit :: nat ⇒ int ⇒ int) n w = w mod 2
^ n"
  by (fact take_bit_eq_mod)

lemma sbintrunc_mod2p: "(signed_take_bit :: nat ⇒ int ⇒ int) n w =
(w + 2 ^ n) mod 2 ^ Suc n - 2 ^ n"
  by (simp add: bintrunc_mod2p signed_take_bit_eq_take_bit_shift)

lemma sbintrunc_eq_take_bit:
  ‹(signed_take_bit :: nat ⇒ int ⇒ int) n k = take_bit (Suc n) (k +
2 ^ n) - 2 ^ n›
  by (fact signed_take_bit_eq_take_bit_shift)
```

```
lemma bintrunc_n_0: "(take_bit :: nat ⇒ int ⇒ int) n 0 = 0"
  by (fact take_bit_of_0)

lemma sbintrunc_n_0: "(signed_take_bit :: nat ⇒ int ⇒ int) n 0 = 0"
  by (fact signed_take_bit_of_0)

lemma sbintrunc_n_minus1: "(signed_take_bit :: nat ⇒ int ⇒ int) n (-
1) = -1"
  by (fact signed_take_bit_of_minus_1)

lemma bintrunc_Suc_numeral:
  "(take_bit :: nat ⇒ int ⇒ int) (Suc n) 1 = 1"
  "(take_bit :: nat ⇒ int ⇒ int) (Suc n) (- 1) = 1 + 2 * (take_bit ::
nat ⇒ int ⇒ int) n (- 1)"
  "(take_bit :: nat ⇒ int ⇒ int) (Suc n) (numeral (Num.Bit0 w)) = 2
* (take_bit :: nat ⇒ int ⇒ int) n (numeral w)"
  "(take_bit :: nat ⇒ int ⇒ int) (Suc n) (numeral (Num.Bit1 w)) = 1
+ 2 * (take_bit :: nat ⇒ int ⇒ int) n (numeral w)"
  "(take_bit :: nat ⇒ int ⇒ int) (Suc n) (- numeral (Num.Bit0 w)) =
2 * (take_bit :: nat ⇒ int ⇒ int) n (- numeral w)"
  "(take_bit :: nat ⇒ int ⇒ int) (Suc n) (- numeral (Num.Bit1 w)) =
1 + 2 * (take_bit :: nat ⇒ int ⇒ int) n (- numeral (w + Num.One))"
  by (simp_all add: take_bit_Suc del: take_bit_minus_one_eq_mask)

lemma sbintrunc_0_numeral [simp]:
  "(signed_take_bit :: nat ⇒ int ⇒ int) 0 1 = -1"
  "(signed_take_bit :: nat ⇒ int ⇒ int) 0 (numeral (Num.Bit0 w)) = 0"
  "(signed_take_bit :: nat ⇒ int ⇒ int) 0 (numeral (Num.Bit1 w)) = -1"
  "(signed_take_bit :: nat ⇒ int ⇒ int) 0 (- numeral (Num.Bit0 w)) =
0"
  "(signed_take_bit :: nat ⇒ int ⇒ int) 0 (- numeral (Num.Bit1 w)) =
-1"
  by simp_all

lemma sbintrunc_Suc_numeral:
  "(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) 1 = 1"
  "(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) (numeral (Num.Bit0 w))
= 2 * (signed_take_bit :: nat ⇒ int ⇒ int) n (numeral w)"
  "(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) (numeral (Num.Bit1 w))
= 1 + 2 * (signed_take_bit :: nat ⇒ int ⇒ int) n (numeral w)"
  "(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) (- numeral (Num.Bit0
w)) = 2 * (signed_take_bit :: nat ⇒ int ⇒ int) n (- numeral w)"
  "(signed_take_bit :: nat ⇒ int ⇒ int) (Suc n) (- numeral (Num.Bit1
w)) = 1 + 2 * (signed_take_bit :: nat ⇒ int ⇒ int) n (- numeral (w +
Num.One))"
  by (simp_all add: signed_take_bit_Suc)

lemma nth_bintr: "(bit :: int ⇒ nat ⇒ bool) ((take_bit :: nat ⇒ int
```

```
⇒ int) m w) n ⟷ n < m ∧ (bit :: int ⇒ nat ⇒ bool) w n"
  by (fact bit_take_bit_iff)

lemma nth_sbintr: "(bit :: int ⇒ nat ⇒ bool) ((signed_take_bit :: nat
⇒ int ⇒ int) m w) n = (if n < m then (bit :: int ⇒ nat ⇒ bool) w n
else (bit :: int ⇒ nat ⇒ bool) w m)"
  by (simp add: bit_signed_take_bit_iff min_def)

lemma bin_nth_Bit0:
  "(bit :: int ⇒ nat ⇒ bool) (numeral (Num.Bit0 w)) n ⟷
    (∃m. n = Suc m ∧ (bit :: int ⇒ nat ⇒ bool) (numeral w) m)"
  using bit_double_iff [of ‹numeral w :: int› n]
  by (auto intro: exI [of _ ‹n - 1›])

lemma bin_nth_Bit1:
  "(bit :: int ⇒ nat ⇒ bool) (numeral (Num.Bit1 w)) n ⟷
    n = 0 ∨ (∃m. n = Suc m ∧ (bit :: int ⇒ nat ⇒ bool) (numeral w)
m)"
  using even_bit_succ_iff [of ‹2 * numeral w :: int› n]
    bit_double_iff [of ‹numeral w :: int› n]
  by auto

lemma bintrunc_bintrunc_l: "n ≤ m ⟹ (take_bit :: nat ⇒ int ⇒ int)
m ((take_bit :: nat ⇒ int ⇒ int) n w) = (take_bit :: nat ⇒ int ⇒ int)
n w"
  by simp

lemma sbintrunc_sbintrunc_l: "n ≤ m ⟹ (signed_take_bit :: nat ⇒ int
⇒ int) m ((signed_take_bit :: nat ⇒ int ⇒ int) n w) = (signed_take_bit
:: nat ⇒ int ⇒ int) n w"
  by simp

lemma bintrunc_bintrunc_ge: "n ≤ m ⟹ (take_bit :: nat ⇒ int ⇒ int)
n ((take_bit :: nat ⇒ int ⇒ int) m w) = (take_bit :: nat ⇒ int ⇒ int)
n w"
  by (rule bin_eqI) (auto simp: nth_bintr)

lemma bintrunc_bintrunc_min [simp]: "(take_bit :: nat ⇒ int ⇒ int)
m ((take_bit :: nat ⇒ int ⇒ int) n w) = (take_bit :: nat ⇒ int ⇒ int)
(min m n) w"
  by (rule take_bit_take_bit)

lemma sbintrunc_sbintrunc_min [simp]: "(signed_take_bit :: nat ⇒ int
⇒ int) m ((signed_take_bit :: nat ⇒ int ⇒ int) n w) = (signed_take_bit
:: nat ⇒ int ⇒ int) (min m n) w"
  by (rule signed_take_bit_signed_take_bit)

lemmas sbintrunc_Suc_Pls =
  signed_take_bit_Suc [where a="0::int", simplified bin_last_numeral_simps
```

```
bin_rest_numeral_simps]

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

lemmas sbintrunc_Sucs = sbintrunc_Suc_Pls sbintrunc_Suc_Min
  sbintrunc_Suc_numeral

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

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

lemmas sbintrunc_0_simps =
  sbintrunc_Pls sbintrunc_Min

lemmas sbintrunc_simps = sbintrunc_0_simps sbintrunc_Sucs

lemma bintrunc_minus: "0 < n ⟹ (take_bit :: nat ⇒ int ⇒ int) (Suc
(n - 1)) w = (take_bit :: nat ⇒ int ⇒ int) n w"
  by auto

lemma sbintrunc_minus: "0 < n ⟹ (signed_take_bit :: nat ⇒ int ⇒ int)
(Suc (n - 1)) w = (signed_take_bit :: nat ⇒ int ⇒ int) n w"
  by auto

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

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

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

lemma sbintrunc_Suc_lem: "(signed_take_bit :: nat ⇒ int ⇒ int) (Suc
n) x = y ⟹ m = Suc n ⟹ (signed_take_bit :: nat ⇒ int ⇒ int) m x
= y"
  by (rule ssubst)
```

```
lemmas sbintrunc_Suc_Ialts =
  sbintrunc_Suc_Is [THEN sbintrunc_Suc_lem]

lemma sbintrunc_bintrunc_lt: "m > n ⟹ (signed_take_bit :: nat ⇒ int
⇒ int) n ((take_bit :: nat ⇒ int ⇒ int) m w) = (signed_take_bit ::
nat ⇒ int ⇒ int) n w"
  by (rule bin_eqI) (auto simp: nth_sbintr nth_bintr)

lemma bintrunc_sbintrunc_le: "m ≤ Suc n ⟹ (take_bit :: nat ⇒ int
⇒ int) m ((signed_take_bit :: nat ⇒ int ⇒ int) n w) = (take_bit ::
nat ⇒ int ⇒ int) m w"
  by (rule take_bit_signed_take_bit)

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

lemma bintrunc_sbintrunc' [simp]: "0 < n ⟹ (take_bit :: nat ⇒ int
⇒ int) n ((signed_take_bit :: nat ⇒ int ⇒ int) (n - 1) w) = (take_bit
:: nat ⇒ int ⇒ int) n w"
  by (metis Suc_diff_1 bintrunc_sbintrunc)

lemma sbintrunc_bintrunc' [simp]: "0 < n ⟹ (signed_take_bit :: nat
⇒ int ⇒ int) (n - 1) ((take_bit :: nat ⇒ int ⇒ int) n w) = (signed_take_bit
:: nat ⇒ int ⇒ int) (n - 1) w"
  by (simp add: sbintrunc_bintrunc_lt)

lemma bin_sbin_eq_iff: "(take_bit :: nat ⇒ int ⇒ int) (Suc n) x = (take_bit
:: nat ⇒ int ⇒ int) (Suc n) y ⟷ (signed_take_bit :: nat ⇒ int ⇒
int) n x = (signed_take_bit :: nat ⇒ int ⇒ int) n y"
  by (simp add: signed_take_bit_eq_iff_take_bit_eq)

lemma bin_sbin_eq_iff':
  "0 < n ⟹ (take_bit :: nat ⇒ int ⇒ int) n x = (take_bit :: nat ⇒
int ⇒ int) n y ⟷ (signed_take_bit :: nat ⇒ int ⇒ int) (n - 1) x =
(signed_take_bit :: nat ⇒ int ⇒ int) (n - 1) y"
  by (simp add: signed_take_bit_eq_iff_take_bit_eq)

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

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



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

```
lemma bintrunc_numeral:
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) x = of_bool (odd x) + 2
* (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (x div 2)"
  by (simp add: numeral_eq_Suc take_bit_Suc mod_2_eq_odd)

lemma sbintrunc_numeral:
  "(signed_take_bit :: nat ⇒ int ⇒ int) (numeral k) x = of_bool (odd
x) + 2 * (signed_take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (x div
2)"
  by (simp add: numeral_eq_Suc signed_take_bit_Suc mod2_eq_if)

lemma bintrunc_numeral_simps [simp]:
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) (numeral (Num.Bit0 w))
=
    2 * (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (numeral w)"
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) (numeral (Num.Bit1 w))
=
    1 + 2 * (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (numeral
w)"
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) (- numeral (Num.Bit0 w))
=
    2 * (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (- numeral w)"
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) (- numeral (Num.Bit1 w))
=
    1 + 2 * (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (- numeral
(w + Num.One))"
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) 1 = 1"
  by (simp_all add: bintrunc_numeral)

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

```
lemma no_bintr_alt1: "(take_bit :: nat ⇒ int ⇒ int) n = (λw. w mod
2 ^ n :: int)"
  by (rule ext) (rule bintrunc_mod2p)

lemma range_bintrunc: "range ((take_bit :: nat ⇒ int ⇒ int) n) = {i.
0 ≤ i ∧ i < 2 ^ n}"
  by (auto simp add: take_bit_eq_mod image_iff) (metis mod_pos_pos_trivial)

lemma no_sbintr_alt2: "signed_take_bit n = (λw. (w + 2 ^ n) mod 2 ^ Suc
n - 2 ^ n :: int)"
  by (rule ext) (simp only: signed_take_bit_eq_take_bit_shift flip: take_bit_eq_mod)

lemma range_sbintrunc: "range ((signed_take_bit :: nat ⇒ int ⇒ int)
n) = {i. - (2 ^ n) ≤ i ∧ i < 2 ^ n}"
proof -
  have ‹surj (λk::int. k + 2 ^ n)›
    by (rule surjI [of _ ‹(λk. k - 2 ^ n)›]) simp
  moreover have ‹(signed_take_bit :: nat ⇒ int ⇒ int) n = ((λk. k -
2 ^ n) ∘ take_bit (Suc n) ∘ (λk. k + 2 ^ n))›
    by (simp add: sbintrunc_eq_take_bit fun_eq_iff)
  ultimately show ?thesis
    apply (simp add: fun.set_map range_bintrunc set_eq_iff image_iff fun_eq_iff)
    by (metis sbintrunc_sbintrunc signed_take_bit_int_eq_self_iff)
qed

lemma sbintrunc_inc:
  ‹k + 2 ^ Suc n ≤ (signed_take_bit :: nat ⇒ int ⇒ int) n k› if ‹k
< - (2 ^ n)›
  using that by (fact signed_take_bit_int_greater_eq)

lemma sbintrunc_dec:
  ‹(signed_take_bit :: nat ⇒ int ⇒ int) n k ≤ k - 2 ^ (Suc n)› if ‹k
≥ 2 ^ n›
  using that by (fact signed_take_bit_int_less_eq)

lemma bintr_ge0: "0 ≤ (take_bit :: nat ⇒ int ⇒ int) n w"
  by (simp add: bintrunc_mod2p)

lemma bintr_lt2p: "(take_bit :: nat ⇒ int ⇒ int) n w < 2 ^ n"
  by (simp add: bintrunc_mod2p)

lemma bintr_Min: "(take_bit :: nat ⇒ int ⇒ int) n (- 1) = 2 ^ n - 1"
  by (simp add: stable_imp_take_bit_eq mask_eq_exp_minus_1)

lemma sbintr_ge: "- (2 ^ n) ≤ (signed_take_bit :: nat ⇒ int ⇒ int)
n w"
  by (fact signed_take_bit_int_greater_eq_minus_exp)
```

```
lemma sbintr_lt: "(signed_take_bit :: nat ⇒ int ⇒ int) n w < 2 ^ n"
  by (fact signed_take_bit_int_less_exp)

lemma bin_rest_trunc: "(λk::int. k div 2) ((take_bit :: nat ⇒ int ⇒
int) n bin) = (take_bit :: nat ⇒ int ⇒ int) (n - 1) ((λk::int. k div
2) bin)"
  by (simp add: take_bit_rec [of n bin])

lemma bin_rest_power_trunc:
  "((λk::int. k div 2) ^^ k) ((take_bit :: nat ⇒ int ⇒ int) n bin) =
(take_bit :: nat ⇒ int ⇒ int) (n - k) (((λk::int. k div 2) ^^ k) bin)"
  by (induct k) (auto simp: bin_rest_trunc)

lemma bin_rest_trunc_i: "(take_bit :: nat ⇒ int ⇒ int) n ((λk::int.
k div 2) bin) = (λk::int. k div 2) ((take_bit :: nat ⇒ int ⇒ int) (Suc
n) bin)"
  by (auto simp add: take_bit_Suc)

lemma bin_rest_strunc: "(λk::int. k div 2) ((signed_take_bit :: nat ⇒
int ⇒ int) (Suc n) bin) = (signed_take_bit :: nat ⇒ int ⇒ int) n ((λk::int.
k div 2) bin)"
  by (simp add: signed_take_bit_Suc)

lemma bintrunc_rest [simp]: "(take_bit :: nat ⇒ int ⇒ int) n ((λk::int.
k div 2) ((take_bit :: nat ⇒ int ⇒ int) n bin)) = (λk::int. k div 2)
((take_bit :: nat ⇒ int ⇒ int) n bin)"
  by (induct n arbitrary: bin) (simp_all add: take_bit_Suc)

lemma sbintrunc_rest [simp]: "(signed_take_bit :: nat ⇒ int ⇒ int)
n ((λk::int. k div 2) ((signed_take_bit :: nat ⇒ int ⇒ int) n bin))
= (λk::int. k div 2) ((signed_take_bit :: nat ⇒ int ⇒ int) n bin)"
  by (induct n arbitrary: bin) (simp_all add: signed_take_bit_Suc mod2_eq_if)

lemma bintrunc_rest': "(take_bit :: nat ⇒ int ⇒ int) n ∘ (λk::int.
k div 2) ∘ (take_bit :: nat ⇒ int ⇒ int) n = (λk::int. k div 2) ∘ (take_bit
:: nat ⇒ int ⇒ int) n"
  by (rule ext) auto

lemma sbintrunc_rest': "(signed_take_bit :: nat ⇒ int ⇒ int) n ∘ (λk::int.
k div 2) ∘ (signed_take_bit :: nat ⇒ int ⇒ int) n = (λk::int. k div
2) ∘ (signed_take_bit :: nat ⇒ int ⇒ int) n"
  by (rule ext) auto

lemma rco_lem:
  assumes "f ∘ g ∘ f = g ∘ f"
  shows "f ∘ (g ∘ f) ^^ n = g ^^ n ∘ f"
proof (induct n)
  case 0
  then show ?case
```

```
    by auto
next
  case (Suc n)
  then show ?case
    by (metis assms comp_assoc funpow_Suc_right)
qed

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

context
  includes bit_operations_syntax
begin

lemmas int_not_def = not_int_def

lemma int_not_simps:
  "NOT (0::int) = -1"
  "NOT (1::int) = -2"
  "NOT (- 1::int) = 0"
  "NOT (numeral w::int) = - numeral (w + Num.One)"
  "NOT (- numeral (Num.Bit0 w)::int) = numeral (Num.BitM w)"
  "NOT (- numeral (Num.Bit1 w)::int) = numeral (Num.Bit0 w)"
  by (simp_all add: not_int_def)

lemma int_not_not: "NOT (NOT x) = x"
  for x :: int
  by (fact bit.double_compl)

lemma int_and_0 [simp]: "0 AND x = 0"
  for x :: int
  by (fact bit.conj_zero_left)

lemma int_and_m1 [simp]: "-1 AND x = x"
  for x :: int
  by (fact and.left_neutral)

lemma int_or_zero [simp]: "0 OR x = x"
  for x :: int
  by (fact or.left_neutral)

lemma int_or_minus1 [simp]: "-1 OR x = -1"
  for x :: int
  by (fact bit.disj_one_left)

lemma int_xor_zero [simp]: "0 XOR x = x"
  for x :: int
```

```
  by (fact xor.left_neutral)

lemma bin_rest_NOT [simp]: "(λk::int. k div 2) (NOT x) = NOT ((λk::int.
k div 2) x)"
  by (fact not_int_div_2)

lemma bin_last_NOT [simp]: "(odd :: int ⇒ bool) (NOT x) ⟷ ¬ (odd
:: int ⇒ bool) x"
  by simp

lemma bin_rest_AND [simp]: "(λk::int. k div 2) (x AND y) = (λk::int.
k div 2) x AND (λk::int. k div 2) y"
  by (subst and_int_rec) auto

lemma bin_last_AND [simp]: "(odd :: int ⇒ bool) (x AND y) ⟷ (odd
:: int ⇒ bool) x ∧ (odd :: int ⇒ bool) y"
  by (subst and_int_rec) auto

lemma bin_rest_OR [simp]: "(λk::int. k div 2) (x OR y) = (λk::int. k
div 2) x OR (λk::int. k div 2) y"
  by (subst or_int_rec) auto

lemma bin_last_OR [simp]: "(odd :: int ⇒ bool) (x OR y) ⟷ (odd ::
int ⇒ bool) x ∨ (odd :: int ⇒ bool) y"
  by (subst or_int_rec) auto

lemma bin_rest_XOR [simp]: "(λk::int. k div 2) (x XOR y) = (λk::int.
k div 2) x XOR (λk::int. k div 2) y"
  by (subst xor_int_rec) auto

lemma bin_last_XOR [simp]: "(odd :: int ⇒ bool) (x XOR y) ⟷ ((odd
:: int ⇒ bool) x ∨ (odd :: int ⇒ bool) y) ∧ ¬ ((odd :: int ⇒ bool)
x ∧ (odd :: int ⇒ bool) y)"
  by (subst xor_int_rec) auto

lemma bin_nth_ops:
  "⋀x y. (bit :: int ⇒ nat ⇒ bool) (x AND y) n ⟷ (bit :: int ⇒ nat
⇒ bool) x n ∧ (bit :: int ⇒ nat ⇒ bool) y n"
  "⋀x y. (bit :: int ⇒ nat ⇒ bool) (x OR y) n ⟷ (bit :: int ⇒ nat
⇒ bool) x n ∨ (bit :: int ⇒ nat ⇒ bool) y n"
  "⋀x y. (bit :: int ⇒ nat ⇒ bool) (x XOR y) n ⟷ (bit :: int ⇒ nat
⇒ bool) x n ≠ (bit :: int ⇒ nat ⇒ bool) y n"
  "⋀x. (bit :: int ⇒ nat ⇒ bool) (NOT x) n ⟷ ¬ (bit :: int ⇒ nat
⇒ bool) x n"
  by (simp_all add: bit_and_iff bit_or_iff bit_xor_iff bit_not_iff)

lemma int_xor_minus1 [simp]: "-1 XOR x = NOT x"
  for x :: int
  by (fact bit.xor_one_left)
```

```
lemma int_xor_extra_simps [simp]:
  "w XOR 0 = w"
  "w XOR -1 = NOT w"
  for w :: int
  by simp_all

lemma int_or_extra_simps [simp]:
  "w OR 0 = w"
  "w OR -1 = -1"
  for w :: int
  by simp_all

lemma int_and_extra_simps [simp]:
  "w AND 0 = 0"
  "w AND -1 = w"
  for w :: int
  by simp_all
```

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"
  by (simp_all add: ac_simps)

lemma bin_ops_same [simp]:
  "x AND x = x"
  "x OR x = x"
  "x XOR x = 0"
  for x :: int
  by simp_all

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

lemma bbw_ao_absorb: "x AND (y OR x) = x ∧ x OR (y AND x) = x"
  for x y :: int
  by (auto simp add: bit_eq_iff bit_simps)

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
  by (auto simp add: bit_eq_iff bit_simps)
```

```
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
  by (auto simp add: bit_eq_iff bit_simps)

lemma int_and_assoc: "(x AND y) AND z = x AND (y AND z)"
  for x y z :: int
  by (auto simp add: bit_eq_iff bit_simps)

lemma int_or_assoc: "(x OR y) OR z = x OR (y OR z)"
  for x y z :: int
  by (auto simp add: bit_eq_iff bit_simps)

lemma int_xor_assoc: "(x XOR y) XOR z = x XOR (y XOR z)"
  for x y z :: int
  by (auto simp add: bit_eq_iff bit_simps)

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
  by (auto simp add: bit_eq_iff bit_simps)

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
  by (auto simp add: bit_eq_iff bit_simps)

lemma bbw_oa_dist: "(x AND y) OR z = (x OR z) AND (y OR z)"
  for x y z :: int
  by (auto simp add: bit_eq_iff bit_simps)

lemma bbw_ao_dist: "(x OR y) AND z = (x AND z) OR (y AND z)"
  for x y z :: int
  by (auto simp add: bit_eq_iff bit_simps)
```

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

```
lemma bin_rest_neg_numeral_BitM [simp]:
  "(λk::int. k div 2) (- numeral (Num.BitM w)) = - numeral w"
  by simp

lemma bin_last_neg_numeral_BitM [simp]:
```

```
  "(odd :: int ⇒ bool) (- numeral (Num.BitM w))"
  by simp
```

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

```
lemma bin_add_not: "x + NOT x = (-1::int)"
  by (simp add: not_int_def)

lemma AND_mod: "x AND (2 ^ n - 1) = x mod 2 ^ n"
  for x :: int
  by (simp flip: take_bit_eq_mod add: take_bit_eq_mask mask_eq_exp_minus_1)

lemma bin_trunc_ao:
  "(take_bit :: nat ⇒ int ⇒ int) n x AND (take_bit :: nat ⇒ int ⇒
int) n y = (take_bit :: nat ⇒ int ⇒ int) n (x AND y)"
  "(take_bit :: nat ⇒ int ⇒ int) n x OR (take_bit :: nat ⇒ int ⇒ int)
n y = (take_bit :: nat ⇒ int ⇒ int) n (x OR y)"
  by simp_all

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

lemma bin_trunc_not: "(take_bit :: nat ⇒ int ⇒ int) n (NOT ((take_bit
:: nat ⇒ int ⇒ int) n x)) = (take_bit :: nat ⇒ int ⇒ int) n (NOT x)"
  by (fact take_bit_not_take_bit)
```

Want theorems of the form of `bin_trunc_xor`.

```
lemma bintr_bintr_i: "x = (take_bit :: nat ⇒ int ⇒ int) n y ⟹ (take_bit
:: nat ⇒ int ⇒ int) n x = (take_bit :: nat ⇒ int ⇒ int) n y"
  by auto

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

lemma not_int_cmp_0 [simp]:
  fixes i :: int shows
  "0 < NOT i ⟷ i < -1"
  "0 ≤ NOT i ⟷ i < 0"
  "NOT i < 0 ⟷ i ≥ 0"
  "NOT i ≤ 0 ⟷ i ≥ -1"
by(simp_all add: int_not_def) arith+

lemma bbw_ao_dist2: "(x :: int) AND (y OR z) = x AND y OR x AND z"
  by (fact bit.conj_disj_distrib)

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"
```

```
  by simp

lemma int_nand_same_middle: fixes x :: int shows "x AND y AND NOT x =
0"
  by (simp add: bit_eq_iff bit_and_iff bit_not_iff)

lemma and_xor_dist: fixes x :: int shows
  "x AND (y XOR z) = (x AND y) XOR (x AND z)"
  by (fact bit.conj_xor_distrib)

lemma int_and_lt0 [simp]:
  ‹x AND y < 0 ⟷ x < 0 ∧ y < 0› for x y :: int
  by (fact and_negative_int_iff)

lemma int_and_ge0 [simp]:
  ‹x AND y ≥ 0 ⟷ x ≥ 0 ∨ y ≥ 0› for x y :: int
  by (fact and_nonnegative_int_iff)

lemma int_and_1: fixes x :: int shows "x AND 1 = x mod 2"
  by (fact and_one_eq)

lemma int_1_and: fixes x :: int shows "1 AND x = x mod 2"
  by (fact one_and_eq)

lemma int_or_lt0 [simp]:
  ‹x OR y < 0 ⟷ x < 0 ∨ y < 0› for x y :: int
  by (fact or_negative_int_iff)

lemma int_or_ge0 [simp]:
  ‹x OR y ≥ 0 ⟷ x ≥ 0 ∧ y ≥ 0› for x y :: int
  by (fact or_nonnegative_int_iff)

lemma int_xor_lt0 [simp]:
  ‹x XOR y < 0 ⟷ (x < 0) ≠ (y < 0)› for x y :: int
  by (fact xor_negative_int_iff)

lemma int_xor_ge0 [simp]:
  ‹x XOR y ≥ 0 ⟷ (x ≥ 0 ⟷ y ≥ 0)› for x y :: int
  by (fact xor_nonnegative_int_iff)

lemma even_conv_AND:
  ‹even i ⟷ i AND 1 = 0› for i :: int
  by (simp add: and_one_eq mod2_eq_if)

lemma bin_last_conv_AND:
  "(odd :: int ⇒ bool) i ⟷ i AND 1 ≠ 0"
  by (simp add: and_one_eq mod2_eq_if)

lemma bitval_bin_last:
```

```
  "of_bool ((odd :: int ⇒ bool) i) = i AND 1"
  by (simp add: and_one_eq mod2_eq_if)

lemma int_not_neg_numeral: "NOT (- numeral n) = (Num.sub n num.One ::
int)"
by(simp add: int_not_def)

lemma int_neg_numeral_pOne_conv_not: "- numeral (n + num.One) = (NOT
(numeral n) :: int)"
by(simp add: int_not_def)

lemma int_0_shiftl: "push_bit n 0 = (0 :: int)"
  by (fact push_bit_of_0)

lemma bin_last_shiftl: "odd (push_bit n x) ⟷ n = 0 ∧ (odd :: int ⇒
bool) x"
  by simp

lemma bin_rest_shiftl: "(λk::int. k div 2) (push_bit n x) = (if n > 0
then push_bit (n - 1) x else (λk::int. k div 2) x)"
  by (cases n) (simp_all add: push_bit_eq_mult)

lemma bin_nth_shiftl: "(bit :: int ⇒ nat ⇒ bool) (push_bit n x) m ⟷
n ≤ m ∧ (bit :: int ⇒ nat ⇒ bool) x (m - n)"
  by (fact bit_push_bit_iff_int)

lemma bin_last_shiftr: "odd (drop_bit n x) ⟷ bit x n" for x :: int
  by (simp add: bit_iff_odd_drop_bit)

lemma bin_rest_shiftr: "(λk::int. k div 2) (drop_bit n x) = drop_bit
(Suc n) x"
  by (simp add: drop_bit_Suc drop_bit_half)

lemma bin_nth_shiftr: "(bit :: int ⇒ nat ⇒ bool) (drop_bit n x) m =
(bit :: int ⇒ nat ⇒ bool) x (n + m)"
  by (simp add: bit_simps)

lemma bin_nth_conv_AND:
  fixes x :: int shows
  "(bit :: int ⇒ nat ⇒ bool) x n ⟷ x AND (push_bit n 1) ≠ 0"
  by (fact bit_iff_and_push_bit_not_eq_0)

lemma int_shiftl_numeral [simp]:
  "push_bit (numeral w') (numeral w :: int) = push_bit (pred_numeral w')
(numeral (num.Bit0 w))"
  "push_bit (numeral w') (- numeral w :: int) = push_bit (pred_numeral
w') (- numeral (num.Bit0 w))"
  by (fact push_bit_numeral push_bit_minus_numeral)+
```

```
lemma int_shiftl_One_numeral [simp]:
  "push_bit (numeral w) (1::int) = push_bit (pred_numeral w) 2"
  using int_shiftl_numeral [of Num.One w]
  by (simp only: numeral_eq_Suc push_bit_Suc) simp

lemma shiftl_ge_0: fixes i :: int shows "push_bit n i ≥ 0 ⟷ i ≥ 0"
  by (fact push_bit_nonnegative_int_iff)

lemma shiftl_lt_0: fixes i :: int shows "push_bit n i < 0 ⟷ i < 0"
  by (fact push_bit_negative_int_iff)

lemma int_shiftl_test_bit: "bit (push_bit i n :: int) m ⟷ m ≥ i ∧
bit n (m - i)"
  by (fact bit_push_bit_iff_int)

lemma int_0shiftr: "drop_bit x (0 :: int) = 0"
  by (fact drop_bit_of_0)

lemma int_minus1_shiftr: "drop_bit x (-1 :: int) = -1"
  by (fact drop_bit_minus_one)

lemma int_shiftr_ge_0: fixes i :: int shows "drop_bit n i ≥ 0 ⟷ i
≥ 0"
  by (fact drop_bit_nonnegative_int_iff)

lemma int_shiftr_lt_0 [simp]: fixes i :: int shows "drop_bit n i < 0
⟷ i < 0"
  by (fact drop_bit_negative_int_iff)

lemma int_shiftr_numeral [simp]:
  "drop_bit (numeral w') (1 :: int) = 0"
  "drop_bit (numeral w') (numeral num.One :: int) = 0"
  "drop_bit (numeral w') (numeral (num.Bit0 w) :: int) = drop_bit (pred_numeral
w') (numeral w)"
  "drop_bit (numeral w') (numeral (num.Bit1 w) :: int) = drop_bit (pred_numeral
w') (numeral w)"
  "drop_bit (numeral w') (- numeral (num.Bit0 w) :: int) = drop_bit (pred_numeral
w') (- numeral w)"
  "drop_bit (numeral w') (- numeral (num.Bit1 w) :: int) = drop_bit (pred_numeral
w') (- numeral (Num.inc w))"
  by (simp_all add: numeral_eq_Suc add_One drop_bit_Suc)

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

```
w)"
  by (simp_all add: drop_bit_Suc add_One)

lemmas bin_log_bintrs = bin_trunc_not bin_trunc_xor bin_trunc_and bin_trunc_or

lemma bintrunc_shiftl:
  "take_bit n (push_bit i m) = push_bit i (take_bit (n - i) m)"
  for m :: int
  by (fact take_bit_push_bit)

lemma bin_mask_conv_pow2:
  "mask n = 2 ^ n - (1 :: int)"
  by (fact mask_eq_exp_minus_1)

lemma bin_mask_ge0: "mask n ≥ (0 :: int)"
  by (fact mask_nonnegative_int)

context
  includes bit_operations_syntax
begin

lemma and_bin_mask_conv_mod: "x AND mask n = x mod 2 ^ n"
  for x :: int
  by (simp flip: take_bit_eq_mod add: take_bit_eq_mask)

end

end

lemma bin_mask_numeral:
  "mask (numeral n) = (1 :: int) + 2 * mask (pred_numeral n)"
  by (fact mask_numeral)

lemma bin_nth_mask: "bit (mask n :: int) i ⟷ i < n"
  by (simp add: bit_mask_iff)

lemma bin_mask_p1_conv_shift: "mask n + 1 = push_bit n (1 :: int)"
  by (simp add: inc_mask_eq_exp)

lemma sbintrunc_eq_in_range:
  "((signed_take_bit :: nat ⇒ int ⇒ int) n x = x) = (x ∈ range ((signed_take_bit
:: nat ⇒ int ⇒ int) n))"
  "(x = (signed_take_bit :: nat ⇒ int ⇒ int) n x) = (x ∈ range ((signed_take_bit
:: nat ⇒ int ⇒ int) n))"
  by (simp add: image_def, metis sbintrunc_sbintrunc)+

lemma sbintrunc_If:
  "- 3 * (2 ^ n) ≤ x ∧ x < 3 * (2 ^ n)
    ⟹ signed_take_bit n x = (if x < - (2 ^ n) then x + 2 * (2 ^ n)
```

```
      else if x ≥ 2 ^ n then x - 2 * (2 ^ n) else x)" for x :: int
  apply (simp add: no_sbintr_alt2)
  by (smt (verit, best) minus_mod_self2 mod_add_self2 mod_pos_pos_trivial)


lemma bintrunc_id:
  "⟦m ≤ int n; 0 < m⟧ ⟹ take_bit n m = m"
  by (simp add: take_bit_int_eq_self_iff le_less_trans)

end


theory More_Bit_Ring
  imports Main
begin



context semiring_bit_operations
begin

context
  includes bit_operations_syntax
begin

lemma disjunctive_add2:
  "(x AND y) = 0 ⟹ x + y = x OR y"
  by (metis disjunctive_add bit_0_eq bit_and_iff bot_apply bot_bool_def)

end
end

context ring_bit_operations
begin

context
  includes bit_operations_syntax
begin

lemma not_xor_is_eqv:
  "NOT (x XOR y) = (x AND y) OR (NOT x AND NOT y)"
  by (simp add: bit.xor_def bit.disj_conj_distrib or.commute)

lemma not_xor_eq_xor_not:
  "(NOT x) XOR y = x XOR (NOT y)"
  by simp

lemma not_minus:
  "NOT (x - y) = y - x - 1"
```

```
  by (simp add: not_eq_complement)

lemma minus_not_eq_plus_1:
  "- NOT x = x + 1"
  by (simp add: minus_eq_not_plus_1)

lemma not_minus_eq_minus_1:
  "NOT (- x) = x - 1"
  by (simp add: not_eq_complement)

lemma and_plus_not_and:
  "(x AND y) + (x AND NOT y) = x"
  by (metis and.left_commute and.right_neutral bit.conj_cancel_right bit.conj_disj_distrib
            bit.conj_zero_right bit.disj_cancel_right disjunctive_add2)

lemma or_eq_and_not_plus:
  "x OR y = (x AND NOT y) + y"
  by (simp add: and.assoc bit.disj_conj_distrib2 disjunctive_add2)

lemma and_eq_not_or_minus:
  "x AND y = (NOT x OR y) - NOT x"
  by (metis and.idem and_eq_not_not_or eq_diff_eq or.commute or.idem or_eq_and_not_plus)

lemma and_not_eq_or_minus:
  "x AND NOT y = (x OR y) - y"
  by (simp add: or_eq_and_not_plus)

lemma and_not_eq_minus_and:
  "x AND NOT y = x - (x AND y)"
  by (simp add: add.commute eq_diff_eq and_plus_not_and)

lemma or_minus_eq_minus_and:
  "(x OR y) - y = x - (x AND y)"
  by (metis and_not_eq_minus_and and_not_eq_or_minus)

lemma plus_eq_and_or:
  "x + y = (x OR y) + (x AND y)"
  using add_commute local.add.semigroup_axioms or_eq_and_not_plus semigroup.assoc
  by (fastforce simp: and_plus_not_and)

lemma xor_eq_or_minus_and:
  "x XOR y = (x OR y) - (x AND y)"
  by (metis (no_types) bit.de_Morgan_conj bit.xor_def2 bit_and_iff bit_or_iff
disjunctive_diff)

lemma not_xor_eq_and_plus_not_or:
  "NOT (x XOR y) = (x AND y) + NOT (x OR y)"
  by (metis (no_types, lifting) not_diff_distrib add.commute bit.de_Morgan_conj
bit.xor_def2
```

```
                                bit_and_iff bit_or_iff disjunctive_diff)

lemma not_xor_eq_and_minus_or:
  "NOT (x XOR y) = (x AND y) - (x OR y) - 1"
  by (metis not_diff_distrib add.commute minus_diff_eq not_minus_eq_minus_1
not_xor_eq_and_plus_not_or)

lemma plus_eq_xor_plus_carry:
  "x + y = (x XOR y) + 2 * (x AND y)"
  by (metis plus_eq_and_or add.commute add.left_commute diff_add_cancel
mult_2 xor_eq_or_minus_and)

lemma plus_eq_or_minus_xor:
  "x + y = 2 * (x OR y) - (x XOR y)"
  by (metis add_diff_cancel_left' diff_diff_eq2 local.mult_2 plus_eq_and_or
xor_eq_or_minus_and)

lemma plus_eq_minus_neg:
  "x + y = x - NOT y - 1"
  using add_commute local.not_diff_distrib not_minus
  by auto

lemma minus_eq_plus_neg:
  "x - y = x + NOT y + 1"
  by (simp add: add.semigroup_axioms diff_conv_add_uminus minus_eq_not_plus_1
semigroup.assoc)

lemma minus_eq_and_not_minus_not_and:
  "x - y = (x AND NOT y) - (NOT x AND y)"
  by (metis bit.de_Morgan_conj bit.double_compl not_diff_distrib plus_eq_and_or)

lemma minus_eq_xor_minus_not_and:
  "x - y = (x XOR y) - 2 * (NOT x AND y)"
  by (metis (no_types) bit.compl_eq_compl_iff bit.xor_compl_left not_diff_distrib
                       plus_eq_xor_plus_carry)

lemma minus_eq_and_not_minus_xor:
  "x - y = 2 * (x AND NOT y) - (x XOR y)"
  by (metis and.commute minus_diff_eq minus_eq_xor_minus_not_and xor.commute)

lemma and_one_neq_simps[simp]:
  "x AND 1 ≠ 0 ⟷ x AND 1 = 1"
  "x AND 1 ≠ 1 ⟷ x AND 1 = 0"
  by (clarsimp simp: and_one_eq)+

end
end

end
```

# 3 Lemmas on words

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

context
  includes bit_operations_syntax
begin
```

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

```
lemma sofl_test:
  ‹sint x + sint y = sint (x + y) ⟷
    drop_bit (size x - 1) ((x + y XOR x) AND (x + y XOR y)) = 0›
  for x y :: ‹'a::len word›
proof -
  obtain n where n: ‹LENGTH('a) = Suc n›
    by (cases ‹LENGTH('a)›) simp_all
  have *: ‹sint x + sint y + 2 ^ Suc n > signed_take_bit n (sint x +
sint y) ⟹ sint x + sint y ≥ - (2 ^ n)›
    ‹signed_take_bit n (sint x + sint y) > sint x + sint y - 2 ^ Suc n
⟹ 2 ^ n > sint x + sint y›
    using signed_take_bit_int_greater_eq [of ‹sint x + sint y› n] signed_take_bit_int_less_
[of n ‹sint x + sint y›]
    by (auto intro: ccontr)
  have ‹sint x + sint y = sint (x + y) ⟷
    (sint (x + y) < 0 ⟷ sint x < 0) ∨ (sint (x + y) < 0 ⟷ sint y
< 0)›
    by (smt (verit) One_nat_def add_diff_cancel_left' signed_take_bit_int_eq_self
sint_greater_eq sint_lt sint_word_ariths(1,2))
  then show ?thesis
    unfolding One_nat_def word_size drop_bit_eq_zero_iff_not_bit_last
bit_and_iff bit_xor_iff
    by (simp add: bit_last_iff)
qed

lemma unat_power_lower [simp]:
  "unat ((2::'a::len word) ^ n) = 2 ^ n" if "n < LENGTH('a::len)"
  using that by transfer simp

lemma unat_p2: "n < LENGTH('a :: len) ⟹ unat (2 ^ n :: 'a word) = 2
^ n"
  by (fact unat_power_lower)

lemma word_div_lt_eq_0:
  "x < y ⟹ x div y = 0" for x :: "'a :: len word"
```

```
  by (fact div_word_less)

lemma word_div_eq_1_iff: "n div m = 1 ⟷ n ≥ m ∧ unat n < 2 * unat
(m :: 'a :: len word)"
  by (metis One_nat_def nat_div_eq_Suc_0_iff of_nat_unat ucast_id unat_1
unat_div word_le_nat_alt)

lemma AND_twice [simp]:
  "(w AND m) AND m = w AND m"
  by (fact and.right_idem)

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›
  by (simp add: bit.conj_disj_distrib)

lemma p2_gt_0:
  "(0 < (2 ^ n :: 'a :: len word)) = (n < LENGTH('a))"
  by (simp add : word_gt_0 not_le)

lemma uint_2p_alt:
  ‹n < LENGTH('a::len) ⟹ uint ((2::'a::len word) ^ n) = 2 ^ n›
  using p2_gt_0 [of n, where ?'a = 'a] by (simp add: uint_2p)

lemma p2_eq_0:
  ‹(2::'a::len word) ^ n = 0 ⟷ LENGTH('a::len) ≤ n›
  by (fact exp_eq_zero_iff)

lemma p2len:
  ‹(2 :: 'a word) ^ LENGTH('a::len) = 0›
  by (fact word_pow_0)

lemma neg_mask_is_div:
  "w AND NOT (mask n) = (w div 2^n) * 2^n"
  for w :: ‹'a::len word›
  by (rule bit_word_eqI)
    (auto simp: bit_simps simp flip: push_bit_eq_mult drop_bit_eq_div)

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

lemma and_mask_arith:
  "w AND mask n = (w * 2^(size w - n)) div 2^(size w - n)"
  for w :: ‹'a::len word›
  by (rule bit_word_eqI)
    (auto simp: bit_simps word_size simp flip: push_bit_eq_mult drop_bit_eq_div)
```

```
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›
  by (rule and_mask_arith)

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

lemma add_mask_fold:
  "x + 2 ^ n - 1 = x + mask n"
  for x :: ‹'a::len word›
  by (simp add: mask_eq_decr_exp)

lemma word_and_mask_le_2pm1: "w AND mask n ≤ 2 ^ n - 1"
  for w :: ‹'a::len word›
  by (simp add: mask_2pm1[symmetric] word_and_le1)

lemma is_aligned_AND_less_0:
  "u AND mask n = 0 ⟹ v < 2^n ⟹ u AND v = 0"
  for u v :: ‹'a::len word›
  by (metis and_zero_eq less_mask_eq word_bw_lcs(1))

lemma and_mask_eq_iff_le_mask:
  ‹w AND mask n = w ⟷ w ≤ mask n›
  for w :: ‹'a::len word›
  by (smt (verit) and.idem mask_eq_iff word_and_le1 word_le_def)

lemma less_eq_mask_iff_take_bit_eq_self:
  ‹w ≤ mask n ⟷ take_bit n w = w›
  for w :: ‹'a::len word›
  by (simp add: and_mask_eq_iff_le_mask take_bit_eq_mask)

lemma NOT_eq:
  "NOT (x :: 'a :: len word) = - x - 1"
  by (fact not_eq_complement)

lemma NOT_mask: "NOT (mask n :: 'a::len word) = - (2 ^ n)"
  by (simp add : NOT_eq mask_2pm1)

lemma le_m1_iff_lt: "(x > (0 :: 'a :: len word)) = ((y ≤ x - 1) = (y
< x))"
  by uint_arith

lemma gt0_iff_gem1: ‹0 < x ⟷ x - 1 < x› for x :: ‹'a::len word›
  using le_m1_iff_lt by blast

lemma power_2_ge_iff:
  ‹2 ^ n - (1 :: 'a::len word) < 2 ^ n ⟷ n < LENGTH('a)›
  using gt0_iff_gem1 p2_gt_0 by blast
```

```
lemma le_mask_iff_lt_2n:
  "n < len_of TYPE ('a) = (((w :: 'a :: len word) ≤ mask n) = (w < 2 ^
n))"
  unfolding mask_2pm1 by (rule trans [OF p2_gt_0 [THEN sym] le_m1_iff_lt])

lemma mask_lt_2pn:
  ‹n < LENGTH('a) ⟹ mask n < (2 :: 'a::len word) ^ n›
  by (simp add: mask_eq_exp_minus_1 power_2_ge_iff)

lemma word_unat_power:
  "(2 :: 'a :: len word) ^ n = of_nat (2 ^ n)"
  by simp

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)"
  by (metis mod_less order_less_trans unat_of_nat word_less_nat_alt xlt)

lemma word_and_max_word:
  fixes a:: "'a::len word"
  shows "x = - 1 ⟹ a AND x = a"
  by simp

lemma word_and_full_mask_simp [simp]:
  ‹x AND mask LENGTH('a) = x› for x :: ‹'a::len word›
  by (simp add: bit_eq_iff bit_simps)

lemma of_int_uint [simp]: "of_int (uint x) = x"
  by (fact word_of_int_uint)

corollary word_plus_and_or_coroll:
  "x AND y = 0 ⟹ x + y = x OR y"
  for x y :: ‹'a::len word›
  by (fact disjunctive_add2)

corollary word_plus_and_or_coroll2:
  "(x AND w) + (x AND NOT w) = x"
  for x w :: ‹'a::len word›
  by (fact and_plus_not_and)

lemma unat_mask_eq:
  ‹unat (mask n :: 'a::len word) = mask (min LENGTH('a) n)›
  by transfer (simp add: nat_mask_eq)

lemma word_plus_mono_left:
  fixes x :: "'a :: len word"
  shows "⟦y ≤ z; x ≤ x + z⟧ ⟹ y + x ≤ z + x"
  by unat_arith
```

```
lemma less_Suc_unat_less_bound:
  "n < Suc (unat (x :: 'a :: len word)) ⟹ n < 2 ^ LENGTH('a)"
  by (auto elim!: order_less_le_trans intro: Suc_leI)

lemma up_ucast_inj:
  "⟦ ucast x = (ucast y::'b::len word); LENGTH('a) ≤ len_of TYPE ('b)
⟧ ⟹ x = (y::'a::len word)"
  by transfer (simp add: min_def split: if_splits)

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))"
  by (fastforce dest: up_ucast_inj)

lemma no_plus_overflow_neg:
  "(x :: 'a :: len word) < -y ⟹ x ≤ x + y"
  by (metis diff_minus_eq_add less_imp_le sub_wrap_lt)

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"
  by (meson le_trans up_ucast_inj)

lemma ucast_0_I:
  "x = 0 ⟹ ucast x = 0"
  by simp

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 (subst mn)
  from mnv mn have nv: "n < LENGTH('a)" and mv: "m < LENGTH('a)"  by
auto
  have uy: "unat y < 2 ^ n"
    using nv unat_mono yv by force
  have ux: "unat x < 2 ^ m"
    using mv unat_mono xv by fastforce
  have "unat x < 2 ^ (LENGTH('a :: len) - n)"
    by (metis exp_eq_zero_iff not_less0 linorder_not_le unat_mono unat_power_lower
```

```
unsigned_0 xv')
  then have *: "unat x * 2 ^ n < 2 ^ LENGTH('a)"
    by (simp add: nat_mult_power_less_eq)
  show "x * 2 ^ n + y < 2 ^ (m + n)" using ux uy nv mnv xv' *
    apply (simp add: word_less_nat_alt unat_word_ariths)
    by (metis less_imp_diff_less mn mod_nat_add nat_add_offset_less unat_power_lower
unsigned_less)
qed

lemma word_less_power_trans:
  fixes n :: "'a :: len word"
  assumes "n < 2 ^ (m - k)" "k ≤ m" "m < len_of TYPE ('a)"
  shows "2 ^ k * n < 2 ^ m"
proof -
  have "2 ^ k * unat n < 2 ^ LENGTH('a)"
  proof -
    have "(1::nat) < 2"
      by simp
    moreover
    have "m - k < len_of (TYPE('a)::'a itself)"
      by (simp add: assms less_imp_diff_less)
    with assms have "unat n < 2 ^ (m - k)"
      by (metis (no_types) unat_power_lower word_less_iff_unsigned)
    ultimately show ?thesis
      by (meson assms order.strict_trans nat_less_power_trans power_strict_increasing)
  qed
  then show ?thesis
    using assms nat_less_power_trans
    by (simp add: word_less_nat_alt unat_word_ariths)
qed

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"
  by (subst field_simps, rule word_less_power_trans)

lemma Suc_unat_diff_1:
  fixes x :: "'a :: len word"
  assumes lt: "1 ≤ x"
  shows "Suc (unat (x - 1)) = unat x"
  by (metis Suc_diff_1 linorder_not_less lt unat_gt_0 unat_minus_one word_less_1)

lemma word_eq_unatI:
  ‹v = w› if ‹unat v = unat w›
  using that by transfer (simp add: nat_eq_iff)

lemma word_div_sub:
  fixes x :: "'a :: len word"
```

```
  assumes "y ≤ x" "0 < y"
shows "(x - y) div y = x div y - 1"
  using assms  by (simp add: word_div_def div_pos_geq uint_minus_simple_alt
uint_sub_lem word_less_def)

lemma word_mult_less_mono1:
  fixes i :: "'a :: len word"
  assumes "i < j" and "0 < k"
    and "unat j * unat k < 2 ^ len_of TYPE ('a)"
  shows  "i * k < j * k"
  by (simp add: assms div_lt_mult word_div_mult)

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"
  using uik ujk ij
  by (auto simp: word_less_nat_alt iffD1 [OF unat_mult_lem] elim: mult_less_mono1)

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)"
  by (rule iffI [OF word_mult_less_dest [OF _ uik ujk] word_mult_less_mono1
[OF _ knz ujk]])

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 -
  note usv = order_le_less_trans [OF usszv szv]

  from usszv obtain q where qv: "sz = us + q"
    by (auto simp: le_iff_add)
  have "Suc (unat (((2:: 'a word) ^ sz - 1) div 2 ^ us)) =
    (2 ^ us + unat ((2:: 'a word) ^ sz - 1)) div 2 ^ us"
    by (simp add: le_div_geq unat_div usv)

  also have "... = ((2 ^ us - 1) + 2 ^ sz) div 2 ^ us" using szv
    by (simp add: unat_minus_one)
  also have "... = (2 ^ us - 1 + 2 ^ us * 2 ^ q) div 2 ^ us"
    by (simp add: power_add qv)
  also have "... = 2 ^ q + ((2 ^ us - 1) div 2 ^ us)"
```

```
    by (metis (no_types) not_less_zero div_mult_self2 take_bit_nat_less_exp)
  also have "... = 2 ^ (sz - us)" using qv by simp
  finally show ?thesis ..
qed

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

lemma length_enum_word_eq:
  ‹length (Enum.enum :: 'a::len word list) = 2 ^ LENGTH('a)›
  by (simp add: enum_word_def)

lemma unat_lt2p [iff]:
  ‹unat x < 2 ^ LENGTH('a)› for x :: ‹'a::len word›
  by transfer simp

lemma of_nat_unat [simp]:
  "of_nat ∘ unat = id"
  by (rule ext, simp)

lemma Suc_unat_minus_one [simp]:
  "x ≠ 0 ⟹ Suc (unat (x - 1)) = unat x"
  by (metis Suc_diff_1 unat_gt_0 unat_minus_one)

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"
  using uik ujk le
  by (auto simp: word_le_nat_alt iffD1 [OF unat_add_lem] elim: add_le_mono1)

lemma word_add_le_mono1:
  fixes i :: "'a :: len word"
  assumes "i ≤ j" and "unat j + unat k < 2 ^ len_of TYPE ('a)"
  shows  "i + k ≤ j + k"
  using assms no_olen_add_nat word_plus_mono_left by fastforce

lemma word_add_le_mono2:
  fixes i :: "'a :: len word"
  shows "⟦i ≤ j; unat j + unat k < 2 ^ LENGTH('a)⟧ ⟹ k + i ≤ k + j"
  by (metis add.commute no_olen_add_nat word_plus_mono_right)

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)"
  using assms word_add_le_dest word_add_le_mono1 by blast

lemma word_add_less_mono1:
  fixes i :: "'a :: len word"
  assumes "i < j" and "unat j + unat k < 2 ^ len_of TYPE ('a)"
  shows  "i + k < j + k"
  using assms no_olen_add_nat not_less_iff_gr_or_eq olen_add_eqv word_l_diffs(2)
by fastforce

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"
  using uik ujk le
  by (auto simp: word_less_nat_alt iffD1 [OF unat_add_lem] elim: add_less_mono1)

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)"
  using assms word_add_less_dest word_add_less_mono1 by blast

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)"
  using assms by (rule word_mult_less_cancel)

lemma word_le_imp_diff_le:
  fixes n :: "'a::len word"
  shows "⟦k ≤ n; n ≤ m⟧ ⟹ n - k ≤ m"
  by (auto simp: unat_sub word_le_nat_alt)

lemma word_less_imp_diff_less:
  fixes n :: "'a::len word"
  shows "⟦k ≤ n; n < m⟧ ⟹ n - k < m"
  by (clarsimp simp: unat_sub word_less_nat_alt
            intro!: less_imp_diff_less)

lemma word_mult_le_mono1:
  fixes i :: "'a :: len word"
```

```
  assumes ij: "i ≤ j"  "0 < k"
  and "unat j * unat k < 2 ^ len_of TYPE ('a)"
  shows  "i * k ≤ j * k"
  by (simp add: assms div_le_mult word_div_mult)

lemma word_mult_le_iff:
  fixes i :: "'a :: len word"
  assumes "0 < k"
  and     "unat i * unat k < 2 ^ len_of TYPE ('a)"
  and     "unat j * unat k < 2 ^ len_of TYPE ('a)"
  shows  "(i * k ≤ j * k) = (i ≤ j)"
  by (metis assms div_le_mult nle_le word_div_mult)


lemma word_diff_less:
  fixes n :: "'a :: len word"
  shows "⟦0 < n; 0 < m; n ≤ m⟧ ⟹ m - n < m"
  by (metis linorder_not_le sub_wrap word_greater_zero_iff)

lemma word_add_increasing:
  fixes x :: "'a :: len word"
  shows "⟦ p + w ≤ x; p ≤ p + w ⟧ ⟹ p ≤ x"
  by unat_arith

lemma word_random:
  fixes x :: "'a :: len word"
  shows "⟦ p ≤ p + x'; x ≤ x' ⟧ ⟹ p ≤ p + x"
  by unat_arith

lemma word_sub_mono:
  "⟦ a ≤ c; d ≤ b; a - b ≤ a; c - d ≤ c ⟧
    ⟹ (a - b) ≤ (c - d :: 'a :: len word)"
  by unat_arith

lemma power_not_zero:
  "n < LENGTH('a::len) ⟹ (2 :: 'a word) ^ n ≠ 0"
  by (metis p2_gt_0 word_neq_0_conv)

lemma word_gt_a_gt_0:
  "a < n ⟹ (0 :: 'a::len word) < n"
  using word_gt_0 word_not_simps(1) by blast

lemma word_power_less_1 [simp]:
  "sz < LENGTH('a::len) ⟹ (2::'a word) ^ sz - 1 < 2 ^ sz"
  using power_2_ge_iff by blast

lemma word_sub_1_le:
  "x ≠ 0 ⟹ x - 1 ≤ (x :: 'a :: len word)"
  by (simp add: word_le_sub1 word_sub_le)
```

```
lemma unat_less_power:
  fixes k :: "'a::len word"
  assumes szv: "sz < LENGTH('a)"
  and     kv:  "k < 2 ^ sz"
  shows   "unat k < 2 ^ sz"
  using szv unat_mono [OF kv] by simp

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 (cases ‹sz < LENGTH('a)›)
  case True
  with assms show ?thesis
    by (simp add: unat_word_ariths take_bit_eq_mod mod_simps unsigned_of_nat)
      (simp add: take_bit_nat_eq_self_iff nat_less_power_trans flip: take_bit_eq_mod)
next
  case False
  with assms show ?thesis
    by simp
qed

lemma word_plus_mcs_4:
  "⟦v + x ≤ w + x; x ≤ v + x⟧ ⟹ v ≤ (w::'a::len word)"
  by uint_arith

lemma word_plus_mcs_3:
  "⟦v ≤ w; x ≤ w + x⟧ ⟹ v + x ≤ w + (x::'a::len word)"
  by unat_arith

lemma word_le_minus_one_leq:
  "x < y ⟹ x ≤ y - 1" for x :: "'a :: len word"
  by transfer (metis le_less_trans less_irrefl take_bit_decr_eq take_bit_nonnegative
zle_diff1_eq)

lemma word_less_sub_le[simp]:
  fixes x :: "'a :: len word"
  assumes nv: "n < LENGTH('a)"
  shows "(x ≤ 2 ^ n - 1) = (x < 2 ^ n)"
  using le_less_trans word_le_minus_one_leq nv power_2_ge_iff by blast

lemma unat_of_nat_len:
  "x < 2 ^ LENGTH('a) ⟹ unat (of_nat x :: 'a::len word) = x"
  by (simp add: unsigned_of_nat take_bit_nat_eq_self_iff)

lemma unat_of_nat_eq:
  "x < 2 ^ LENGTH('a) ⟹ unat (of_nat x ::'a::len word) = x"
  by (rule unat_of_nat_len)
```

```
lemma unat_eq_of_nat:
  "n < 2 ^ LENGTH('a) ⟹ (unat (x :: 'a::len word) = n) = (x = of_nat
n)"
  by transfer
    (auto simp: take_bit_of_nat nat_eq_iff take_bit_nat_eq_self_iff intro:
sym)

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 -
  have kn: "unat (of_nat k :: 'a word) * unat ((2::'a word) ^ n) < 2 ^
LENGTH('a)"
    using assms
    by (metis le_unat_uoi mult.commute nat_le_linear nat_less_power_trans
not_less unat_power_lower)

  have "unat a div 2 ^ n * 2 ^ n ≠ unat a"
    using assms
    by (metis Abs_fnat_hom_0 mod_mult_self2_is_0 unat_power_lower word_arith_nat_mod)

  then have "a div 2 ^ n * 2 ^ n < a" using assms
    by (metis add_cancel_right_right le_less word_div_mult_le word_mod_div_equality)

  also from xk le have "... ≤ of_nat k * 2 ^ n" by (simp add: field_simps)
  finally show ?thesis using sz kv
    by (smt (verit) div_mult_le kn order_le_less_trans unat_div unsigned_less
word_mult_less_dest)
qed

lemma mask_out_sub_mask:
  "(x AND NOT (mask n)) = x - (x AND (mask n))"
  for x :: ‹'a::len word›
  by (fact and_not_eq_minus_and)

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›
  by (auto simp: and_not_eq_minus_and)

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

```
  using take_bit_int_eq_self_iff [of n ‹take_bit LENGTH('a) (uint w)›]
  by (transfer fixing: n) auto

lemma word_power_increasing:
  assumes x: "2 ^ x < (2 ^ y::'a::len word)" "x < LENGTH('a)" "y < LENGTH('a)"
  shows "x < y" using x
  using assms by transfer simp

lemma mask_twice:
  "(x AND mask n) AND mask m = x AND mask (min m n)"
  for x :: ‹'a::len word›
  by (simp flip: take_bit_eq_mask)

lemma plus_one_helper[elim!]:
  "x < n + (1 :: 'a :: len word) ⟹ x ≤ n"
  using inc_le linorder_not_le by blast

lemma plus_one_helper2:
  "⟦ x ≤ n; n + 1 ≠ 0 ⟧ ⟹ x < n + (1 :: 'a :: len word)"
  by (simp add: word_less_nat_alt word_le_nat_alt field_simps
                unatSuc)

lemma less_x_plus_1:
  "x ≠ - 1 ⟹ (y < x + 1) = (y < x ∨ y = x)" for x :: "'a::len word"
  by (meson max_word_wrap plus_one_helper plus_one_helper2 word_le_less_eq)

lemma word_Suc_leq:
  fixes k:: "'a::len word" shows "k ≠ - 1 ⟹ x < k + 1 ⟷ x ≤ k"
  using less_x_plus_1 word_le_less_eq by auto

lemma word_Suc_le:
   fixes k:: "'a::len word" shows "x ≠ - 1 ⟹ x + 1 ≤ k ⟷ x < k"
  by (meson not_less word_Suc_leq)

lemma word_lessThan_Suc_atMost:
  ‹{..< k + 1} = {..k}› if ‹k ≠ - 1› for k :: ‹'a::len word›
  using that by (simp add: lessThan_def atMost_def word_Suc_leq)

lemma word_atLeastLessThan_Suc_atLeastAtMost:
  ‹{l ..< u + 1} = {l..u}› if ‹u ≠ - 1› for l :: ‹'a::len word›
  using that by (simp add: atLeastAtMost_def atLeastLessThan_def word_lessThan_Suc_atMost)

lemma word_atLeastAtMost_Suc_greaterThanAtMost:
  ‹{m<..u} = {m + 1..u}› if ‹m ≠ - 1› for m :: ‹'a::len word›
  using that by (simp add: greaterThanAtMost_def greaterThan_def atLeastAtMost_def
atLeast_def word_Suc_le)

lemma word_atLeastLessThan_Suc_atLeastAtMost_union:
  fixes l:: "'a::len word"
```

```
  assumes "m ≠ - 1" and "l ≤ m" and "m ≤ u"
  shows "{l..m} ∪ {m+1..u} = {l..u}"
  by (metis assms ivl_disj_un_two(8) word_atLeastAtMost_Suc_greaterThanAtMost)

lemma max_word_less_eq_iff [simp]:
  ‹- 1 ≤ w ⟷ w = - 1› for w :: ‹'a::len word›
  by (fact word_order.extremum_unique)

lemma word_or_zero:
  "(a OR b = 0) = (a = 0 ∧ b = 0)"
  for a b :: ‹'a::len word›
  by (fact or_eq_0_iff)

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"
  by (smt (verit) suc_n le_less_trans lessI nat_less_le nat_mult_less_cancel_disj
p2_gt_0
         power_Suc power_Suc unat_power_lower word_less_nat_alt x)

lemma power_overflow:
  "n ≥ LENGTH('a) ⟹ 2 ^ n = (0 :: 'a::len word)"
  by simp

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)"
  by (fact signed_1)

lemma ucast_of_nat:
  "is_down (ucast :: 'a :: len word ⇒ 'b :: len word)
    ⟹ ucast (of_nat n :: 'a word) = (of_nat n :: 'b word)"
  by transfer simp
```

```
lemma scast_1':
  "(scast (1::'a::len word) :: 'b::len word) =
   (word_of_int (signed_take_bit (LENGTH('a::len) - Suc 0) (1::int)))"
  by transfer simp

lemma scast_1:
  "(scast (1::'a::len word) :: 'b::len word) = (if LENGTH('a) = 1 then
-1 else 1)"
  by (fact signed_1)

lemma unat_minus_one_word:
  "unat (-1 :: 'a :: len word) = 2 ^ LENGTH('a) - 1"
  by (simp add: mask_eq_exp_minus_1 unsigned_minus_1_eq_mask)

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"
  by (simp add: word_le_nat_alt)

lemma word_leq_le_minus_one:
  "⟦ x ≤ y; x ≠ 0 ⟧ ⟹ x - 1 < (y :: 'a :: len word)"
  by (meson le_m1_iff_lt linorder_not_less word_greater_zero_iff)

lemma neg_mask_combine:
  "NOT(mask a) AND NOT(mask b) = NOT(mask (max a b) :: 'a::len word)"
  by (rule bit_word_eqI) (auto simp: bit_simps)

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›
  by (rule bit_word_eqI) (auto simp: bit_simps)

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›
  by (metis (no_types, lifting) add.commute add_diff_eq and.assoc and_not_eq_minus_and
and_plus_not_and mask_twice min_def)

lemma word_of_nat_less: "n < unat x ⟹ of_nat n < x"
  by (metis le_unat_uoi nat_less_le word_less_nat_alt)

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

```
  by (metis mask_eq_exp_minus_1 min.commute unat_mask_eq)

lemma mask_over_length:
  "LENGTH('a) ≤ n ⟹ mask n = (-1::'a::len word)"
  by (simp add: mask_eq_decr_exp)

lemma Suc_2p_unat_mask:
  "n < LENGTH('a) ⟹ Suc (2 ^ n * k + unat (mask n :: 'a::len word))
= 2 ^ n * (k+1)"
  by (simp add: unat_mask)

lemma sint_of_nat_ge_zero:
  "x < 2 ^ (LENGTH('a) - 1) ⟹ sint (of_nat x :: 'a :: len word) ≥ 0"
  by (simp add: bit_iff_odd signed_of_nat)

lemma int_eq_sint:
  assumes "x < 2 ^ (LENGTH('a) - 1)"
  shows "sint (of_nat x :: 'a :: len word) = int x"
proof -
  have "int x < 2 ^ (len_of (TYPE('a)::'a itself) - 1)"
    by (metis assms of_nat_less_iff of_nat_numeral of_nat_power)
  then show ?thesis
    by (smt (verit) One_nat_def id_apply of_int_eq_id of_nat_0_le_iff
signed_of_nat signed_take_bit_int_eq_self)
qed

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)"
  by (simp add: int_eq_sint less_imp_diff_less)

lemma word_le_not_less:
  fixes b :: "'a::len word"
  shows "b ≤ a ⟷ ¬ a < b"
  by fastforce

lemma less_is_non_zero_p1:
  fixes a :: "'a :: len word"
  shows "a < k ⟹ a + 1 ≠ 0"
  using linorder_not_le max_word_wrap by auto

lemma unat_add_lem':
  fixes y :: "'a::len word"
  shows "(unat x + unat y < 2 ^ LENGTH('a)) ⟹ (unat (x + y) = unat
x + unat y)"
  using unat_add_lem by blast

lemma word_less_two_pow_divI:
```

```
  "⟦ (x :: 'a::len word) < 2 ^ (n - m); m ≤ n; n < LENGTH('a) ⟧ ⟹ x
< 2 ^ n div 2 ^ m"
  by (simp add: word_less_nat_alt power_minus_is_div unat_div)

lemma word_less_two_pow_divD:
  fixes x :: "'a::len word"
  assumes "x < 2 ^ n div 2 ^ m"
  shows "n ≥ m ∧ (x < 2 ^ (n - m))"
proof -
  have f2: "unat x < unat ((2::'a word) ^ n div 2 ^ m)"
    using assms by (simp add: word_less_nat_alt)
  then have f3: "0 < unat ((2::'a word) ^ n div 2 ^ m)"
    using order_le_less_trans by blast
  have f4: "n < LENGTH('a)"
    by (metis assms div_0 possible_bit_def possible_bit_word word_zero_le
[THEN leD])
  then have "2 ^ n div 2 ^ m = unat ((2::'a word) ^ n div 2 ^ m)"
    by (metis div_by_0 exp_eq_zero_iff f3 linorder_not_le unat_div unat_gt_0
unat_power_lower)
  then show ?thesis
    by (metis assms f2 power_minus_is_div two_pow_div_gt_le unat_div word_arith_nat_div
word_unat_power)
qed

lemma of_nat_less_two_pow_div_set:
  assumes "n < LENGTH('a)"
  shows  "{x. x < (2 ^ n div 2 ^ m :: 'a::len word)} = of_nat ` {k. k
< 2 ^ n div 2 ^ m}"
proof -
  have "∃k<2 ^ n div 2 ^ m. w = word_of_nat k"
    if "w < 2 ^ n div 2 ^ m" for w :: "'a word"
    using that assms
    by (metis less_imp_diff_less power_minus_is_div unat_less_power unat_of_nat_len
word_less_two_pow_divD word_nchotomy)
  moreover have "(word_of_nat k::'a word) < 2 ^ n div 2 ^ m"
    if "k < 2 ^ n div 2 ^ m" for k
    using that assms
    by (metis order_le_less_trans two_pow_div_gt_le unat_div unat_power_lower
word_of_nat_less)
  ultimately show ?thesis
    by (auto simp: word_of_nat_less)
qed

lemma ucast_less:
  "LENGTH('b) < LENGTH('a) ⟹
   (ucast (x :: 'b :: len word) :: ('a :: len word)) < 2 ^ LENGTH('b)"
  by transfer simp

lemma ucast_range_less:
```

```
  "LENGTH('a :: len) < LENGTH('b :: len) ⟹
   range (ucast :: 'a word ⇒ 'b word) = {x. x < 2 ^ len_of TYPE ('a)}"
  apply safe
  apply (simp add: ucast_less)
  by (metis (mono_tags, opaque_lifting) UNIV_I Word.of_nat_unat image_eqI
unat_eq_of_nat unat_less_power unat_lt2p)

lemma word_power_less_diff:
  fixes q :: "'a::len word"
  assumes  "2 ^ n * q < 2 ^ m" and "q < 2 ^ (LENGTH('a) - n)"
    shows "q < 2 ^ (m - n)"
proof (cases "m ≥ LENGTH('a) ∨ n ≥ LENGTH('a) ∨ n=0")
  case True
  then show ?thesis
    using assms
    by (elim context_disjE; simp add: power_overflow)
next
  case False
  have "2 ^ n * unat q < 2 ^ m"
    by (metis assms p2_gt_0 unat_eq_of_nat unat_less_power unat_lt2p unat_mult_power_lem
word_gt_a_gt_0)
  with assms have "unat q < 2 ^ (m - n)"
    using nat_power_less_diff by blast
  then show ?thesis
    using False word_less_nat_alt by fastforce
qed

lemma word_less_sub_1:
  "x < (y :: 'a :: len word) ⟹ x ≤ y - 1"
  by (fact word_le_minus_one_leq)

lemma word_sub_mono2:
  "⟦ a + b ≤ c + d; c ≤ a; b ≤ a + b; d ≤ c + d ⟧ ⟹ b ≤ (d :: 'a ::
len word)"
  using add_diff_cancel_left' word_le_minus_mono by fastforce

lemma word_not_le:
  "(¬ x ≤ (y :: 'a :: len word)) = (y < x)"
  by (fact not_le)

lemma word_subset_less:
  fixes s :: "'a :: len word"
  assumes "{x..x + r - 1} ⊆ {y..y + s - 1}"
      and xy: "x ≤ x + r - 1" "y ≤ y + s - 1"
      and "s ≠ 0"
    shows "r ≤ s"
proof -
  obtain "x ≤ y + (s - 1)" "y ≤ x" "x + (r - 1) ≤ y + (s - 1)"
    using assms by (auto simp flip: add_diff_eq)
```

```
  then have "r - 1 ≤ s - 1"
    by (metis add_diff_eq xy olen_add_eqv word_sub_mono2)
  then show ?thesis
    using ‹s ≠ 0› word_le_minus_cancel word_le_sub1 by auto
qed

lemma uint_power_lower:
  "n < LENGTH('a) ⟹ uint (2 ^ n :: 'a :: len word) = (2 ^ n :: int)"
  by (rule uint_2p_alt)

lemma power_le_mono:
  "⟦2 ^ n ≤ (2::'a::len word) ^ m; n < LENGTH('a); m < LENGTH('a)⟧ ⟹
n ≤ m"
  by (simp add: word_le_nat_alt)

lemma two_power_eq:
  "⟦n < LENGTH('a); m < LENGTH('a)⟧
   ⟹ ((2::'a::len word) ^ n = 2 ^ m) = (n = m)"
  by (metis nle_le power_le_mono)

lemma unat_less_helper:
  "x < of_nat n ⟹ unat x < n"
  by (metis not_less_iff_gr_or_eq word_less_nat_alt word_of_nat_less)

lemma nat_uint_less_helper:
  "nat (uint y) = z ⟹ x < y ⟹ nat (uint x) < z"
  using nat_less_eq_zless uint_lt_0 word_less_iff_unsigned by blast

lemma of_nat_0:
  "⟦of_nat n = (0::'a::len word); n < 2 ^ LENGTH('a)⟧ ⟹ n = 0"
  by (auto simp: word_of_nat_eq_0_iff)

lemma of_nat_inj:
  "⟦x < 2 ^ LENGTH('a); y < 2 ^ LENGTH('a)⟧ ⟹
   (of_nat x = (of_nat y :: 'a :: len word)) = (x = y)"
  by (metis unat_of_nat_len)

lemma div_to_mult_word_lt:
  "⟦ (x :: 'a :: len word) ≤ y div z ⟧ ⟹ x * z ≤ y"
  by (cases "z = 0") (simp_all add: div_le_mult word_neq_0_conv)

lemma ucast_ucast_mask:
  "(ucast :: 'a :: len word ⇒ 'b :: len word) (ucast x) = x AND mask
(len_of TYPE ('a))"
  by (metis Word.of_int_uint and_mask_bintr unsigned_ucast_eq)

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

```
  by (simp add: less_mask_eq ucast_ucast_mask)

lemma ucast_ucast_id:
  "LENGTH('a) < LENGTH('b) ⟹ ucast (ucast (x::'a::len word)::'b::len
word) = x"
  using is_up less_or_eq_imp_le ucast_up_ucast_id by blast

lemma unat_ucast:
  "unat (ucast x :: ('a :: len) word) = unat x mod 2 ^ (LENGTH('a))"
  by (metis Word.of_nat_unat unat_of_nat)

lemma ucast_less_ucast:
  "LENGTH('a) ≤ LENGTH('b) ⟹
   (ucast x < ((ucast (y :: 'a::len word)) :: 'b::len word)) = (x < y)"
  by (metis Word.of_nat_unat is_up not_less_iff_gr_or_eq ucast_up_ucast_id
word_of_nat_less)

— This weaker version was previously called ucast_less_ucast. We retain it to
support existing proofs.
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)"
  by (metis add.commute max_word_wrap unatSuc)

lemma word_div_1:
  "(n :: 'a :: len word) div 1 = n"
  by (fact div_by_1)

lemma word_minus_one_le:
  "-1 ≤ (x :: 'a :: len word) = (x = -1)"
  by (fact word_order.extremum_unique)

lemma up_scast_inj:
  "⟦ scast x = (scast y :: 'b :: len word); size x ≤ LENGTH('b) ⟧ ⟹
x = y"
  by (metis is_up scast_up_scast_id word_size)

lemma up_scast_inj_eq:
  "LENGTH('a) ≤ len_of TYPE ('b) ⟹
  (scast x = (scast y::'b::len word)) = (x = (y::'a::len word))"
  by (metis is_up scast_up_scast_id)

lemma word_le_add:
  fixes x :: "'a :: len word"
  shows "x ≤ y ⟹ ∃n. y = x + of_nat n"
  by (rule exI [where x = "unat (y - x)"]) simp
```

```
lemma word_plus_mcs_4':
  "⟦x + v ≤ x + w; x ≤ x + v⟧ ⟹ v ≤ w" for x :: "'a::len word"
  by (meson olen_add_eqv order_refl word_add_increasing word_sub_mono2)

lemma unat_eq_1:
  ‹unat x = Suc 0 ⟷ x = 1›
  by (auto intro!: unsigned_word_eqI [where ?'a = nat])

lemma word_unat_Rep_inject1:
  ‹unat x = unat 1 ⟷ x = 1›
  by (simp add: unat_eq_1)

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›
  by (rule bit_word_eqI) (auto simp: bit_simps)

lemma word_less_cases:
  "x < y ⟹ x = y - 1 ∨ x < y - (1 ::'a::len word)"
  by (meson order_le_imp_less_or_eq word_le_minus_one_leq)

lemma mask_and_mask:
  "mask a AND mask b = (mask (min a b) :: 'a::len word)"
  by (simp flip: take_bit_eq_mask ac_simps)

lemma mask_eq_0_eq_x:
  "(x AND w = 0) = (x AND NOT w = x)"
  for x w :: ‹'a::len word›
  by (simp add: and_not_eq_minus_and)

lemma mask_eq_x_eq_0:
  "(x AND w = x) = (x AND NOT w = 0)"
  for x w :: ‹'a::len word›
  by (metis and_not_eq_minus_and eq_iff_diff_eq_0)

lemma compl_of_1: "NOT 1 = (-2 :: 'a :: len word)"
  by (fact not_one_eq)

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›
  by (metis word_bw_comms(1) word_plus_and_or_coroll2)

lemma word_FF_is_mask:
  "0xFF = (mask 8 :: 'a::len word)"
  by (simp add: mask_eq_decr_exp)

lemma word_1FF_is_mask:
  "0x1FF = (mask 9 :: 'a::len word)"
```

```
  by (simp add: mask_eq_decr_exp)

lemma ucast_of_nat_small:
  "x < 2 ^ LENGTH('a) ⟹ ucast (of_nat x :: 'a :: len word) = (of_nat
x :: 'b :: len word)"
  by (metis Word.of_nat_unat of_nat_inverse)

lemma word_le_make_less:
  fixes x :: "'a :: len word"
  shows "y ≠ -1 ⟹ (x ≤ y) = (x < (y + 1))"
  by (simp add: word_Suc_leq)

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

lemma word_to_1_set:
  "{0 ..< (1 :: 'a :: len word)} = {0}"
  by fastforce

lemma word_leq_minus_one_le:
  fixes x :: "'a::len word"
  shows "⟦y ≠ 0; x ≤ y - 1 ⟧ ⟹ x < y"
  using le_m1_iff_lt word_neq_0_conv by blast

lemma word_count_from_top:
  "n ≠ 0 ⟹ {0 ..< n :: 'a :: len word} = {0 ..< n - 1} ∪ {n - 1}"
  using word_leq_minus_one_le word_less_cases by force

lemma word_minus_one_le_leq:
  "⟦ x - 1 < y ⟧ ⟹ x ≤ (y :: 'a :: len word)"
  using diff_add_cancel inc_le by force

lemma word_must_wrap:
  "⟦ x ≤ n - 1; n ≤ x ⟧ ⟹ n = (0 :: 'a :: len word)"
  using dual_order.trans sub_wrap word_less_1 by blast

lemma range_subset_card:
  "⟦ {a :: 'a :: len word .. b} ⊆ {c .. d}; b ≥ a ⟧ ⟹ d ≥ c ∧ d - c
≥ b - a"
  using word_sub_le word_sub_mono by fastforce

lemma less_1_simp:
  "n - 1 < m = (n ≤ (m :: 'a :: len word) ∧ n ≠ 0)"
  by unat_arith

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)"
  by (metis drop_bit_eq_div drop_bit_take_bit take_bit_eq_mod)
```

```
lemma word_range_minus_1':
  fixes a :: "'a :: len word"
  shows "a ≠ 0 ⟹ {a-1<..b} = {a..b}"
  by (simp add: word_atLeastAtMost_Suc_greaterThanAtMost)

lemma word_range_minus_1:
  fixes a :: "'a :: len word"
  shows "b ≠ 0 ⟹ {a..b - 1} = {a..<b}"
  by (auto simp: word_le_minus_one_leq word_leq_minus_one_le)

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

lemma overflow_plus_one_self:
  "(1 + p ≤ p) = (p = (-1 :: 'a :: len word))"
  by (metis add.commute order_less_irrefl word_Suc_le word_order.extremum)

lemma plus_1_less:
  "(x + 1 ≤ (x :: 'a :: len word)) = (x = -1)"
  using word_Suc_leq by blast

lemma pos_mult_pos_ge:
  "[|x > (0::int); n>=0 |] ==> n * x >= n*1"
  by (simp add: mult_le_cancel_left1)

lemma word_plus_strict_mono_right:
  fixes x :: "'a :: len word"
  shows "⟦y < z; x ≤ x + z⟧ ⟹ x + y < x + z"
  by unat_arith

lemma word_div_mult:
  "0 < c ⟹ a < b * c ⟹ a div c < b" for a b c :: "'a::len word"
  by (metis antisym_conv3 div_lt_mult leD order.asym word_div_mult_le)

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"
  by (simp add: word_less_power_trans2 word_of_nat_less)

lemma word_1_le_power:
  "n < LENGTH('a) ⟹ (1 :: 'a :: len word) ≤ 2 ^ n"
  by (metis bot_nat_0.extremum power_0 two_power_increasing)

lemma unat_1_0:
  "1 ≤ (x::'a::len word) = (0 < unat x)"
  by (auto simp: word_le_nat_alt)
```

```
lemma x_less_2_0_1':
  fixes x :: "'a::len word"
  shows "⟦LENGTH('a) ≠ 1; x < 2⟧ ⟹ x = 0 ∨ x = 1"
  by (metis One_nat_def less_2_cases of_nat_numeral unat_less_helper unsigned_0
unsigned_1 unsigned_word_eqI)

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"
  by (simp add: word_of_nat_less)

lemma of_nat_n_less_equal_power_2:
  "n < LENGTH('a::len) ⟹ ((of_nat n)::'a word) < 2 ^ n"
  by (simp add: More_Word.of_nat_power)

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"
  by (metis and_mask_less' eqm sz)

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))"
  by (simp add: unat_of_nat_len word_less_nat_alt xlt ylt)

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)"
  by (metis unat_of_nat_len word_less_eq_iff_unsigned)

lemma mask_AND_NOT_mask:
  "(w AND NOT (mask n)) AND mask n = 0"
  for w :: ‹'a::len word›
  by (simp add: mask_eq_0_eq_x)

lemma AND_NOT_mask_plus_AND_mask_eq:
  "(w AND NOT (mask n)) + (w AND mask n) = w"
  for w :: ‹'a::len word›
  by (simp add: add.commute word_plus_and_or_coroll2)

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"
  using m1 m2 split_word_eq_on_mask by blast

lemma neq_0_no_wrap:
  fixes x :: "'a :: len word"
  shows "⟦ x ≤ x + y; x ≠ 0 ⟧ ⟹ x + y ≠ 0"
  by clarsimp

lemma unatSuc2:
  fixes n :: "'a :: len word"
  shows "n + 1 ≠ 0 ⟹ unat (n + 1) = Suc (unat n)"
  by (simp add: add.commute unatSuc)

lemma word_of_nat_le:
  "n ≤ unat x ⟹ of_nat n ≤ x"
  by (simp add: le_unat_uoi word_le_nat_alt)

lemma word_unat_less_le:
  "a ≤ of_nat b ⟹ unat a ≤ b"
  by (metis eq_iff le_cases le_unat_uoi word_of_nat_le)

lemma mask_Suc_0 : "mask (Suc 0) = (1 :: 'a::len word)"
  by (simp add: mask_eq_decr_exp)

lemma bool_mask':
  fixes x :: "'a :: len word"
  shows "2 < LENGTH('a) ⟹ (0 < x AND 1) = (x AND 1 = 1)"
  by (simp add: and_one_eq mod_2_eq_odd)

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"
  by transfer (smt (verit, ccfv_threshold) min_def take_bit_add take_bit_take_bit)

lemma lt1_neq0:
  fixes x :: "'a :: len word"
  shows "(1 ≤ x) = (x ≠ 0)" by unat_arith

lemma word_plus_one_nonzero:
  fixes x :: "'a :: len word"
  shows "⟦x ≤ x + y; y ≠ 0⟧ ⟹ x + 1 ≠ 0"
  using max_word_wrap by fastforce

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

```
  by (metis diff_add_cancel word_plus_one_nonzero word_sub_le)

lemma word_le_minus_mono_right:
  fixes x :: "'a :: len word"
  shows "⟦ z ≤ y; y ≤ x; z ≤ x ⟧ ⟹ x - y ≤ x - z"
  using range_subset_card by auto

lemma word_0_sle_from_less:
  ‹0 ≤s x› if ‹x < 2 ^ (LENGTH('a) - 1)› for x :: ‹'a::len word›
  using that
  by (metis p2_gt_0 signed_0 sint_of_nat_ge_zero unat_eq_of_nat unat_less_power
unat_lt2p word_gt_a_gt_0 word_sle_eq)

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)"
  by (metis Word.of_nat_unat assms(1) of_nat_diff unat_sub word_less_eq_iff_unsigned)

lemma word_1_0:
  "⟦a + (1::('a::len) word) ≤ b; a < of_nat x⟧ ⟹ a < b"
  by (metis word_Suc_le word_order.extremum_strict)

lemma unat_of_nat_less: "⟦ a < b; unat b = c ⟧ ⟹ a < of_nat c"
  by fastforce

lemma word_le_plus_1: "⟦ (y::('a::len) word) < y + n; a < n ⟧ ⟹ y +
a ≤ y + a + 1"
  by unat_arith

lemma word_le_plus: "⟦(a::('a::len) word) < a + b; c < b⟧ ⟹ a ≤ a +
c"
  by (metis order_less_imp_le word_random)

lemma sint_minus1 [simp]: "(sint x = -1) = (x = -1)"
  by (metis signed_word_eqI sint_n1)

lemma sint_0 [simp]: "(sint x = 0) = (x = 0)"
  by (fact signed_eq_0_iff)


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 (cases ‹LENGTH('a) = 1›)
  case True
```

```
  then have ‹a = 0 ∨ a = 1›
    by transfer auto
  with True that show ?thesis
    by auto
next
  case False
  with that show ?thesis
    by (simp add: less_le Suc_le_eq)
qed

lemma sint_int_min:
  "sint (- (2 ^ (LENGTH('a) - Suc 0)) :: ('a::len) word) = - (2 ^ (LENGTH('a)
- Suc 0))"
proof (cases ‹LENGTH('a)›)
  case 0
  then show ?thesis
    by simp
next
  case Suc
  then show ?thesis
    by transfer (simp add: signed_take_bit_int_eq_self)
qed

lemma sint_int_max_plus_1:
  "sint (2 ^ (LENGTH('a) - Suc 0) :: ('a::len) word) = - (2 ^ (LENGTH('a)
- Suc 0))"
proof (cases ‹LENGTH('a)›)
  case 0
  then show ?thesis
    by simp
next
  case Suc
  then show ?thesis
    by transfer (simp add: signed_take_bit_eq_take_bit_shift take_bit_eq_mod
word_of_int_2p)
qed

lemma uint_range': ‹0 ≤ uint x ∧ uint x < 2 ^ LENGTH('a)› for x :: ‹'a::len
word›
  by simp

lemma sint_of_int_eq:
  "⟦ - (2 ^ (LENGTH('a) - 1)) ≤ x; x < 2 ^ (LENGTH('a) - 1) ⟧ ⟹ sint
(of_int x :: ('a::len) word) = x"
  by (simp add: signed_take_bit_int_eq_self signed_of_int)

lemma of_int_sint:
  "of_int (sint a) = a"
  by simp
```

```
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"
  by transfer (simp add: signed_take_bit_take_bit)

lemma word_less_nowrapI':
  "(x :: 'a :: len word) ≤ z - k ⟹ k ≤ z ⟹ 0 < k ⟹ x < x + k"
  by uint_arith

lemma mask_plus_1:
  "mask n + 1 = (2 ^ n :: 'a::len word)"
  by (clarsimp simp: mask_eq_decr_exp)

lemma unat_inj: "inj unat"
  by (metis eq_iff injI word_le_nat_alt)

lemma unat_ucast_upcast:
  "is_up (ucast :: 'b word ⇒ 'a word)
      ⟹ unat (ucast x :: ('a::len) word) = unat (x :: ('b::len) word)"
  by (metis Word.of_nat_unat of_nat_eq_iff uint_up_ucast)

lemma ucast_mono:
  "⟦ (x :: 'b :: len word) < y; y < 2 ^ LENGTH('a) ⟧
   ⟹ ucast x < ((ucast y) :: 'a :: len word)"
  by (metis Word.of_nat_unat of_nat_mono_maybe p2_gt_0 unat_less_power
unat_mono word_gt_a_gt_0)

lemma ucast_mono_le:
  "⟦x ≤ y; y < 2 ^ LENGTH('b)⟧ ⟹ (ucast (x :: 'a :: len word) :: 'b
:: len word) ≤ ucast y"
  by (metis order_class.order_eq_iff ucast_mono word_le_less_eq)

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›
  by (auto simp: word_less_nat_alt intro: ucast_mono_le)

lemma neg_mask_add_mask:
  "((x:: 'a :: len word) AND NOT (mask n)) + (2 ^ n - 1) = x OR mask n"
  by (simp add: mask_eq_decr_exp or_eq_and_not_plus)

lemma le_step_down_word: "⟦(i::('a::len) word) ≤ n; i = n ⟶ P; i ≤
n - 1 ⟶ P⟧ ⟹ P"
  by unat_arith

lemma le_step_down_word_2:
```

```
  fixes x :: "'a::len word"
  shows "⟦x ≤  y; x ≠ y⟧ ⟹ x ≤ y - 1"
  by (meson le_step_down_word)

lemma NOT_mask_AND_mask[simp]: "(w AND mask n) AND NOT (mask n) = 0"
  by (simp add: and.assoc)

lemma and_and_not[simp]: "(a AND b) AND NOT b = 0"
  for a b :: ‹'a::len word›
  using AND_twice mask_eq_x_eq_0 by blast

lemma ex_mask_1[simp]: "(∃x. mask x = (1 :: 'a::len word))"
  using mask_1 by blast

lemma not_switch: "NOT a = x ⟹ a = NOT x"
  by auto

lemma test_bit_eq_iff: "bit u = bit v ⟷ u = v"
  for u v :: "'a::len word"
  by (auto intro: bit_eqI simp add: fun_eq_iff)

lemma test_bit_size: "bit w n ⟹ n < size w"
  for w :: "'a::len word"
  by transfer simp

lemma word_eq_iff: "x = y ⟷ (∀n<LENGTH('a). bit x n = bit y n)"
  for x y :: "'a::len word"
  using bit_word_eqI by blast

lemma word_eqI: "(⋀n. n < size u ⟹ bit u n = bit v n) ⟹ u = v"
  for u :: "'a::len word"
  by (simp add: word_size word_eq_iff)

lemma word_eqD: "u = v ⟹ bit u x = bit v x"
  for u v :: "'a::len word"
  by simp

lemma test_bit_bin': "bit w n ⟷ n < size w ∧ bit (uint w) n"
  by transfer (simp add: bit_take_bit_iff)

lemmas test_bit_bin = test_bit_bin' [unfolded word_size]

lemma word_test_bit_def: ‹bit a = bit (uint a)›
  using bit_uint_iff test_bit_bin by blast

lemmas test_bit_def' = word_test_bit_def [THEN fun_cong]

lemma word_test_bit_transfer [transfer_rule]:
  "(rel_fun pcr_word (rel_fun (=) (=)))
```

```
    (λx n. n < LENGTH('a) ∧ bit x n) (bit :: 'a::len word ⇒ _)"
  by transfer_prover

lemma word_ops_nth_size:
  "n < size x ⟹
    bit (x OR y) n = (bit x n | bit y n) ∧
    bit (x AND y) n = (bit x n ∧ bit y n) ∧
    bit (x XOR y) n = (bit x n ≠ bit y n) ∧
    bit (NOT x) n = (¬ bit x n)"
  for x :: "'a::len word"
  by transfer (simp add: bit_or_iff bit_and_iff bit_xor_iff bit_not_iff)

lemma word_ao_nth:
  "bit (x OR y) n = (bit x n | bit y n) ∧
    bit (x AND y) n = (bit x n ∧ bit y n)"
  for x :: "'a::len word"
  using bit_and_iff bit_or_iff by blast

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

lemma nth_sint:
  fixes w :: "'a::len word"
  defines "l ≡ LENGTH('a)"
  shows "bit (sint w) n = (if n < l - 1 then bit w n else bit w (l - 1))"
  unfolding sint_uint l_def
  by (auto simp: bit_signed_take_bit_iff word_test_bit_def not_less min_def)

lemma test_bit_2p: "bit (word_of_int (2 ^ n)::'a::len word) m ⟷ m
= n ∧ m < LENGTH('a)"
  by transfer (auto simp: bit_exp_iff)

lemma nth_w2p: "bit ((2::'a::len word) ^ n) m ⟷ m = n ∧ m < LENGTH('a::len)"
  by transfer (auto simp: bit_exp_iff)

lemma bang_is_le: "bit x m ⟹ 2 ^ m ≤ x"
  for x :: "'a::len word"
  by (metis bit_take_bit_iff less_irrefl linorder_le_less_linear take_bit_word_eq_self_iff)

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

lemma nth_0: "¬ bit (0 :: 'a::len word) n"
  by simp

lemma nth_minus1: "bit (-1 :: 'a::len word) n ⟷ n < LENGTH('a)"
  by simp

lemma nth_ucast_weak:
```

```
  "bit (ucast w::'a::len word) n = (bit w n ∧ n < LENGTH('a))"
  using bit_ucast_iff by blast

lemma nth_ucast:
  "bit (ucast (w::'a::len word)::'b::len word) n =
   (bit w n ∧ n < min LENGTH('a) LENGTH('b))"
  by (metis bit_word_ucast_iff min_less_iff_conj nth_ucast_weak)

lemma nth_mask:
  ‹bit (mask n :: 'a::len word) i ⟷ i < n ∧ i < size (mask n :: 'a
word)›
  by (auto simp: word_size Word.bit_mask_iff)

lemma nth_slice: "bit (slice n w :: 'a::len word) m = (bit w (m + n)
∧ m < LENGTH('a))"
  using bit_imp_le_length
  by (fastforce simp: bit_simps less_diff_conv dest: bit_imp_le_length)

— keep quantifiers for use in simplification
lemma test_bit_split':
  "word_split c = (a, b) ⟶
    (∀n m.
      bit b n = (n < size b ∧ bit c n) ∧
      bit a m = (m < size a ∧ bit c (m + size b)))"
  by (auto simp: word_split_bin' bit_unsigned_iff word_size bit_drop_bit_eq
ac_simps
           dest: bit_imp_le_length)

lemma test_bit_split:
  "word_split c = (a, b) ⟹
    (∀n::nat. bit b n ⟷ n < size b ∧ bit c n) ∧
    (∀m::nat. bit a m ⟷ m < size a ∧ bit c (m + size b))"
  by (simp add: test_bit_split')

lemma test_bit_rcat:
  "sw = size (hd wl) ⟹ rc = word_rcat wl ⟹ bit rc n =
    (n < size rc ∧ n div sw < size wl ∧ bit ((rev wl) ! (n div sw)) (n
mod sw))"
  for wl :: "'a::len word list"
  by (simp add: word_size word_rcat_def rev_map bit_horner_sum_uint_exp_iff
bit_simps not_le)

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

lemma max_test_bit: "bit (- 1::'a::len word) n ⟷ n < LENGTH('a)"
  by (fact nth_minus1)

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

```
  by (simp flip: map_replicate_const)

lemma word_and_1:
  "n AND 1 = (if bit n 0 then 1 else 0)" for n :: "_ word"
  by (rule bit_word_eqI) (auto simp: bit_and_iff bit_1_iff intro: gr0I)

lemma nth_w2p_same:
  "bit (2^n :: 'a :: len word) n = (n < LENGTH('a))"
  by (simp add: nth_w2p)

lemma word_leI:
  fixes u :: "'a::len word"
  assumes "⋀n.  ⟦n < size u; bit u n ⟧ ⟹ bit (v::'a::len word) n"
  shows "u ≤ v"
proof (rule order_trans)
  show "u ≤ u AND v"
    by (metis assms bit_and_iff word_eqI word_le_less_eq)
qed (simp add: word_and_le1)

lemma bang_eq:
  fixes x :: "'a::len word"
  shows "(x = y) = (∀n. bit x n = bit y n)"
  by (auto intro!: bit_eqI)

lemma neg_mask_test_bit:
  "bit (NOT(mask n) :: 'a :: len word) m = (n ≤ m ∧ m < LENGTH('a))"
  by (auto simp: bit_simps)

lemma upper_bits_unset_is_l2p:
  ‹(∀n' ≥ n. n' < LENGTH('a) ⟶ ¬ bit p n') ⟷ (p < 2 ^ n)› (is ‹?P
⟷ ?Q›)
    if ‹n < LENGTH('a)›
    for p :: "'a :: len word"
proof
  assume ?Q
  then show ?P
    by (meson bang_is_le le_less_trans not_le word_power_increasing)
next
  assume P: ?P
  have ‹take_bit n p = p›
  proof (rule bit_word_eqI)
    fix q
    assume q: ‹q < LENGTH('a)›
    show ‹bit (take_bit n p) q ⟷ bit p q›
    proof (cases ‹q < n›)
      case True
      then show ?thesis
        by (auto simp: bit_simps)
    next
```

```
      case False
      then show ?thesis
        by (simp add: P q bit_take_bit_iff)
    qed
  qed
  with that show ?Q
    using take_bit_word_eq_self_iff [of n p] by auto
qed

lemma less_2p_is_upper_bits_unset:
  "p < 2 ^ n ⟷ n < LENGTH('a) ∧ (∀n' ≥ n. n' < LENGTH('a) ⟶ ¬ bit
p n')" for p :: "'a :: len word"
  by (meson le_less_trans le_mask_iff_lt_2n upper_bits_unset_is_l2p word_zero_le)

lemma test_bit_over:
  "n ≥ size (x::'a::len word) ⟹ (bit x n) = False"
  by transfer auto

lemma le_mask_high_bits:
  "w ≤ mask n ⟷ (∀i ∈ {n ..< size w}. ¬ bit w i)"
  for w :: ‹'a::len word›
  by (metis atLeastLessThan_iff bit_take_bit_iff less_eq_mask_iff_take_bit_eq_self
linorder_not_le nth_mask word_leI wsst_TYs(3))

lemma test_bit_conj_lt:
  "(bit x m ∧ m < LENGTH('a)) = bit x m" for x :: "'a :: len word"
  using test_bit_bin by blast

lemma neg_test_bit:
  "bit (NOT x) n = (¬ bit x n ∧ n < LENGTH('a))" for x :: "'a::len word"
  by (cases "n < LENGTH('a)") (auto simp: test_bit_over word_ops_nth_size
word_size)

lemma nth_bounded:
  "⟦bit (x :: 'a :: len word) n; x < 2 ^ m; m ≤ len_of TYPE ('a)⟧ ⟹
n < m"
  by (meson less_2p_is_upper_bits_unset linorder_not_le test_bit_conj_lt)

lemma and_neq_0_is_nth:
  ‹x AND y ≠ 0 ⟷ bit x n› if ‹y = 2 ^ n› for x y :: ‹'a::len word›
  by (simp add: and_exp_eq_0_iff_not_bit that)

lemma nth_is_and_neq_0:
  "bit (x::'a::len word) n = (x AND 2 ^ n ≠ 0)"
  by (subst and_neq_0_is_nth; rule refl)

lemma max_word_not_less [simp]:
   "¬ - 1 < x" for x :: ‹'a::len word›
  by (fact word_order.extremum_strict)
```

```
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"
  by (rule bit_eqI) (auto simp: bit_simps)

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"
  by (rule bit_eqI) (auto simp: bit_simps max_def)

lemma swap_with_xor:
  "⟦(x::'a::len word) = a XOR b; y = b XOR x; z = x XOR y⟧ ⟹ z = b ∧
y = a"
  by (auto intro: bit_word_eqI simp add: bit_simps)

lemma le_mask_imp_and_mask:
  "(x::'a::len word) ≤ mask n ⟹ x AND mask n = x"
  by (metis and_mask_eq_iff_le_mask)

lemma or_not_mask_nop:
  "((x::'a::len word) OR NOT (mask n)) AND mask n = x AND mask n"
  by (metis word_and_not word_ao_dist2 word_bw_comms(1) word_log_esimps(3))

lemma mask_subsume:
  "⟦n ≤ m⟧ ⟹ ((x::'a::len word) OR y AND mask n) AND NOT (mask m) =
x AND NOT (mask m)"
  by (rule bit_word_eqI) (auto simp: bit_simps word_size)

lemma and_mask_0_iff_le_mask:
  fixes w :: "'a::len word"
  shows "(w AND NOT(mask n) = 0) = (w ≤ mask n)"
  by (simp add: mask_eq_0_eq_x le_mask_imp_and_mask and_mask_eq_iff_le_mask)

lemma mask_twice2:
  "n ≤ m ⟹ ((x::'a::len word) AND mask m) AND mask n = x AND mask n"
  by (metis mask_twice min_def)

lemma uint_2_id:
  "LENGTH('a) ≥ 2 ⟹ uint (2::('a::len) word) = 2"
  by simp

lemma div_of_0_id[simp]: "(0::('a::len) word) div n = 0"
  by (simp add: word_div_def)

lemma degenerate_word: "LENGTH('a) = 1 ⟹ (x::('a::len) word) = 0 ∨
x = 1"
  by (metis One_nat_def less_irrefl_nat sint_1_cases)
```

```
lemma div_less_dividend_word:
  fixes x :: "('a::len) word"
  assumes "x ≠ 0" "n ≠ 1"
  shows "x div n < x"
proof (cases <n = 0>)
  case True
  then show ?thesis
    by (simp add: assms word_greater_zero_iff)
next
  case False
  then have "n > 1"
    by (metis assms(2) lt1_neq0 nless_le)
  then show ?thesis
    by (simp add: assms(1) unat_div unat_gt_0 word_less_nat_alt)
qed

lemma word_less_div:
  fixes x :: "('a::len) word"
    and y :: "('a::len) word"
  shows "x div y = 0 ⟹ y = 0 ∨ x < y"
  by (metis div_greater_zero_iff linorder_not_le unat_div unat_gt_0 word_less_eq_iff_unsign

lemma not_degenerate_imp_2_neq_0: "LENGTH('a) > 1 ⟹ (2::('a::len) word)
≠ 0"
  by (metis numerals(1) power_not_zero power_zero_numeral)

lemma word_overflow: "(x::('a::len) word) + 1 > x ∨ x + 1 = 0"
  using plus_one_helper2 by auto

lemma word_overflow_unat: "unat ((x::('a::len) word) + 1) = unat x +
1 ∨ x + 1 = 0"
  by (metis Suc_eq_plus1 add.commute unatSuc)

lemma even_word_imp_odd_next:
  "even (unat (x::('a::len) word)) ⟹ x + 1 = 0 ∨ odd (unat (x + 1))"
  by (metis even_plus_one_iff word_overflow_unat)

lemma odd_word_imp_even_next: "odd (unat (x::('a::len) word)) ⟹ x +
1 = 0 ∨ even (unat (x + 1))"
  using even_plus_one_iff word_overflow_unat by force

lemma overflow_imp_lsb: "(x::('a::len) word) + 1 = 0 ⟹ bit x 0"
  using even_plus_one_iff [of x] by (simp add: bit_0)

lemma odd_iff_lsb: "odd (unat (x::('a::len) word)) = bit x 0"
  by transfer (simp add: even_nat_iff bit_0)

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)"
  by (metis unat_of_nat)

lemma lsb_this_or_next: "¬ (bit ((x::('a::len) word) + 1) 0) ⟹ bit
x 0"
  by (simp add: bit_0)

lemma mask_or_not_mask:
  "x AND mask n OR x AND NOT (mask n) = x"
  for x :: ‹'a::len word›
  by (metis and.right_neutral bit.disj_cancel_right word_combine_masks)

lemma word_gr0_conv_Suc: "(m::'a::len word) > 0 ⟹ ∃n. m = n + 1"
  by (metis add.commute add_minus_cancel)

lemma revcast_down_us [OF refl]:
  "rc = revcast ⟹ source_size rc = target_size rc + n ⟹ rc w = ucast
(signed_drop_bit n w)"
  for w :: "'a::len word"
  by (simp add: source_size_def target_size_def bit_word_eqI bit_simps
ac_simps)

lemma revcast_down_ss [OF refl]:
  "rc = revcast ⟹ source_size rc = target_size rc + n ⟹ rc w = scast
(signed_drop_bit n w)"
  for w :: "'a::len word"
  by (simp add: source_size_def target_size_def bit_word_eqI bit_simps
ac_simps)

lemma revcast_down_uu [OF refl]:
  "rc = revcast ⟹ source_size rc = target_size rc + n ⟹ rc w = ucast
(drop_bit n w)"
  for w :: "'a::len word"
  by (simp add: source_size_def target_size_def bit_word_eqI bit_simps
ac_simps)

lemma revcast_down_su [OF refl]:
  "rc = revcast ⟹ source_size rc = target_size rc + n ⟹ rc w = scast
(drop_bit n w)"
  for w :: "'a::len word"
  by (simp add: source_size_def target_size_def bit_word_eqI bit_simps
ac_simps)

lemma cast_down_rev [OF refl]:
  "uc = ucast ⟹ source_size uc = target_size uc + n ⟹ uc w = revcast
(push_bit n w)"
  for w :: "'a::len word"
  by (simp add: source_size_def target_size_def bit_word_eqI bit_simps)
```

```
lemma revcast_up [OF refl]:
  "rc = revcast ⟹ source_size rc + n = target_size rc ⟹
    rc w = push_bit n (ucast w :: 'a::len word)"
  by (metis add_diff_cancel_left' le_add1 linorder_not_le revcast_def
slice1_def wsst_TYs)

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

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

lemma word_power_nonzero:
  "⟦ (x :: 'a::len word) < 2 ^ (LENGTH('a) - n); n < LENGTH('a); x ≠ 0
⟧
  ⟹ x * 2 ^ n ≠ 0"
  by (metis gr0I mult.commute not_less_eq p2_gt_0 power_0 word_less_1
word_power_less_diff zero_less_diff)

lemma less_1_helper:
  "n ≤ m ⟹ (n - 1 :: int) < m"
  by arith

lemma div_power_helper:
  "⟦ x ≤ y; y < LENGTH('a) ⟧ ⟹ (2 ^ y - 1) div (2 ^ x :: 'a::len word)
= 2 ^ (y - x) - 1"
  by (metis Suc_div_unat_helper add_diff_cancel_left' of_nat_Suc unat_div
word_arith_nat_div word_unat_power)

lemma max_word_mask:
  "(- 1 :: 'a::len word) = mask LENGTH('a)"
  by (fact minus_1_eq_mask)

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›
  by (rule bit_word_eqI) (auto simp: bit_simps word_size)

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›
  by (rule bit_word_eqI) (auto simp: bit_simps word_size)

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›
  by (simp add: and.assoc neg_mask_twice)

lemma ucast_and_neg_mask:
  "ucast (x AND NOT (mask n)) = ucast x AND NOT (mask n)"
proof (rule bit_word_eqI)
  fix m
  show "bit (ucast (x AND NOT (mask n))::'a word) m = bit ((ucast x::'a
word) AND NOT (mask n)) m"
    by (smt (verit) bit_and_iff bit_word_ucast_iff neg_mask_test_bit)
qed

lemma ucast_and_mask:
  "ucast (x AND mask n) = ucast x AND mask n"
  by (metis take_bit_eq_mask unsigned_take_bit_eq)

lemma ucast_mask_drop:
  "LENGTH('a :: len) ≤ n ⟹ (ucast (x AND mask n) :: 'a word) = ucast
x"
  by (metis mask_twice2 ucast_and_mask word_and_full_mask_simp)

lemma mask_exceed:
  "n ≥ LENGTH('a) ⟹ (x::'a::len word) AND NOT (mask n) = 0"
  by (rule bit_word_eqI) (simp add: bit_simps)

lemma word_add_no_overflow: "(x::'a::len word) < - 1 ⟹ x < x + 1"
  using less_x_plus_1 order_less_le by blast

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)"
  by (metis add.commute bound max_word_max word_Suc_leq word_not_le word_of_nat_less)

lemma unat_ucast_up_simp:
  fixes x :: "'a::len word"
  assumes "LENGTH('a) ≤ LENGTH('b)"
  shows "unat (ucast x :: 'b::len word) = unat x"
  by (simp add: assms is_up unat_ucast_upcast)

lemma unat_ucast_less_no_overflow:
  "⟦n < 2 ^ LENGTH('a); unat f < n⟧ ⟹ (f::('a::len) word) < of_nat n"
  by (simp add: unat_of_nat_len word_less_nat_alt)
```

```
lemma unat_ucast_less_no_overflow_simp:
  "n < 2 ^ LENGTH('a) ⟹ (unat f < n) = ((f::('a::len) word) < of_nat
n)"
  using unat_less_helper unat_ucast_less_no_overflow by blast

lemma unat_ucast_no_overflow_le:
  fixes f :: "'a::len word" and b :: "'b :: len word"
  assumes no_overflow: "unat b < (2 :: nat) ^ LENGTH('a)"
  and upward_cast: "LENGTH('a) < LENGTH('b)"
  shows "(ucast f < b) = (unat f < unat b)"
proof -
  have LR: "ucast f < b ⟹ unat f < unat b"
    by (simp add: less_or_eq_imp_le unat_ucast_up_simp upward_cast word_less_nat_alt)
  have RL: "unat f < unat b ⟹ ucast f < b"
  proof-
    assume ineq: "unat f < unat b"
    have "ucast f < ((ucast (ucast b ::'a::len word)) :: 'b :: len word)"
      by (metis ineq no_overflow ucast_nat_def unat_of_nat_len word_nchotomy
word_of_nat_less)
    then show ?thesis
      using ineq word_of_nat_less by fastforce
  qed
  then show ?thesis by (simp add:RL LR iffI)
qed

lemmas ucast_up_mono = ucast_less_ucast[THEN iffD2]

lemma minus_one_word:
  "(-1 :: 'a :: len word) = 2 ^ LENGTH('a) - 1"
  by simp

lemma le_2p_upper_bits:
  "⟦ (p::'a::len word) ≤ 2^n - 1; n < LENGTH('a) ⟧ ⟹
  ∀n'≥n. n' < LENGTH('a) ⟶ ¬ bit p n'"
  by (subst upper_bits_unset_is_l2p; simp)

lemma le2p_bits_unset:
  "p ≤ 2 ^ n - 1 ⟹ ∀n'≥n. n' < LENGTH('a) ⟶ ¬ bit (p::'a::len word)
n'"
  using upper_bits_unset_is_l2p [where p=p]
  by (cases "n < LENGTH('a)") auto

lemma complement_nth_w2p:
  shows "n' < LENGTH('a) ⟹ bit (NOT (2 ^ n :: 'a::len word)) n' = (n'
≠ n)"
  by (fastforce simp: word_ops_nth_size word_size nth_w2p)

lemma word_unat_and_lt:
```

```
  "unat x < n ∨ unat y < n ⟹ unat (x AND y) < n"
  by (meson le_less_trans word_and_le1 word_and_le2 word_le_nat_alt)

lemma word_unat_mask_lt:
  "m ≤ size w ⟹ unat ((w::'a::len word) AND mask m) < 2 ^ m"
  by (rule word_unat_and_lt) (simp add: unat_mask word_size)

lemma word_sless_sint_le: "x <s y ⟹ sint x ≤ sint y - 1"
  by (metis word_sless_alt zle_diff1_eq)

lemma upper_trivial:
  fixes x :: "'a::len word"
  shows "x ≠ 2 ^ LENGTH('a) - 1 ⟹ x < 2 ^ LENGTH('a) - 1"
  by (simp add: less_le)

lemma constraint_expand:
  fixes x :: "'a::len word"
  shows "x ∈ {y. lower ≤ y ∧ y ≤ upper} = (lower ≤ x ∧ x ≤ upper)"
  by (rule mem_Collect_eq)

lemma card_map_elide:
  "card ((of_nat :: nat ⇒ 'a::len word) ` {0..<n}) = card {0..<n}"
  if "n ≤ CARD('a::len word)"
proof -
  let ?of_nat = "of_nat :: nat ⇒ 'a word"
  have "inj_on ?of_nat {i. i < CARD('a word)}"
    by (rule inj_onI) (simp add: card_word of_nat_inj)
  moreover have "{0..<n} ⊆ {i. i < CARD('a word)}"
    using that by auto
  ultimately have "inj_on ?of_nat {0..<n}"
    by (rule inj_on_subset)
  then show ?thesis
    by (simp add: card_image)
qed

lemma card_map_elide2:
  "n ≤ CARD('a::len word) ⟹ card ((of_nat::nat ⇒ 'a::len word) ` {0..<n})
= n"
  by (subst card_map_elide) clarsimp+

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"
  by transfer simp

lemma le_ucast_ucast_le:
  "x ≤ ucast y ⟹ ucast x ≤ y"
  for x :: "'a::len word" and y :: "'b::len word"
  by (smt (verit) le_unat_uoi linorder_not_less order_less_imp_le ucast_nat_def
```

```
unat_arith_simps(1))

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"
  by (metis ucast_nat_def unat_mono unat_ucast_up_simp word_of_nat_less)

lemma ucast_le_ucast:
  "LENGTH('a) ≤ LENGTH('b) ⟹ (ucast x ≤ (ucast y::'b::len word)) =
(x ≤ y)"
  for x :: "'a::len word"
  by (simp add: unat_arith_simps(1) unat_ucast_up_simp)

lemmas ucast_up_mono_le = ucast_le_ucast[THEN iffD2]

lemma ucast_or_distrib:
  fixes x :: "'a::len word"
  fixes y :: "'a::len word"
  shows "(ucast (x OR y) :: ('b::len) word) = ucast x OR ucast y"
  by (fact unsigned_or_eq)

lemma word_exists_nth:
  "(w::'a::len word) ≠ 0 ⟹ ∃i. bit w i"
  by (auto simp: bit_eq_iff)

lemma max_word_not_0 [simp]:
  "- 1 ≠ (0 :: 'a::len word)"
  by simp

lemma unat_max_word_pos[simp]: "0 < unat (- 1 :: 'a::len word)"
  using unat_gt_0 [of ‹- 1 :: 'a::len word›] by simp




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 (rule bit_word_eqI)
  fix q
  assume ‹q < LENGTH('a)›
  from eq have ‹push_bit n x = push_bit n y›
    by (simp add: push_bit_eq_mult)
  moreover from le have §: ‹take_bit m x = x› ‹take_bit m y = y›
    by (simp_all add: less_eq_mask_iff_take_bit_eq_self)
  ultimately have ‹push_bit n (take_bit m x) = push_bit n (take_bit m
y)›
```

```
    by simp_all
  with § ‹q < LENGTH('a)› ws show ‹bit x q ⟷ bit y q›
    apply (simp add: push_bit_take_bit bit_eq_iff bit_simps not_le)
    by (metis (full_types) add.commute add_diff_cancel_right' add_less_cancel_right
le_add2 less_le_trans)
qed

lemma word_of_nat_inj:
  assumes "x < 2 ^ LENGTH('a)" "y < 2 ^ LENGTH('a)"
  assumes "of_nat x = (of_nat y :: 'a::len word)"
  shows "x = y"
  using assms of_nat_inj by blast

lemma word_of_int_bin_cat_eq_iff:
  "(word_of_int (concat_bit LENGTH('b) (uint b) (uint a))::'c::len word)
=
    word_of_int (concat_bit LENGTH('b) (uint d) (uint c)) ⟷ b = d ∧
a = c" (is "?L=?R")
  if "LENGTH('a) + LENGTH('b) ≤ LENGTH('c)"
  for a:: "'a::len word" and b:: "'b::len word"
proof
  assume ?L
  then have "take_bit LENGTH('c) (concat_bit LENGTH('b) (uint b) (uint
a)) =
             take_bit LENGTH('c) (concat_bit LENGTH('b) (uint d) (uint
c))"
    by (simp add: word_of_int_eq_iff)
  then show "b = d ∧ a = c"
    using that concat_bit_eq_iff
    by (metis (no_types, lifting) concat_bit_assoc concat_bit_of_zero_2
concat_bit_take_bit_eq
        take_bit_tightened uint_sint unsigned_word_eqI)
qed auto

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"
  using word_of_int_bin_cat_eq_iff [OF that, of b a d c]
  by (simp add: word_cat_eq' ac_simps)

lemma p2_eq_1: "2 ^ n = (1::'a::len word) ⟷ n = 0"
proof -
  have "2 ^ n = (1::'a word) ⟹ n = 0"
    by (metis One_nat_def not_less one_less_numeral_iff p2_eq_0 p2_gt_0
power_0 power_0
        power_inject_exp semiring_norm(76) unat_power_lower zero_neq_one)
  then show ?thesis by auto
qed
```

```
end

lemmas word_div_less = div_word_less

lemma word_mod_by_0: "k mod (0::'a::len word) = k"
  by (simp add: word_arith_nat_mod)

— the binary operations only
lemmas word_log_binary_defs =
  word_and_def word_or_def word_xor_def

— limited hom result
lemma word_cat_hom:
  "LENGTH('a::len) ≤ LENGTH('b::len) + LENGTH('c::len) ⟹
    (word_cat (word_of_int w :: 'b word) (b :: 'c word) :: 'a word) =
    word_of_int ((λk n l. concat_bit n l k) w (size b) (uint b))"
  by (rule bit_eqI) (auto simp: bit_simps size_word_def)

lemma uint_shiftl:
  "uint (push_bit i n) = take_bit (size n) (push_bit i (uint n))"
  by (simp add: unsigned_push_bit_eq word_size)

lemma sint_range':
  ‹- (2 ^ (LENGTH('a) - Suc 0)) ≤ sint x ∧ sint x < 2 ^ (LENGTH('a) -
Suc 0)›
  for x :: ‹'a::len word›
  by transfer simp_all

lemma signed_arith_eq_checks_to_ord:
  ‹sint a + sint b = sint (a + b) ⟷ (a ≤s a + b ⟷ 0 ≤s b)› (is ?P)
  ‹sint a - sint b = sint (a - b) ⟷ (0 ≤s a - b ⟷ b ≤s a)› (is ?Q)
  ‹- sint a = sint (- a) ⟷ (0 ≤s - a ⟷ a ≤s 0)› (is ?R)
proof -
  define n where ‹n = LENGTH('a) - Suc 0›
  then have [simp]: ‹LENGTH('a) = Suc n›
    by simp
  define k l where ‹k = sint a› ‹l = sint b›
  then have [simp]: ‹sint a = k› ‹sint b = l›
    by simp_all
  have self_eq_iff: ‹k + l = signed_take_bit n (k + l) ⟷ - (2 ^ n)
≤ k + l ∧ k + l < 2 ^ n›
    using signed_take_bit_int_eq_self_iff [of n ‹k + l›]
    by auto
  from sint_range' [where x=a] sint_range' [where x=b]
  have assms: ‹- (2 ^ n) ≤ k› ‹k < 2 ^ n› ‹- (2 ^ n) ≤ l› ‹l < 2 ^
n›
    by simp_all
  have aux: ‹signed_take_bit n x
```

```
          = (if x < - (2 ^ n) then x + 2 * (2 ^ n)
             else if x ≥ 2 ^ n then x - 2 * (2 ^ n) else x)›
    if x: ‹- 3 * (2 ^ n) ≤ x ∧ x < 3 * (2 ^ n)› for x :: int
  proof -
    have "2 ^ n + (x + 2 ^ n) mod (2 * 2 ^ n) = x"
      if "x < 3 * 2 ^ n" "2 ^ n ≤ x"
      using that by (smt (verit) minus_mod_self2 mod_pos_pos_trivial)
    moreover have "(x + 2 ^ n) mod (2 * 2 ^ n) = 3 * 2 ^ n + x"
      if "- (3 * 2 ^ n) ≤ x" and "x < - (2 ^ n)"
      using that by (smt (verit) mod_add_self2 mod_pos_pos_trivial)
    ultimately show ?thesis
      using x by (auto simp: signed_take_bit_eq_take_bit_shift take_bit_eq_mod)
  qed
  show ?P ?Q ?R
    using assms
    by (auto simp: sint_word_ariths word_sle_eq word_sless_alt aux)
qed

lemma signed_mult_eq_checks_double_size:
  assumes mult_le: "(2 ^ (LENGTH('a) - 1) + 1) ^ 2 ≤ (2 :: int) ^ (LENGTH('b)
- 1)"
          and le: "2 ^ (LENGTH('a) - 1) ≤ (2 :: int) ^ (LENGTH('b) -
1)"
  shows "(sint (a :: 'a :: len word) * sint b = sint (a * b))
      = (scast a * scast b = (scast (a * b) :: 'b :: len word))"
proof -
  have P: "signed_take_bit (size a - 1) (sint a * sint b) ∈ range (signed_take_bit
(size a - 1))"
    by simp

  have abs: "!! x :: 'a word. abs (sint x) < 2 ^ (size a - 1) + 1"
    by (smt (verit) sint_ge sint_lt wsst_TYs(3))
  have abs_ab: "abs (sint a * sint b) < 2 ^ (LENGTH('b) - 1)"
    using abs_mult_less[OF abs[where x=a] abs[where x=b]] mult_le
    by (simp add: abs_mult power2_eq_square word_size)
  define r s where ‹r = LENGTH('a) - 1› ‹s = LENGTH('b) - 1›
  then have ‹LENGTH('a) = Suc r› ‹LENGTH('b) = Suc s›
    ‹size a = Suc r› ‹size b = Suc r›
    by (simp_all add: word_size)
  with abs_ab le show ?thesis
    by (smt (verit) One_nat_def Word.of_int_sint of_int_mult sint_greater_eq
sint_less sint_of_int_eq)
qed

context
  includes bit_operations_syntax
begin

lemma wils1:
```

```
  ‹word_of_int (NOT (uint (word_of_int x :: 'a word))) = (word_of_int
(NOT x) :: 'a::len word)›
  ‹word_of_int (uint (word_of_int x :: 'a word) XOR uint (word_of_int
y :: 'a word)) = (word_of_int (x XOR y) :: 'a::len word)›
  ‹word_of_int (uint (word_of_int x :: 'a word) AND uint (word_of_int
y :: 'a word)) = (word_of_int (x AND y) :: 'a::len word)›
  ‹word_of_int (uint (word_of_int x :: 'a word) OR uint (word_of_int y
:: 'a word)) = (word_of_int (x OR y) :: 'a::len word)›
  by (simp_all add: word_of_int_eq_iff uint_word_of_int_eq take_bit_not_take_bit)

end

end
```

# 4 Shift operations with infix syntax

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

context semiring_bit_operations
begin

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

lemma bit_shiftl_iff [bit_simps]:
  ‹bit (a << m) n ⟷ m ≤ n ∧ possible_bit TYPE('a) n ∧ bit a (n - m)›
  by (simp add: shiftl_def bit_simps)

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

lemma bit_shiftr_eq [bit_simps]:
  ‹bit (a >> n) = bit a ∘ (+) n›
  by (simp add: shiftr_def bit_simps)

end

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

lemma bit_sshiftr_iff [bit_simps]:
  ‹bit (w >>> m) n ⟷ bit w (if LENGTH('a) - m ≤ n ∧ n < LENGTH('a)
then LENGTH('a) - 1 else m + n)›
  for w :: ‹'a::len word›
  by (simp add: sshiftr_def bit_simps)

context
```

```
  includes lifting_syntax
begin

lemma shiftl_word_transfer [transfer_rule]:
  ‹(pcr_word ===> (=) ===> pcr_word) (λk n. push_bit n k) (<<)›
  apply (unfold shiftl_def)
  apply transfer_prover
  done

lemma shiftr_word_transfer [transfer_rule]:
  ‹((pcr_word :: int ⇒ 'a::len word ⇒ _) ===> (=) ===> pcr_word)
    (λk n. drop_bit n (take_bit LENGTH('a) k))
    (>>)›
proof -
  have ‹((pcr_word :: int ⇒ 'a::len word ⇒ _) ===> (=) ===> pcr_word)
    (λk n. (drop_bit n ∘ take_bit LENGTH('a)) k)
    (>>)›
    by (unfold shiftr_def) transfer_prover
  then show ?thesis
    by simp
qed

lemma sshiftr_transfer [transfer_rule]:
  ‹((pcr_word :: int ⇒ 'a::len word ⇒ _) ===> (=) ===> pcr_word)
    (λk n. drop_bit n (signed_take_bit (LENGTH('a) - Suc 0) k))
    (>>>)›
proof -
  have ‹((pcr_word :: int ⇒ 'a::len word ⇒ _) ===> (=) ===> pcr_word)
    (λk n. (drop_bit n ∘ signed_take_bit (LENGTH('a) - Suc 0)) k)
    (>>>)›
    by (unfold sshiftr_def) transfer_prover
  then show ?thesis
    by simp
qed

end

context semiring_bit_operations
begin

lemma shiftl_0 [simp]:
  ‹0 << n = 0›
  by (simp add: shiftl_def)

lemma shiftl_of_0 [simp]:
  ‹a << 0 = a›
  by (simp add: shiftl_def)

lemma shiftl_of_Suc [simp]:
```

```
  ‹a << Suc n = (a * 2) << n›
  by (simp add: shiftl_def)

lemma shiftl_1 [simp]:
  ‹1 << n = 2 ^ n›
  by (simp add: shiftl_def)

lemma shiftl_numeral_Suc [simp]:
  ‹numeral m << Suc n = push_bit (Suc n) (numeral m)›
  by (fact shiftl_def)

lemma shiftl_numeral_numeral [simp]:
  ‹numeral m << numeral n = push_bit (numeral n) (numeral m)›
  by (fact shiftl_def)

lemma shiftr_0 [simp]:
  ‹0 >> n = 0›
  by (simp add: shiftr_def)

lemma shiftr_of_0 [simp]:
  ‹a >> 0 = a›
  by (simp add: shiftr_def)

lemma shiftr_1 [simp]:
  ‹1 >> n = of_bool (n = 0)›
  by (simp add: shiftr_def)

lemma shiftr_numeral_Suc [simp]:
  ‹numeral m >> Suc n = drop_bit (Suc n) (numeral m)›
  by (fact shiftr_def)

lemma shiftr_numeral_numeral [simp]:
  ‹numeral m >> numeral n = drop_bit (numeral n) (numeral m)›
  by (fact shiftr_def)

lemma shiftl_eq_mult:
  ‹x << n = x * 2 ^ n›
  unfolding shiftl_def by (fact push_bit_eq_mult)

lemma shiftr_eq_div:
  ‹x >> n = x div 2 ^ n›
  unfolding shiftr_def by (fact drop_bit_eq_div)

end

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

lemma int_shiftl_BIT: fixes x :: int
```

```
  shows int_shiftl0: "x << 0 = x"
  and int_shiftl_Suc: "x << Suc n = 2 * x << n"
  by (auto simp add: shiftl_int_def)

context ring_bit_operations
begin

context
  includes bit_operations_syntax
begin

lemma shiftl_minus_1_numeral [simp]:
  ‹- 1 << numeral n = NOT (mask (numeral n))›
  by (simp add: shiftl_def)

end

end

lemma shiftl_Suc_0 [simp]:
  ‹Suc 0 << n = 2 ^ n›
  by (simp add: shiftl_def)

lemma shiftr_Suc_0 [simp]:
  ‹Suc 0 >> n = of_bool (n = 0)›
  by (simp add: shiftr_def)

lemma sshiftr_numeral_Suc [simp]:
  ‹numeral m >>> Suc n = signed_drop_bit (Suc n) (numeral m)›
  by (fact sshiftr_def)

lemma sshiftr_numeral_numeral [simp]:
  ‹numeral m >>> numeral n = signed_drop_bit (numeral n) (numeral m)›
  by (fact sshiftr_def)

context ring_bit_operations
begin

lemma shiftl_minus_numeral_Suc [simp]:
  ‹- numeral m << Suc n = push_bit (Suc n) (- numeral m)›
  by (fact shiftl_def)

lemma shiftl_minus_numeral_numeral [simp]:
  ‹- numeral m << numeral n = push_bit (numeral n) (- numeral m)›
  by (fact shiftl_def)

lemma shiftr_minus_numeral_Suc [simp]:
  ‹- numeral m >> Suc n = drop_bit (Suc n) (- numeral m)›
  by (fact shiftr_def)
```

```
lemma shiftr_minus_numeral_numeral [simp]:
  ‹- numeral m >> numeral n = drop_bit (numeral n) (- numeral m)›
  by (fact shiftr_def)

end

lemma sshiftr_0 [simp]:
  ‹0 >>> n = 0›
  by (simp add: sshiftr_def)

lemma sshiftr_of_0 [simp]:
  ‹w >>> 0 = w›
  by (simp add: sshiftr_def)

lemma sshiftr_1 [simp]:
  ‹(1 :: 'a::len word) >>> n = of_bool (LENGTH('a) = 1 ∨ n = 0)›
  by (simp add: sshiftr_def)

lemma sshiftr_minus_numeral_Suc [simp]:
  ‹- numeral m >>> Suc n = signed_drop_bit (Suc n) (- numeral m)›
  by (fact sshiftr_def)

lemma sshiftr_minus_numeral_numeral [simp]:
  ‹- numeral m >>> numeral n = signed_drop_bit (numeral n) (- numeral
m)›
  by (fact sshiftr_def)

end
```

# 5 Word Alignment

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

begin

context
  includes bit_operations_syntax
begin

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

lemma is_aligned_iff_udvd:
```

```
  ‹is_aligned w n ⟷ 2 ^ n udvd w›
  by transfer (simp flip: take_bit_eq_0_iff add: min_def)

lemma is_aligned_iff_take_bit_eq_0:
  ‹is_aligned w n ⟷ take_bit n w = 0›
  by (simp add: is_aligned_iff_udvd take_bit_eq_0_iff exp_dvd_iff_exp_udvd)

lemma is_aligned_iff_dvd_int:
  ‹is_aligned ptr n ⟷ 2 ^ n dvd uint ptr›
  by transfer simp

lemma is_aligned_iff_dvd_nat:
  ‹is_aligned ptr n ⟷ 2 ^ n dvd unat ptr›
proof -
  have ‹unat ptr = nat |uint ptr|›
    by transfer simp
  then have ‹2 ^ n dvd unat ptr ⟷ 2 ^ n dvd uint ptr›
    by (simp only: dvd_nat_abs_iff) simp
  then show ?thesis
    by (simp add: is_aligned_iff_dvd_int)
qed

lemma is_aligned_0 [simp]:
  ‹is_aligned 0 n›
  by transfer simp

lemma is_aligned_at_0 [simp]:
  ‹is_aligned w 0›
  by transfer simp

lemma is_aligned_beyond_length:
  ‹is_aligned w n ⟷ w = 0› if ‹LENGTH('a) ≤ n› for w :: ‹'a::len word›
  using that by (simp add: is_aligned_iff_take_bit_eq_0 take_bit_word_beyond_length_eq)

lemma is_alignedI [intro?]:
  ‹is_aligned x n› if ‹x = 2 ^ n * k› for x :: ‹'a::len word›
proof (unfold is_aligned_iff_udvd)
  from that show ‹2 ^ n udvd x›
    using dvd_triv_left exp_dvd_iff_exp_udvd by blast
qed

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 (cases ‹n < LENGTH('a)›)
  case False
  with assms have ‹w = 0›
```

```
    by (simp add: is_aligned_beyond_length)
  with that [of 0] show thesis
    by simp
next
  case True
  moreover define m where ‹m = LENGTH('a) - n›
  ultimately have l: ‹LENGTH('a) = n + m› and ‹m ≠ 0›
    by simp_all
  from ‹n < LENGTH('a)› have *: ‹unat (2 ^ n :: 'a word) = 2 ^ n›
    by transfer simp
  from assms have ‹2 ^ n udvd w›
    by (simp add: is_aligned_iff_udvd)
  then obtain v :: ‹'a word›
    where ‹unat w = unat (2 ^ n :: 'a word) * unat v› ..
  moreover define q where ‹q = unat v›
  ultimately have unat_w: ‹unat w = 2 ^ n * q›
    by (simp add: *)
  then have ‹word_of_nat (unat w) = (word_of_nat (2 ^ n * q) :: 'a word)›
    by simp
  then have w: ‹w = 2 ^ n * word_of_nat q›
    by simp
  moreover have ‹q < 2 ^ (LENGTH('a) - n)›
  proof (rule ccontr)
    assume ‹¬ q < 2 ^ (LENGTH('a) - n)›
    then have ‹2 ^ (LENGTH('a) - n) ≤ q›
      by simp
    then have ‹2 ^ LENGTH('a) ≤ 2 ^ n * q›
      by (simp add: l power_add)
    with unat_w [symmetric] show False
      by (metis le_antisym nat_less_le unsigned_less)
  qed
  ultimately show thesis
    using that by blast
qed

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 -
  from assms
  obtain q where ‹w = 2 ^ n * word_of_nat q› ‹q < 2 ^ (LENGTH('a) -
n)›
    by (rule is_alignedE)
  then have ‹w = push_bit n (word_of_nat q)›
    by (simp add: push_bit_eq_mult)
  with that show thesis
    using ‹q < 2 ^ (LENGTH('a) - n)› .
```

```
qed

lemma is_aligned_mask:
  ‹is_aligned w n ⟷ w AND mask n = 0›
  by (simp add: is_aligned_iff_take_bit_eq_0 take_bit_eq_mask)

lemma is_aligned_imp_not_bit:
  ‹¬ bit w m› if ‹is_aligned w n› and ‹m < n›
  for w :: ‹'a::len word›
proof -
  from ‹is_aligned w n›
  obtain q where ‹w = push_bit n (word_of_nat q)› ‹q < 2 ^ (LENGTH('a)
- n)› ..
  moreover have ‹¬ bit (push_bit n (word_of_nat q :: 'a word)) m›
    using ‹m < n› by (simp add: bit_simps)
  ultimately show ?thesis
    by simp
qed

lemma is_aligned_weaken:
  "⟦ is_aligned w x; x ≥ y ⟧ ⟹ is_aligned w y"
  unfolding is_aligned_iff_dvd_nat
  by (erule dvd_trans [rotated]) (simp add: le_imp_power_dvd)

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)"
  using aligned is_alignedE by blast

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 cases
  assume nlt: "n < LENGTH('a)"
  show ?thesis
    unfolding is_aligned_iff_dvd_nat dvd_def
  proof -
    from aligned2 obtain q2 where yv: "y = 2 ^ m * of_nat q2"
      and q2v: "q2 < 2 ^ (LENGTH('a) - m)"
      by (auto elim: is_alignedE)

    from lt obtain k where kv: "m + k = n" by (auto simp: le_iff_add)
    with aligned1 obtain q1 where xv: "x = 2 ^ (m + k) * of_nat q1"
      and q1v: "q1 < 2 ^ (LENGTH('a) - (m + k))"
```

```
  by (auto elim: is_alignedE)

have l1: "2 ^ (m + k) * q1 < 2 ^ LENGTH('a)"
  by (rule nat_less_power_trans [OF q1v])
     (subst kv, rule order_less_imp_le [OF nlt])

have l2: "2 ^ m * q2 < 2 ^ LENGTH('a)"
  by (rule nat_less_power_trans [OF q2v],
      rule order_less_imp_le [OF order_le_less_trans])
     fact+

have "x = of_nat (2 ^ (m + k) *  q1)" using xv
  by simp

moreover have "y = of_nat (2 ^ m * q2)" using yv
  by simp

ultimately have upls: "unat x + unat y = 2 ^ m * (2 ^ k * q1 + q2)"
proof -
  have f1: "unat x = 2 ^ (m + k) * q1"
    using q1v unat_mult_power_lem xv by blast
  have "unat y = 2 ^ m * q2"
    using q2v unat_mult_power_lem yv by blast
  then show ?thesis
    using f1 by (simp add: power_add semiring_normalization_rules(34))
qed


show "∃d. unat (x + y) = 2 ^ m * d"
proof (cases "unat x + unat y < 2 ^ LENGTH('a)")
  case True

  have "unat (x + y) = unat x + unat y"
    by (subst unat_plus_if', rule if_P) fact

  also have "... = 2 ^ m * (2 ^ k * q1 + q2)" by (rule upls)
  finally show ?thesis ..
next
  case False
  then have "unat (x + y) = (unat x + unat y) mod 2 ^ LENGTH('a)"
    by (subst unat_word_ariths(1)) simp

  also have "... = (2 ^ m * (2 ^ k * q1 + q2)) mod 2 ^ LENGTH('a)"
    by (subst upls, rule refl)

  also
  have "... = 2 ^ m * ((2 ^ k * q1 +  q2) mod 2 ^ (LENGTH('a) - m))"
  proof -
    have "m ≤ len_of (TYPE('a))"
```

```
        by (meson le_trans less_imp_le_nat lt nlt)
      then show ?thesis
        by (metis mult_mod_right ordered_cancel_comm_monoid_diff_class.add_diff_inverse
power_add)
    qed

    finally show ?thesis ..
  qed
 qed
next
 assume "¬ n < LENGTH('a)"
 with assms
 show ?thesis
  by (simp add: is_aligned_mask not_less take_bit_eq_mod power_overflow
word_arith_nat_defs(7) flip: take_bit_eq_mask)
qed

corollary aligned_sub_aligned:
 "⟦is_aligned (x::'a::len word) n; is_aligned y m; m ≤ n⟧
  ⟹ is_aligned (x - y) m"
 by (metis (no_types, lifting) diff_zero is_aligned_mask is_aligned_weaken
mask_eqs(4))

lemma is_aligned_shift:
 fixes k::"'a::len word"
 shows "is_aligned (k << m) m"
proof cases
 assume mv: "m < LENGTH('a)"
 from mv obtain q where mq: "m + q = LENGTH('a)" and "0 < q"
  by (auto dest: less_imp_add_positive)

 have "(2::nat) ^ m dvd unat (push_bit m k)"
 proof
  have kv: "(unat k div 2 ^ q) * 2 ^ q + unat k mod 2 ^ q = unat k"
   by (rule div_mult_mod_eq)

  have "unat (push_bit m k) = unat (2 ^ m * k)"
   by (simp add: push_bit_eq_mult ac_simps)
  also have "... = (2 ^ m * unat k) mod (2 ^ LENGTH('a))" using mv
   by (simp add: unat_word_ariths(2))
  also have "... = 2 ^ m * (unat k mod 2 ^ q)"
   by (subst mq [symmetric], subst power_add, subst mod_mult2_eq) simp
  finally show "unat (push_bit m k) = 2 ^ m * (unat k mod 2 ^ q)" .
 qed
 then show ?thesis by (unfold is_aligned_iff_dvd_nat shiftl_def)
next
 assume "¬ m < LENGTH('a)"
 then show ?thesis
  by (simp add: not_less power_overflow is_aligned_mask word_size shiftl_def)
```

```
qed

lemma aligned_mod_eq_0:
  fixes p::"'a::len word"
  assumes al: "is_aligned p sz"
  shows   "p mod 2 ^ sz = 0"
proof cases
  assume szv: "sz < LENGTH('a)"
  with al
  show ?thesis
    unfolding is_aligned_iff_dvd_nat
    by (simp add: and_mask_dvd_nat p2_gt_0 word_mod_2p_is_mask)
next
  assume "¬ sz < LENGTH('a)"
  with al show ?thesis
    by (simp add: is_aligned_mask flip: take_bit_eq_mask take_bit_eq_mod)
qed

lemma is_aligned_triv: "is_aligned (2 ^ n ::'a::len word) n"
  by (rule is_alignedI [where k = 1], simp)

lemma is_aligned_mult_triv1: "is_aligned (2 ^ n * x  ::'a::len word)
n"
  by (rule is_alignedI [OF refl])

lemma is_aligned_mult_triv2: "is_aligned (x * 2 ^ n ::'a::len word) n"
  by (subst mult.commute, simp add: is_aligned_mult_triv1)

lemma word_power_less_0_is_0:
  fixes x :: "'a::len word"
  shows "x < a ^ 0 ⟹ x = 0" by simp

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 -
  have szv: "sz < LENGTH('a)"
  using off p2_gt_0 word_neq_0_conv by fastforce

  from al obtain q where ptrq: "ptr = 2 ^ sz * of_nat q" and
    qv: "q < 2 ^ (LENGTH('a) - sz)" by (auto elim: is_alignedE)

  show ?thesis
  proof (cases "sz = 0")
    case True
    then show ?thesis using off ptrq qv
```

```
      by simp
  next
    case False
    then have sne: "0 < sz" ..
    show ?thesis
    proof -
      have uq: "unat (of_nat q ::'a::len word) = q"
        by (metis diff_less le_unat_uoi len_gt_0 order_less_imp_le qv
sne unat_power_lower)
      have uptr: "unat ptr = 2 ^ sz * q"
        by (simp add: ptrq qv unat_mult_power_lem)
      show "unat ptr + unat off < 2 ^ LENGTH('a)" using szv
        by (metis le_add_diff_inverse2 less_imp_le mult.commute nat_add_offset_less
off qv unat_less_power uptr)
    qed
  qed
qed

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"
  by (subst no_plus_overflow_unat_size, subst word_size, rule is_aligned_no_wrap)
fact+

lemma is_aligned_no_overflow':
  fixes p :: "'a::len word"
  assumes al: "is_aligned p n"
  shows "p ≤ p + (2 ^ n - 1)"
proof cases
  assume "n<LENGTH('a)"
  with al
  have "2^n - (1::'a::len word) < 2^n"
    by (simp add: word_less_nat_alt unat_sub_if_size)
  with al
  show ?thesis by (rule is_aligned_no_wrap')
next
  assume "¬ n<LENGTH('a)"
  with al
  show ?thesis
  by (simp add: not_less power_overflow is_aligned_mask mask_2pm1)
qed

lemma is_aligned_no_overflow:
  "is_aligned ptr sz ⟹ ptr ≤ ptr + 2^sz - 1"
  by (drule is_aligned_no_overflow') (simp add: field_simps)

lemma replicate_not_True:
```

```
  "⋀n. xs = replicate n False ⟹ True ∉ set xs"
  by (induct xs) auto

lemma map_zip_replicate_False_xor:
  "n = length xs ⟹ map (λ(x, y). x = (¬ y)) (zip xs (replicate n False))
= xs"
  by (induct xs arbitrary: n, auto)

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 (induct xs)
  case Nil then show ?case by simp
next
  case (Cons y ys)
  show ?case
  proof (cases "n = Suc (length ys)")
    case True
    then show ?thesis
      using Cons.prems(3) rev_nth by fastforce
  next
    case False
    with Cons show ?thesis
      apply (simp add: rev_nth nth_append)
      by (simp add: Cons_nth_drop_Suc Suc_diff_le)
  qed
qed

lemma drop_minus:
  "⟦ n < length xs; n' = length xs ⟧ ⟹ drop (n' - Suc n) xs = rev xs
! n  # drop (n' - n) xs"
  by (simp add: Suc_diff_Suc drop_minus_lem)

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 -
  have ‹¬ bit x n›
    using that by (rule is_aligned_imp_not_bit)
  then have ‹x + 2 ^ n = x OR 2 ^ n›
    by (subst disjunctive_add) (auto simp add: bit_simps disjunctive_add)
  moreover have ‹(x OR 2 ^ n) XOR 2 ^ n = x›
    by (rule bit_word_eqI) (auto simp add: bit_simps ‹¬ bit x n›)
  ultimately show ?thesis
    by simp
qed

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'"
  by (metis aligned_add_aligned is_alignedI mult.assoc mult.commute)

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'"
  by (simp add: aligned_add_aligned is_aligned_mult_triv2)

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)"
  by (metis is_alignedE le_add_diff_inverse2 less_imp_le mult.commute
nat_add_offset_less unat_mult_power_lem)



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

lemma aligned_small_is_0:
  "⟦ is_aligned x n; x < 2 ^ n ⟧ ⟹ x = 0"
  by (simp add: is_aligned_mask less_mask_eq)

corollary is_aligned_less_sz:
  "⟦is_aligned a sz; a ≠ 0⟧ ⟹ ¬ a < 2 ^ sz"
  by (rule notI, drule(1) aligned_small_is_0, erule(1) notE)

lemma aligned_at_least_t2n_diff:
  assumes x: "is_aligned x n"
      and y: "is_aligned y n"
      and "x < y"
    shows "x ≤ y - 2 ^ n"
  by (meson assms is_aligned_iff_udvd udvd_minus_le')

lemma is_aligned_no_overflow'':
  "⟦is_aligned x n; x + 2 ^ n ≠ 0⟧ ⟹ x ≤ x + 2 ^ n"
  using is_aligned_no_overflow order_trans word_sub_1_le by blast

lemma is_aligned_bitI:
  ‹is_aligned p m› if ‹⋀n. n < m ⟹ ¬ bit p n›
  by (simp add: bit_take_bit_iff bit_word_eqI is_aligned_iff_take_bit_eq_0
that)
```

```
lemma is_aligned_nth:
  "is_aligned p m = (∀n < m. ¬ bit p n)"
  using is_aligned_bitI is_aligned_imp_not_bit by blast

lemma range_inter:
  "({a..b} ∩ {c..d} = {}) = (∀x. ¬(a ≤ x ∧ x ≤ b ∧ c ≤ x ∧ x ≤ d))"
  by auto

lemma aligned_inter_non_empty:
  "⟦ {p..p + (2 ^ n - 1)} ∩ {p..p + 2 ^ m - 1} = {};
     is_aligned p n; is_aligned p m⟧ ⟹ False"
  by (simp add: is_aligned_no_overflow is_aligned_no_overflow')

lemma not_aligned_mod_nz:
  assumes al: "¬ is_aligned a n"
  shows "a mod 2 ^ n ≠ 0"
  by (meson assms is_aligned_iff_udvd mod_eq_0_imp_udvd)

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 (subst mn)
  from yv obtain qy where "y + qy = 2 ^ n"
    by (auto simp: le_iff_add)

  have "x * 2 ^ n + y ≤ x * 2 ^ n + 2 ^ n"
    using yv xv by simp
  also have "... = (x + 1) * 2 ^ n" by simp
  also have "... ≤ 2 ^ (m + n)" using xv
    by (subst power_add) (rule mult_le_mono1, simp)
  finally show "x * 2 ^ n + y ≤ 2 ^ (m + n)" .
qed

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 -
  from al obtain q where ptrq: "ptr = 2 ^ sz * of_nat q" and
    qv: "q < 2 ^ (LENGTH('a) - sz)" by (auto elim: is_alignedE)

  show ?thesis
  proof (cases "sz = 0")
```

```
    case True
    then show ?thesis using off ptrq qv
      by (auto simp add: le_Suc_eq Suc_le_eq) (simp add: le_less)
  next
    case False
    then have sne: "0 < sz" ..

    show ?thesis
    proof -
      have uq: "unat (of_nat q :: 'a word) = q"
        by (metis diff_less le_unat_uoi len_gt_0 order_le_less qv sne
unat_power_lower)
      have uptr: "unat ptr = 2 ^ sz * q"
        by (simp add: ptrq qv unat_mult_power_lem)
      show "unat ptr + off ≤ 2 ^ LENGTH('a)" using szv
        by (metis le_add_diff_inverse2 less_imp_le mult.commute nat_add_offset_le
off qv uptr)
    qed
  qed
qed

lemma is_aligned_neg_mask:
  "m ≤ n ⟹ is_aligned (x AND NOT (mask n)) m"
  by (rule is_aligned_bitI) (simp add: bit_simps)

lemma unat_minus:
  "unat (- (x :: 'a :: len word)) = (if x = 0 then 0 else 2 ^ size x -
unat x)"
  using unat_sub_if_size[where x="2 ^ size x" and y=x]
  by (simp add: unat_eq_0 word_size)

lemma is_aligned_minus:
  ‹is_aligned (- p) n› if ‹is_aligned p n› for p :: ‹'a::len word›
  by (metis is_alignedE is_alignedI mult_minus_right that)

lemma add_mask_lower_bits:
  fixes x :: "'a :: len word"
  assumes "is_aligned x n"
    and "∀n'≥n. n' < LENGTH('a) ⟶ ¬ bit p n'"
  shows "x + p AND NOT (mask n) = x"
proof -
  have "x AND p = 0"
    by (metis assms bit_and_iff bit_imp_le_length is_aligned_nth not_le_imp_less
word_exists_nth)
  moreover
  have "(x OR p) AND NOT (mask n) = x"
  proof (rule bit_word_eqI)
    fix k
    assume "k < LENGTH('a)"
```

```
    show "bit ((x OR p) AND NOT (mask n)) k = bit x k"
      by (metis assms is_aligned_mask mask_eq_0_eq_x neg_mask_test_bit
word_ao_nth)
  qed
  ultimately show ?thesis
    by (simp add: word_plus_and_or_coroll)
qed

lemma is_aligned_andI1:
  "is_aligned x n ⟹ is_aligned (x AND y) n"
  by (simp add: is_aligned_nth bit_simps)

lemma is_aligned_andI2:
  "is_aligned y n ⟹ is_aligned (x AND y) n"
  by (simp add: is_aligned_nth bit_simps)

lemma is_aligned_shiftl:
  "is_aligned w (n - m) ⟹ is_aligned (w << m) n"
  by (simp add: is_aligned_nth bit_simps)

lemma is_aligned_shiftr:
  "is_aligned w (n + m) ⟹ is_aligned (w >> m) n"
  by (simp add: is_aligned_nth bit_simps)

lemma is_aligned_shiftl_self:
  "is_aligned (p << n) n"
  by (rule is_aligned_shift)

lemma is_aligned_neg_mask_eq:
  "is_aligned p n ⟹ p AND NOT (mask n) = p"
  by (simp add: is_aligned_mask mask_eq_x_eq_0)

lemma is_aligned_shiftr_shiftl:
  "is_aligned w n ⟹ w >> n << n = w"
  apply (rule bit_word_eqI)
  by (metis bit_shiftl_iff bit_shiftr_eq is_aligned_nth leI le_add_diff_inverse
o_apply possible_bit_word)

lemma aligned_shiftr_mask_shiftl:
  assumes "is_aligned x n"
  shows "((x >> n) AND mask v) << n = x AND mask (v + n)"
proof -
  have "bit x m ⟹ m ≥ n" for m
    using assms is_aligned_imp_not_bit leI by blast
  with assms show ?thesis
    apply (intro word_eqI)
    by (metis bit_and_iff is_aligned_neg_mask_eq is_aligned_shiftr_shiftl
push_bit_and push_bit_mask_eq shiftl_def)
qed
```

```
lemma mask_zero:
  "is_aligned x a ⟹ x AND mask a = 0"
  by (metis is_aligned_mask)

lemma is_aligned_neg_mask_eq_concrete:
  "⟦ is_aligned p n; msk AND NOT (mask n) = NOT (mask n) ⟧
   ⟹ p AND msk = p"
  by (metis word_bw_assocs(1) word_bw_comms(1) is_aligned_neg_mask_eq)

lemma is_aligned_and_not_zero:
  "⟦ is_aligned n k; n ≠ 0 ⟧ ⟹ 2 ^ k ≤ n"
  using is_aligned_less_sz leI by blast

lemma is_aligned_and_2_to_k:
  "(n AND 2 ^ k - 1) = 0 ⟹ is_aligned (n :: 'a :: len word) k"
  by (simp add: is_aligned_mask mask_eq_decr_exp)

lemma is_aligned_power2:
  "b ≤ a ⟹ is_aligned (2 ^ a) b"
  by (metis is_aligned_triv is_aligned_weaken)

lemma aligned_sub_aligned':
  "⟦ is_aligned (a :: 'a :: len word) n; is_aligned b n; n < LENGTH('a)
⟧
   ⟹ is_aligned (a - b) n"
  by (simp add: aligned_sub_aligned)

lemma is_aligned_neg_mask_weaken:
  "⟦ is_aligned p n; m ≤ n ⟧ ⟹ p AND NOT (mask m) = p"
   using is_aligned_neg_mask_eq is_aligned_weaken by blast

lemma is_aligned_neg_mask2 [simp]:
  "is_aligned (a AND NOT (mask n)) n"
  by (rule is_aligned_bitI) (simp add: bit_simps)

lemma is_aligned_0':
  "is_aligned 0 n"
  by (fact is_aligned_0)

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 cases
  assume szv: "sz < LENGTH('a)"
  from al obtain k where xv: "x = 2 ^ sz * (of_nat k)"
```

```
    and kl: "k < 2 ^ (LENGTH('a) - sz)"
    by (auto elim: is_alignedE)

  show ?thesis using szv
    using al is_aligned_no_wrap offv by blast
next
  assume "¬ sz < LENGTH('a)"
  with offv show ?thesis by (simp add: not_less power_overflow )
qed

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 cases
  assume szv: "sz < LENGTH('a)"

  have ux: "unat x + unat k < 2 ^ LENGTH('a)"
    by (rule aligned_add_offset_no_wrap) fact+

  show ?thesis using al szv
    by (metis add_0 assms(2) less_mask_eq mask_eqs(1) mask_zero word_gt_a_gt_0
word_mod_2p_is_mask)
next
  assume "¬ sz < LENGTH('a)"
  with al show ?thesis
    by (simp add: not_less power_overflow is_aligned_mask mask_eq_decr_exp
                  word_mod_by_0)
qed

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 cases
  assume szv: "sz < LENGTH('a)"
  show ?thesis
  proof (rule equals0I, clarsimp)
    fix z
    assume xb: "x ≤ z" and xt: "z ≤ x + (2 ^ sz - 1)"
      and yb: "y ≤ z" and yt: "z ≤ y + (2 ^ sz - 1)"

    have rl: "⋀(p::'a word) k w. ⟦uint p + uint k < 2 ^ LENGTH('a); w
= p + k; w ≤ p + (2 ^ sz - 1) ⟧
      ⟹ k < 2 ^ sz"
      by (smt (verit, best) no_olen_add szv uint_plus_simple_iff word_le_def
```

```
word_less_sub_le)
    from xb obtain kx where
      kx: "z = x + kx" and
      kxl: "uint x + uint kx < 2 ^ LENGTH('a)"
      by (clarsimp dest!: word_le_exists')

    from yb obtain ky where
      ky: "z = y + ky" and
      kyl: "uint y + uint ky < 2 ^ LENGTH('a)"
      by (clarsimp dest!: word_le_exists')

    have "x = y"
    proof -
      have "kx = z mod 2 ^ sz"
      proof (subst kx, rule sym, rule aligned_add_offset_mod)
        show "kx < 2 ^ sz" by (rule rl) fact+
      qed fact+

      also have "... = ky"
      proof (subst ky, rule aligned_add_offset_mod)
        show "ky < 2 ^ sz"
          using kyl ky yt by (rule rl)
      qed fact+

      finally have kxky: "kx = ky" .
      moreover have "x + kx = y + ky" by (simp add: kx [symmetric] ky
[symmetric])
      ultimately show ?thesis by simp
    qed
    then show False using neq by simp
  qed
next
  assume "¬ sz < LENGTH('a)"
  with neq alx aly
  have False by (simp add: is_aligned_mask mask_eq_decr_exp power_overflow)
  then show ?thesis ..
qed

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)"
  by (metis add_diff_cancel_left' add_mask_lower_bits less_2p_is_upper_bits_unset
subtract_mask(2))

lemmas mask_inner_mask = mask_eqs(1)

lemma mask_add_aligned:
  "is_aligned p n ⟹ (p + q) AND mask n = q AND mask n"
  by (metis add_0 mask_inner_mask mask_zero)
```

```
lemma mask_out_add_aligned:
  assumes al: "is_aligned p n"
  shows "p + (q AND NOT (mask n)) = (p + q) AND NOT (mask n)"
  using mask_add_aligned [OF al]
  by (simp add: mask_out_sub_mask)

lemma is_aligned_add_or:
  "⟦is_aligned p n; d < 2 ^ n⟧ ⟹ p + d = p OR d"
  by (metis and_zero_eq less_mask_eq mask_zero word_bw_assocs(1) word_bw_comms(1)
word_plus_and_or_coroll)

lemma not_greatest_aligned:
  "⟦ x < y; is_aligned x n; is_aligned y n ⟧ ⟹ x + 2 ^ n ≠ 0"
  by (metis NOT_mask add_diff_cancel_right' diff_0 is_aligned_neg_mask_eq
not_le word_and_le1)

lemma neg_mask_mono_le:
  "x ≤ y ⟹ x AND NOT(mask n) ≤ y AND NOT(mask n)" for x :: "'a :: len
word"
proof (rule ccontr, simp add: linorder_not_le, cases "n < LENGTH('a)")
  case False
  then show "y AND NOT(mask n) < x AND NOT(mask n) ⟹ False"
    by (simp add: mask_eq_decr_exp linorder_not_less power_overflow)
next
  case True
  assume a: "x ≤ y" and b: "y AND NOT(mask n) < x AND NOT(mask n)"
  have word_bits: "n < LENGTH('a)" by fact
  have "y ≤ (y AND NOT(mask n)) + (y AND mask n)"
    by (simp add: word_plus_and_or_coroll2 add.commute)
  also have "... ≤ (y AND NOT(mask n)) + 2 ^ n"
  proof (rule word_plus_mono_right)
    show "y AND mask n ≤ 2 ^ n"
      by (metis and_mask_less' b linorder_not_le mask_exceed order_less_imp_le)
  next
    show "y AND NOT (mask n) ≤ (y AND NOT (mask n)) + 2 ^ n"
      using b is_aligned_neg_mask2 is_aligned_no_overflow'' not_greatest_aligned
by blast
  qed
  also have "... ≤ x AND NOT(mask n)"
  proof -
    have "y AND NOT (mask n) ≤ (x AND NOT (mask n)) - 2 ^ n"
      by (simp add: aligned_at_least_t2n_diff b)
    moreover have "2 ^ n ≤ x AND NOT (mask n)"
      by (metis b is_aligned_mask is_aligned_neg_mask2 less_mask_eq linorder_not_le
word_and_le2)
    ultimately show ?thesis
      by (metis add.commute le_plus)
  qed
```

```
  also have "... ≤ x" by (rule word_and_le2)
  also have "x ≤ y" by fact
  finally
  show "False" using b by simp
qed

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›
  by (metis eq_iff neg_mask_mono_le word_and_le1 word_and_le2 word_bw_same(1))

lemma neg_mask_le_high_bits:
  ‹NOT (mask n) ≤ w ⟷ (∀i ∈ {n ..< size w}. bit w i)› (is ‹?P ⟷
?Q›)
  for w :: ‹'a::len word›
proof
  assume ?Q
  then have ‹w AND NOT (mask n) = NOT (mask n)›
    by (auto simp add: bit_simps word_size intro: bit_word_eqI)
  then show ?P
    by (simp add: and_neg_mask_eq_iff_not_mask_le)
next
  assume ?P
  then have *: ‹w AND NOT (mask n) = NOT (mask n)›
    by (simp add: and_neg_mask_eq_iff_not_mask_le)
  show ‹?Q›
  proof (rule ccontr)
    assume ‹¬ (∀i∈{n..<size w}. bit w i)›
    then obtain m where m: ‹¬ bit w m› ‹n ≤ m› ‹m < LENGTH('a)›
      by (auto simp add: word_size)
    from * have ‹bit (w AND NOT (mask n)) m ⟷ bit (NOT (mask n ::
'a word)) m›
      by auto
    with m show False by (auto simp add: bit_simps)
  qed
qed

lemma is_aligned_add_less_t2n:
  fixes p :: "'a::len word"
  assumes "is_aligned p n"
      and "d < 2 ^ n"
      and "n ≤ m"
      and "p < 2 ^ m"
    shows "p + d < 2^m"
proof (cases "m < LENGTH('a)")
  case True
  have "m < size (p + d)"
    by (simp add: True word_size)
  moreover
```

```
  have "p + d AND mask m = p + d"
    using assms
    by (smt (verit, ccfv_SIG) is_aligned_add_helper less_mask_eq mask_eq_x_eq_0
mask_lower_twice)
  ultimately show ?thesis
    using True assms by (metis and_mask_less')
next
  case False
  then show ?thesis
    using ‹p < 2 ^ m› less_2p_is_upper_bits_unset by blast
qed

lemma aligned_offset_non_zero:
  "⟦ is_aligned x n; y < 2 ^ n; x ≠ 0 ⟧ ⟹ x + y ≠ 0"
  by (simp add: is_aligned_no_wrap' neq_0_no_wrap)

lemma is_aligned_over_length:
  "⟦ is_aligned p n; LENGTH('a) ≤ n ⟧ ⟹ (p::'a::len word) = 0"
  by (simp add: is_aligned_mask mask_over_length)

lemma is_aligned_no_overflow_mask:
  "is_aligned x n ⟹ x ≤ x + mask n"
  by (simp add: mask_eq_decr_exp) (erule is_aligned_no_overflow')

lemma aligned_mask_step:
  fixes p' :: "'a::len word"
  assumes "n' ≤ n"
      and "p' ≤ p + mask n"
      and "is_aligned p n"
      and "is_aligned p' n'"
    shows "p' + mask n' ≤ p + mask n"
proof (cases "LENGTH('a) ≤ n")
  case True
  then show ?thesis
    using assms(3) is_aligned_over_length mask_over_length by fastforce
next
  case False
  obtain k k' where kk: "2 ^ n' * k' ≤ 2 ^ n * k + unat (mask n::'a word)"
    "unat p = 2 ^ n * k" "unat p' = 2 ^ n' * k'"
    using assms
    by (metis is_alignedE is_aligned_no_overflow_mask unat_mult_power_lem
unat_plus_simple word_le_nat_alt)
  then have "2 ^ n' * (k' + 1) ≤ 2 ^ n * (k + 1)"
    by (metis False Suc_2p_unat_mask assms(1) leI le_imp_less_Suc power_2_mult_step_le)
  then have "2 ^ n' * k' + unat (mask n'::'a word) ≤ 2 ^ n * k + unat
(mask n::'a word)"
    by (smt (verit, best) False Suc_2p_unat_mask assms(1) leI not_less_eq_eq
order_le_less_trans)
  with assms kk show ?thesis
```

```
    by (metis is_aligned_no_overflow_mask unat_plus_simple word_le_nat_alt)
qed

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)"
  using al is_aligned_no_overflow_mask no_olen_add_nat offv word_random
by blast

lemma aligned_less_plus_1:
  assumes "is_aligned x n" and "0 < n"
  shows "x < x + 1"
proof (rule plus_one_helper2)
  show "x + 1 ≠ 0"
    using assms is_aligned_nth overflow_imp_lsb by blast
qed auto

lemma aligned_add_offset_less:
  assumes x: "is_aligned x n"
      and y: "is_aligned y n"
      and "x < y"
      and z: "z < 2 ^ n"
    shows "x + z < y"
proof (cases "y = 0 ∨ z = 0")
  case True
  then show ?thesis
    using ‹x<y› by auto
next
  case False
  with y is_aligned_get_word_bits have §: "n < LENGTH('a)" "z ≠ 0"
    by auto
  then have "x ≤ y - 2 ^ n"
    by (simp add: aligned_at_least_t2n_diff assms(3) x y)
  with assms show ?thesis
    by (smt (verit, ccfv_SIG) False diff_add_cancel is_aligned_and_not_zero
is_aligned_no_wrap'
                                not_less olen_add_eqv word_sub_mono2)
qed

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"
  by (simp add: aligned_add_offset_less)

lemma is_aligned_add_step_le:
```

```
  "⟦ is_aligned (a::'a::len word) n; is_aligned b n; a < b; b ≤ a + mask
n ⟧ ⟹ False"
  by (metis gap_between_aligned is_aligned_get_word_bits leD linorder_neq_iff
mask_eq_decr_exp)

lemma aligned_add_mask_lessD:
  "⟦ x + mask n < y; is_aligned x n ⟧ ⟹ x < y" for y::"'a::len word"
  by (metis is_aligned_no_overflow' mask_2pm1 order_le_less_trans)

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"
  using aligned_add_mask_lessD is_aligned_add_step_le word_le_not_less
by blast

lemma is_aligned_diff:
  fixes m :: "'a::len word"
  assumes alm: "is_aligned m s1"
  and     aln: "is_aligned n s2"
  and    s2wb: "s2 < LENGTH('a)"
  and      nm: "m ∈ {n .. n + (2 ^ s2 - 1)}"
  and    s1s2: "s1 ≤ s2"
shows  "∃q. m - n = of_nat q * 2 ^ s1 ∧ q < 2 ^ (s2 - s1)"
proof (cases "s1=0")
  case True
  with nm have "unat(m-n) < 2 ^ s2"
    by simp (metis add.commute s2wb unat_less_power word_diff_ls'(4) word_less_sub_le)
  with True nm show ?thesis
    using unat_eq_of_nat by force
next
  case False
  have rl: "⋀m s. ⟦ m < 2 ^ (LENGTH('a) - s); s < LENGTH('a) ⟧ ⟹ unat
((2::'a word) ^ s * of_nat m) = 2 ^ s * m"
  proof -
    fix m :: nat and  s
    assume m: "m < 2 ^ (LENGTH('a) - s)" and s: "s < LENGTH('a)"
    then have "unat ((of_nat m) :: 'a word) = m"
      by (metis diff_le_self le_unat_uoi nat_less_le unat_of_nat_len unat_power_lower)
    then show "?thesis m s" using s m
      using unat_mult_power_lem by blast
  qed
  have s1wb: "s1 < LENGTH('a)" using s2wb s1s2 by simp
  from alm obtain mq where mmq: "m = 2 ^ s1 * of_nat mq" and mq: "mq
< 2 ^ (LENGTH('a) - s1)"
    by (auto elim: is_alignedE simp: field_simps)
  from aln obtain nq where nnq: "n = 2 ^ s2 * of_nat nq" and nq: "nq
< 2 ^ (LENGTH('a) - s2)"
    by (auto elim: is_alignedE simp: field_simps)
```

```
  from s1s2 obtain sq where sq: "s2 = s1 + sq" by (auto simp: le_iff_add)

  note us1 = rl [OF mq s1wb]
  note us2 = rl [OF nq s2wb]

  from nm have "n ≤ m" by clarsimp
  then have "(2::'a word) ^ s2 * of_nat nq ≤ 2 ^ s1 * of_nat mq" us-
ing nnq mmq by simp
  then have "2 ^ s2 * nq ≤ 2 ^ s1 * mq" using s1wb s2wb
    by (simp add: word_le_nat_alt us1 us2)
  then have nqmq: "2 ^ sq * nq ≤ mq" using sq by (simp add: power_add)

  have "m - n = 2 ^ s1 * of_nat mq - 2 ^ s2 * of_nat nq" using mmq nnq
by simp
  also have "... = 2 ^ s1 * of_nat mq - 2 ^ s1 * 2 ^ sq * of_nat nq" us-
ing sq by (simp add: power_add)
  also have "... = 2 ^ s1 * (of_nat mq - 2 ^ sq * of_nat nq)" by (simp
add: field_simps)
  also have "... = 2 ^ s1 * of_nat (mq - 2 ^ sq * nq)" using s1wb s2wb
us1 us2 nqmq
    by simp
  finally have mn: "m - n = of_nat (mq - 2 ^ sq * nq) * 2 ^ s1" by simp
  moreover
  from nm have "m - n ≤ 2 ^ s2 - 1"
    by - (rule word_diff_ls', (simp add: field_simps)+)
  then have §: "(2::'a word) ^ s1 * of_nat (mq - 2 ^ sq * nq) < 2 ^ s2"
    using mn s2wb by (simp add: field_simps)
  then have "of_nat (mq - 2 ^ sq * nq) < (2::'a word) ^ (s2 - s1)"
  proof (rule word_power_less_diff)
    have mm: "mq - 2 ^ sq * nq < 2 ^ (LENGTH('a) - s1)" using mq by simp
    moreover have "LENGTH('a) - s1 < LENGTH('a)"
      using False diff_less by blast
    ultimately show "of_nat (mq - 2 ^ sq * nq) < (2::'a word) ^ (LENGTH('a)
- s1)"
      using of_nat_power by blast
  qed
  then have "mq - 2 ^ sq * nq < 2 ^ (s2 - s1)" using mq s2wb
    by (smt (verit, best) § diff_le_self nat_power_less_diff order_le_less_trans
unat_less_power
                           unat_mult_power_lem)
  ultimately show ?thesis
    by auto
qed

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"
  using al2
```

```
proof (rule is_aligned_get_word_bits)
  assume "x = 0" then show ?thesis using al1 by simp
next
  assume nv: "n < LENGTH('a)"
  from al1 obtain q1
    where xy: "x + y = 2 ^ n * of_nat q1" and "q1 < 2 ^ (LENGTH('a) -
n)"
    by (rule is_alignedE)
  moreover from al2 obtain q2
    where x: "x = 2 ^ n * of_nat q2" and "q2 < 2 ^ (LENGTH('a) - n)"
    by (rule is_alignedE)
  ultimately have "y = 2 ^ n * (of_nat q1 - of_nat q2)"
    by (simp add: field_simps)
  then show ?thesis using nv by (simp add: is_aligned_mult_triv1)
qed

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

lemma is_aligned_add:
  "⟦is_aligned p n; is_aligned q n⟧ ⟹ is_aligned (p + q) n"
  by (simp add: is_aligned_mask mask_add_aligned)

lemma aligned_shift:
  fixes x :: "'a::len word"
  assumes x: "x < 2 ^ n"
      and "is_aligned y n"
      and "n ≤ LENGTH('a::len)"
    shows "(x + y) >> n = y >> n"
proof (subst word_plus_and_or_coroll)
  show "x AND y = 0"
    using assms
    by (meson le_less_trans is_aligned_andI2 is_aligned_less_sz word_and_le2)
next
  show "x OR y >> n = y >> n"
    unfolding shiftr_def
    by (metis x drop_bit_or less_mask_eq or.left_neutral take_bit_eq_mask
              take_bit_eq_self_iff_drop_bit_eq_0)
qed

lemma aligned_shift':
  "⟦x < 2 ^ n; is_aligned (y :: 'a :: len word) n;n ≤ LENGTH('a)⟧
   ⟹ (y + x) >> n = y >> n"
  by (simp add: add.commute aligned_shift)

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

```
  by (metis le_word_or2 or_eq_and_not_plus)

lemma word_neg_and_le:
  "ptr ≤ (ptr AND NOT (mask n)) + (2 ^ n - 1)"
  for ptr :: ‹'a::len word›
  by (simp add: and_neg_mask_plus_mask_mono mask_2pm1[symmetric])

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)"
  by (drule(1) is_aligned_add_helper, simp)

lemma is_aligned_after_mask:
  "⟦is_aligned k m;m≤ n⟧ ⟹ is_aligned (k AND mask n) m"
  by (rule is_aligned_andI1)

lemma and_mask_plus:
  assumes "is_aligned ptr m"
    and "m ≤ n"
    and "a < 2 ^ m"
  shows "ptr + a AND mask n = (ptr AND mask n) + a"
proof (rule mask_eqI)
  show "(ptr + a AND mask n) AND mask m = (ptr AND mask n) + a AND mask
m"
    by (metis assms(2) mask_inner_mask mask_twice2)
next
  show "(ptr + a AND mask n) AND NOT (mask m) = (ptr AND mask n) + a
AND NOT (mask m)"
    by (metis assms(1,3) is_aligned_add_helper is_aligned_neg_mask2 word_bw_assocs(1)
word_bw_comms(1))
qed

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"
  by (metis is_aligned_addD1)

lemma neg_mask_add_aligned:
  "⟦ is_aligned p n; q < 2 ^ n ⟧ ⟹ (p + q) AND NOT (mask n) = p AND NOT
(mask n)"
  by (metis is_aligned_add_helper is_aligned_neg_mask_eq)

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 (cases "m = 0")
  case True
  then show ?thesis using off ak by simp
next
  case False
  from ak have ak1: "a + 1 ≤ k" by (rule inc_le)
  then have "(a + 1) * 2 ^ m ≠ 0"
    by (meson ak kw less_is_non_zero_p1 mw order_le_less_trans word_power_nonzero)
  then have "(a * 2 ^ m) + off < ((a + 1) * 2 ^ m)" using kw mw
    by (simp add: distrib_right is_aligned_mult_triv2 is_aligned_no_overflow''
off
                  word_plus_strict_mono_right)
  also have "... ≤ k * 2 ^ m" using ak1 mw kw False
    by (simp add: less_2p_is_upper_bits_unset nat_mult_power_less_eq unat_less_power
                  word_mult_le_mono1)
  finally show ?thesis .
qed

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"
  by (metis of_nat_power Word.of_nat_neq_0 add.commute add.right_neutral
is_aligned_addD1
            is_aligned_less_sz is_aligned_no_wrap''' unat_0)

lemma le_or_mask:
  "w ≤ w' ⟹ w OR mask x ≤ w' OR mask x"
  for w w' :: <'a::len word>
  by (metis neg_mask_add_mask add.commute le_word_or1 mask_2pm1 neg_mask_mono_le
word_plus_mono_left)

end

end
```

# 6 Increment and Decrement Machine Words Without Wrap-Around

```
theory Next_and_Prev
imports
  Aligned
begin
```

Previous and next words addresses, without wrap around.

```
lift_definition word_next :: <'a::len word ⇒ 'a word>
  is <λk. if 2 ^ LENGTH('a) dvd k + 1 then - 1 else k + 1>
  by (simp flip: take_bit_eq_0_iff) (metis take_bit_add)
```

```
lift_definition word_prev :: ‹'a::len word ⇒ 'a word›
  is ‹λk. if 2 ^ LENGTH('a) dvd k then 0 else k - 1›
  by (simp flip: take_bit_eq_0_iff) (metis take_bit_diff)

lemma word_next_unfold:
  ‹word_next w = (if w = - 1 then - 1 else w + 1)›
  by transfer (simp flip: take_bit_eq_mask_iff_exp_dvd)

lemma word_prev_unfold:
  ‹word_prev w = (if w = 0 then 0 else w - 1)›
  by transfer (simp flip: take_bit_eq_0_iff)

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)›
  by transfer
    (simp add: mask_eq_exp_minus_1 take_bit_incr_eq flip: take_bit_eq_mask_iff_exp_dvd)

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)›
  by transfer (simp add: take_bit_eq_0_iff take_bit_decr_eq)

lemma word_adjacent_union:
  "word_next e = s' ⟹ s ≤ e ⟹ s' ≤ e' ⟹ {s..e} ∪ {s'..e'} = {s
.. e'}"
  apply (simp add: word_next_unfold ivl_disj_un_two_touch split: if_splits)
  apply (drule sym)
  apply simp
  apply (subst word_atLeastLessThan_Suc_atLeastAtMost_union)
     apply (simp_all add: word_Suc_le)
  done

end
```

# 7 Signed division on word

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

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

```
instantiation word :: (len) signed_division
begin
```

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

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›
  by (simp flip: signed_take_bit_decr_length_iff)

lemma sdiv_word_def:
  ‹v sdiv w = word_of_int (sint v sdiv sint w)›
  for v w :: ‹'a::len word›
  by transfer simp

lemma smod_word_def:
  ‹v smod w = word_of_int (sint v smod sint w)›
  for v w :: ‹'a::len word›
  by transfer simp

instance proof
  fix v w :: ‹'a word›
  have ‹sint v sdiv sint w * sint w + sint v smod sint w = sint v›
    by (fact sdiv_mult_smod_eq)
  then have ‹word_of_int (sint v sdiv sint w * sint w + sint v smod sint
w) = (word_of_int (sint v) :: 'a word)›
    by simp
  then show ‹v sdiv w * w + v smod w = v›
    by (simp add: sdiv_word_def smod_word_def)
qed

end

lemma signed_divide_word_code [code]:
  ‹v sdiv w =
   (let v' = sint v; w' = sint w;
        negative = (v' < 0) ≠ (w' < 0);
        result = |v'| div |w'|
    in word_of_int (if negative then - result else result))›
  for v w :: ‹'a::len word›
  by (simp add: sdiv_word_def signed_divide_int_def sgn_if)

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

```
  by (simp add: smod_word_def signed_modulo_int_def sgn_if)

lemma sdiv_smod_id:
  ‹(a sdiv b) * b + (a smod b) = a›
  for a b :: ‹'a::len word›
  by (fact sdiv_mult_smod_eq)

lemma signed_div_arith:
    "sint ((a::('a::len) word) sdiv b) = signed_take_bit (LENGTH('a) -
1) (sint a sdiv sint b)"
  by (simp add: sdiv_word_def sint_sbintrunc')

lemma signed_mod_arith:
    "sint ((a::('a::len) word) smod b) = signed_take_bit (LENGTH('a) -
1) (sint a smod sint b)"
  by (simp add: sint_sbintrunc' smod_word_def)

lemma word_sdiv_div0 [simp]:
    "(a :: ('a::len) word) sdiv 0 = 0"
  by (auto simp: sdiv_word_def signed_divide_int_def sgn_if)

lemma smod_word_zero [simp]:
  ‹w smod 0 = w› for w :: ‹'a::len word›
  by transfer (simp add: take_bit_signed_take_bit)

lemma word_sdiv_div1 [simp]:
    "(a :: ('a::len) word) sdiv 1 = a"
proof -
  have "sint (- (1::'a word)) = - 1"
    by simp
  then show ?thesis
    by (metis int_sdiv_simps(1) mult_1 mult_minus_left scast_eq scast_id
        sdiv_minus_eq sdiv_word_def signed_1 wi_hom_neg)
qed

lemma smod_word_one [simp]:
  ‹w smod 1 = 0› for w :: ‹'a::len word›
  by (simp add: smod_word_def signed_modulo_int_def)

lemma word_sdiv_div_minus1 [simp]:
    "(a :: ('a::len) word) sdiv -1 = -a"
  by (simp add: sdiv_word_def)

lemma smod_word_minus_one [simp]:
  ‹w smod - 1 = 0› for w :: ‹'a::len word›
  by (simp add: smod_word_def signed_modulo_int_def)

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

```
proof (cases ‹1 < |sint w|›)
  case True
  then show ?thesis
    by (auto simp add: sdiv_word_def signed_divide_int_def split: if_splits)
next
  case False
  then have ‹|sint w| ≤ 1›
    by simp
  then have ‹sint w ∈ {- 1, 0, 1}›
    by auto
  then have ‹(word_of_int (sint w) :: 'a::len word) ∈ word_of_int ‘ {-
1, 0, 1}›
    by blast
  then have ‹w ∈ {- 1, 0, 1}›
    by simp
  then show ?thesis by auto
qed

lemma one_smod_word_eq [simp]:
  ‹1 smod w = 1 - of_bool (w = 1 ∨ w = - 1)› for w :: ‹'a::len word›
  using sdiv_smod_id [of 1 w] by auto

lemma minus_one_sdiv_word_eq [simp]:
  ‹- 1 sdiv w = - (1 sdiv w)› for w :: ‹'a::len word›
  by (metis (mono_tags, opaque_lifting) minus_sdiv_eq of_int_minus sdiv_word_def
signed_1 sint_n1
      word_sdiv_div1 word_sdiv_div_minus1)

lemma minus_one_smod_word_eq [simp]:
  ‹- 1 smod w = - (1 smod w)› for w :: ‹'a::len word›
  using sdiv_smod_id [of ‹- 1› w] by auto

lemma smod_word_alt_def:
  "(a :: ('a::len) word) smod b = a - (a sdiv b) * b"
  by (simp add: minus_sdiv_mult_eq_smod)

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

```
  sdiv_word_def [of ‹- numeral a› ‹- numeral b›, simplified sint_sbintrunc
sint_sbintrunc_neg]
  for a b

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

lemmas word_sdiv_0 = word_sdiv_div0

lemma sdiv_word_min:
    "- (2 ^ (size a - 1)) ≤ sint (a :: ('a::len) word) sdiv sint (b ::
('a::len) word)"
  by (smt (verit, ccfv_SIG) atLeastAtMost_iff sdiv_int_range sint_ge sint_lt
wsst_TYs(3))

lemma sdiv_word_max:
  ‹sint a sdiv sint b ≤ 2 ^ (size a - Suc 0)›
  for a b :: ‹'a::len word›
proof (cases ‹sint a = 0 ∨ sint b = 0 ∨ sgn (sint a) ≠ sgn (sint b)›)
  case True then show ?thesis
  proof -
    have "⋀i j::int. i sdiv j ≤ |i|"
      by (meson atLeastAtMost_iff sdiv_int_range)
    then show ?thesis
      by (smt (verit) sint_range_size)
  qed
next
  case False
  then have ‹|sint a| div |sint b| ≤ |sint a|›
    by (subst nat_le_eq_zle [symmetric]) (simp_all add: div_abs_eq_div_nat)
  also have ‹|sint a| ≤ 2 ^ (size a - Suc 0)›
    using sint_range_size [of a] by auto
  finally show ?thesis
    using False by (simp add: signed_divide_int_def)
qed
```

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

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

lemma smod_word_mod_0:
  "x smod (0 :: ('a::len) word) = x"
  by (fact smod_word_zero)

lemma smod_word_0_mod [simp]:
  "0 smod (x :: ('a::len) word) = 0"
  by (clarsimp simp: smod_word_def)

lemma smod_word_max:
  "sint (a::'a word) smod sint (b::'a word) < 2 ^ (LENGTH('a::len) - Suc
0)"
proof (cases ‹sint b = 0 ∨ LENGTH('a) = 0›)
  case True
  then show ?thesis
    by (force simp: sint_less)
next
  case False
  then show ?thesis
    by (smt (verit) sint_greater_eq sint_less smod_int_compares)
qed

lemma smod_word_min:
  "- (2 ^ (LENGTH('a::len) - Suc 0)) ≤ sint (a::'a word) smod sint (b::'a
word)"
  by (smt (verit) sint_greater_eq sint_less smod_int_compares smod_int_mod_0)

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

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

end


theory Bitwise
  imports
    "HOL-Library.Word"
```

```
    More_Arithmetic
    Reversed_Bit_Lists
    Bit_Shifts_Infix_Syntax
```

**begin**

Helper constants used in defining addition

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

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

lemma carry_simps:
  "carry True a b = (a ∨ b)"
  "carry a True b = (a ∨ b)"
  "carry a b True = (a ∨ b)"
  "carry False a b = (a ∧ b)"
  "carry a False b = (a ∧ b)"
  "carry a b False = (a ∧ b)"
  by (auto simp add: carry_def)

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)"
  by (simp_all add: xor3_def)
```

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))"
  by simp

lemma rbl_word_1: "rev (to_bl (1 :: 'a::len word)) = takefill False (LENGTH('a))
[True]"
  by (metis rev_rev_ident rev_singleton_conv word_1_bl word_rev_tf)

lemma rbl_word_if: "rev (to_bl (if P then x else y)) = map2 (If P) (rev
(to_bl x)) (rev (to_bl y))"
  by (simp add: split_def)

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)"
  by (simp add: carry_def xor3_def)
```

```
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 (induction rule: list_induct2)
  case Nil
  then show ?case by simp
next
  case (Cons x xs y ys)
  then show ?case
    using rbl_add_carry_Cons by auto
qed

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)"
  using rbl_add_suc_carry_fold[where xs="rev (to_bl x)" and ys="rev (to_bl
y)"]
  by (smt (verit) id_apply length_rev word_add_rbl word_rotate.lbl_lbl
zip_rev)

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 [] = []"
  by (simp_all add: rbl_plus_def)

lemma rbl_word_plus: "rev (to_bl (x + y)) = rbl_plus False (rev (to_bl
x)) (rev (to_bl y))"
  by (simp add: rbl_plus_def to_bl_plus_carry zip_rev)

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"
  by (simp_all add: rbl_succ2_def)

lemma twos_complement: "- x = word_succ (not x)"
  using arg_cong[OF word_add_not[where x=x], where f="λa. a - x + 1"]
  by (simp add: word_succ_p1 word_sp_01[unfolded word_succ_p1] del: word_add_not)
```

```
lemma rbl_word_neg: "rev (to_bl (- x)) = rbl_succ2 True (map Not (rev
(to_bl x)))"
  for x :: ‹'a::len word›
  by (simp add: twos_complement word_succ_rbl[OF refl] bl_word_not rev_map
rbl_succ2_def)

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))"
  by (simp add: word_cat_bl word_rev_tf)

lemma rbl_word_slice:
  "rev (to_bl (slice n w :: 'a::len word)) =
    takefill False (LENGTH('a)) (drop n (rev (to_bl w)))"
  by (simp add: drop_rev slice_take word_rev_tf)

lemma rbl_word_ucast:
  "rev (to_bl (ucast x :: 'a::len word)) = takefill False (LENGTH('a))
(rev (to_bl x))"
  by (simp add: takefill_alt ucast_bl word_rev_tf)

lemma rbl_shiftl:
  "rev (to_bl (w << n)) = takefill False (size w) (replicate n False @
rev (to_bl w))"
  by (simp add: bl_shiftl takefill_alt word_size rev_drop)

lemma rbl_shiftr:
  "rev (to_bl (w >> n)) = takefill False (size w) (drop n (rev (to_bl
w)))"
  by (simp add: shiftr_slice rbl_word_slice word_size)

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]"
  by (simp_all add: drop_nonempty_def)

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"
  by (simp_all add: takefill_last_def) (simp_all add: takefill_alt)
```

```
lemma rbl_sshiftr:
  "rev (to_bl (w >>> n)) = takefill_last False (size w) (drop_nonempty
False n (rev (to_bl w)))"
proof (cases "n < size w")
  case True
  then show ?thesis
    by (simp add: bl_sshiftr takefill_last_def word_size takefill_alt
          rev_take last_rev drop_nonempty_def)
next
  case False
  then have §: "(w >>> n) = of_bl (replicate (size w) (msb w))"
    by (intro word_eqI) (simp add: bit_simps word_size msb_nth)
  with False show ?thesis
   apply (simp add: word_size takefill_last_def takefill_alt
      last_rev word_msb_alt word_rev_tf drop_nonempty_def take_Cons')
    by (metis Suc_pred len_gt_0 replicate_Suc)
qed

lemma nth_word_of_int:
  "bit (word_of_int x :: 'a::len word) n = (n < LENGTH('a) ∧ bit x n)"
  by (simp add: bit_word_of_int_iff)

lemma nth_scast:
  "bit (scast (x :: 'a::len word) :: 'b::len word) n =
    (n < LENGTH('b) ∧
    (if n < LENGTH('a) - 1 then bit x n
     else bit x (LENGTH('a) - 1)))"
  by (simp add: bit_signed_iff)

lemma rbl_word_scast:
  "rev (to_bl (scast x :: 'a::len word)) = takefill_last False (LENGTH('a))
(rev (to_bl x))"
proof (rule nth_equalityI)
  show "length (rev (to_bl (scast x::'a word))) = length (takefill_last
False (len_of (TYPE('a)::'a itself)) (rev (to_bl x)))"
    by (simp add: word_size takefill_last_def)
next
  fix i
  assume "i < length (rev (to_bl (scast x::'a word)))"
  then show "rev (to_bl (scast x::'a word)) ! i = takefill_last False
(LENGTH('a)) (rev (to_bl x)) ! i"
  apply (cases "LENGTH('b)")
     apply (auto simp: nth_scast takefill_last_def nth_takefill word_size
rev_nth
        to_bl_nth less_Suc_eq_le last_rev msb_nth simp flip: word_msb_alt)
  done
qed

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 = []"
  by (simp_all add: rbl_mul_def)

lemma takefill_le2: "length xs ≤ n ⟹ takefill x m (takefill x n xs)
= takefill x m xs"
  by (simp add: takefill_alt replicate_add[symmetric])

lemma take_rbl_plus: "∀n b. take n (rbl_plus b xs ys) = rbl_plus b (take
n xs) (take n ys)"
  unfolding rbl_plus_def take_zip[symmetric]
  by (rule list.induct) (auto simp: take_Cons' split_def)

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 (induct xs)
  case Nil
  show ?case by (simp add: rbl_mul_simps)
next
  case (Cons z zs)

  have rbl_word_plus': "to_bl (x + y) = rev (rbl_plus False (rev (to_bl
x)) (rev (to_bl y)))"
    for x y :: "'a word"
    by (simp add: rbl_word_plus[symmetric])

  have mult_bit: "to_bl (of_bl [z] * y) = map ((∧) z) (to_bl y)"
    by (cases z) (simp cong: map_cong, simp add: map_replicate_const cong:
map_cong)

  have shiftl: "of_bl xs * 2 * y = (of_bl xs * y) << 1" for xs
    by (simp add: push_bit_eq_mult shiftl_def)

  have zip_take_triv: "⋀xs ys n. n = length ys ⟹ zip (take n xs) ys
= zip xs ys"
    by (rule nth_equalityI) simp_all

  from Cons
  have "rbl_plus False (map ((∧) z) (rev (to_bl y)))
        (False # take (length zs) (rev (to_bl (of_bl (rev zs) * y))))
=
```

```
        rbl_plus False
         (take (Suc (length zs)) (map ((∧) z) (rev (to_bl y))))
         (take (Suc (length zs)) (rev (to_bl (of_bl (rev zs) * y * 2))))"
    unfolding word_size
    by (simp add: rbl_plus_def zip_take_triv mult.commute [of _ 2] to_bl_double_eq
take_butlast
        flip: butlast_rev)
  with Cons show ?case
    by (simp add: trans [OF of_bl_append add.commute]
        rbl_mul_simps rbl_word_plus' distrib_right mult_bit shiftl rev_map
take_rbl_plus)
qed

lemma rbl_word_mul: "rev (to_bl (x * y)) = rbl_mul (rev (to_bl x)) (rev
(to_bl y))"
  for x :: "'a::len word"
  using word_rbl_mul_induct[where xs="rev (to_bl x)" and y=y] by (simp
add: word_size)
```

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"
   apply (simp_all add: rev_bl_order_def)
  using less_Suc_eq_0_disj by fastforce

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))"
  by (induct arbitrary: F rule: list_induct2) (auto simp: rev_bl_order_simps)

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 (induct xs ys rule: list_induct2)
  case Nil
  then show ?case
    by (auto simp: rev_bl_order_simps(1))
```

```
next
  case (Cons x xs y ys)
  then show ?case
    apply (simp add: rev_bl_order_simps bl_to_bin_app_cat)
    apply (auto simp add: bl_to_bin_def add1_zle_eq concat_bit_Suc)
    done
qed

lemma word_le_rbl: "x ≤ y ⟷ rev_bl_order True (rev (to_bl x)) (rev
(to_bl y))"
  for x y :: "'a::len word"
  by (simp add: rev_bl_order_bl_to_bin word_le_def)

lemma word_less_rbl: "x < y ⟷ rev_bl_order False (rev (to_bl x)) (rev
(to_bl y))"
  for x y :: "'a::len word"
  by (simp add: word_less_alt rev_bl_order_bl_to_bin)

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)"
  by (simp_all add: map_last_def)

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 -
  have "length (to_bl x) = length (to_bl y)"
    by auto
  with word_msb_alt[where w=x] word_msb_alt[where w=y]
  show ?thesis
    unfolding word_sle_msb_le word_le_rbl
    by (cases "to_bl x"; cases "to_bl y"; auto simp: map_last_def rev_bl_order_rev_simp)
qed

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)))"
  by (metis (no_types, lifting) rev_bl_order_def signed.less_le signed.not_less
word_sle_rbl)
```

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))"
  by (simp_all add: bin_to_bl_aux_append bin_to_bl_zero_aux bin_to_bl_minus1_aux
replicate_append_same)

lemma to_bl_upt: "to_bl x = rev (map (bit x) [0 ..< size x])"
  by (simp add: to_bl_eq_rev word_size rev_map)

lemma rev_to_bl_upt: "rev (to_bl x) = map (bit x) [0 ..< size x]"
  by (simp add: to_bl_upt)

lemma upt_eq_list_intros:
  "j ≤ i ⟹ [i ..< j] = []"
  "i = x ⟹ x < j ⟹ [x + 1 ..< j] = xs ⟹ [i ..< j] = (x # xs)"
  by (simp_all add: upt_eq_Cons_conv)
```

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)"
  by auto

ML ‹
structure Word_Bitwise_Tac =
struct

val word_ss = simpset_of theory_context ‹Word›;

fun mk_nat_clist ns =
  fold_rev (Thm.mk_binop cterm ‹Cons :: nat ⇒ _›)
    ns cterm ‹[] :: nat list›;

fun upt_conv ctxt ct =
  case Thm.term_of ct of
```

```
    Const_ ‹upt for n m› =>
      let
        val (i, j) = apply2 (snd o HOLogic.dest_number) (n, m);
        val ns = map (Numeral.mk_cnumber ctyp ‹nat›) (i upto (j - 1))
          |> mk_nat_clist;
        val prop =
          Thm.mk_binop cterm ‹(=) :: nat list ⇒ _› ct ns
          |> Thm.apply cterm ‹Trueprop›;
      in
        try (fn () =>
          Goal.prove_internal ctxt [] prop
            (K (REPEAT_DETERM (resolve_tac ctxt @{thms upt_eq_list_intros}
1
                ORELSE simp_tac (put_simpset word_ss ctxt) 1))) |> mk_meta_eq)
()
      end
  | _ => NONE;

val expand_upt_simproc = simproc_setup ‹passive expand_upt ("upt x y")
= ‹K upt_conv› ›;

fun word_len_simproc_fn ctxt ct =
  (case Thm.term_of ct of
    Const_ ‹len_of _ for t› =>
     (let
        val T = fastype_of t |> dest_Type |> snd |> the_single
        val n = Numeral.mk_cnumber ctyp ‹nat› (Word_Lib.dest_binT T);
        val prop =
          Thm.mk_binop cterm ‹(=) :: nat ⇒ _› ct n
          |> Thm.apply cterm ‹Trueprop›;
      in
        Goal.prove_internal ctxt [] prop (K (simp_tac (put_simpset word_ss
ctxt) 1))
        |> mk_meta_eq |> SOME
      end handle TERM _ => NONE | TYPE _ => NONE)
  | _ => NONE);

val word_len_simproc =
  simproc_setup ‹passive word_len ("len_of x") = ‹K word_len_simproc_fn› ›;

(* convert 5 or nat 5 to Suc 4 when n_sucs = 1, Suc (Suc 4) when n_sucs
= 2,
   or just 5 (discarding nat) when n_sucs = 0 *)

fun nat_get_Suc_simproc_fn n_sucs ctxt ct =
  let
    val (f, arg) = dest_comb (Thm.term_of ct);
    val n =
      (case arg of term ‹nat› $ n => n | n => n)
```

```
      |> HOLogic.dest_number |> snd;
    val (i, j) = if n > n_sucs then (n_sucs, n - n_sucs) else (n, 0);
    val arg' = funpow i HOLogic.mk_Suc (HOLogic.mk_number ***typ*** ‹nat› j);
    val _ = if arg = arg' then raise TERM ("", []) else ();
    fun propfn g =
      HOLogic.mk_eq (g arg, g arg')
      |> HOLogic.mk_Trueprop |> Thm.cterm_of ctxt;
    val eq1 =
      Goal.prove_internal ctxt [] (propfn I)
        (K (simp_tac (put_simpset word_ss ctxt) 1));
  in
    Goal.prove_internal ctxt [] (propfn (curry (op $) f))
      (K (simp_tac (put_simpset HOL_ss ctxt addsimps [eq1]) 1))
    |> mk_meta_eq |> SOME
  end handle TERM _ => NONE;

fun nat_get_Suc_simproc n_sucs ts =
  Simplifier.make_simproc ***context***
   {name = "nat_get_Suc",
    kind = Simproc,
    lhss = map (fn t => t $ ***term*** ‹n :: nat›) ts,
    proc = K (nat_get_Suc_simproc_fn n_sucs),
    identifier = []};

val no_split_ss =
  simpset_of (put_simpset HOL_ss ***context***
    |> Splitter.del_split @{thm if_split});

val expand_word_eq_sss =
  (simpset_of (put_simpset HOL_basic_ss ***context*** addsimps
       @{thms word_eq_rbl_eq word_le_rbl word_less_rbl word_sle_rbl word_sless_rbl}),
  map simpset_of [
   put_simpset no_split_ss ***context*** addsimps
    @{thms rbl_word_plus rbl_word_and rbl_word_or rbl_word_not
                                rbl_word_neg bl_word_sub rbl_word_xor
                                rbl_word_cat rbl_word_slice rbl_word_scast
                                rbl_word_ucast rbl_shiftl rbl_shiftr rbl_sshiftr
                                rbl_word_if},
   put_simpset no_split_ss ***context*** addsimps
    @{thms to_bl_numeral to_bl_neg_numeral to_bl_0 rbl_word_1},
   put_simpset no_split_ss ***context*** addsimps
    @{thms rev_rev_ident rev_replicate rev_map to_bl_upt word_size}
          addsimprocs [word_len_simproc],
   put_simpset no_split_ss ***context*** addsimps
    @{thms list.simps split_conv replicate.simps list.map
                                zip_Cons_Cons zip_Nil drop_Suc_Cons drop_0
drop_Nil
                                foldr.simps list.map zip.simps(1) zip_Nil
zip_Cons_Cons takefill_Suc_Cons
```

```
                                takefill_Suc_Nil takefill.Z rbl_succ2_simps
                                rbl_plus_simps rev_bin_to_bl_simps append.simps
                                takefill_last_simps drop_nonempty_simps
                                rev_bl_order_simps}
          addsimprocs [expand_upt_simproc,
                       nat_get_Suc_simproc 4
                         [term ‹replicate›, term ‹takefill x›,
                          term ‹drop›, term ‹bin_to_bl›,
                          term ‹takefill_last x›,
                          term ‹drop_nonempty x›]],
    put_simpset no_split_ss context addsimps @{thms xor3_simps carry_simps
if_bool_simps}
  ])

fun tac ctxt =
  let
    val (ss, sss) = expand_word_eq_sss;
  in
    foldr1 (op THEN_ALL_NEW)
      ((CHANGED o safe_full_simp_tac (put_simpset ss ctxt)) ::
        map (fn ss => safe_full_simp_tac (put_simpset ss ctxt)) sss)
  end;

end
›

method_setup word_bitwise =
  ‹Scan.succeed (fn ctxt => Method.SIMPLE_METHOD (Word_Bitwise_Tac.tac
ctxt 1))›
  "decomposer for word equalities and inequalities into bit propositions
on concrete word lengths"

end
```

# 8 Comprehension syntax for `int`

```
theory Bit_Comprehension_Int
  imports
    Bit_Comprehension
begin

instantiation int :: bit_comprehension
begin

definition
  ‹set_bits f = (
      if ∃n. ∀m≥n. f m = f n then
      let n = LEAST n. ∀m≥n. f m = f n
      in signed_take_bit n (horner_sum of_bool 2 (map f [0..<Suc n]))
```

```
    else 0 :: int)›

instance proof
  fix k :: int
  from int_bit_bound [of k]
  obtain n where *: ‹⋀m. n ≤ m ⟹ bit k m ⟷ bit k n›
    and **: ‹n > 0 ⟹ bit k (n - 1) ≠ bit k n›
    by blast
  have l: ‹(LEAST q. ∀m≥q. bit k m ⟷ bit k q) = n›
  proof (rule Least_equality)
    show "∀m≥n. bit k m = bit k n"
      using * by blast
    show "⋀y. ∀m≥y. bit k m = bit k y ⟹ n ≤ y"
      by (metis "**" One_nat_def Suc_pred le_cases le0 neq0_conv not_less_eq_eq)
  qed
  have "signed_take_bit n (take_bit (Suc n) k) = k"
    apply (rule bit_eqI)
    by (metis "*" bit_signed_take_bit_iff bit_take_bit_iff leI lessI less_SucI
min.absorb4 min.order_iff)
  then show ‹set_bits (bit k) = k›
    unfolding * set_bits_int_def horner_sum_bit_eq_take_bit l
    using "*" by auto
qed

end

lemma int_set_bits_K_False [simp]: "(BITS _. False) = (0 :: int)"
  by (simp add: set_bits_int_def)

lemma int_set_bits_K_True [simp]: "(BITS _. True) = (-1 :: int)"
  by (simp add: set_bits_int_def)

lemma set_bits_code [code]:
  "set_bits = Code.abort (STR ''set_bits is unsupported on type int'')
(λ_. set_bits :: _ ⇒ int)"
  by simp

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 (cases ‹∃n. ∀m≥n. f m ⟷ f n›)
  case True
  then obtain q where q: ‹∀m≥q. f m ⟷ f q›
```

```
    by blast
  define n where ‹n = (LEAST n. ∀m≥n. f m ⟷ f n)›
  have ‹∀m≥n. f m ⟷ f n›
    unfolding n_def
    using q by (rule LeastI [of _ q])
  then have n: ‹⋀m. n ≤ m ⟹ f m ⟷ f n›
    by blast
  from n_def have n_eq: ‹(LEAST q. ∀m≥q. f m ⟷ f n) = n›
    by (smt (verit, best) Least_le ‹∀m≥n. f m = f n› dual_order.antisym
wellorder_Least_lemma(1))
  show ?thesis
  proof (cases ‹f n›)
    case False
    with n have *: ‹∃n. ∀n'≥n. ¬ f n'›
      by blast
    have **: ‹(LEAST n. ∀n'≥n. ¬ f n') = n›
      using False n_eq by simp
    from * False show ?thesis
      unfolding set_bits_int_def n_def [symmetric] **
      by (auto simp add: take_bit_horner_sum_bit_eq bit_horner_sum_bit_iff
take_map
          signed_take_bit_def set_bits_int_def horner_sum_bit_eq_take_bit
simp del: upt.upt_Suc)
  next
    case True
    with n obtain *: ‹∃n. ∀n'≥n. f n'› ‹¬ (∃n. ∀n'≥n. ¬ f n')›
      by (metis linorder_linear)
    have **: ‹(LEAST n. ∀n'≥n. f n') = n›
      using True n_eq by simp
    from * True show ?thesis
      unfolding set_bits_int_def n_def [symmetric] **
      by (auto simp add: take_bit_horner_sum_bit_eq
      bit_horner_sum_bit_iff take_map
      signed_take_bit_def set_bits_int_def
      horner_sum_bit_eq_take_bit nth_append simp del: upt.upt_Suc)
  qed
next
  case False
  then show ?thesis
    by (auto simp add: set_bits_int_def)
qed

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'))"
by(auto simp add: wf_set_bits_int.simps)

lemma wf_set_bits_int_const [simp]: "wf_set_bits_int (λ_. b)"
by(cases b)(auto intro: wf_set_bits_int.intros)

lemma wf_set_bits_int_fun_upd [simp]:
  "wf_set_bits_int (f(n := b)) ⟷ wf_set_bits_int f" (is "?lhs ⟷ ?rhs")
proof
  assume ?lhs
  then obtain n'
    where "(∀n''≥n'. ¬ (f(n := b)) n'') ∨ (∀n''≥n'. (f(n := b)) n'')"
    by(auto simp add: wf_set_bits_int_simps)
  hence "(∀n''≥max (Suc n) n'. ¬ f n'') ∨ (∀n''≥max (Suc n) n'. f n'')"
by auto
  thus ?rhs by(auto simp only: wf_set_bits_int_simps)
next
  assume ?rhs
  then obtain n' where "(∀n''≥n'. ¬ f n'') ∨ (∀n''≥n'. f n'')" (is
"?wf f n'")
    by(auto simp add: wf_set_bits_int_simps)
  hence "?wf (f(n := b)) (max (Suc n) n')" by auto
  thus ?lhs by(auto simp only: wf_set_bits_int_simps)
qed

lemma wf_set_bits_int_Suc [simp]:
  "wf_set_bits_int (λn. f (Suc n)) ⟷ wf_set_bits_int f" (is "?lhs ⟷
?rhs")
by(auto simp add: wf_set_bits_int_simps intro: le_SucI dest: Suc_le_D)

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)"
using wff proof cases
  case (zeros n)
  show ?thesis
  proof(cases "∀n. ¬ f n")
    case True
    hence "f = (λ_. False)" by auto
    thus ?thesis using True by(simp add: o_def)
  next
    case False
    then obtain n' where "f n'" by blast
    with zeros have "(LEAST n. ∀n'≥n. ¬ f n') = Suc (LEAST n. ∀n'≥Suc
n. ¬ f n')"
```

```
      by(auto intro: Least_Suc)
    also have "(λn. ∀n'≥Suc n. ¬ f n') = (λn. ∀n'≥n. ¬ f (Suc n'))"
by(auto dest: Suc_le_D)
    also from zeros have "∀n'≥n. ¬ f (Suc n')" by auto
    ultimately show ?thesis using zeros
      apply (simp (no_asm_simp) add: set_bits_int_unfold' exI
        del: upt.upt_Suc flip: map_map split del: if_split)
      apply (simp only: map_Suc_upt upt_conv_Cons)
      apply simp
      done
  qed
next
  case (ones n)
  show ?thesis
  proof(cases "∀n. f n")
    case True
    hence "f = (λ_. True)" by auto
    thus ?thesis using True by(simp add: o_def)
  next
    case False
    then obtain n' where "¬ f n'" by blast
    with ones have "(LEAST n. ∀n'≥n. f n') = Suc (LEAST n. ∀n'≥Suc n.
f n')"
      by(auto intro: Least_Suc)
    also have "(λn. ∀n'≥Suc n. f n') = (λn. ∀n'≥n. f (Suc n'))" by(auto
dest: Suc_le_D)
    also from ones have "∀n'≥n. f (Suc n')" by auto
    moreover from ones have "(∃n. ∀n'≥n. ¬ f n') = False"
      by(auto intro!: exI[where x="max n m" for n m] simp add: max_def
split: if_split_asm)
    moreover hence "(∃n. ∀n'≥n. ¬ f (Suc n')) = False"
      by(auto elim: allE[where x="Suc n" for n] dest: Suc_le_D)
    ultimately show ?thesis using ones
      apply (simp (no_asm_simp) add: set_bits_int_unfold' exI split del:
if_split)
      apply (auto simp add: Let_def hd_map map_tl[symmetric] map_map[symmetric]
map_Suc_upt upt_conv_Cons signed_take_bit_Suc
        not_le simp del: map_map)
      done
  qed
qed

lemma bin_last_set_bits [simp]:
  "odd (set_bits f :: int) = f 0"
  by (subst int_set_bits_unfold_BIT) simp_all

lemma bin_rest_set_bits [simp]:
  "set_bits f div (2 :: int) = set_bits (f ∘ Suc)"
  by (subst int_set_bits_unfold_BIT) simp_all
```

```
lemma bin_nth_set_bits [simp]:
  "bit (set_bits f :: int) m ⟷ f m"
using wff proof (induction m arbitrary: f)
  case 0
  then show ?case
    by (simp add: Bit_Comprehension_Int.bin_last_set_bits bit_0)
next
  case Suc
  from Suc.IH [of "f ∘ Suc"] Suc.prems show ?case
    by (simp add: Bit_Comprehension_Int.bin_rest_set_bits comp_def bit_Suc)
qed

end

end
```

# 9 Signed Words

```
theory Signed_Words
  imports "HOL-Library.Word"
begin
```

Signed words as separate (isomorphic) word length class. Useful for tagging words in C.

```
typedef ('a::len0) signed = "UNIV :: 'a set" ..

lemma card_signed [simp]: "CARD (('a::len0) signed) = CARD('a)"
  unfolding type_definition.card [OF type_definition_signed]
  by simp

instantiation signed :: (len0) len0
begin

definition
  len_signed [simp]: "len_of (x::'a::len0 signed itself) = LENGTH('a)"

instance ..

end

instance signed :: (len) len
  by (intro_classes, simp)

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)"
  by (auto simp: is_up scast_up_scast_id)
```

```
lemma ucast_scast_id [simp]:
  "ucast (scast (x :: 'a::len signed word) :: 'a word) = x"
  by transfer (simp add: take_bit_signed_take_bit)

lemma scast_of_nat [simp]:
  "scast (of_nat x :: 'a::len signed word) = (of_nat x :: 'a word)"
  by transfer (simp add: take_bit_signed_take_bit)

lemma scast_ucast_id [simp]:
  "scast (ucast (x :: 'a::len word) :: 'a signed word) = x"
  by transfer (simp add: take_bit_signed_take_bit)

lemma scast_eq_scast_id [simp]:
  "((scast (a :: 'a::len signed word) :: 'a word) = scast b) = (a = b)"
  by (metis ucast_scast_id)

lemma ucast_eq_ucast_id [simp]:
  "((ucast (a :: 'a::len word) :: 'a signed word) = ucast b) = (a = b)"
  by (metis scast_ucast_id)

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)"
  by (metis scast_ucast_id ucast_scast_id)+

lemma scast_2_power [simp]: "scast ((2 :: 'a::len signed word) ^ x) =
((2 :: 'a word) ^ x)"
  by (rule bit_word_eqI) (auto simp add: bit_simps)

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

lemma zero_sle_ucast_up:
  "¬ is_down (ucast :: 'a word ⇒ 'b signed word) ⟹
         (0 <=s ((ucast (b::('a::len) word)) :: ('b::len) signed word))"
  by transfer (simp add: bit_simps)

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›
  apply (cases ‹LENGTH('b)›)
  apply simp_all
  apply transfer
  apply (simp add: signed_take_bit_take_bit)
  apply (metis add.commute mask_eq_exp_minus_1 take_bit_incr_eq zle_add1_eq_le)
  done
```

```
lemma zero_sle_ucast:
  "(0 <=s ((ucast (b::('a::len) word)) :: ('a::len) signed word))
               = (uint b < 2 ^ (LENGTH('a) - 1))"
  apply transfer
  apply (cases ‹LENGTH('a)›)
   apply (simp_all add: take_bit_Suc_from_most bit_simps)
  apply (simp_all add: bit_simps disjunctive_add)
  done

lemma nth_w2p_scast:
  "(bit (scast ((2::'a::len signed word) ^ n) :: 'a word) m)
        ⟷ (bit (((2::'a::len  word) ^ n) :: 'a word) m)"
  by (simp add: bit_simps)

lemma scast_nop_1 [simp]:
  "((scast ((of_int x)::('a::len) word))::'a signed word) = of_int x"
  by transfer (simp add: take_bit_signed_take_bit)

lemma scast_nop_2 [simp]:
  "((scast ((of_int x)::('a::len) signed word))::'a word) = of_int x"
  by transfer (simp add: take_bit_signed_take_bit)

lemmas scast_nop = scast_nop_1 scast_nop_2 scast_id

type_synonym 'a sword = "'a signed word"

end
```

# 10 Bitwise tactic for Signed Words

```
theory Bitwise_Signed
imports
  "HOL-Library.Word"
  Bitwise
  Signed_Words
begin

ML ‹fun bw_tac_signed ctxt = let
  val (ss, sss) = Word_Bitwise_Tac.expand_word_eq_sss
  val sss = nth_map 2 (fn ss => put_simpset ss ctxt addsimps @{thms len_signed}
|> simpset_of) sss
in
    foldr1 (op THEN_ALL_NEW)
      ((CHANGED o safe_full_simp_tac (put_simpset ss ctxt)) ::
        map (fn ss => safe_full_simp_tac (put_simpset ss ctxt)) sss)
  end;
›
```

```
method_setup word_bitwise_signed =
  ‹Scan.succeed (fn ctxt => Method.SIMPLE_METHOD (bw_tac_signed ctxt 1))›
  "decomposer for word equalities and inequalities into bit propositions
on concrete word lengths"

end
```

# 11 Enumeration Instances for Words

```
theory Enumeration_Word
  imports More_Word Enumeration Even_More_List
begin

lemma length_word_enum: "length (enum :: 'a :: len word list) = 2 ^ LENGTH('a)"
  by (simp add: enum_word_def)

lemma fromEnum_unat[simp]: "fromEnum (x :: 'a::len word) = unat x"
proof -
  have "enum ! the_index enum x = x" by (auto intro: nth_the_index)
  moreover
  have "the_index enum x < length (enum::'a::len word list)" by (auto
intro: the_index_bounded)
  moreover
  { fix y assume "of_nat y = x"
    moreover assume "y < 2 ^ LENGTH('a)"
    ultimately have "y = unat x" using of_nat_inverse by fastforce
  }
  ultimately
  show ?thesis by (simp add: fromEnum_def enum_word_def)
qed

lemma toEnum_of_nat[simp]: "n < 2 ^ LENGTH('a) ⟹ (toEnum n :: 'a ::
len word) = of_nat n"
  by (simp add: toEnum_def length_word_enum enum_word_def)

instantiation word :: (len) enumeration_both
begin

definition
  enum_alt_word_def: "enum_alt ≡ alt_from_ord (enum :: ('a :: len) word
list)"

instance
  by (intro_classes, simp add: enum_alt_word_def)

end

definition
  upto_enum_step :: "('a :: len) word ⇒ 'a word ⇒ 'a word ⇒ 'a word
```

```
list"
    (<(<notation=<mixfix upto_enum_step>>[_ , _ .e. _])>)
where
  "[a , b .e. c] ≡ if c < a then [] else map (λx. a + x * (b - a)) [0
.e. (c - a) div (b - a)]"


lemma maxBound_word:
  "(maxBound::'a::len word) = -1"
  by (simp add: maxBound_def enum_word_def last_map of_nat_diff)

lemma minBound_word:
  "(minBound::'a::len word) = 0"
  by (simp add: minBound_def enum_word_def upt_conv_Cons)

lemma maxBound_max_word:
  "(maxBound::'a::len word) = - 1"
  by (fact maxBound_word)

lemma leq_maxBound [simp]:
  "(x::'a::len word) ≤ maxBound"
  by (simp add: maxBound_max_word)

lemma upto_enum_red':
  assumes lt: "1 ≤ X"
  shows "[(0::'a :: len word) .e. X - 1] =  map of_nat [0 ..< unat X]"
proof -
  have lt': "unat X < 2 ^ LENGTH('a)"
    by (rule unat_lt2p)
  have "map (toEnum::nat ⇒ 'a word) [0..<unat X] = map word_of_nat [0..<unat
X]"
    using order_less_trans by fastforce
  then show ?thesis
    by (simp add: Suc_unat_diff_1 lt upto_enum_red)
qed

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]" using szv
  by (simp add: upto_enum_red' word_1_le_power)

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 -
  have "⋀n. ⟦n < 2 ^ (sz - us)⟧ ⟹ toEnum n * 2 ^ us = (word_of_nat n
```

```
* 2 ^ us :: 'a word)"
    using szv nat_less_le order_le_less_trans by fastforce
  with assms show ?thesis
    by (simp add: upto_enum_step_def upto_enum_red Suc_div_unat_helper)
qed

lemma upto_enum_word: "[x .e. y] = map of_nat [unat x ..< Suc (unat y)]"
  unfolding upto_enum_red
  using order_less_trans toEnum_of_nat by force

lemma word_upto_Cons_eq:
  "x < y ⟹ [x::'a::len word .e. y] = x # [x + 1 .e. y]"
  unfolding upto_enum_red
  using lessI less_is_non_zero_p1 unatSuc2 unat_mono upt_conv_Cons by
fastforce

lemma distinct_enum_upto:
  "distinct [(0 :: 'a::len word) .e. b]"
proof -
  have "⋀(b::'a word). [0 .e. b] = nths enum {..< Suc (fromEnum b)}"
    unfolding upto_enum_red nths_upt_eq_take enum_word_def
    using order_less_trans toEnum_of_nat
    by (fastforce simp: take_map Suc_leI)
  then show ?thesis
    by (rule ssubst) (rule distinct_nthsI, simp)
qed

lemma upto_enum_set_conv [simp]:
  fixes a :: "'a :: len word"
  shows "set [a .e. b] = {x. a ≤ x ∧ x ≤ b}"
proof -
  have "a ≤ b"
    if "unat a ≤ unat b"
    using that word_less_eq_iff_unsigned by blast
  moreover have "a ≤ toEnum m"
    if "unat a ≤ m" "m < unat b" for m
    using that
    by (metis fromEnum_unat le_unat_uoi nat_less_le to_from_enum word_le_nat_alt)
  moreover have "toEnum n ≤ b"
    if "unat a ≤ n" "n < unat b" for n
    using that
    by (metis fromEnum_unat le_unat_uoi nat_less_le to_from_enum word_of_nat_le)
  moreover have "w ∈ toEnum ` {x. unat a ≤ unat b ∧ (x = unat b ∨ unat
a ≤ x ∧ x < unat b)}"
    if "a ≤ w" and "w ≤ b" for w :: "'a word"
    using that
    by (smt (verit, del_insts) order.order_iff_strict order.trans fromEnum_unat
imageI mem_Collect_eq to_from_enum unat_mono)
  ultimately show ?thesis
```

```
    by (auto simp: upto_enum_red)
qed

lemma upto_enum_less:
  assumes "x ∈ set [(a::'a::len word).e.2 ^ n - 1]" and "n < LENGTH('a::len)"
  shows   "x < 2 ^ n"
  using assms by auto

lemma upto_enum_len_less:
  "⟦ n ≤ length [a, b .e. c]; n ≠ 0 ⟧ ⟹ a ≤ c"
  unfolding upto_enum_step_def
  by (simp split: if_split_asm)

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"
  unfolding upto_enum_step_def
  by (simp add: upto_enum_red)

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]"
  by (simp add: ax word_upto_Cons_eq)

lemma upto_enum_set_conv2:
  fixes a :: "'a::len word"
  shows "set [a .e. b] = {a .. b}"
  by auto

lemma length_upto_enum [simp]:
  fixes a :: "'a :: len word"
  shows "length [a .e. b] = Suc (unat b) - unat a"
  by (metis length_map length_upt upto_enum_word)

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)"
  by (simp add: word_le_nat_alt)

lemma length_upto_enum_less_one:
  "⟦a ≤ b; b ≠ 0⟧ ⟹ length [a .e. b - 1] = unat (b - a)"
  by (simp add: unat_sub)

lemma drop_upto_enum: "drop (unat n) [0 .e. m] = [n .e. m]"
  by (induction m) (auto simp: upto_enum_def drop_map)
```

```
lemma distinct_enum_upto' [simp]:
  "distinct [a::'a::len word .e. b]"
  by (metis distinct_drop distinct_enum_upto drop_upto_enum)

lemma length_interval:
  "⟦set xs = {x. (a::'a::len word) ≤ x ∧ x ≤ b}; distinct xs⟧
  ⟹ length xs = Suc (unat b) - unat a"
  by (metis distinct_card distinct_enum_upto' length_upto_enum upto_enum_set_conv)

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 -
  have §: "[0..<2 ^ LENGTH('a)] = ([0..<unat v] @ [unat v..<2 ^ LENGTH('a)])"
    by (simp add: order_less_imp_le upt_add_eq_append')
  have "⋀n. ⟦unat v < n; n < 2 ^ LENGTH('a)⟧ ⟹ v < word_of_nat n"
    using unat_ucast_less_no_overflow by blast
  moreover
  have "∀w∈set (map word_of_nat [0..<unat v]). w < v"
    using word_of_nat_less by force
  ultimately show ?thesis
    unfolding enum_word_def order_less_imp_le upt_add_eq_append' §
    by (force simp add: upt_conv_Cons)
qed

lemma remdups_enum_upto:
  fixes s::"'a::len word"
  shows "remdups [s .e. e] = [s .e. e]"
  by simp

lemma card_enum_upto:
  fixes s::"'a::len word"
  shows "card (set [s .e. e]) = Suc (unat e) - unat s"
  by (subst List.card_set) (simp add: remdups_enum_upto)

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]"
  unfolding upto_enum_step_def
proof (subst upto_enum_red, subst if_not_P [OF leD [OF lt3]], clarsimp,
rule conjI)
  show "unat ((z - x) div (y - x)) = 0"
  proof (subst unat_div, rule div_less)
    have syx: "unat (y - x) = unat y - unat x"
      by (rule unat_sub [OF order_less_imp_le]) fact
    moreover have "unat (z - x) = unat z - unat x"
```

```
    by (rule unat_sub) fact

  ultimately show "unat (z - x) < unat (y - x)"
    using lt2 lt3 unat_mono word_less_minus_mono_left by blast
qed

then show "(z - x) div (y - x) * (y - x) = 0"
  by (simp add: unat_div) (simp add: word_arith_nat_defs(6))
qed

end
```

# 12 Print Words in Hex

```
theory Hex_Words
imports
  "HOL-Library.Word"
begin
```

Print words in hex.

```
typed_print_translation \<open>
let
  fun dest_num (Const (@{const_syntax Num.Bit0}, _) $ n) = 2 * dest_num
n
    | dest_num (Const (@{const_syntax Num.Bit1}, _) $ n) = 2 * dest_num
n + 1
    | dest_num (Const (@{const_syntax Num.One}, _)) = 1;

  fun dest_bin_hex_str tm =
  let
    val num = dest_num tm;
    val pre = if num < 10 then "" else "0x"
  in
    pre ^ (Int.fmt StringCvt.HEX num)
  end;

  fun num_tr' sign ctxt T [n] =
    let
      val k = dest_bin_hex_str n;
      val t' = Syntax.const @{syntax_const "_Numeral"} $
        Syntax.free (sign ^ k);
    in
      case T of
        Type (@{type_name fun}, [_, T' as Type("Word.word",_)]) =>
          if not (Config.get ctxt show_types) andalso can Term.dest_Type
T'
          then t'
          else Syntax.const @{syntax_const "_constrain"} $ t' $
                           Syntax_Phases.term_of_typ ctxt T'
```

```
      | T' => if T' = dummyT then t' else raise Match
    end;
in [(@{const_syntax numeral}, num_tr' "")] end
›
```

**end**

# 13 Normalising Word Numerals

**theory** Norm_Words
  **imports** Signed_Words
**begin**

Normalise word numerals, including negative ones apart from `- 1`, to the interval `[0..2^len_of 'a)`. Only for concrete word lengths.

**lemma** bintrunc_numeral:
```
  "(take_bit :: nat ⇒ int ⇒ int) (numeral k) x = of_bool (odd x) + 2
* (take_bit :: nat ⇒ int ⇒ int) (pred_numeral k) (x div 2)"
```
  **by** (simp add: numeral_eq_Suc take_bit_Suc mod_2_eq_odd)

**lemma** neg_num_bintr:
```
  "(- numeral x :: 'a::len word) = word_of_int (take_bit LENGTH('a) (-
numeral x))"
```
  **by** transfer simp

**ML** ‹
```
  fun is_refl (Const _ ‹Pure.eq _ for x y›) = (x = y)
    | is_refl _ = false;

  fun signed_dest_wordT (Type ‹word (Type ‹signed T›)›) = Word_Lib.dest_binT
T
    | signed_dest_wordT T = Word_Lib.dest_wordT T

  fun typ_size_of t = signed_dest_wordT (type_of (Thm.term_of t));

  fun num_len (Const _ ‹Num.Bit0 for n›) = num_len n + 1
    | num_len (Const _ ‹Num.Bit1 for n›) = num_len n + 1
    | num_len (Const _ ‹Num.One›) = 1
    | num_len (Const _ ‹numeral _ for t›) = num_len t
    | num_len (Const _ ‹uminus _ for t›) = num_len t
    | num_len t = raise TERM ("num_len", [t])

  fun unsigned_norm is_neg _ ctxt ct =
  (if is_neg orelse num_len (Thm.term_of ct) > typ_size_of ct then let
      val btr = if is_neg
                then @{thm neg_num_bintr} else @{thm num_abs_bintr}
      val th = [Thm.reflexive ct, mk_eq btr] MRS transitive_thm

      (* will work in context of theory Word as well *)
```

```
      val ss = simpset_of (@{context} addsimps @{thms bintrunc_numeral}
delsimps @{thms take_bit_minus_one_eq_mask})
        (* TODO: completely explicitly determined simpset *)
      val cnv = simplify (put_simpset ss ctxt) th
    in if is_refl (Thm.prop_of cnv) then NONE else SOME cnv end
    else NONE)
  handle TERM ("num_len", _) => NONE
       | TYPE ("dest_binT", _, _) => NONE
›

simproc_setup
  unsigned_norm ("numeral n::'a::len word") = ‹unsigned_norm false›

simproc_setup
  unsigned_norm_neg0 ("-numeral (num.Bit0 num)::'a::len word") = ‹unsigned_norm
true›

simproc_setup
  unsigned_norm_neg1 ("-numeral (num.Bit1 num)::'a::len word") = ‹unsigned_norm
true›

lemma minus_one_norm:
  "(-1 :: 'a :: len word) = of_nat (2 ^ LENGTH('a) - 1)"
  by (simp add:of_nat_diff)

lemmas minus_one_norm_num =
  minus_one_norm [where 'a="'b::len bit0"] minus_one_norm [where 'a="'b::len0
bit1"]

context
begin

private lemma "f (7 :: 2 word) = f 3" by simp

private lemma "f 7 = f (3 :: 2 word)" by simp

private lemma "f (-2) = f (21 + 1 :: 3 word)" by simp

private lemma "f (-2) = f (13 + 1 :: 'a::len word)"
  apply simp
  oops

private lemma "f (-2) = f (0xFFFFFFFE :: 32 word)" by simp

private lemma "(-1 :: 2 word) = 3" by simp

private lemma "f (-2) = f (0xFFFFFFFE :: 32 signed word)" by simp
```

We leave `- 1` untouched by default, because it is often useful and its normal

form can be large. To include it in the normalisation, add `minus_one_norm_num`. The additional normalisation is restricted to concrete numeral word lengths, like the rest.

```
context
  notes minus_one_norm_num [simp]
begin

private lemma "f (-1) = f (15 :: 4 word)" by simp

private lemma "f (-1) = f (7 :: 3 word)" by simp

private lemma "f (-1) = f (0xFFFF :: 16 word)" by simp

private lemma "f (-1) = f (0xFFFF + 1 :: 'a::len word)"
  apply simp
  oops

end

end

end
```

# 14 Syntax bundles for traditional infix syntax

```
theory Syntax_Bundles
  imports "HOL-Library.Word"
begin

bundle bit_projection_infix_syntax
begin

notation bit  (infixl <!!> 100)

end

end
```

```
theory Sgn_Abs
  imports Most_significant_bit
begin
```

# 15 `sgn` and `abs` for `'a word`

## 15.1 Instances

`sgn` on words returns -1, 0, or 1.

**instantiation** word :: (len) sgn
**begin**

**definition** sgn_word :: ‹'a word ⇒ 'a word› **where**
  ‹sgn w = (if w = 0 then 0 else if 0 <s w then 1 else -1)›

**instance** ..

**end**

**lemma** word_sgn_0[simp]:
  "sgn 0 = (0::'a::len word)"
  **by** (simp add: sgn_word_def)

**lemma** word_sgn_1[simp]:
  "sgn 1 = (1::'a::len word)"
  **by** (simp add: sgn_word_def)

**lemma** word_sgn_max_word[simp]:
  "sgn (- 1) = (-1::'a::len word)"
  **by** (clarsimp simp: sgn_word_def word_sless_alt)

**lemmas** word_sgn_numeral[simp] = sgn_word_def[**where** w="numeral w" **for** w]

abs on words is the usual definition.

**instantiation** word :: (len) abs
**begin**

**definition** abs_word :: ‹'a word ⇒ 'a word› **where**
  ‹abs w = (if w ≤s 0 then -w else w)›

**instance** ..

**end**

**lemma** word_abs_0[simp]:
  "|0| = (0::'a::len word)"
  **by** (simp add: abs_word_def)

**lemma** word_abs_1[simp]:
  "|1| = (1::'a::len word)"
  **by** (simp add: abs_word_def)

```
lemma word_abs_max_word[simp]:
  "|- 1| = (1::'a::len word)"
  by (clarsimp simp: abs_word_def word_sle_eq)

lemma word_abs_msb:
  "|w| = (if msb w then -w else w)" for w::"'a::len word"
  by (simp add: abs_word_def msb_word_iff_sless_0 word_sless_eq)

lemmas word_abs_numeral[simp] = word_abs_msb[where w="numeral w" for
w]
```

## 15.2 Properties

```
lemma word_sgn_0_0:
  "sgn a = 0 ⟷ a = 0" for a::"'a::len word"
  by (simp add: sgn_word_def)

lemma word_sgn_1_pos:
  "1 < LENGTH('a) ⟹ sgn a = 1 ⟷ 0 <s a" for a::"'a::len word"
  unfolding sgn_word_def by simp

lemma word_sgn_1_neg:
  "sgn a = - 1 ⟷ a <s 0"
  unfolding sgn_word_def
  using sint_1_cases by force

lemma word_sgn_pos[simp]:
  "0 <s a ⟹ sgn a = 1"
  by (simp add: sgn_word_def)

lemma word_sgn_neg[simp]:
  "a <s 0 ⟹ sgn a = - 1"
  by (simp only: word_sgn_1_neg)

lemma word_abs_sgn:
  "|k| = k * sgn k" for k :: "'a::len word"
  unfolding sgn_word_def abs_word_def
  by auto

lemma word_sgn_greater[simp]:
  "0 <s sgn a ⟷ 0 <s a" for a::"'a::len word"
  by (smt (verit) signed_eq_0_iff sint_1_cases sint_n1 word_sgn_0_0 word_sgn_neg
word_sgn_pos
                  word_sless_alt)

lemma word_sgn_less[simp]:
  "sgn a <s 0 ⟷ a <s 0" for a::"'a::len word"
  unfolding sgn_word_def
```

```
  using degenerate_word signed.antisym_conv3 word_sless_alt by force

lemma word_abs_sgn_eq_1[simp]:
  "a ≠ 0 ⟹ |sgn a| = 1" for a::"'a::len word"
  unfolding abs_word_def sgn_word_def
  by (clarsimp simp: word_sle_eq)

lemma word_abs_sgn_eq:
  "|sgn a| = (if a = 0 then 0 else 1)" for a::"'a::len word"
  by clarsimp

lemma word_sgn_mult_self_eq[simp]:
  "sgn a * sgn a = of_bool (a ≠ 0)" for a::"'a::len word"
  by (cases "0 <s a"; simp)

end
```

# 16 Displaying Phantom Types for Word Operations

```
theory Type_Syntax
  imports "HOL-Library.Word"
begin

ML ‹
structure Word_Syntax =
struct

  val show_word_types = Attrib.setup_config_bool @{binding show_word_types}
(K true)

  fun tr' cnst ctxt typ ts = if Config.get ctxt show_word_types then
      case (Term.binder_types typ, Term.body_type typ) of
        ([Type ‹word S›], Type ‹word T›) =>
          list_comb
            (Syntax.const cnst $ Syntax_Phases.term_of_typ ctxt S $ Syntax_Phases.term_of_ty
ctxt T
            , ts)
        | _ => raise Match
  else raise Match

end
›


syntax
  "_Ucast" :: "type ⇒ type ⇒ logic"
    (‹(‹indent=1 notation=‹mixfix UCAST› ›UCAST/(‹indent=1 notation=‹infix
```

```
cast>>'(_ → _')))>)
syntax_consts
  "_Ucast" == ucast
translations
  "UCAST('s → 't)" => "CONST ucast :: ('s word ⇒ 't word)"
typed_print_translation
  < [(@{const_syntax ucast}, Word_Syntax.tr' @{syntax_const "_Ucast"})]
>


syntax
  "_Scast" :: "type ⇒ type ⇒ logic"
    (<(<indent=1 notation=<mixfix SCAST>>SCAST/(<indent=1 notation=<infix
cast>>'(_ → _')))>)
syntax_consts
  "_Scast" == scast
translations
  "SCAST('s → 't)" => "CONST scast :: ('s word ⇒ 't word)"
typed_print_translation
  < [(@{const_syntax scast}, Word_Syntax.tr' @{syntax_const "_Scast"})]
>


syntax
  "_Revcast" :: "type ⇒ type ⇒ logic"
    (<(<indent=1 notation=<mixfix REVCAST>>REVCAST/(<indent=1 notation=<infix
cast>>'(_ → _')))>)
syntax_consts
  "_Revcast" == revcast
translations
  "REVCAST('s → 't)" => "CONST revcast :: ('s word ⇒ 't word)"
typed_print_translation
  < [(@{const_syntax revcast}, Word_Syntax.tr' @{syntax_const "_Revcast"})]
>



end
```

# 17 Solving Word Equalities

```
theory Word_EqI
  imports
    More_Word
    Aligned
    "HOL-Eisbach.Eisbach_Tools"
begin
```

Some word equalities can be solved by considering the problem bitwise for

all `n < LENGTH('a)`. This is similar to the existing method `word_bitwise` and expanding into an explicit list of bits. The `word_bitwise` only works on concrete word lengths, but can treat a wider number of operators (in particular a mix of arithmetic, order, and bit operations). The `word_eqI` method below works on words of abstract size (`'a word`) and produces smaller, more abstract goals, but does not deal with arithmetic operations.

**lemmas** `le_mask_high_bits_len = le_mask_high_bits[unfolded word_size]`
**lemmas** `neg_mask_le_high_bits_len = neg_mask_le_high_bits[unfolded word_size]`

**named_theorems** `word_eqI_simps`

**lemmas** `[word_eqI_simps] =`
```
  word_or_zero
  neg_mask_test_bit
  nth_ucast
  less_2p_is_upper_bits_unset
  le_mask_high_bits_len
  neg_mask_le_high_bits_len
  bang_eq
  is_up
  is_down
  is_aligned_nth
  word_size
```

**lemmas** `word_eqI_folds [symmetric] =`
```
  push_bit_eq_mult
  drop_bit_eq_div
  take_bit_eq_mod
```

**lemmas** `word_eqI_rules = word_eqI [rule_format, unfolded word_size] bit_eqI`

**lemma** `test_bit_lenD:`
```
  "bit x n ⟹ n < LENGTH('a) ∧ bit x n" for x :: "'a :: len word"
  by (fastforce dest: test_bit_size simp: word_size)
```

— Method to reduce goals of the form `P ⟹ x = y` for words of abstract length to reasoning on bits of the words. Leaves open goal if unsolved.
**method** `word_eqI uses simp simp_del split split_del cong flip =`
```
  (
   rule word_eqI_rules,

   (simp only: word_eqI_folds)?,

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

   ((drule less_mask_eq)+)?,
```

```
   (simp only: bit_simps word_eqI_simps)?,

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

   ((drule test_bit_lenD)+)?,

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

   (simp add: simp test_bit_conj_lt del: simp_del flip: flip split: split
split del: split_del cong: cong)?)
```

— Method to reduce goals of the form `P ⟹ x = y` for words of abstract length to reasoning on bits of the words. Fails if goal unsolved, but tries harder than `word_eqI`.

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

end


theory Boolean_Inequalities
  imports Word_EqI
begin
```

# 18 All inequalities between binary Boolean operations on `'a word`

Enumerates all binary functions resulting from Boolean operations on `'a word` and derives all inequalities of the form `f x y ≤ g x y` between them. We leave out the trivial `0 ≤ g x y`, `f x y ≤ - 1`, and `f x y ≤ f x y`, because these are already readily available to the simplifier and other methods.

This leaves 36 inequalities. Some of these are subsumed by each other, but we generate the full list to avoid too much manual processing.

All inequalities produced here are in simp normal form.

**context**

```
  includes bit_operations_syntax
begin


definition all_bool_word_funs :: "('a::len word ⇒ 'a word ⇒ 'a word)
list" where
  "all_bool_word_funs ≡ [
     λx y. 0,
     λx y. x AND y,
     λx y. x AND NOT y,
     λx y. x,
     λx y. NOT x AND y,
     λx y. y,
     λx y. x XOR y,
     λx y. x OR y,
     λx y. NOT (x OR y),
     λx y. NOT (x XOR y),
     λx y. NOT y,
     λx y. x OR NOT y,
     λx y. NOT x,
     λx y. NOT x OR y,
     λx y. NOT (x AND y),
     λx y. -1
   ]"
```

The inequalities on `'a word` follow directly from implications on propositional Boolean logic, which `simp` can solve automatically. This means, we can simply enumerate all combinations, reduce from `'a word` to `bool`, and attempt to solve by `simp` to get the complete list.

```
local_setup ‹
let
  (* derived from Numeral.mk_num, but returns a term, not syntax. *)
  fun mk_num n =
    if n > 0 then
      (case Integer.quot_rem n 2 of
        (0, 1) => const ‹Num.One›
      | (n, 0) => const ‹Num.Bit0› $ mk_num n
      | (n, 1) => const ‹Num.Bit1› $ mk_num n)
    else raise Match

  (* derived from Numeral.mk_number, but returns a term, not syntax. *)
  fun mk_number n =
    if n = 0 then term ‹0::nat›
    else if n = 1 then term ‹1::nat›
    else term ‹numeral::num ⇒ nat› $ mk_num n;

  (* generic form of the goal statement *)
  val goal = @{term "λn m. (all_bool_word_funs!n) x y ≤ (all_bool_word_funs!m)
x y"}
```

```
  (* instance of the goal statement for a pair (i,j) of Boolean functions
*)
  fun stmt (i,j) = HOLogic.Trueprop $ (goal $ mk_number i $ mk_number
j)

  (* attempt to prove an inequality between functions i and j *)
  fun le_thm ctxt (i,j) = Goal.prove ctxt ["x", "y"] [] (stmt (i,j)) (fn
_ =>
    (asm_full_simp_tac (ctxt addsimps [@{thm all_bool_word_funs_def}])
     THEN_ALL_NEW resolve_tac ctxt @{thms word_leI}
     THEN_ALL_NEW asm_full_simp_tac (ctxt addsimps @{thms word_eqI_simps
bit_simps})) 1)

  (* generate all combinations for (i,j), collect successful inequality
theorems,
     unfold all_bool_word_funs, and put into simp normal form. We leave
out 0 (bottom)
     and 15 (top), as well as reflexive thms to remove trivial lemmas
from the list.*)
  fun thms ctxt =
    map_product (fn x => fn y => (x,y)) (1 upto 14) (1 upto 14)
    |> filter (fn (x,y) => x <> y)
    |> map_filter (try (le_thm ctxt))
    |> map (Simplifier.simplify ctxt o Local_Defs.unfold ctxt @{thms all_bool_word_funs_def
in
  fn ctxt =>
    Local_Theory.notes [((Binding.name "word_bool_le_funs", []), [(thms
ctxt, [])])] ctxt |> #2
end
›
```

**lemma**
```
  "x AND y ≤ x" for x :: "'a::len word"
  by (rule word_bool_le_funs(1))
```

**lemma**
```
  "NOT x ≤ NOT x OR NOT y" for x :: "'a::len word"
  by (rule word_bool_le_funs(36))
```

**lemma**
```
  "x XOR y ≤ NOT x OR NOT y" for x :: "'a::len word"
  by (rule word_bool_le_funs)
```

**lemma** word_xor_le_nand:
```
  "x XOR y ≤ NOT (x AND y)" for x :: "'a::len word"
  by (simp add: word_bool_le_funs)
```

**end**

**end**

# 19 Lemmas with Generic Word Length

**theory** Word_Lemmas
  **imports**
    Type_Syntax
    Signed_Division_Word
    Signed_Words
    More_Word
    Most_significant_bit
    Enumeration_Word
    Aligned
    Bit_Shifts_Infix_Syntax
    Boolean_Inequalities
    Word_EqI

**begin**

**context**
  **includes** bit_operations_syntax
**begin**

**lemma** word_max_le_or:
  "max x y ≤ x OR y" **for** x :: "'a::len word"
  **by** (simp add: word_bool_le_funs)

**lemma** word_and_le_min:
  "x AND y ≤ min x y" **for** x :: "'a::len word"
  **by** (simp add: word_bool_le_funs)

**lemma** word_not_le_eq:
  "(NOT x ≤ y) = (NOT y ≤ x)" **for** x :: "'a::len word"
  **by** transfer (auto simp: take_bit_not_eq_mask_diff)

**lemma** word_not_le_not_eq[simp]:
  "(NOT y ≤ NOT x) = (x ≤ y)" **for** x :: "'a::len word"
  **by** (subst word_not_le_eq) simp

**lemma** not_min_eq:
  "NOT (min x y) = max (NOT x) (NOT y)" **for** x :: "'a::len word"
  **unfolding** min_def max_def
  **by** auto

**lemma** not_max_eq:
  "NOT (max x y) = min (NOT x) (NOT y)" **for** x :: "'a::len word"
  **unfolding** min_def max_def

```
  by auto

lemma ucast_le_ucast_eq:
  fixes x y :: "'a::len word"
  assumes x: "x < 2 ^ n"
  assumes y: "y < 2 ^ n"
  assumes n: "n = LENGTH('b::len)"
  shows "(UCAST('a → 'b) x ≤ UCAST('a → 'b) y) ⟷ (x ≤ y)" (is "?L=_")
proof
  assume ?L then show "x ≤ y"
    by (metis x less_mask_eq le_ucast_ucast_le n ucast_ucast_mask)
qed (use n ucast_mono_le y in auto)

lemma ucast_zero_is_aligned:
  ‹is_aligned w n› if ‹UCAST('a::len → 'b::len) w = 0› ‹n ≤ LENGTH('b)›
proof (rule is_aligned_bitI)
  fix q
  assume ‹q < n›
  moreover have ‹bit (UCAST('a::len → 'b::len) w) q = bit 0 q›
    using that by simp
  with ‹q < n› ‹n ≤ LENGTH('b)› show ‹¬ bit w q›
    by (simp add: bit_simps)
qed

lemma unat_ucast_eq_unat_and_mask:
  "unat (UCAST('b::len → 'a::len) w) = unat (w AND mask LENGTH('a))"
  by (metis take_bit_eq_mask unsigned_take_bit_eq unsigned_ucast_eq)

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 -
  from that have a1: ‹x ≤ word_of_int (uint (word_of_int (2 ^ LENGTH('b)
- 1) :: 'b word))›
    by (simp add: of_int_mask_eq)
  have f2: "((∃i ia. (0::int) ≤ i ∧ ¬ 0 ≤ i + - 1 * ia ∧ i mod ia ≠
i) ∨
            ¬ (0::int) ≤ - 1 + 2 ^ LENGTH('b) ∨ (0::int) ≤ - 1 + 2 ^
LENGTH('b) + - 1 * 2 ^ LENGTH('b) ∨
            (- (1::int) + 2 ^ LENGTH('b)) mod 2 ^ LENGTH('b) =
              - 1 + 2 ^ LENGTH('b)) = ((∃i ia. (0::int) ≤ i ∧ ¬ 0 ≤
i + - 1 * ia ∧ i mod ia ≠ i) ∨
            ¬ (1::int) ≤ 2 ^ LENGTH('b) ∨
            2 ^ LENGTH('b) + - (1::int) * ((- 1 + 2 ^ LENGTH('b)) mod
2 ^ LENGTH('b)) = 1)"
    by force
  have f3: "∀i ia. ¬ (0::int) ≤ i ∨ 0 ≤ i + - 1 * ia ∨ i mod ia = i"
    using mod_pos_pos_trivial by force
```

```
  have "(1::int) ≤ 2 ^ LENGTH('b)"
    by simp
  then have "2 ^ LENGTH('b) + - (1::int) * ((- 1 + 2 ^ LENGTH('b)) mod
2 ^ len_of TYPE ('b)) = 1"
    using f3 f2 by blast
  then have f4: "- (1::int) + 2 ^ LENGTH('b) = (- 1 + 2 ^ LENGTH('b))
mod 2 ^ LENGTH('b)"
    by linarith
  have f5: "x ≤ word_of_int (uint (word_of_int (- 1 + 2 ^ LENGTH('b))::'b
word))"
    using a1 by force
  have f6: "2 ^ LENGTH('b) + - (1::int) = - 1 + 2 ^ LENGTH('b)"
    by force
  have f7: "- (1::int) * 1 = - 1"
    by auto
  have "∀x0 x1. (x1::int) - x0 = x1 + - 1 * x0"
    by force
  then have §: "x ≤ 2 ^ LENGTH('b) - 1"
    using f7 f6 f5 f4 by (metis uint_word_of_int wi_homs(2) word_arith_wis(8)
word_of_int_2p)
  then have ‹uint x ≤ uint (2 ^ LENGTH('b) - (1 :: 'a word))›
    by (simp add: word_le_def)
  then have ‹uint x ≤ 2 ^ LENGTH('b) - 1›
    by (simp add: uint_word_ariths)
      (metis ‹1 ≤ 2 ^ LENGTH('b)› ‹uint x ≤ uint (2 ^ LENGTH('b) - 1)›
linorder_not_less lt2p_lem uint_1 uint_minus_simple_alt uint_power_lower
word_le_def zle_diff1_eq)
  then show ?thesis
    by (metis § and_mask_eq_iff_le_mask mask_eq_exp_minus_1 ucast_ucast_mask)
qed

lemma uint_shiftr_eq:
  ‹uint (w >> n) = uint w div 2 ^ n›
  by word_eqI

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›
  by (simp add: bit_simps)

lemma bit_shiftr_word_iff:
  ‹bit (w >> m) n ⟷ bit w (m + n)›
  for w :: ‹'a::len word›
  by (simp add: bit_simps)

lemma uint_sshiftr_eq:
  ‹uint (w >>> n) = take_bit LENGTH('a) (sint w div 2 ^  n)›
  for w :: ‹'a::len word›
  by (word_eqI_solve dest: test_bit_lenD)
```

```
lemma sshiftr_n1: "-1 >>> n = -1"
  by (simp add: sshiftr_def)

lemma nth_sshiftr:
  "bit (w >>> m) n = (n < size w ∧ (if n + m ≥ size w then bit w (size
w - 1) else bit w (n + m)))"
  by (simp add: add.commute bit_sshiftr_iff test_bit_over wsst_TYs(3))

lemma sshiftr_numeral:
  ‹(numeral k >>> numeral n :: 'a::len word) =
    word_of_int (signed_take_bit (LENGTH('a) - 1) (numeral k) >> numeral
n)›
  using signed_drop_bit_word_numeral [of n k] by (simp add: sshiftr_def
shiftr_def)

lemma sshiftr_div_2n: "sint (w >>> n) = sint w div 2 ^ n"
  by word_eqI (cases ‹n < LENGTH('a)›; fastforce simp: le_diff_conv2)

lemma mask_eq:
  ‹mask n = (1 << n) - (1 :: 'a::len word)›
  by (simp add: mask_eq_exp_minus_1 shiftl_def)

lemma nth_shiftl': "bit (w << m) n ⟷ n < size w ∧ n >= m ∧ bit w (n
- m)"
  for w :: "'a::len word"
  by (simp add: bit_simps word_size ac_simps)

lemmas nth_shiftl = nth_shiftl' [unfolded word_size]

lemma nth_shiftr: "bit (w >> m) n = bit w (n + m)"
  for w :: "'a::len word"
  by (simp add: bit_simps ac_simps)

lemma shiftr_div_2n: "uint (shiftr w n) = uint w div 2 ^ n"
  by (fact uint_shiftr_eq)

lemma shiftl_rev: "shiftl w n = word_reverse (shiftr (word_reverse w)
n)"
  by word_eqI_solve

lemma rev_shiftl: "word_reverse w << n = word_reverse (w >> n)"
  by (simp add: shiftl_rev)

lemma shiftr_rev: "w >> n = word_reverse (word_reverse w << n)"
  by (simp add: rev_shiftl)

lemma rev_shiftr: "word_reverse w >> n = word_reverse (w << n)"
  by (simp add: shiftr_rev)
```

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

lemma shiftl_zero_size: "size x ≤ n ⟹ x << n = 0"
  for x :: "'a::len word"
  by (simp add: shiftl_def word_size)

lemma shiftl_t2n: "shiftl w n = 2 ^ n * w"
  for w :: "'a::len word"
  by (simp add: shiftl_def push_bit_eq_mult)

lemma word_shift_by_2:
  "x * 4 = (x::'a::len word) << 2"
  by (simp add: shiftl_t2n)

lemma word_shift_by_3:
  "x * 8 = (x::'a::len word) << 3"
  by (simp add: shiftl_t2n)

lemma slice_shiftr: "slice n w = ucast (w >> n)"
  by word_eqI (cases ‹n ≤ LENGTH('b)›; fastforce simp: ac_simps dest:
bit_imp_le_length)

lemma shiftr_zero_size: "size x ≤ n ⟹ x >> n = 0"
  for x :: "'a :: len word"
  by word_eqI

lemma shiftr_x_0 [simp]: "x >> 0 = x"
  for x :: "'a::len word"
  by (simp add: shiftr_def)

lemma shiftl_x_0 [simp]: "x << 0 = x"
  for x :: "'a::len word"
  by (simp add: shiftl_def)

lemmas shiftl0 = shiftl_x_0

lemma shiftr_1 [simp]: "(1::'a::len word) >> n = (if n = 0 then 1 else
0)"
  by (simp add: shiftr_def)

lemma and_not_mask:
  "w AND NOT (mask n) = (w >> n) << n"
  for w :: ‹'a::len word›
  by word_eqI_solve
```

```
lemma and_mask:
  "w AND mask n = (w << (size w - n)) >> (size w - n)"
  for w :: ‹'a::len word›
  by word_eqI_solve

lemma shiftr_div_2n_w: "w >> n = w div (2^n :: 'a :: len word)"
  by (fact shiftr_eq_div)

lemma le_shiftr:
  "u ≤ v ⟹ u >> (n :: nat) ≤ (v :: 'a :: len word) >> n"
  unfolding shiftr_def
  apply transfer
  apply (simp add: take_bit_drop_bit)
  apply (simp add: drop_bit_eq_div zdiv_mono1)
  done

lemma le_shiftr':
  "⟦ u >> n ≤ v >> n ; u >> n ≠ v >> n ⟧ ⟹ (u::'a::len word) ≤ v"
  by (metis le_cases le_shiftr verit_la_disequality)

lemma shiftr_mask_le:
  "n ≤ m ⟹ mask n >> m = (0 :: 'a::len word)"
  by word_eqI

lemma shiftr_mask [simp]:
  ‹mask m >> m = (0::'a::len word)›
  by (rule shiftr_mask_le) simp

lemma le_mask_iff:
  "(w ≤ mask n) = (w >> n = 0)"
  for w :: ‹'a::len word›
  by (simp add: less_eq_mask_iff_take_bit_eq_self shiftr_def take_bit_eq_self_iff_drop_bit_

lemma and_mask_eq_iff_shiftr_0:
  "(w AND mask n = w) = (w >> n = 0)"
  for w :: ‹'a::len word›
  by (simp flip: take_bit_eq_mask add: shiftr_def take_bit_eq_self_iff_drop_bit_eq_0)

lemma mask_shiftl_decompose:
  "mask m << n = mask (m + n) AND NOT (mask n :: 'a::len word)"
  by word_eqI_solve

lemma shiftl_over_and_dist:
  fixes a::"'a::len word"
  shows "(a AND b) << c = (a << c) AND (b << c)"
  by (unfold shiftl_def) (fact push_bit_and)

lemma shiftr_over_and_dist:
  fixes a::"'a::len word"
```

```
  shows "a AND b >> c = (a >> c) AND (b >> c)"
  by (unfold shiftr_def) (fact drop_bit_and)

lemma sshiftr_over_and_dist:
  fixes a::"'a::len word"
  shows "a AND b >>> c = (a >>> c) AND (b >>> c)"
  by word_eqI

lemma shiftl_over_or_dist:
  fixes a::"'a::len word"
  shows "a OR b << c = (a << c) OR (b << c)"
  by (unfold shiftl_def) (fact push_bit_or)

lemma shiftr_over_or_dist:
  fixes a::"'a::len word"
  shows "a OR b >> c = (a >> c) OR (b >> c)"
  by (unfold shiftr_def) (fact drop_bit_or)

lemma sshiftr_over_or_dist:
  fixes a::"'a::len word"
  shows "a OR b >>> c = (a >>> c) OR (b >>> c)"
  by word_eqI

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)"
  by (word_eqI_solve simp: add.commute add.left_commute)

lemma shiftr_shiftr:
  fixes a::"'a::len word"
  shows "a >> b >> c = a >> (b + c)"
  by word_eqI (simp add: add.left_commute add.commute)

lemma shiftl_shiftr1:
  fixes a::"'a::len word"
  shows "c ≤ b ⟹ a << b >> c = a AND (mask (size a - b)) << (b - c)"
  by word_eqI (auto simp: ac_simps)

lemma shiftl_shiftr2:
  fixes a::"'a::len word"
  shows "b < c ⟹ a << b >> c = (a >> (c - b)) AND (mask (size a - c))"
  by word_eqI_solve

lemma shiftr_shiftl1:
```

```
  fixes a::"'a::len word"
  shows "c ≤ b ⟹ a >> b << c = (a >> (b - c)) AND (NOT (mask c))"
  by word_eqI_solve

lemma shiftr_shiftl2:
  fixes a::"'a::len word"
  shows "b < c ⟹ a >> b << c = (a << (c - b)) AND (NOT (mask c))"
  by word_eqI (auto simp: ac_simps)

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)"
  by word_eqI_solve

lemma word_shiftl_add_distrib:
  fixes x :: "'a :: len word"
  shows "(x + y) << n = (x << n) + (y << n)"
  by (simp add: shiftl_t2n ring_distribs)

lemma mask_shift:
  "(x AND NOT (mask y)) >> y = x >> y"
  for x :: ‹'a::len word›
  by word_eqI

lemma shiftr_div_2n':
  "unat (w >> n) = unat w div 2 ^ n"
  by word_eqI

lemma shiftl_shiftr_id:
  "⟦ n < LENGTH('a); x < 2 ^ (LENGTH('a) - n) ⟧ ⟹ x << n >> n = (x::'a::len
word)"
  by word_eqI (metis add.commute less_diff_conv)

lemma ucast_shiftl_eq_0:
  fixes w :: "'a :: len word"
  shows "⟦ n ≥ LENGTH('b) ⟧ ⟹ ucast (w << n) = (0 :: 'b :: len word)"
  by (transfer fixing: n) (simp add: take_bit_push_bit)

lemma word_shift_nonzero:
  fixes x:: "'a::len word"
  assumes "x ≤ 2 ^ m"
      and mn: "m + n < LENGTH('a::len)"
      and "x ≠ 0"
    shows "x << n ≠ 0"
proof -
```

```
  have "0 < unat x"
    by (simp add: assms unat_gt_0)
  moreover
  have "unat x ≤ 2 ^ m"
    by (simp add: assms word_unat_less_le)
  then have §: "2 ^ n * unat x < 2 ^ LENGTH('a)"
    by (metis add_diff_cancel_right' mn le_add2 nat_le_power_trans order_le_less_trans
unat_lt2p unat_power_lower)
  ultimately have "0 < 2 ^ n * unat x mod 2 ^ LENGTH('a)"
    by simp
  with § show ?thesis
    by (metis add.commute add_lessD1 less_zeroE mn mult.commute shiftl_eq_mult
unat_0 unat_power_lower unat_word_ariths(2))
qed

lemma word_shiftr_lt:
  fixes w :: "'a::len word"
  shows "unat (w >> n) < (2 ^ (LENGTH('a) - n))"
  by (metis mult.commute add_lessD1 div_mod_decomp nat_power_less_diff
shiftr_div_2n' unat_lt2p)

lemma shiftr_less_t2n':
  "⟦ x AND mask (n + m) = x; m < LENGTH('a) ⟧ ⟹ x >> n < 2 ^ m" for x
:: "'a :: len word"
  by (metis and_mask_eq_iff_shiftr_0 eq_mask_less shiftr_shiftr)

lemma shiftr_less_t2n:
  "x < 2 ^ (n + m) ⟹ x >> n < 2 ^ m" for x :: "'a :: len word"
  by (meson le_add2 less_mask_eq order_le_less_trans p2_gt_0 shiftr_less_t2n'
word_gt_a_gt_0)

lemma shiftr_eq_0: "n ≥ LENGTH('a) ⟹ ((w::'a::len word) >> n) = 0"
  by (metis drop_bit_word_beyond shiftr_def)

lemma shiftl_less_t2n:
  fixes x :: "'a :: len word"
  shows "⟦ x < (2 ^ (m - n)); m < LENGTH('a) ⟧ ⟹ (x << n) < 2 ^ m"
  by (simp add: shiftl_def take_bit_push_bit word_size flip: mask_eq_iff_w2p
take_bit_eq_mask)

lemma shiftl_less_t2n':
  "(x::'a::len word) < 2 ^ m ⟹ m+n < LENGTH('a) ⟹ x << n < 2 ^ (m
+ n)"
  by (simp add: shiftl_less_t2n)

lemma scast_bit_test [simp]:
  "scast ((1 :: 'a::len signed word) << n) = (1 :: 'a word) << n"
  by word_eqI
```

```
lemma signed_shift_guard_to_word:
  ‹unat x * 2 ^ y < 2 ^ n ⟷ x = 0 ∨ x < 1 << n >> y›
    if ‹n < LENGTH('a)› ‹0 < n›
    for x :: ‹'a::len word›
proof (cases ‹x = 0›)
  case True
  then show ?thesis
    by simp
next
  case False
  then have ‹unat x ≠ 0›
    by (simp add: unat_eq_0)
  then have ‹unat x ≥ 1›
    by simp
  show ?thesis
  proof (cases ‹y < n›)
    case False
    then have ‹n ≤ y›
      by simp
    then obtain q where ‹y = n + q›
      using le_Suc_ex by blast
    moreover have ‹(2 :: nat) ^ n >> n + q ≤ 1›
      by (simp add: drop_bit_eq_div power_add shiftr_def)
    ultimately show ?thesis
      using ‹x ≠ 0› ‹unat x ≥ 1› ‹n < LENGTH('a)›
      by (simp add: power_add not_less word_le_nat_alt unat_drop_bit_eq
shiftr_def shiftl_def)
  next
    case True
    with that have ‹y < LENGTH('a)›
      by simp
    show ?thesis
    proof (cases ‹2 ^ n = unat x * 2 ^ y›)
      case True
      moreover have ‹unat x * 2 ^ y < 2 ^ LENGTH('a)›
        using ‹n < LENGTH('a)› by (simp flip: True)
      moreover have ‹(word_of_nat (2 ^ n) :: 'a word) = word_of_nat (unat
x * 2 ^ y)›
        using True by simp
      then have ‹2 ^ n = x * 2 ^ y›
        by simp
      ultimately show ?thesis
        using ‹y < LENGTH('a)›
        by (auto simp: drop_bit_eq_div word_less_nat_alt unat_div unat_word_ariths
          shiftr_def shiftl_def)
    next
      case False
      with ‹y < n› have *: ‹unat x ≠ 2 ^ n div 2 ^ y›
        by (auto simp flip: power_sub power_add)
```

```
      have ‹unat x * 2 ^ y < 2 ^ n ⟷ unat x * 2 ^ y ≤ 2 ^ n›
        using False by (simp add: less_le)
      also have ‹... ⟷ unat x ≤ 2 ^ n div 2 ^ y›
        by (simp add: less_eq_div_iff_mult_less_eq)
      also have ‹... ⟷ unat x < 2 ^ n div 2 ^ y›
        using * by (simp add: less_le)
      finally show ?thesis
      using that ‹x ≠ 0› by (simp flip: push_bit_eq_mult drop_bit_eq_div
        add: shiftr_def shiftl_def unat_drop_bit_eq word_less_iff_unsigned
[where ?'a = nat])
    qed
  qed
qed

lemma shiftr_not_mask_0:
  "n+m ≥ LENGTH('a :: len) ⟹ ((w::'a::len word) >> n) AND NOT (mask
m) = 0"
  by word_eqI

lemma shiftl_mask_is_0[simp]:
  "(x << n) AND mask n = 0"
  for x :: ‹'a::len word›
  by (simp flip: take_bit_eq_mask add: take_bit_push_bit shiftl_def)

lemma rshift_sub_mask_eq:
  "(a >> (size a - b)) AND mask b = a >> (size a - b)"
  for a :: ‹'a::len word›
  by (simp add: and_mask_eq_iff_shiftr_0 shiftr_shiftr shiftr_zero_size)

lemma shiftl_shiftr3:
  "b ≤ c ⟹ a << b >> c = (a >> c - b) AND mask (size a - c)"
  for a :: ‹'a::len word›
  by (cases "b = c") (simp_all add: shiftl_shiftr1 shiftl_shiftr2)

lemma and_mask_shiftr_comm:
  "m ≤ size w ⟹ (w AND mask m) >> n = (w >> n) AND mask (m-n)"
  for w :: ‹'a::len word›
  by (simp add: and_mask shiftr_shiftr) (simp add: word_size shiftl_shiftr3)

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›
  by (simp add: and_mask word_size shiftl_shiftl) (simp add: shiftl_shiftr1)

lemma le_mask_shiftl_le_mask: "s = m + n ⟹ x ≤ mask n ⟹ x << m ≤
mask s"
  for x :: ‹'a::len word›
  by (simp add: le_mask_iff shiftl_shiftr3)
```

```
lemma word_and_1_shiftl:
  "x AND (1 << n) = (if bit x n then (1 << n) else 0)" for x :: "'a ::
len word"
  by word_eqI_solve

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›
  by word_eqI_solve

lemma shift_times_fold:
  "(x :: 'a :: len word) * (2 ^ n) << m = x << (m + n)"
  by (simp add: shiftl_t2n ac_simps power_add)

lemma of_bool_nth:
  "of_bool (bit x v) = (x >> v) AND 1"
  for x :: ‹'a::len word›
  by (simp add: bit_iff_odd_drop_bit word_and_1 shiftr_def)

lemma shiftr_mask_eq:
  "(x >> n) AND mask (size x - n) = x >> n" for x :: "'a :: len word"
  by (word_eqI_solve dest: test_bit_lenD)

lemma shiftr_mask_eq':
  "m = (size x - n) ⟹ (x >> n) AND mask m = x >> n" for x :: "'a ::
len word"
  by (simp add: shiftr_mask_eq)

lemma and_eq_0_is_nth:
  fixes x :: "'a :: len word"
  shows "y = 1 << n ⟹ ((x AND y) = 0) = (¬ (bit x n))"
  by (simp add: and_exp_eq_0_iff_not_bit shiftl_def)

lemma word_shift_zero:
  "⟦ x << n = 0; x ≤ 2^m; m + n < LENGTH('a)⟧ ⟹ (x::'a::len word) =
0"
  using word_shift_nonzero by blast

lemma mask_shift_and_negate[simp]:"(w AND mask n << m) AND NOT (mask
n << m) = 0"
  for w :: ‹'a::len word›
  by word_eqI
```

```
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›
  by word_eqI_solve

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›
  by word_eqI_solve

lemma shiftr1_unfold: "x div 2 = x >> 1"
  by (simp add: drop_bit_eq_div shiftr_def)

lemma shiftr1_is_div_2: "(x::('a::len) word) >> 1 = x div 2"
  by (simp add: drop_bit_eq_div shiftr_def)

lemma shiftl1_is_mult: "(x << 1) = (x :: 'a::len word) * 2"
  by (metis One_nat_def mult_2 mult_2_right one_add_one
        power_0 power_Suc shiftl_t2n)

lemma shiftr1_lt:"x ≠ 0 ⟹ (x::('a::len) word) >> 1 < x"
  by (simp add: div_less_dividend_word drop_bit_eq_div shiftr_def)

lemma shiftr1_0_or_1:"(x::('a::len) word) >> 1 = 0 ⟹ x = 0 ∨ x = 1"
  by (metis le_mask_iff mask_1 not_less not_less_iff_gr_or_eq word_less_1)

lemma shiftr1_irrelevant_lsb: "bit (x::('a::len) word) 0 ∨ x >> 1 =
(x + 1) >> 1"
  by (auto simp: bit_0 shiftr_def drop_bit_Suc ac_simps elim: evenE)

lemma shiftr1_0_imp_only_lsb:"((x::('a::len) word) + 1) >> 1 = 0 ⟹
x = 0 ∨ x + 1 = 0"
  by (metis One_nat_def shiftr1_0_or_1 word_less_1 word_overflow)

lemma shiftr1_irrelevant_lsb': "¬ (bit (x::('a::len) word) 0) ⟹ x
>> 1 = (x + 1) >> 1"
  using shiftr1_irrelevant_lsb [of x] by simp


lemma cast_chunk_assemble_id:
  assumes n: "n = LENGTH('a::len)"
    and "m = LENGTH('b::len)"
    and "n * 2 = m"
  shows "UCAST('a → 'b) (UCAST('b → 'a) x::'a word) OR (UCAST('a →
'b) (UCAST('b → 'a) (x >> n)::'a word) << n) = x"
```

```
proof -
  have "((ucast ((ucast (x >> n))::'a word))::'b word) = x >> n"
    by (metis and_mask_eq_iff_shiftr_0 assms mult_2_right order_refl shiftr_eq_0
shiftr_shiftr ucast_ucast_mask)
  with n show ?thesis
    by (auto simp: ucast_ucast_mask simp flip: and_not_mask word_ao_dist2)
qed

lemma cast_chunk_scast_assemble_id:
  fixes x:: "'b::len word"
  assumes "n = LENGTH('a::len)"
      and "m = LENGTH('b)"
      and "n * 2 = m"
    shows "UCAST('a → 'b) (SCAST('b → 'a) x) OR (UCAST('a → 'b) (SCAST('b
→ 'a) (x >> n)) << n) = x"
proof -
  have "SCAST('b → 'a) x = UCAST('b → 'a) x"
    by (metis assms down_cast_same is_up is_up_down le_add2 mult_2_right)
  then show ?thesis
    by (metis assms cast_chunk_assemble_id down_cast_same is_down le_add2
mult_2_right)
qed

lemma unat_shiftr_less_t2n:
  fixes x :: "'a :: len word"
  shows "unat x < 2 ^ (n + m) ⟹ unat (x >> n) < 2 ^ m"
  by (simp add: shiftr_div_2n' power_add mult.commute less_mult_imp_div_less)

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)"
  by (meson add_lessD1 order_less_le_trans shiftl_less_t2n' ucast_less)


lemma NOT_mask_shifted_lenword:
  "NOT (mask len << (LENGTH('a) - len) ::'a::len word) = mask (LENGTH('a)
- len)"
  by word_eqI_solve



lemma shiftr_less:
  "(w::'a::len word) < k ⟹ w >> n < k"
  by (metis div_le_dividend le_less_trans shiftr_div_2n' unat_arith_simps(2))

lemma word_and_notzeroD:
  "w AND w' ≠ 0 ⟹ w ≠ 0 ∧ w' ≠ 0"
  by auto
```

```
lemma shiftr_le_0:
  "unat (w::'a::len word) < 2 ^ n ⟹ w >> n = (0::'a::len word)"
  by (auto simp: take_bit_word_eq_self_iff word_less_nat_alt shiftr_def
    simp flip: take_bit_eq_self_iff_drop_bit_eq_0 intro: ccontr)

lemma of_nat_shiftl:
  "(of_nat x << n) = (of_nat (x * 2 ^ n) :: ('a::len) word)"
proof -
  have "(of_nat x::'a word) << n = of_nat (2 ^ n) * of_nat x"
    using shiftl_t2n by (metis word_unat_power)
  thus ?thesis by simp
qed

lemma shiftl_1_not_0:
  "n < LENGTH('a) ⟹ (1::'a::len word) << n ≠ 0"
  by (simp add: shiftl_t2n)




lemma bitmagic_zeroLast_leq_or1Last:
  "(a::('a::len) word) AND (mask len << x - len) ≤ a OR mask (y - len)"
  by (meson le_word_or2 order_trans word_and_le2)

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
  have helper3: "x OR y = x OR y AND NOT x" for x y ::"'a::len word" by
(simp add: word_oa_dist2)
  from assms show "base = NOT (mask (LENGTH('a) - len)) AND a ⟹
                   a ∈ {base..base OR mask (LENGTH('a) - len)}"
    by (metis and.commute atLeastAtMost_iff helper3 le_word_or2 or.commute
word_and_le1)
next
  assume "a ∈ {base..base OR mask (LENGTH('a) - len)}"
  hence a: "base ≤ a ∧ a ≤ base OR mask (LENGTH('a) - len)" by simp
  show "base = NOT (mask (LENGTH('a) - len)) AND a"
  proof -
    have f2: "∀x₀. base AND NOT (mask x₀) ≤ a AND NOT (mask x₀)"
      using a neg_mask_mono_le by blast
    have f3: "∀x₀. a AND NOT (mask x₀) ≤ (base OR mask (LENGTH('a) -
len)) AND NOT (mask x₀)"
      using a neg_mask_mono_le by blast
    have f4: "base = base AND NOT (mask (LENGTH('a) - len))"
      using valid_prefix by (metis mask_eq_0_eq_x word_bw_comms(1))
    hence f5: "∀x₆. (base OR x₆) AND NOT (mask (LENGTH('a) - len)) =
```

```
                    base OR x6 AND NOT (mask (LENGTH('a) - len))"
      using word_ao_dist by (metis)
    have f6: "∀x2 x3. a AND NOT (mask x2) ≤ x3 ∨
                    ¬ (base OR mask (LENGTH('a) - len)) AND NOT (mask
x2) ≤ x3"
      using f3 dual_order.trans by auto
    have "base = (base OR mask (LENGTH('a) - len)) AND NOT (mask (LENGTH('a)
- len))"
      using f5 by auto
    hence "base = a AND NOT (mask (LENGTH('a) - len))"
      using f2 f4 f6 by (metis eq_iff)
    thus "base = NOT (mask (LENGTH('a) - len)) AND a"
      by (metis word_bw_comms(1))
  qed
qed

lemma of_nat_eq_signed_scast:
  "(of_nat x = (y :: ('a::len) signed word))
   = (of_nat x = (scast y :: 'a word))"
  by (metis scast_of_nat scast_scast_id(2))

lemma word_aligned_add_no_wrap_bounded:
  "⟦ w + 2^n ≤ x; w + 2^n ≠ 0; is_aligned w n ⟧ ⟹ (w::'a::len word)
< x"
  by (blast dest: is_aligned_no_overflow le_less_trans word_leq_le_minus_one)

lemma mask_Suc:
  "mask (Suc n) = (2 :: 'a::len word) ^ n + mask n"
  by (simp add: mask_eq_decr_exp)

lemma mask_mono:
  "sz' ≤ sz ⟹ mask sz' ≤ (mask sz :: 'a::len word)"
  by (simp add: le_mask_iff shiftr_mask_le)

lemma aligned_mask_disjoint:
  "⟦ is_aligned (a :: 'a :: len word) n; b ≤ mask n ⟧ ⟹ a AND b = 0"
  by (metis and_zero_eq is_aligned_mask le_mask_imp_and_mask word_bw_lcs(1))

lemma word_and_or_mask_aligned:
  "⟦ is_aligned a n; b ≤ mask n ⟧ ⟹ a + b = a OR b"
  by (simp add: aligned_mask_disjoint word_plus_and_or_coroll)

lemma word_and_or_mask_aligned2:
  ‹is_aligned b n ⟹ a ≤ mask n ⟹ a + b = a OR b›
  using word_and_or_mask_aligned [of b n a] by (simp add: ac_simps)

lemma is_aligned_ucastI:
  "is_aligned w n ⟹ is_aligned (ucast w) n"
  by (simp add: bit_ucast_iff is_aligned_nth)
```

```
lemma ucast_le_maskI:
  "a ≤ mask n ⟹ UCAST('a::len → 'b::len) a ≤ mask n"
  by (metis and_mask_eq_iff_le_mask ucast_and_mask)

lemma ucast_add_mask_aligned:
  "⟦ a ≤ mask n; is_aligned b n ⟧ ⟹ UCAST ('a::len → 'b::len) (a +
b) = ucast a + ucast b"
  by (metis add.commute is_aligned_ucastI ucast_le_maskI ucast_or_distrib
word_and_or_mask_aligned)

lemma ucast_shiftl:
  "LENGTH('b) ≤ LENGTH ('a) ⟹ UCAST ('a::len → 'b::len) x << n = ucast
(x << n)"
  by word_eqI_solve

lemma ucast_leq_mask:
  "LENGTH('a) ≤ n ⟹ ucast (x::'a::len word) ≤ mask n"
  by (metis and_mask_eq_iff_le_mask ucast_and_mask ucast_id ucast_mask_drop)

lemma shiftl_inj:
  ‹x = y›
    if ‹x << n = y << n› ‹x ≤ mask (LENGTH('a) - n)› ‹y ≤ mask (LENGTH('a)
- n)›
    for x y :: ‹'a::len word›
proof (cases ‹n < LENGTH('a)›)
  case False
  with that show ?thesis
    by simp
next
  case True
  moreover from that have ‹take_bit (LENGTH('a) - n) x = x› ‹take_bit
(LENGTH('a) - n) y = y›
    by (simp_all add: less_eq_mask_iff_take_bit_eq_self)
  ultimately show ?thesis
    using ‹x << n = y << n› by (metis diff_less gr_implies_not0 linorder_cases
linorder_not_le shiftl_shiftr_id shiftl_x_0 take_bit_word_eq_self_iff)
qed

lemma distinct_word_add_ucast_shift_inj:
  ‹p' = p ∧ off' = off›
  if *: ‹p + (UCAST('a::len → 'b::len) off << n) = p' + (ucast off' <<
n)›
    and ‹is_aligned p n'› ‹is_aligned p' n'› ‹n' = n + LENGTH('a)› ‹n'
< LENGTH('b)›
proof -
  from ‹n' = n + LENGTH('a)›
  have [simp]: ‹n' - n = LENGTH('a)› ‹n + LENGTH('a) = n'›
    by simp_all
```

```
  from ‹is_aligned p n'› obtain q
    where p: ‹p = push_bit n' (word_of_nat q)› ‹q < 2 ^ (LENGTH('b)
- n')›
    by (rule is_alignedE')
  from ‹is_aligned p' n'› obtain q'
    where p': ‹p' = push_bit n' (word_of_nat q')› ‹q' < 2 ^ (LENGTH('b)
- n')›
    by (rule is_alignedE')
  define m :: nat where ‹m = unat off›
  then have off: ‹off = word_of_nat m›
    by simp
  define m' :: nat where ‹m' = unat off'›
  then have off': ‹off' = word_of_nat m'›
    by simp
  have ‹push_bit n' q + take_bit n' (push_bit n m) < 2 ^ LENGTH('b)›
    by (metis id_apply is_aligned_no_wrap''' of_nat_eq_id of_nat_push_bit
p(1) p(2) take_bit_nat_eq_self_iff take_bit_nat_less_exp take_bit_push_bit
that(2) that(5) unsigned_of_nat)
  moreover have ‹push_bit n' q' + take_bit n' (push_bit n m') < 2 ^ LENGTH('b)›
    by (metis ‹n' - n = LENGTH('a)› id_apply is_aligned_no_wrap''' m'_def
of_nat_eq_id of_nat_push_bit off' p'(1) p'(2) take_bit_nat_eq_self_iff
take_bit_push_bit that(3) that(5) unsigned_of_nat)
  ultimately have ‹push_bit n' q + take_bit n' (push_bit n m) = push_bit
n' q' + take_bit n' (push_bit n m')›
    using * by (simp add: p p' off off' push_bit_of_nat push_bit_take_bit
word_of_nat_inj unsigned_of_nat shiftl_def flip: of_nat_add)
  then have ‹int (push_bit n' q + take_bit n' (push_bit n m))
    = int (push_bit n' q' + take_bit n' (push_bit n m'))›
    by simp
  then have ‹concat_bit n' (int (push_bit n m)) (int q)
    = concat_bit n' (int (push_bit n m')) (int q')›
    by (simp add: of_nat_push_bit of_nat_take_bit concat_bit_eq)
  then show ?thesis
    by (simp add: p p' off off' take_bit_of_nat take_bit_push_bit word_of_nat_eq_iff
concat_bit_eq_iff)
      (simp add: push_bit_eq_mult)
qed

lemma word_upto_Nil:
  "y < x ⟹ [x .e. y ::'a::len word] = []"
  by (simp add: upto_enum_red not_le word_less_nat_alt)

lemma word_enum_decomp_elem:
  assumes "[x .e. (y ::'a::len word)] = as @ a # bs"
  shows "x ≤ a ∧ a ≤ y"
proof -
  have "set as ⊆ set [x .e. y] ∧ a ∈ set [x .e. y]"
    using assms by (auto dest: arg_cong[where f=set])
  then show ?thesis by auto
```

```
qed

lemma word_enum_prefix:
  "[x .e. (y ::'a::len word)] = as @ a # bs ⟹ as = (if x < a then [x
.e. a - 1] else [])"
proof (induction as arbitrary: x)
  case Nil
  show ?case
  proof (cases "x < y")
    case True
    then show ?thesis
      using local.Nil word_upto_Cons_eq by force
  next
    case False
    then show ?thesis
      using local.Nil not_less_iff_gr_or_eq word_upto_Nil by fastforce
  qed
next
  case (Cons b as)
  show ?case
  proof (cases x y rule: linorder_cases)
    case less
    with Cons.prems have "b + 1 ≤ a"
      using word_enum_decomp_elem word_upto_Cons_eq by fastforce
    moreover have "x + 1 ≤ a"
      using Cons.prems less word_enum_decomp_elem word_upto_Cons_eq by
fastforce
    moreover have "(x + 1 ≤ a) = (x < a)"
      by (metis less word_Suc_le word_not_simps(3))
    ultimately
    show ?thesis
      using Cons less word_1_diffs(2) less_is_non_zero_p1 olen_add_eqv
unat_plus_simple
        word_overflow_unat word_upto_Cons_eq by fastforce
  next
    case equal
    then show ?thesis
      using Cons.prems by auto
  next
    case greater
    then show ?thesis
      using Cons
      by (simp add: word_upto_Nil)
  qed
qed

lemma word_enum_decomp_set:
  "[x .e. (y ::'a::len word)] = as @ a # bs ⟹ a ∉ set as"
  by (metis distinct_append distinct_enum_upto' not_distinct_conv_prefix)
```

```
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 -
  from assms
  have "set as ⊆ set [x .e. y] ∧ a ∈ set [x .e. y]"
    by (auto dest: arg_cong[where f=set])
  with word_enum_decomp_set[OF assms]
  show ?thesis by auto
qed

lemma of_nat_unat_le_mask_ucast:
  "⟦of_nat (unat t) = w; t ≤ mask LENGTH('a)⟧ ⟹ t = UCAST('a::len →
'b::len) w"
  by (clarsimp simp: ucast_nat_def ucast_ucast_mask simp flip: and_mask_eq_iff_le_mask)

lemma less_diff_gt0:
  "a < b ⟹ (0 :: 'a :: len word) < b - a"
  by unat_arith

lemma unat_plus_gt:
  "unat ((a :: 'a :: len word) + b) ≤ unat a + unat b"
  by (clarsimp simp: unat_plus_if_size)

lemma const_less:
  "⟦ (a :: 'a :: len word) - 1 < b; a ≠ b ⟧ ⟹ a < b"
  by (metis less_1_simp word_le_less_eq)

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›
  by (metis (no_types, opaque_lifting)
            add.assoc add.commute add.right_neutral add_uminus_conv_diff
            mask_eq_decr_exp mask_eqs(2) mask_eqs(6) mult.commute mult_zero_left
            subtract_mask(1) that)

lemma unat_of_nat_minus_1:
  "⟦ n < 2 ^ LENGTH('a); n ≠ 0 ⟧ ⟹ unat ((of_nat n:: 'a :: len word)
- 1) = n - 1"
  by (simp add: of_nat_diff unat_eq_of_nat)

lemma word_eq_zeroI:
  "a ≤ a - 1 ⟹ a = 0" for a :: "'a :: len word"
  by (simp add: word_must_wrap)

lemma word_add_format:
  "(-1 :: 'a :: len  word) + b + c = b + (c - 1)"
```

```
  by simp

lemma upto_enum_word_nth:
  assumes "i ≤ j" and "k ≤ unat (j - i)"
  shows "[i .e. j] ! k = i + of_nat k"
proof -
  have "unat i + unat (j-i) < 2 ^ LENGTH('a)"
    by (metis add.commute ‹i ≤ j› eq_diff_eq no_olen_add_nat)
  then have "toEnum (unat i + k) = i + word_of_nat k"
    using assms by auto
  moreover have "[j] ! (k + unat i - unat j) = i + word_of_nat k"
    if "¬ k < unat j - unat i"  "unat i ≤ unat j"
    using that assms unat_sub by fastforce
  moreover have "[] ! k = i + word_of_nat k" if "¬ unat i ≤ unat j"
    using that ‹i ≤ j› word_less_eq_iff_unsigned by blast
  ultimately show ?thesis
    by (auto simp: upto_enum_def nth_append)
qed

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)"
  by (clarsimp simp: upto_enum_step_def not_less[symmetric] upto_enum_word_nth)

lemma upto_enum_inc_1_len:
  fixes a :: "'a::len word"
  assumes "a < - 1"
  shows "[(0 :: 'a :: len word) .e. 1 + a] = [0 .e. a] @ [1 + a]"
proof -
  have "unat (1+a) = 1 + unat a"
    by (simp add: add_eq_0_iff assms unatSuc word_order.not_eq_extremum)
  with assms show ?thesis
    by (simp add: upto_enum_word)
qed

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›
  by (clarsimp simp: mask_out_sub_mask mask_eqs(7)[symmetric] mask_twice)

lemma shiftr_shiftl_shiftr[simp]:
  "(x :: 'a :: len word) >> a << a >> a = x >> a"
  by (word_eqI_solve dest: bit_imp_le_length)

lemma add_right_shift:
  fixes x y :: ‹'a::len word›
  assumes "x AND mask n = 0" and "y AND mask n = 0" and "x ≤ x + y"
  shows "(x + y) >> n = (x >> n) + (y >> n)"
```

```
proof -
  obtain §: "is_aligned x n" "is_aligned y n" "unat x + unat y < 2 ^ LENGTH('a)"
    using assms is_aligned_mask no_olen_add_nat by blast
  then have "unat x div 2 ^ n + unat y div 2 ^ n < 2 ^ LENGTH('a)"
    by (metis add_le_mono add_lessD1 div_le_dividend le_Suc_ex)
  with § show ?thesis
    by (metis (no_types, lifting) div_add is_aligned_iff_dvd_nat shiftr_div_2n'
unat_plus_if' word_unat_eq_iff)
qed

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)"
  by (smt (verit) add_diff_cancel_left' add_right_shift diff_0_right diff_add_cancel
mask_eqs(4))

lemma and_and_mask_simple:
  "y AND mask n = mask n ⟹ (x AND y) AND mask n = x AND mask n"
  by (simp add: ac_simps)

lemma and_and_mask_simple_not:
  "y AND mask n = 0 ⟹ (x AND y) AND mask n = 0"
  by (simp add: ac_simps)

lemma word_and_le':
  "b ≤ c ⟹ (a :: 'a :: len word) AND b ≤ c"
  by (metis word_and_le1 order_trans)

lemma word_and_less':
  "b < c ⟹ (a :: 'a :: len word) AND b < c"
  by transfer simp

lemma shiftr_w2p:
  "x < LENGTH('a) ⟹ 2 ^ x = (2 ^ (LENGTH('a) - 1) >> (LENGTH('a) - 1
- x) :: 'a :: len word)"
  by word_eqI_solve

lemma t2p_shiftr:
  "⟦ b ≤ a; a < LENGTH('a) ⟧ ⟹ (2 :: 'a :: len word) ^ a >> b = 2 ^
(a - b)"
  by word_eqI_solve

lemma scast_1[simp]:
  "scast (1 :: 'a :: len signed word) = (1 :: 'a word)"
  by simp

lemma unsigned_uminus1 [simp]:
  ‹(unsigned (-1::'b::len word)::'c::len word) = mask LENGTH('b)›
  by (fact unsigned_minus_1_eq_mask)
```

```
lemma ucast_ucast_mask_eq:
  "⟦ UCAST('a::len → 'b::len) x = y; x AND mask LENGTH('b) = x ⟧ ⟹ x
= ucast y"
  by (drule sym) (simp flip: take_bit_eq_mask add: unsigned_ucast_eq)

lemma ucast_up_eq:
  "⟦ ucast x = (ucast y::'b::len word); LENGTH('a) ≤ LENGTH ('b) ⟧
   ⟹ ucast x = (ucast y::'a::len word)"
  by (simp add: word_eq_iff bit_simps)

lemma ucast_up_neq:
  "⟦ ucast x ≠ (ucast y::'b::len word); LENGTH('b) ≤ LENGTH ('a) ⟧
   ⟹ ucast x ≠ (ucast y::'a::len word)"
  by (fastforce dest: ucast_up_eq)

lemma mask_AND_less_0:
  "⟦ x AND mask n = 0; m ≤ n ⟧ ⟹ x AND mask m = 0"
  for x :: ‹'a::len word›
  by (metis mask_twice2 word_and_notzeroD)

lemma mask_len_id:
  "(x :: 'a :: len word) AND mask LENGTH('a) = x"
  by simp

lemma scast_ucast_down_same:
  "LENGTH('b) ≤ LENGTH('a) ⟹ SCAST('a → 'b) = UCAST('a::len → 'b::len)"
  by (simp add: down_cast_same is_down)

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"
  by (simp add: word_plus_and_or_coroll aligned_mask_disjoint word_or_zero)

lemma mask_eq1_nochoice:
  "⟦ LENGTH('a) > 1; (x :: 'a :: len word) AND 1 = x ⟧ ⟹ x = 0 ∨ x =
1"
  by (metis word_and_1)

lemma shiftr_and_eq_shiftl:
  "(w >> n) AND x = y ⟹ w AND (x << n) = (y << n)" for y :: "'a:: len
word"
  by (smt (verit, best) and_not_mask mask_eq_0_eq_x shiftl_mask_is_0 shiftl_over_and_dist
word_bw_lcs(1))

lemma add_mask_lower_bits':
  "⟦ len = LENGTH('a); is_aligned (x :: 'a :: len word) n;
     ∀n' ≥ n. n' < len ⟶ ¬ bit p n' ⟧
```

```
   ⟹ x + p AND NOT(mask n) = x"
  using add_mask_lower_bits by auto

lemma leq_mask_shift:
  "(x :: 'a :: len word) ≤ mask (low_bits + high_bits) ⟹ (x >> low_bits)
≤ mask high_bits"
  by (simp add: le_mask_iff shiftr_shiftr ac_simps)

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"
  by (simp add: and_mask_eq_iff_le_mask leq_mask_shift ucast_ucast_mask)

lemma const_le_unat:
  "⟦ b < 2 ^ LENGTH('a); of_nat b ≤ a ⟧ ⟹ b ≤ unat (a :: 'a :: len word)"
  by (simp add: word_le_nat_alt unsigned_of_nat take_bit_nat_eq_self)

lemma upt_enum_offset_trivial:
  "⟦ x < 2 ^ LENGTH('a) - 1 ; n ≤ unat x ⟧
   ⟹ ([(0 :: 'a :: len word) .e. x] ! n) = of_nat n"
  by (induct x arbitrary: n) (auto simp: upto_enum_word_nth)

lemma word_le_mask_out_plus_2sz:
  "x ≤ (x AND NOT(mask sz)) + 2 ^ sz - 1"
  for x :: ‹'a::len word›
  by (metis add_diff_eq word_neg_and_le)

lemma ucast_add:
  "ucast (a + (b :: 'a :: len word)) = ucast a + (ucast b :: ('a signed
word))"
  by transfer (simp add: take_bit_add)

lemma ucast_minus:
  "ucast (a - (b :: 'a :: len word)) = ucast a - (ucast b :: ('a signed
word))"
  by (metis (no_types, opaque_lifting) add_diff_cancel_left' add_diff_eq
ucast_add)

lemma scast_ucast_add_one [simp]:
  "scast (ucast (x :: 'a::len word) + (1 :: 'a signed word)) = x + 1"
  by (metis scast_ucast_id ucast_1 ucast_add)

lemma word_and_le_plus_one:
  "a > 0 ⟹ (x :: 'a :: len word) AND (a - 1) < a"
  by (simp add: gt0_iff_gem1 word_and_less')

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)"
  by (simp add: shiftr_div_2n' unat_ucast_up_simp)

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)"
  by (metis ucast_and_mask unat_ucast_up_simp)

lemma shiftr_less_t2n3:
  "⟦ (2 :: 'a word) ^ (n + m) = 0; m < LENGTH('a) ⟧
   ⟹ (x :: 'a :: len word) >> n < 2 ^ m"
  by (fastforce intro: shiftr_less_t2n' simp: mask_eq_decr_exp power_overflow)

lemma unat_shiftr_le_bound:
  "⟦ 2 ^ (LENGTH('a :: len) - n) - 1 ≤ bnd; 0 < n ⟧
   ⟹ unat ((x :: 'a word) >> n) ≤ bnd"
  by (metis add.commute le_diff_conv less_Suc_eq_le order_less_le_trans
plus_1_eq_Suc word_shiftr_lt)

lemma shiftr_eqD:
  "⟦ x >> n = y >> n; is_aligned x n; is_aligned y n ⟧
   ⟹ x = y"
  by (metis is_aligned_shiftr_shiftl)

lemma word_shiftr_shiftl_shiftr_eq_shiftr:
  "a ≥ b ⟹ (x :: 'a :: len word) >> a << b >> b = x >> a"
  by (simp add: mask_shift shiftr_shiftl1 shiftr_shiftr)

lemma of_int_uint_ucast:
   "of_int (uint (x :: 'a::len word)) = (ucast x :: 'b::len word)"
  by (fact Word.of_int_uint)

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›
  by (simp add: word_mod_2p_is_mask word_bw_assocs)

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))"
  by (metis ucast_and_mask ucast_id ucast_ucast_mask ucast_up_eq)

lemma of_nat_less_t2n:
  "of_nat i < (2 :: ('a :: len) word) ^ n ⟹ n < LENGTH('a) ∧ unat (of_nat
i :: 'a word) < 2 ^ n"
  by (metis order_less_trans p2_gt_0 unat_less_power word_neq_0_conv)
```

```
lemma two_power_increasing_less_1:
  "⟦ n ≤ m; m ≤ LENGTH('a) ⟧ ⟹ (2 :: 'a :: len word) ^ n - 1 ≤ 2 ^
m - 1"
  by (meson le_m1_iff_lt order_le_less_trans p2_gt_0 two_power_increasing
word_1_le_power word_le_minus_mono_left)

lemma word_sub_mono4:
  "⟦ y + x ≤ z + x; y ≤ y + x; z ≤ z + x ⟧ ⟹ y ≤ z" for y :: "'a ::
len word"
  by (simp add: word_add_le_iff2)

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"
  by (meson le_less_trans nat_less_le unat_less_power word_power_less_1)

lemma mask_sub:
  "n ≤ m ⟹ mask m - mask n = mask m AND NOT(mask n :: 'a::len word)"
  by (metis (full_types) and_mask_eq_iff_shiftr_0 mask_out_sub_mask shiftr_mask_le
word_bw_comms(1))

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 -
  assume lt: "sz' ≤ sz"
  hence "?lhs = ptr AND (mask sz AND NOT(mask sz'))"
    by (metis add_diff_cancel_left' multiple_mask_trivia)
  also have "... ≤ ?rhs" using lt
    by (metis (mono_tags) add_diff_eq diff_eq_eq eq_iff mask_2pm1 mask_sub
word_and_le')
  finally show ?thesis by simp
qed

lemma mask_out_eq_0:
  "⟦ idx < 2 ^ sz; sz < LENGTH('a) ⟧ ⟹ (of_nat idx :: 'a :: len word)
AND NOT(mask sz) = 0"
  by (simp add: of_nat_power less_mask_eq mask_eq_0_eq_x)

lemma is_aligned_neg_mask_eq':
  "is_aligned ptr sz = (ptr AND NOT(mask sz) = ptr)"
  using is_aligned_mask mask_eq_0_eq_x by blast

lemma neg_mask_mask_unat:
  "sz < LENGTH('a)
```

```
   ⟹ unat ((ptr :: 'a :: len word) AND NOT(mask sz)) + unat (ptr AND
mask sz) = unat ptr"
  by (metis AND_NOT_mask_plus_AND_mask_eq unat_plus_simple word_and_le2)

lemma unat_pow_le_intro:
  "LENGTH('a) ≤ n ⟹ unat (x :: 'a :: len word) < 2 ^ n"
  by (metis lt2p_lem not_le of_nat_le_iff of_nat_numeral semiring_1_class.of_nat_power
uint_nat)

lemma unat_shiftl_less_t2n:
  ‹unat (x << n) < 2 ^ m›
  if ‹unat (x :: 'a :: len word) < 2 ^ (m - n)› ‹m < LENGTH('a)›
proof (cases ‹n ≤ m›)
  case False
  with that show ?thesis
    by (simp add: unsigned_eq_0_iff)
next
  case True
  moreover define q r where ‹q = m - n› and ‹r = LENGTH('a) - n - q›
  ultimately have ‹m - n = q› ‹m = n + q› ‹LENGTH('a) = r + q + n›
    using that by simp_all
  with that show ?thesis
    by (metis le_add2 order_le_less_trans shiftl_less_t2n unat_power_lower
word_less_iff_unsigned)
qed

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"
  by (metis add_diff_cancel_left' add_mask_lower_bits is_aligned_add_helper
order_le_less_trans
       subtract_mask(2) unat_power_lower word_less_nat_alt)

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"
  by (fastforce intro: unat_is_aligned_add[where p=0 and n=c, simplified,
THEN conjunct2]
                       unat_shiftl_less_t2n unat_shiftr_less_t2n unat_pow_le_intro)

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›
  by (simp add: leq_mask_shift le_mask_imp_and_mask)
```

```
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›
  using and_mask_eq_iff_shiftr_0 by force

lemma unat_less_iff:
  "⟦ unat (a :: 'a :: len word) = b; c < 2 ^ LENGTH('a) ⟧ ⟹ (a < of_nat
c) = (b < c)"
  using unat_ucast_less_no_overflow_simp by blast

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)"
  by (meson is_aligned_no_wrap' le_m1_iff_lt not_le word_less_sub_1 word_plus_mono_right)

lemma mask_add_aligned_right:
  "is_aligned p n ⟹ (q + p) AND mask n = q AND mask n"
  by (simp add: mask_add_aligned add.commute)

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)"
  by (metis le_mask_shiftl_le_mask)

lemma word_two_power_neg_ineq:
  assumes "2 ^ m ≠ (0::'a word)"
  shows "2 ^ n ≤ - (2 ^ m :: 'a :: len word)"
proof (cases "n < LENGTH('a) ∧ m < LENGTH('a)")
  case True
  with assms show ?thesis
  by (metis bit_minus_exp_iff linorder_not_le nat_less_le nth_bounded
possible_bit_word)
next
  case False
  with assms show ?thesis
    by (force simp: power_overflow)
qed

lemma unat_shiftl_absorb:
  fixes x :: "'a :: len word"
  shows "⟦ x ≤ 2 ^ p; p + k < LENGTH('a) ⟧ ⟹ unat x * 2 ^ k = unat
(x * 2 ^ k)"
  by (smt (verit) add_diff_cancel_right' add_lessD1 le_add2 le_less_trans
mult.commute nat_le_power_trans
         unat_lt2p unat_mult_lem unat_power_lower word_le_nat_alt)

lemma word_plus_mono_right_split:
```

```
  fixes x :: "'a :: len word"
  assumes "unat (x AND mask sz) + unat z < 2 ^ sz" and "sz < LENGTH('a)"
  shows "x ≤ x + z"
proof -
  have *: "is_aligned (x AND NOT (mask sz)) sz" "word_of_nat (unat z)
= z"
          "word_of_nat (unat (x AND mask sz)) = x AND mask sz"
    by auto
  with assms have "x AND mask sz ≤ (x AND mask sz) + z"
    by (metis (mono_tags, lifting) le_unat_uoi of_nat_add order_less_imp_le
unat_plus_simple unat_power_lower)
  then have "(x AND NOT(mask sz)) + (x AND mask sz) ≤ (x AND NOT(mask
sz)) + ((x AND mask sz) + z)"
    by (metis (no_types, lifting) of_nat_power assms * is_aligned_no_wrap'
of_nat_add word_plus_mono_right)
  then show ?thesis
    by (simp add: and_not_eq_minus_and)
qed

lemma mul_not_mask_eq_neg_shiftl:
  "NOT(mask n :: 'a::len word) = -1 << n"
  by (simp add: NOT_mask shiftl_t2n)

lemma shiftr_mul_not_mask_eq_and_not_mask:
  "(x >> n) * NOT(mask n) = - (x AND NOT(mask n))"
  for x :: ‹'a::len word›
  by (metis NOT_mask and_not_mask mult_minus_left mult.commute shiftl_t2n)

lemma mask_eq_n1_shiftr:
  "n ≤ LENGTH('a) ⟹ (mask n :: 'a :: len word) = -1 >> (LENGTH('a) -
n)"
  by (metis diff_diff_cancel eq_refl mask_full shiftr_mask2)

lemma is_aligned_mask_out_add_eq:
  "is_aligned p n ⟹ (p + x) AND NOT(mask n) = p + (x AND NOT(mask n))"
  by (simp add: mask_out_add_aligned)

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"
  by (drule is_aligned_mask_out_add_eq[where x="-1"]) (simp add: NOT_mask)

lemma unat_2tp_if:
  "unat (2 ^ n :: ('a :: len) word) = (if n < LENGTH ('a) then 2 ^ n else
0)"
  by (simp add: unsigned_eq_0_iff)
```

```
lemma mask_of_mask:
  "mask (n::nat) AND mask (m::nat) = (mask (min m n) :: 'a::len word)"
  by word_eqI_solve

lemma unat_signed_ucast_less_ucast:
  "LENGTH('a) ≤ LENGTH('b) ⟹ unat (ucast (x :: 'a :: len word) :: 'b
:: len signed word) = unat x"
  by (simp add: unat_ucast_up_simp)

lemma toEnum_of_ucast:
  "LENGTH('b) ≤ LENGTH('a) ⟹
   (toEnum (unat (b::'b :: len word))::'a :: len word) = of_nat (unat
b)"
  by (simp add: unat_pow_le_intro)

lemma plus_mask_AND_NOT_mask_eq:
  "x AND NOT(mask n) = x ⟹ (x + mask n) AND NOT(mask n) = x" for x::‹'a::len
word›
  by (metis AND_NOT_mask_plus_AND_mask_eq is_aligned_neg_mask2 mask_AND_NOT_mask
mask_out_add_aligned word_and_not)

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))"
  by word_eqI_solve

lemma unat_ucast_le:
  "unat (ucast (x :: 'a :: len word) :: 'b :: len word) ≤ unat x"
  by (simp add: ucast_nat_def word_unat_less_le)

lemma ucast_le_up_down_iff:
  "⟦ LENGTH('a) ≤ LENGTH('b); (x :: 'b :: len word) ≤ ucast (- 1 :: 'a
:: len word) ⟧
   ⟹ (ucast x ≤ (y :: 'a word)) = (x ≤ ucast y)"
  using le_max_word_ucast_id ucast_le_ucast by metis

lemma ucast_ucast_mask_shift:
  "a ≤ LENGTH('a) + b
   ⟹ ucast (ucast (p AND mask a >> b) :: 'a :: len word) = p AND mask
a >> b"
  by (simp add: mask_mono ucast_ucast_eq_mask_shift word_and_le')

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)"
  by (metis linear ucast_ucast_mask_shift unat_ucast_up_simp)
```

```
lemma mask_overlap_zero:
  "a ≤ b ⟹ (p AND mask a) AND NOT(mask b) = 0"
  for p :: ‹'a::len word›
  by (metis NOT_mask_AND_mask mask_lower_twice2 max_def)

lemma mask_shifl_overlap_zero:
  "a + c ≤ b ⟹ (p AND mask a << c) AND NOT(mask b) = 0"
  for p :: ‹'a::len word›
  by (metis and_mask_0_iff_le_mask mask_mono mask_shiftl_decompose order_trans
shiftl_over_and_dist word_and_le' word_and_le2)

lemma mask_overlap_zero':
  "a ≥ b ⟹ (p AND NOT(mask a)) AND mask b = 0"
  for p :: ‹'a::len word›
  using mask_AND_NOT_mask mask_AND_less_0 by blast

lemma mask_rshift_mult_eq_rshift_lshift:
  "((a :: 'a :: len word) >> b) * (1 << c) = (a >> b << c)"
  by (simp add: shiftl_t2n)

lemma shift_alignment:
  "a ≥ b ⟹ is_aligned (p >> a << a) b"
  using is_aligned_shift is_aligned_weaken by blast

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›
  by (simp add: add.commute multiple_mask_trivia word_bw_comms(1) word_bw_lcs(1)
word_plus_and_or_coroll2)

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›
  by (simp add: and_not_mask)

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)"
  by (metis and_not_mask mask_rshift_mult_eq_rshift_lshift mask_split_sum_twice
word_eq_unatI)

lemma is_up_compose:
  "⟦ is_up uc; is_up uc' ⟧ ⟹ is_up (uc' ∘ uc)"
  unfolding is_up_def by (simp add: Word.target_size Word.source_size)

lemma of_int_sint_scast:
  "of_int (sint (x :: 'a :: len word)) = (scast x :: 'b :: len word)"
```

```
  by (fact Word.of_int_sint)

lemma scast_of_nat_to_signed [simp]:
  "scast (of_nat x :: 'a :: len word) = (of_nat x :: 'a signed word)"
  by (rule bit_word_eqI) (simp add: bit_simps)

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)"
  by (metis of_nat_add scast_of_nat)

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)"
  by (metis Abs_fnat_hom_add scast_of_nat_to_signed)

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]"
  using and_mask_less' len unat_less_power by (fastforce simp add: image_iff
Bex_def)

lemma sint_eq_uint_2pl:
  "⟦ (a :: 'a :: len word) < 2 ^ (LENGTH('a) - 1) ⟧
   ⟹ sint a = uint a"
  by (simp add: not_msb_from_less sint_eq_uint word_2p_lem word_size)

lemma pow_sub_less:
  "⟦ a + b ≤ LENGTH('a); unat (x :: 'a :: len word) = 2 ^ a ⟧
   ⟹ unat (x * 2 ^ b - 1) < 2 ^ (a + b)"
  by (metis eq_or_less_helperD le_eq_less_or_eq power_add unat_eq_of_nat
unat_lt2p word_unat_power)

lemma sle_le_2pl:
  "⟦ (b :: 'a :: len word) < 2 ^ (LENGTH('a) - 1); a ≤ b ⟧ ⟹ a <=s b"
  by (simp add: not_msb_from_less word_sle_msb_le)

lemma sless_less_2pl:
  "⟦ (b :: 'a :: len word) < 2 ^ (LENGTH('a) - 1); a < b ⟧ ⟹ a <s b"
  using not_msb_from_less word_sless_msb_less by blast

lemma and_mask2:
  "w << n >> n = w AND mask (size w - n)"
  for w :: ‹'a::len word›
  by (rule bit_word_eqI) (auto simp: bit_simps word_size)

lemma aligned_sub_aligned_simple:
  "⟦ is_aligned a n; is_aligned b n ⟧ ⟹ is_aligned (a - b) n"
```

```
  by (simp add: aligned_sub_aligned)

lemma minus_one_shift:
  "- (1 << n) = (-1 << n :: 'a::len word)"
  by (simp add: shiftl_def minus_exp_eq_not_mask)

lemma ucast_eq_mask:
  "(UCAST('a::len → 'b::len) x = UCAST('a → 'b) y) =
   (x AND mask LENGTH('b) = y AND mask LENGTH('b))"
  by transfer (simp flip: take_bit_eq_mask add: ac_simps)

context
  fixes w :: "'a::len word"
begin

private lemma sbintrunc_uint_ucast:
  ‹signed_take_bit n (uint (ucast w :: 'b word)) = signed_take_bit n (uint
w)› if ‹Suc n = LENGTH('b::len)›
  by (rule bit_eqI) (use that in ‹auto simp: bit_simps›)

private lemma test_bit_sbintrunc:
  assumes "i < LENGTH('a)"
  shows "bit (word_of_int (signed_take_bit n (uint w)) :: 'a word) i
           = (if n < i then bit w n else bit w i)"
  using assms by (simp add: bit_simps)

private lemma test_bit_sbintrunc_ucast:
  assumes len_a: "i < LENGTH('a)"
  shows "bit (word_of_int (signed_take_bit (LENGTH('b) - 1) (uint (ucast
w :: 'b word))) :: 'a word) i
           = (if LENGTH('b::len) ≤ i then bit w (LENGTH('b) - 1) else bit
w i)"
  using len_a by (auto simp: sbintrunc_uint_ucast bit_simps)

lemma scast_ucast_high_bits:
  ‹scast (ucast w :: 'b::len word) = w
     ⟷ (∀ i ∈ {LENGTH('b) ..< size w}. bit w i = bit w (LENGTH('b)
- 1))›
proof (cases ‹LENGTH('a) ≤ LENGTH('b)›)
  case True
  moreover define m where ‹m = LENGTH('b) - LENGTH('a)›
  ultimately have ‹LENGTH('b) = m + LENGTH('a)›
    by simp
  then show ?thesis
    by (simp add: signed_ucast_eq word_size) word_eqI
next
  case False
  define q where ‹q = LENGTH('b) - 1›
  then have ‹LENGTH('b) = Suc q›
```

```
    by simp
  moreover define m where ‹m = Suc LENGTH('a) - LENGTH('b)›
  with False ‹LENGTH('b) = Suc q› have ‹LENGTH('a) = m + q›
    by (simp add: not_le)
  ultimately show ?thesis
    apply (simp add: signed_ucast_eq word_size)
    apply transfer
    apply (simp add: signed_take_bit_take_bit)
    apply (simp add: bit_eq_iff bit_take_bit_iff bit_signed_take_bit_iff
min_def)
    by (metis atLeastLessThan_iff linorder_not_le nat_less_le not_less_eq)
qed

lemma scast_ucast_mask_compare:
  "scast (ucast w :: 'b::len word) = w
   ⟷ (w ≤ mask (LENGTH('b) - 1) ∨ NOT(mask (LENGTH('b) - 1)) ≤ w)"
  apply (auto simp: Ball_def le_mask_high_bits neg_mask_le_high_bits scast_ucast_high_bits
word_size)
  by (metis decr_length_less_iff nless_le)

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)"
  by (meson add_lessD1 order_less_le_trans shiftl_less_t2n' ucast_less)

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))"
  by word_eqI_solve

lemma ucast_NOT:
  "ucast (NOT x) = NOT(ucast x) AND mask (LENGTH('a))" for x::"'a::len
word"
  by word_eqI_solve

lemma ucast_NOT_down:
  "is_down UCAST('a::len → 'b::len) ⟹ UCAST('a → 'b) (NOT x) = NOT(UCAST('a
→ 'b) x)"
  by word_eqI

lemma upto_enum_step_shift:
  assumes "is_aligned p n"
  shows "([p , p + 2 ^ m .e. p + 2 ^ n - 1]) = map ((+) p) [0, 2 ^ m .e.
2 ^ n - 1]"
proof -
  consider "n < LENGTH('a)" | "p = 0" "n ≥ LENGTH('a)"
```

```
    by (meson assms is_aligned_get_word_bits)
  then show ?thesis
  proof cases
    case 1
    with assms show ?thesis
      using is_aligned_no_overflow linorder_not_le
      by (force simp: upto_enum_step_def)
  qed (auto simp: map_idI)
qed


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)]"
  by (simp add: upto_enum_step_red upto_enum_step_shift)

lemma upto_enum_step_subset:
  "set [x, y .e. z] ⊆ {x .. z}"
proof -
  have "⋀w. ⟦x ≤ z; w ≤ (z - x) div (y - x)⟧
         ⟹ x ≤ x + w * (y - x) ∧ x + w * (y - x) ≤ z"
    by (metis add.commute div_to_mult_word_lt eq_diff_eq le_plus' word_plus_mono_right2)
  then show ?thesis
    by (auto simp: upto_enum_step_def linorder_not_less)
qed

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)"
  unfolding ucast_eq lift_M
  by (metis lift_M' local.distrib is_down ucast_down_wi uint_word_of_int
word_of_int_uint)

lemma ucast_down_add:
    "is_down (ucast:: 'a word ⇒ 'b word) ⟹  ucast ((a :: 'a::len word)
+ b) = (ucast a + ucast b :: 'b::len word)"
  by (metis (mono_tags, opaque_lifting) of_int_add ucast_down_wi word_of_int_Ex)
```

```
lemma ucast_down_minus:
    "is_down (ucast:: 'a word ⇒ 'b word) ⟹  ucast ((a :: 'a::len word)
- b) = (ucast a - ucast b :: 'b::len word)"
  by (metis add_diff_cancel_right' diff_add_cancel ucast_down_add)

lemma ucast_down_mult:
    "is_down (ucast:: 'a word ⇒ 'b word) ⟹  ucast ((a :: 'a::len word)
* b) = (ucast a * ucast b :: 'b::len word)"
  by (simp add: mod_mult_eq take_bit_eq_mod ucast_distrib uint_word_arith_bintrs(3))

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 -
  have §: "is_down UCAST('a → 'b)"
    using is_up_down is_down by blast
  then have "UCAST('a → 'b) (M a b) = M' (UCAST('a → 'b) a) (UCAST('a
→ 'b) b)"
    using lift_M lift_M' local.distrib ucast_distrib by blast
  with § show ?thesis
    using down_cast_same by fastforce
qed

lemma scast_down_add:
    "is_down (scast:: 'a word ⇒ 'b word) ⟹  scast ((a :: 'a::len word)
+ b) = (scast a + scast b :: 'b::len word)"
  by (metis down_cast_same is_up_down ucast_down_add)

lemma scast_down_minus:
    "is_down (scast:: 'a word ⇒ 'b word) ⟹  scast ((a :: 'a::len word)
- b) = (scast a - scast b :: 'b::len word)"
  by (metis down_cast_same is_up_down ucast_down_minus)

lemma scast_down_mult:
    "is_down (scast:: 'a word ⇒ 'b word) ⟹  scast ((a :: 'a::len word)
* b) = (scast a * scast b :: 'b::len word)"
  by (metis down_cast_same is_up_down ucast_down_mult)

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"
  by (metis down_cast_same ucast_eq ucast_down_wi)

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"
  by (metis down_cast_same ucast_eq ucast_down_wi)

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"
  by (metis down_cast_same ucast_eq ucast_down_wi)

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"
  by (metis scast_eq sint_up_scast)

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"
  by (metis scast_eq ucast_down_wi)

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"
  by (metis scast_eq ucast_down_wi)

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"
  by (metis down_cast_same scast_eq sint_up_scast)

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"
  by (metis down_cast_same ucast_eq ucast_down_wi)

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"
  by (metis ucast_up_ucast)

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"
  by (metis down_cast_same is_down scast_eq ucast_down_wi)

lemma scast_down_wi [OF refl]:
  "uc = scast ⟹ is_down uc ⟹ uc (word_of_int x) = word_of_int x"
  by (metis down_cast_same is_up_down ucast_down_wi)

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 (rule classical)
  assume not_thesis: "¬ ?thesis"

  have not_zero: "b ≠ 0"
    using not_thesis by force

  let ?range = <{- (2 ^ (size a - 1))..<2 ^ (size a - 1)} :: int set>

  have result_range: "sint a sdiv sint b ∈ ?range ∪ {2 ^ (size a - 1)}"
    using sdiv_word_min [of a b] sdiv_word_max [of a b] by auto
```

```
  have result_range_overflow: "(sint a sdiv sint b = 2 ^ (size a - 1))
= (?a_int_min ∧ ?b_minus1)"
  proof -
    have False
      if "sint a sdiv sint b = 2 ^ (size a - 1)" "¬ (a = - (2 ^ (size
a - 1)) ∧ b = - 1)"
    proof (cases "?a_int_min")
      case True
      with that show ?thesis
        by (smt (verit, best) One_nat_def int_sdiv_negated_is_minus1 sint_int_min
sint_minus1
            wsst_TYs(3) zero_less_power)
    next
      case False
      with that have "|sint a| < 2 ^ (size a - 1)"
        by (smt (verit, best) One_nat_def signed_word_eqI sint_ge sint_int_min
sint_less wsst_TYs(3))
      then show ?thesis
        by (metis atLeastAtMost_iff not_less sdiv_int_range that(1))
    qed
    then  show ?thesis
      by (smt (verit, ccfv_SIG) One_nat_def int_sdiv_simps(3) sint_int_min
sint_n1 wsst_TYs(3))
  qed
  then show ?thesis
    using result_range by auto
qed

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))"
  by (auto simp: sint_word_ariths word_size signed_div_arith signed_mod_arith
signed_take_bit_int_eq_self_iff intro: sym dest: sym)

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)"
  by (subst (asm) signed_arith_ineq_checks_to_eq; simp)+

lemma nasty_split_lt:
  ‹x * 2 ^ n + (2 ^ n - 1) ≤ 2 ^ m - 1›
    if ‹x < 2 ^ (m - n)› ‹n ≤ m› ‹m < LENGTH('a::len)›
    for x :: ‹'a::len word›
proof -
  define q where ‹q = m - n›
  with ‹n ≤ m› have ‹m = q + n›
    by simp
  with ‹x < 2 ^ (m - n)› have *: ‹i < q› if ‹bit x i› for i
    using that by simp (metis bit_take_bit_iff take_bit_word_eq_self_iff)
  from ‹m = q + n› have ‹push_bit n x OR mask n ≤ mask m›
    by (auto simp: le_mask_high_bits word_size bit_simps dest!: *)
  then have ‹push_bit n x + mask n ≤ mask m›
    by (simp add: disjunctive_add bit_simps)
  then show ?thesis
    by (simp add: mask_eq_exp_minus_1 push_bit_eq_mult)
qed

end

end
```

# 20 Words of Length 8

**theory** Word_8
**imports**
  More_Word
  Enumeration_Word
  Even_More_List
  Signed_Words
  Word_Lemmas
**begin**

**context**
  **includes** bit_operations_syntax
**begin**

**lemma** len8: "len_of (x :: 8 itself) = 8" **by** simp

**lemma** word8_and_max_simp:
  ‹x AND 0xFF = x› **for** x :: ‹8 word›
  **using** word_and_full_mask_simp [of x]
  **by** (simp add: numeral_eq_Suc mask_Suc_exp)

**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 -
  have ‹map unat ?lhs = [0..<256]›
    by (simp add: enum_word_def comp_def take_bit_nat_eq_self map_idem_upt_eq
unsigned_of_nat)
  also have ‹... = map unat ?rhs›
    by (simp add: upt_zero_numeral_unfold)
  finally show ?thesis
    using unat_inj by (rule map_injective)
qed

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}"
  by (simp add: enum_word8_eq)

lemma set_strip_insert: "⟦ x ∈ insert a S; x ≠ a ⟧ ⟹ x ∈ S"
  by simp

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"
  apply (subgoal_tac "x ∈ set enum", subst (asm) set_enum_word8_def)
    apply (drule set_strip_insert, assumption)+
   apply (erule emptyE)
  apply (subst enum_UNIV, rule UNIV_I)
  done

end

end
```

# 21 Words of Length 16

```
theory Word_16
imports
  More_Word
  Signed_Words
begin

lemma len16: "len_of (x :: 16 itself) = 16" by simp

context
  includes bit_operations_syntax
begin

lemma word16_and_max_simp:
  ‹x AND 0xFFFF = x› for x :: ‹16 word›
  using word_and_full_mask_simp [of x]
  by (simp add: numeral_eq_Suc mask_Suc_exp)

end

end
```

# 22 Additional Syntax for Word Bit Operations

```
theory Word_Syntax
imports
```

```
  "HOL-Library.Word"
begin
```

Additional bit and type syntax that forces word types.

```
context
  includes bit_operations_syntax
begin

abbreviation
  wordNOT  :: "'a::len word ⇒ 'a word"      (‹(‹open_block notation=‹prefix
~~› ›~~ _)› [70] 71)
where
  "~~ x == NOT x"

abbreviation
  wordAND  :: "'a::len word ⇒ 'a word ⇒ 'a word" (infixr ‹&&› 64)
where
  "a && b == a AND b"

abbreviation
  wordOR   :: "'a::len word ⇒ 'a word ⇒ 'a word" (infixr ‹||›  59)
where
  "a || b == a OR b"

abbreviation
  wordXOR  :: "'a::len word ⇒ 'a word ⇒ 'a word" (infixr ‹xor› 59)
where
  "a xor b == a XOR b"

end

end
```

# 23 Names of Specific Word Lengths

```
theory Word_Names
  imports Signed_Words
begin

type_synonym word8 = "8 word"
type_synonym word16 = "16 word"
type_synonym word32 = "32 word"
type_synonym word64 = "64 word"

type_synonym sword8 =  "8 sword"
type_synonym sword16 = "16 sword"
type_synonym sword32 = "32 sword"
type_synonym sword64 = "64 sword"
```

**end**

# 24 Misc word operations

**theory** More_Word_Operations
  **imports**
    "HOL-Library.Word"
    Aligned Reversed_Bit_Lists More_Misc Signed_Words
    Word_Lemmas Many_More Word_EqI

**begin**

**context**
  **includes** bit_operations_syntax
**begin**

**definition**
  ptr_add :: "'a :: len word ⇒ nat ⇒ 'a word" **where**
  "ptr_add ptr n ≡ ptr + of_nat n"

**definition**
  alignUp :: "'a::len word ⇒ nat ⇒ 'a word" **where**
 "alignUp x n ≡ x + 2 ^ n - 1 AND NOT (2 ^ n - 1)"

**lemma** alignUp_unfold:
  ‹alignUp w n = (w + mask n) AND NOT (mask n)›
  **by** (simp add: alignUp_def mask_eq_exp_minus_1 add_mask_fold)

**abbreviation** mask_range :: "'a::len word ⇒ nat ⇒ 'a word set" **where**
  "mask_range p n ≡ {p .. p + mask n}"

**definition**
  w2byte :: "'a :: len word ⇒ 8 word" **where**
  "w2byte ≡ ucast"

**definition**
  word_clz :: "'a::len word ⇒ nat"
**where**
  "word_clz w ≡ length (takeWhile Not (to_bl w))"

**definition**
  word_ctz :: "'a::len word ⇒ nat"
**where**
  "word_ctz w ≡ length (takeWhile Not (rev (to_bl w)))"

**lemma** word_ctz_unfold:

```
  ‹word_ctz w = length (takeWhile (Not ∘ bit w) [0..<LENGTH('a)])› for
w :: ‹'a::len word›
  by (simp add: word_ctz_def rev_to_bl_eq takeWhile_map)

lemma word_ctz_unfold':
  ‹word_ctz w = Min (insert LENGTH('a) {n. bit w n})› for w :: ‹'a::len
word›
proof (cases ‹∃n. bit w n›)
  case True
  then obtain n where ‹bit w n› ..
  then have "Min (Collect (bit w)) = min LENGTH('a) (Min (Collect (bit
w)))"
    by (metis Collect_empty_eq Min_in finite_bit_word mem_Collect_eq min.absorb4
test_bit_conj_lt)
  with ‹bit w n› show ?thesis
    unfolding word_ctz_unfold
    by (metis Collect_empty_eq Min.insert Min_eq_length_takeWhile finite_bit_word
test_bit_conj_lt)
next
  case False
  then have ‹bit w = bot›
    by auto
  then have ‹word_ctz w = LENGTH('a)›
    by (simp add: word_ctz_def rev_to_bl_eq bot_fun_def map_replicate_const)
  with ‹bit w = bot› show ?thesis
    by simp
qed

lemma word_ctz_le: "word_ctz (w :: ('a::len word)) ≤ LENGTH('a)"
proof -
  have "length (takeWhile Not (rev (to_bl w))) ≤ LENGTH('a)"
    by (metis length_rev length_takeWhile_le word_bl_Rep')
  then show ?thesis
    by (simp add: word_ctz_def)
qed

lemma word_ctz_less:
  assumes "w ≠ 0"
  shows "word_ctz (w :: ('a::len word)) < LENGTH('a)"
proof -
  have "length (takeWhile Not (rev (to_bl w))) < LENGTH('a)"
    if "True ∈ set (to_bl w)"
    using that by (metis length_rev length_takeWhile_less set_rev word_bl_Rep')
  with assms show ?thesis
    by (auto simp: word_ctz_def eq_zero_set_bl)
qed

lemma take_bit_word_ctz_eq [simp]:
  ‹take_bit LENGTH('a) (word_ctz w) = word_ctz w›
```

```
  for w :: ‹'a::len word›
  by (force simp: take_bit_nat_eq_self_iff word_ctz_def to_bl_unfold intro:
le_less_trans [OF length_takeWhile_le])

lemma word_ctz_not_minus_1:
  ‹word_of_nat (word_ctz (w :: 'a :: len word)) ≠ (- 1 :: 'a::len word)›
if ‹1 < LENGTH('a)›
proof -
  note word_ctz_le
  also from that have ‹LENGTH('a) < mask LENGTH('a)›
    by (simp add: less_mask)
  finally have ‹word_ctz w < mask LENGTH('a)› .
  then have ‹word_of_nat (word_ctz w) < (word_of_nat (mask LENGTH('a))
:: 'a word)›
    by (simp add: of_nat_word_less_iff)
  also have ‹... = - 1›
    by (rule bit_word_eqI) (simp add: bit_simps)
  finally show ?thesis
    by simp
qed

lemma unat_of_nat_ctz_mw:
  "unat (of_nat (word_ctz (w :: 'a :: len word)) :: 'a :: len word) =
word_ctz w"
  by (simp add: unsigned_of_nat)

lemma unat_of_nat_ctz_smw:
  "unat (of_nat (word_ctz (w :: 'a :: len word)) :: 'a :: len signed word)
= word_ctz w"
  by (simp add: unsigned_of_nat)

definition
  word_log2 :: "'a::len word ⇒ nat"
where
  "word_log2 (w::'a::len word) ≡ size w - 1 - word_clz w"


definition
  pop_count :: "('a::len) word ⇒ nat"
where
  "pop_count w ≡ length (filter id (to_bl w))"


definition
  sign_extend :: "nat ⇒ 'a::len word ⇒ 'a word"
where
  "sign_extend n w ≡ if bit w n then w OR NOT (mask n) else w AND mask
n"
```

```
lemma sign_extend_eq_signed_take_bit:
  ‹sign_extend = signed_take_bit›
proof (rule ext)+
  fix n and w :: ‹'a::len word›
  show ‹sign_extend n w = signed_take_bit n w›
  proof (rule bit_word_eqI)
    fix q
    assume ‹q < LENGTH('a)›
    then show ‹bit (sign_extend n w) q ⟷ bit (signed_take_bit n w)
q›
      by (simp add: bit.disj_conj_distrib2 sign_extend_def signed_take_bit_eq_if_negative
          signed_take_bit_eq_if_positive take_bit_eq_mask)
  qed
qed

definition
  sign_extended :: "nat ⇒ 'a::len word ⇒ bool"
where
  "sign_extended n w ≡ ∀i. n < i ⟶ i < size w ⟶ bit w i = bit w n"

lemma ptr_add_0 [simp]:
  "ptr_add ref 0 = ref "
  unfolding ptr_add_def by simp

lemma pop_count_0[simp]:
  "pop_count 0 = 0"
  by (clarsimp simp:pop_count_def)

lemma pop_count_1[simp]:
  "pop_count 1 = 1"
  by (clarsimp simp:pop_count_def to_bl_1)

lemma pop_count_0_imp_0:
  "(pop_count w = 0) = (w = 0)"
proof
  assume "pop_count w = 0"
  then have "∀x∈set (to_bl w). ¬ id x"
    by (auto simp: pop_count_def filter_empty_conv)
  then show "w = 0"
    using eq_zero_set_bl by fastforce
qed auto

lemma word_log2_zero_eq [simp]:
  ‹word_log2 0 = 0›
  by (simp add: word_log2_def word_clz_def word_size)

lemma word_log2_unfold:
  ‹word_log2 w = (if w = 0 then 0 else Max {n. bit w n})›
  for w :: ‹'a::len word›
```

```
proof (cases ‹w = 0›)
  case True
  then show ?thesis
    by simp
next
  case False
  then obtain r where ‹bit w r›
    by (auto simp add: bit_eq_iff)
  then have ‹Max {m. bit w m} = LENGTH('a) - Suc (length
    (takeWhile (Not ∘ bit w) (rev [0..<LENGTH('a)])))›
    by (subst Max_eq_length_takeWhile [of _ ‹LENGTH('a)›])
      (auto simp add: bit_imp_le_length)
  then have ‹word_log2 w = Max {x. bit w x}›
    by (simp add: word_log2_def word_clz_def word_size to_bl_unfold rev_map
takeWhile_map)
  with ‹w ≠ 0› show ?thesis
    by simp
qed

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 -
  from ‹w ≠ 0› have ‹word_log2 w = Max {n. bit w n}›
    by (simp add: word_log2_unfold)
  also have ‹Max {n. bit w n} = n›
    using that by (auto intro: Max_eqI)
  finally show ?thesis .
qed

lemma bit_word_log2:
  ‹bit w (word_log2 w)› if ‹w ≠ 0›
proof -
  from ‹w ≠ 0› have ‹∃r. bit w r›
    by (auto intro: bit_eqI)
  then obtain r where ‹bit w r› ..
  from ‹w ≠ 0› have ‹word_log2 w = Max {n. bit w n}›
    by (simp add: word_log2_unfold)
  also have ‹Max {n. bit w n} ∈ {n. bit w n}›
    using ‹bit w r› by (subst Max_in) auto
  finally show ?thesis
    by simp
qed

lemma word_log2_maximum:
  ‹n ≤ word_log2 w› if ‹bit w n›
proof -
  have ‹n ≤ Max {n. bit w n}›
```

```
    using that by (auto intro: Max_ge)
  also from that have ‹w ≠ 0›
    by force
  then have ‹Max {n. bit w n} = word_log2 w›
    by (simp add: word_log2_unfold)
  finally show ?thesis .
qed

lemma word_log2_nth_not_set:
  "⟦ word_log2 w < i ; i < size w ⟧ ⟹ ¬ bit w i"
  using word_log2_maximum [of w i] by auto

lemma word_log2_max:
  "word_log2 w < size w"
  by (metis bit_word_log2 test_bit_size word_log2_zero_eq word_size_gt_0)

lemma word_clz_0[simp]:
  "word_clz (0::'a::len word) = LENGTH('a)"
  unfolding word_clz_def by simp

lemma word_clz_minus_one[simp]:
  "word_clz (-1::'a::len word) = 0"
  unfolding word_clz_def by simp

lemma is_aligned_alignUp[simp]:
  "is_aligned (alignUp p n) n"
  by (simp add: alignUp_def is_aligned_mask mask_eq_decr_exp word_bw_assocs)

lemma alignUp_le[simp]:
  "alignUp p n ≤ p + 2 ^ n - 1"
  unfolding alignUp_def by (rule word_and_le2)

lemma alignUp_idem:
  fixes a :: "'a::len word"
  assumes "is_aligned a n" "n < LENGTH('a)"
  shows "alignUp a n = a"
  using assms unfolding alignUp_def
  by (metis add_cancel_right_right add_diff_eq and_mask_eq_iff_le_mask
mask_eq_decr_exp mask_out_add_aligned order_refl word_plus_and_or_coroll2)

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 -
  from ‹n < LENGTH('a)› have ‹(2::int) ^ n < 2 ^ LENGTH('a)›
    by simp
  have anz: "a mod 2 ^ n ≠ 0"
```

```
    by (rule not_aligned_mod_nz) fact+
  then have um: "unat (a mod 2 ^ n - 1) div 2 ^ n = 0"
    by (simp add: sz less_imp_diff_less p2_gt_0 unat_less_power unat_minus_one
word_mod_less_divisor)
  have "a + 2 ^ n - 1 = (a div 2 ^ n) * 2 ^ n + (a mod 2 ^ n) + 2 ^ n
- 1"
    by (simp add: word_mod_div_equality)
  also have "... = (a mod 2 ^ n - 1) + (a div 2 ^ n + 1) * 2 ^ n"
    by (simp add: field_simps)
  finally have §: "a + 2 ^ n - 1 = a mod 2 ^ n - 1 + (a div 2 ^ n + 1)
* 2 ^ n" .
  have "2 ^ n + word_of_nat (unat a div 2 ^ n) * 2 ^ n
          = (word_of_nat (unat a div 2 ^ n) + 1) * 2 ^ n"
    by (smt (verit) sz add.commute mult.commute mult_Suc_right of_nat_Suc
of_nat_add of_nat_mult word_unat_power)
  then show "alignUp a n = (a div 2 ^ n + 1) * 2 ^ n" using sz
    unfolding alignUp_def
    apply (simp add: § flip: mask_eq_decr_exp)
    by (smt (verit) div_eq_0_iff add.commute is_alignedI mult.commute
power_eq_0_iff um
        unat_is_aligned_add word_eq_unatI zero_neq_numeral)
qed

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 (cases "is_aligned a n")
  case True
  then show ?thesis using sz
    by (subst alignUp_idem, simp_all)
next
  case False

  have lt0: "unat a div 2 ^ n < 2 ^ (LENGTH('a) - n)" using sz
    by (metis le_add_diff_inverse2 less_mult_imp_div_less order_less_imp_le
power_add unsigned_less)

  have eq0: "⟦2 ^ n * (unat a div 2 ^ n + 1) = 2 ^ LENGTH('a)⟧
           ⟹ (a div 2 ^ n + 1) * 2 ^ n = 0"
    by (smt (verit) sz mult.commute of_nat_1 of_nat_add of_nat_mult unat_power_lower
word_arith_nat_div word_exp_length_eq_0 word_unat_power)
  have"2 ^ n * (unat a div 2 ^ n + 1) ≤ 2 ^ LENGTH('a)" using sz
    by (metis One_nat_def Suc_leI add.right_neutral add_Suc_right lt0
nat_le_power_trans nat_less_le)
  moreover have "2 ^ n * (unat a div 2 ^ n + 1) ≠ 2 ^ LENGTH('a)" us-
ing nowrap sz
    using eq0 alignUp_not_aligned_eq [OF False sz] by argo
```

```
  ultimately have lt: "2 ^ n * (unat a div 2 ^ n + 1) < 2 ^ LENGTH('a)"
by simp

  have "a = a div 2 ^ n * 2 ^ n + a mod 2 ^ n" by (rule word_mod_div_equality
[symmetric])
  also have "... < (a div 2 ^ n + 1) * 2 ^ n" using sz lt
    by (simp add: field_simps lt0 p2_gt_0 unat_mult_power_lem word_add_less_mono1
word_arith_nat_div word_mod_less_divisor)
  also have "... =  alignUp a n"
    by (rule alignUp_not_aligned_eq [symmetric]) fact+
  finally show ?thesis by (rule order_less_imp_le)
qed

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 (cases "is_aligned a n")
  case True
  then show ?thesis using sz le by (simp add: alignUp_idem)
next
  case False
  then have anz: "a mod 2 ^ n ≠ 0"
    by (rule not_aligned_mod_nz)
  from al obtain k where xk: "x = 2 ^ n * of_nat k" and kv: "k < 2 ^
(LENGTH('a) - n)"
    by (auto elim!: is_alignedE)
  then have kn: "unat (of_nat k :: 'a word) * unat ((2::'a word) ^ n)
< 2 ^ LENGTH('a)"
    using sz
    by (metis nat_mult_power_less_eq nat_power_minus_less unat_of_nat_len
unat_power_lower zero_less_numeral)
  have au: "alignUp a n = (a div 2 ^ n + 1) * 2 ^ n"
    by (rule alignUp_not_aligned_eq) fact+
  also have "... ≤ of_nat k * 2 ^ n"
  proof (rule word_mult_le_mono1 [OF inc_le _ kn])
    show "a div 2 ^ n < of_nat k" using kv xk le sz anz
      by (simp add: alignUp_div_helper)
    show "(0:: 'a word) < 2 ^ n" using sz by (simp add: p2_gt_0 sz)
  qed

  finally show ?thesis using xk by (simp add: field_simps)
qed

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 (cases "is_aligned a n")
  case True
  then have "alignUp a n = a" using sz by (simp add: alignUp_idem)
  then show ?thesis using az by simp
next
  case False
  then have anz: "a mod 2 ^ n ≠ 0"
    by (rule not_aligned_mod_nz)

  {
    assume asm: "alignUp a n = 0"

    have lt0: "unat a div 2 ^ n < 2 ^ (LENGTH('a) - n)" using sz
      by (metis le_add_diff_inverse2 less_mult_imp_div_less order_less_imp_le
power_add unsigned_less)

    have leq: "2 ^ n * (unat a div 2 ^ n + 1) ≤ 2 ^ LENGTH('a)" using
sz
      by (metis One_nat_def Suc_leI add.right_neutral add_Suc_right lt0
nat_le_power_trans
                order_less_imp_le)

    from al obtain k where  kv: "k < 2 ^ (LENGTH('a) - n)" and xk: "x
= 2 ^ n * of_nat k"
      by (auto elim!: is_alignedE)

    then have "a div 2 ^ n < of_nat k" using ax sz anz
      by (rule alignUp_div_helper)

    then have r: "unat a div 2 ^ n < k" using sz
      by (simp flip: drop_bit_eq_div unat_drop_bit_eq) (metis leI le_unat_uoi
unat_mono)

    have "alignUp a n = (a div 2 ^ n + 1) * 2 ^ n"
      by (rule alignUp_not_aligned_eq) fact+

    then have "... = 0" using asm by simp
    then have "2 ^ LENGTH('a) dvd 2 ^ n * (unat a div 2 ^ n + 1)"
      using sz by (simp add: unat_arith_simps ac_simps)
                  (simp add: unat_word_ariths mod_simps mod_eq_0_iff_dvd)
    with leq have "2 ^ n * (unat a div 2 ^ n + 1) = 2 ^ LENGTH('a)"
      by (force elim!: le_SucE)
    then have "unat a div 2 ^ n = 2 ^ LENGTH('a) div 2 ^ n - 1"
      by (metis (no_types, opaque_lifting) Groups.add_ac(2) add.right_neutral
                add_diff_cancel_left' div_le_dividend div_mult_self4 gr_implies_not0
```

```
              le_neq_implies_less power_eq_0_iff zero_neq_numeral)
    then have "unat a div 2 ^ n = 2 ^ (LENGTH('a) - n) - 1"
      using sz by (simp add: power_sub)
    then have "2 ^ (LENGTH('a) - n) - 1 < k" using r
      by simp
    then have False using kv by simp
  } then show ?thesis by clarsimp
qed

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
  from al have xl: "x ≤ x + 2 ^ n - 1" by (simp add: is_aligned_no_overflow)

  from xl sub have ax: "a ≤ x"
    by auto

  show "a ≤ alignUp a n"
  proof (rule alignUp_ge)
    show "alignUp a n ≠ 0" using al sz ax anz
      by (rule alignUp_is_aligned_nz)
  qed fact+

  show "alignUp a n + 2 ^ n - 1 ≤ b"
  proof (rule order_trans)
    from xl show tp: "x + 2 ^ n - 1 ≤ b" using sub
      by auto

    from ax have "alignUp a n ≤ x"
      by (rule alignUp_le_greater_al) fact+
    then have "alignUp a n + (2 ^ n - 1) ≤ x + (2 ^ n - 1)"
      using xl al is_aligned_no_overflow' olen_add_eqv word_plus_mcs_3
by blast
    then show "alignUp a n + 2 ^ n - 1 ≤ x + 2 ^ n - 1"
      by (simp add: field_simps)
  qed
qed

lemma alignUp_def2:
  "alignUp a sz = a + 2 ^ sz - 1 AND NOT (mask sz)"
  by (simp add: alignUp_def flip: mask_eq_decr_exp)

lemma alignUp_def3:
  "alignUp a sz = 2^ sz + (a - 1 AND NOT (mask sz))"
```

```
  by (simp add: alignUp_def2 is_aligned_triv field_simps mask_out_add_aligned)

lemma  alignUp_plus:
  "is_aligned w us ⟹ alignUp (w + a) us  = w + alignUp a us"
  by (clarsimp simp: alignUp_def2 mask_out_add_aligned field_simps)

lemma alignUp_distance:
  "alignUp (q :: 'a :: len word) sz - q ≤ mask sz"
  by (metis (no_types) add.commute add_diff_cancel_left alignUp_def2 diff_add_cancel
                       mask_2pm1 subtract_mask(2) word_and_le1 word_sub_le_iff)

lemma is_aligned_diff_neg_mask:
  assumes "is_aligned p sz"
  shows "(p - q AND NOT (mask sz)) = (p - ((alignUp q sz) AND NOT (mask
sz)))"
proof -
  have "- q AND NOT (mask sz) = - (alignUp q sz AND NOT (mask sz))"
    by (simp add: eq_neg_iff_add_eq_0 add.commute alignUp_distance is_aligned_neg_mask_eq
mask_out_add_aligned and_mask_eq_iff_le_mask flip: mask_eq_x_eq_0)
  with assms show ?thesis
    by (metis diff_conv_add_uminus mask_out_add_aligned)
qed

lemma word_clz_max: "word_clz w ≤ size (w::'a::len word)"
  unfolding word_clz_def
  by (metis length_takeWhile_le word_size_bl)

lemma word_clz_nonzero_max:
  fixes w :: "'a::len word"
  assumes nz: "w ≠ 0"
  shows "word_clz w < size (w::'a::len word)"
proof -
  {
    assume a: "word_clz w = size (w::'a::len word)"
    hence "length (takeWhile Not (to_bl w)) = length (to_bl w)"
      by (simp add: word_clz_def word_size)
    hence allj: "∀j∈set(to_bl w). ¬ j"
      by (metis a length_takeWhile_less less_irrefl_nat word_clz_def)
    hence "to_bl w = replicate (length (to_bl w)) False"
      using eq_zero_set_bl nz by fastforce
    hence "w = 0"
      by (metis to_bl_0 word_bl.Rep_eqD word_bl_Rep')
    with nz have False by simp
  }
  thus ?thesis using word_clz_max
    by (fastforce intro: le_neq_trans)
qed
```

```
lemma bin_sign_extend_iff [bit_simps]:
  ‹bit (sign_extend e w) i ⟷ bit w (min e i)›
  if ‹i < LENGTH('a)› for w :: ‹'a::len word›
  using that by (simp add: sign_extend_def bit_simps min_def)

lemma sign_extend_bitwise_if:
  "i < size w ⟹ bit (sign_extend e w) i ⟷ (if i < e then bit w i else
bit w e)"
  by (simp add: word_size bit_simps)

lemma sign_extend_bitwise_if'  [word_eqI_simps]:
  ‹i < LENGTH('a) ⟹ bit (sign_extend e w) i ⟷ (if i < e then bit
w i else bit w e)›
  for w :: ‹'a::len word›
  using sign_extend_bitwise_if [of i w e] by (simp add: word_size)

lemma sign_extend_bitwise_disj:
  "i < size w ⟹ bit (sign_extend e w) i ⟷ i ≤ e ∧ bit w i ∨ e ≤
i ∧ bit w e"
  by (auto simp: sign_extend_bitwise_if)

lemma sign_extend_bitwise_cases:
  "i < size w ⟹ bit (sign_extend e w) i ⟷ (i ≤ e ⟶ bit w i) ∧ (e
≤ i ⟶ bit w e)"
  by (auto simp: sign_extend_bitwise_if)

lemmas sign_extend_bitwise_disj' = sign_extend_bitwise_disj[simplified
word_size]
lemmas sign_extend_bitwise_cases' = sign_extend_bitwise_cases[simplified
word_size]


lemma sign_extend_def':
  "sign_extend n w = (if bit w n then w OR NOT (mask (Suc n)) else w AND
mask (Suc n))"
  by (rule bit_word_eqI) (auto simp add: bit_simps sign_extend_eq_signed_take_bit
min_def less_Suc_eq_le)

lemma sign_extended_sign_extend:
  "sign_extended n (sign_extend n w)"
  by (clarsimp simp: sign_extended_def word_size sign_extend_bitwise_if)

lemma sign_extended_iff_sign_extend:
  "sign_extended n w ⟷ sign_extend n w = w"
  by (metis linorder_not_le sign_extend_bitwise_cases sign_extend_bitwise_disj
sign_extended_def word_eqI)

lemma sign_extended_weaken:
```

```
  "sign_extended n w ⟹ n ≤ m ⟹ sign_extended m w"
  unfolding sign_extended_def by (cases "n < m") auto

lemma sign_extend_sign_extend_eq:
  "sign_extend m (sign_extend n w) = sign_extend (min m n) w"
  by (rule bit_word_eqI) (simp add: sign_extend_eq_signed_take_bit bit_simps)

lemma sign_extended_high_bits:
  "⟦ sign_extended e p; j < size p; e ≤ i; i < j ⟧ ⟹ bit p i = bit p
j"
  by (drule (1) sign_extended_weaken; simp add: sign_extended_def)

lemma sign_extend_eq:
  "w AND mask (Suc n) = v AND mask (Suc n) ⟹ sign_extend n w = sign_extend
n v"
  by (simp flip: take_bit_eq_mask add: sign_extend_eq_signed_take_bit
signed_take_bit_eq_iff_take_bit_eq)

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 (cases "e < size p")
  case True
  note and_or = is_aligned_add_or[OF p f]
  have "¬ bit f e"
    using True e less_2p_is_upper_bits_unset[THEN iffD1, OF f]
    by (fastforce simp: word_size)
  hence i: "bit (p + f) e = bit p e"
    by (simp add: and_or bit_simps)
  have fm: "f AND mask e = f"
    by (fastforce intro: subst[where P="λf. f AND mask e = f", OF less_mask_eq[OF
f]]
                  simp: mask_twice e)
  show ?thesis
    using assms
    by (simp add: and_or bit_or_iff less_2p_is_upper_bits_unset sign_extended_def
wsst_TYs(3))
next
  case False thus ?thesis
    by (simp add: sign_extended_def word_size)
qed

lemma sign_extended_neq_mask:
  "⟦sign_extended n ptr; m ≤ n⟧ ⟹ sign_extended n (ptr AND NOT (mask
m))"
  by (fastforce simp: sign_extended_def word_size neg_mask_test_bit bit_simps)
```

```
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"
  unfolding limited_and_def
  by (metis and_and_not and_zero_eq word_bw_lcs(1))

lemma limited_and_eq_id:
  "⟦ limited_and x z; y AND z = z ⟧ ⟹ x AND y = x"
  unfolding limited_and_def
  by (erule subst, fastforce simp: word_bw_lcs word_bw_assocs word_bw_comms)

lemma lshift_limited_and:
  "limited_and x z ⟹ limited_and (x << n) (z << n)"
  using push_bit_and [of n x z] by (simp add: limited_and_def shiftl_def)

lemma rshift_limited_and:
  "limited_and x z ⟹ limited_and (x >> n) (z >> n)"
  using drop_bit_and [of n x z] by (simp add: limited_and_def shiftr_def)

lemmas limited_and_simps1 = limited_and_eq_0 limited_and_eq_id

lemmas is_aligned_limited_and
    = is_aligned_neg_mask_eq[unfolded mask_eq_decr_exp, folded limited_and_def]

lemmas limited_and_simps = limited_and_simps1
       limited_and_simps1[OF is_aligned_limited_and]
       limited_and_simps1[OF lshift_limited_and]
       limited_and_simps1[OF rshift_limited_and]
       limited_and_simps1[OF rshift_limited_and, OF is_aligned_limited_and]
       not_one_eq

definition
  from_bool :: "bool ⇒ 'a::len word" where
  "from_bool b ≡ case b of True ⇒ of_nat 1
                         | False ⇒ of_nat 0"

lemma from_bool_eq: <from_bool = of_bool>
  by (simp add: fun_eq_iff from_bool_def)

lemma from_bool_0: "(from_bool x = 0) = (¬ x)"
  by (simp add: from_bool_def split: bool.split)

lemma from_bool_eq_if':
  "((if P then 1 else 0) = from_bool Q) = (P = Q)"
  by (cases Q) (simp_all add: from_bool_def)
```

```
definition
  to_bool :: "'a::len word ⇒ bool" where
  "to_bool ≡ (≠) 0"

lemma to_bool_and_1:
  "to_bool (x AND 1) ⟷ bit x 0"
  by (simp add: to_bool_def word_and_1)

lemma to_bool_from_bool [simp]: "to_bool (from_bool r) = r"
  unfolding from_bool_def to_bool_def
  by (simp split: bool.splits)

lemma from_bool_neq_0 [simp]: "(from_bool b ≠ 0) = b"
  by (simp add: from_bool_def split: bool.splits)

lemma from_bool_mask_simp [simp]:
  "(from_bool r :: 'a::len word) AND 1 = from_bool r"
  unfolding from_bool_def
  by (clarsimp split: bool.splits)

lemma from_bool_1 [simp]: "(from_bool P = 1) = P"
  by (simp add: from_bool_def split: bool.splits)

lemma ge_0_from_bool [simp]: "(0 < from_bool P) = P"
  by (simp add: from_bool_def split: bool.splits)

lemma limited_and_from_bool:
  "limited_and (from_bool b) 1"
  using from_bool_mask_simp limited_and_def by blast

lemma to_bool_1 [simp]: "to_bool 1" by (simp add: to_bool_def)
lemma to_bool_0 [simp]: "¬to_bool 0" by (simp add: to_bool_def)

lemma from_bool_eq_if:
  "(from_bool Q = (if P then 1 else 0)) = (P = Q)"
  by (cases Q) (simp_all add: from_bool_def)

lemma to_bool_eq_0:
  "(¬ to_bool x) = (x = 0)"
  by (simp add: to_bool_def)

lemma to_bool_neq_0:
  "(to_bool x) = (x ≠ 0)"
  by (simp add: to_bool_def)

lemma from_bool_all_helper:
  "(∀bool. from_bool bool = val ⟶ P bool)
      = ((∃bool. from_bool bool = val) ⟶ P (val ≠ 0))"
  by (auto simp: from_bool_0)
```

```
lemma fold_eq_0_to_bool:
  "(v = 0) = (¬ to_bool v)"
  by (simp add: to_bool_def)

lemma from_bool_to_bool_iff:
  "w = from_bool b ⟷ to_bool w = b ∧ (w = 0 ∨ w = 1)"
  by (cases b) (auto simp: from_bool_def to_bool_def)

lemma from_bool_eqI:
  "from_bool x = from_bool y ⟹ x = y"
  unfolding from_bool_def
  by (auto split: bool.splits)

lemma neg_mask_in_mask_range:
  assumes "is_aligned ptr bits"
  shows "(ptr' AND NOT(mask bits) = ptr) = (ptr' ∈ mask_range ptr bits)"
proof -
  have "ptr' AND NOT (mask bits) ≤ ptr'"
    if "ptr = ptr' AND NOT (mask bits)"
    using that word_and_le2 by auto
  moreover have "ptr' ≤ (ptr' AND NOT (mask bits)) + mask bits"
    if "ptr = ptr' AND NOT (mask bits)"
    using that by (simp add: and_neg_mask_plus_mask_mono)
  moreover have "ptr' AND NOT (mask bits) = ptr"
    if "ptr ≤ ptr'" and "ptr' ≤ ptr + mask bits"
    using that assms
    by (meson and_neg_mask_plus_mask_mono order.trans is_aligned_add_step_le
is_aligned_neg_mask2 linorder_less_linear word_and_le2)
  ultimately show ?thesis
    using assms atLeastAtMost_iff by blast
qed

lemma aligned_offset_in_range:
  assumes "is_aligned x m"
      and "y < 2 ^ m"
      and "is_aligned p n"
      and "m ≤ n"
    shows "(x + y ∈ {p .. p + mask n}) = (x ∈ mask_range p n)"
proof (subst disjunctive_add)
  fix k show "¬ bit x k ∨ ¬ bit y k"
    using assms by (metis bit_take_bit_iff is_aligned_nth take_bit_word_eq_self_iff)
next
  have ‹y AND NOT (mask n) = 0›
    using assms by (metis less_mask_eq mask_overlap_zero)
  with assms show "(x OR y ∈ mask_range p n) = (x ∈ mask_range p n)"
    by (metis less_mask_eq mask_subsume neg_mask_in_mask_range)
qed
```

```
lemma mask_range_to_bl':
  assumes "is_aligned (ptr :: 'a :: len word) bits" "bits < LENGTH('a)"
  shows "mask_range ptr bits
       = {x. take (LENGTH('a) - bits) (to_bl x) = take (LENGTH('a) - bits)
(to_bl ptr)}"
proof -
  have "take (LENGTH('a) - bits) (to_bl x) = take (LENGTH('a) - bits)
(to_bl ptr)"
    if x: "ptr ≤ x" "x ≤ ptr + mask bits"
    for x :: "'a word"
  proof -
    obtain y where "x = ptr + y" "y < 2 ^ bits"
      by (metis x and_mask_less' assms atLeastAtMost_iff bit.double_compl
neg_mask_in_mask_range
          word_plus_and_or_coroll2)
    then show ?thesis
      using that assms by (simp add: is_aligned_add_conv)
  qed
  moreover have "ptr ≤ x" "x ≤ ptr + mask bits"
    if x: "take (LENGTH('a) - bits) (to_bl x) = take (LENGTH('a) - bits)
(to_bl ptr)"
    for x :: "'a word"
  proof -
    obtain y where y: "y < 2 ^ bits" "to_bl (ptr + y) = to_bl x"
    proof
      let ?y = "of_bl (drop (LENGTH('a) - bits) (to_bl x)) :: 'a word"
      have "length (drop (LENGTH('a) - bits) (to_bl x)) < LENGTH('a)"
        by (simp add: assms)
      then show "?y < 2 ^ bits"
        by (simp add: of_bl_length_less)
      then show "to_bl (ptr + ?y) = to_bl x"
        using x assms is_aligned_add_conv
        by (metis (no_types) append_eq_conv_conj atd_lem bl_and_mask length_replicate
            of_bl_rep_False word_bl.Rep_inverse)
    qed
    show "ptr ≤ x"
      using assms y is_aligned_no_wrap' by auto
    show "x ≤ ptr + mask bits"
      by (metis assms y le_mask_iff_lt_2n word_bl.Rep_eqD word_plus_mono_right
          is_aligned_no_overflow_mask)
  qed
  ultimately show ?thesis
    using assms atLeastAtMost_iff by blast
qed

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)}"
  apply (frule is_aligned_get_word_bits, assumption)
  using mask_range_to_bl'
   apply blast
  using power_overflow mask_eq_decr_exp
  by (smt (verit, del_insts) Collect_cong Collect_mem_eq diff_is_0_eq'
is_aligned_beyond_length is_aligned_neg_mask2 linorder_not_le mask_range_to_bl'
neg_mask_in_mask_range take0)

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'"
  apply (clarsimp simp add: mask_range_to_bl set_eq_iff)
  by (smt (verit, ccfv_threshold) le_refl min.orderE min_le_iff_disj take_take)

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"
  using al
proof (rule is_aligned_get_word_bits)
  assume p0: "ptr = 0" and szv': "LENGTH ('a) ≤ sz"
  then have "(2 ::'a word) ^ sz = 0" by simp
  show ?thesis using p0
    by (simp add: ‹2 ^ sz = 0› mask_eq_decr_exp)
next
  assume szv': "sz < LENGTH('a)"
  hence blah: "2 ^ (sz - sz') < (2 :: nat) ^ LENGTH('a)"
    using szv by auto
  show ?thesis
  proof -
    have "ptr ≤ ptr + x"
      if "ptr + x ≤ ptr + x + mask sz'"
      using that al xsz is_aligned_no_wrap' by blast
    moreover have "ptr + x + mask sz' ≤ ptr + mask sz"
      if "ptr + x ≤ ptr + x + mask sz'"
      using that aligned_add_aligned aligned_mask_step
    proof -
      have "ptr ≤ ptr + mask sz"
        by (simp add: al is_aligned_no_overflow_mask)
      moreover have "x ≤ mask sz"
        using xsz le_mask_iff_lt_2n szv' by blast
      moreover have "is_aligned (ptr + x) sz'"
        using al al' aligned_add_aligned szv by blast
```

```
      ultimately show ?thesis
        using al aligned_mask_step szv word_plus_mono_right by blast
    qed
    ultimately show ?thesis
      by auto
  qed
qed

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' = {}"
  using aligned_mask_range_cases
  by (auto simp: neg_mask_in_mask_range)

lemma aligned_mask_ranges_disjoint2:
  assumes "is_aligned p n"
      and "is_aligned ptr bits"
      and "m ≤ n"
      and "n < size p"
      and "m ≤ bits"
      and "∀y<2 ^ (n - m). p + (y << m) ∉ mask_range ptr bits"
    shows "mask_range p n ∩ mask_range ptr bits = {}"
proof -
  have False
    if xp: "p ≤ x" "x ≤ p + mask n"
      and xptr: "ptr ≤ x" "x ≤ ptr + mask bits"
    for x :: "'a word"
  proof -
    have eq_p: "x AND NOT (mask n) = p"
      by (simp add: assms(1) neg_mask_in_mask_range xp)
    have eq_ptr: "x AND NOT (mask bits) = ptr"
      by (simp add: assms(2) neg_mask_in_mask_range xptr)
    have *: "x AND mask n >> m < 2 ^ (n - m)"
      using that assms
      by (metis and_mask_less' le_add_diff_inverse shiftr_less_t2n wsst_TYs(3))
    then have "take_bit (n - m) (drop_bit m x) < 2 ^ (n - m)"
      by (metis drop_bit_take_bit shiftr_def take_bit_eq_mask)
    then have "p + (x AND mask n >> m << m) AND NOT (mask bits) = ptr"
      using assms
      by (smt (verit) AND_NOT_mask_plus_AND_mask_eq and_not_mask eq_p
eq_ptr is_aligned_neg_mask_weaken
          word_bw_assocs(1) word_bw_comms(1))
    then show ?thesis
      using * assms(2,6) neg_mask_in_mask_range by blast
  qed
  then show ?thesis
    using range_inter by blast
```

**qed**

**lemma** word_clz_sint_upper[simp]:
  "LENGTH('a) ≥ 3 ⟹ sint (of_nat (word_clz (w :: 'a :: len word)) ::
'a sword) ≤ int (LENGTH('a))"
  **using** word_clz_max [of w]
  **by** (smt (verit, ccfv_SIG) id_apply of_int_eq_id of_nat_le_0_iff of_nat_mono
semiring_1_class.of_nat_0
      signed_of_nat signed_take_bit_int_eq_self sint_range' wsst_TYs(3))

**lemma** word_clz_sint_lower[simp]:
  **assumes** "LENGTH('a) ≥ 3"
  **shows** "- sint (of_nat (word_clz (w :: 'a :: len word)) :: 'a signed
word) ≤ int (LENGTH('a))"
**proof** -
  **have** "∀w. ¬ 2 ^ (LENGTH('a) - 1) < - sint (w::'a signed word)"
    **by** (smt (verit, ccfv_SIG) len_signed sint_ge)
  **have** "word_clz w < 2 ^ LENGTH('a)"
    **by** (metis pow_mono_leq_imp_lt word_clz_max wsst_TYs(3))
  **then show** ?thesis
    **using** assms small_powers_of_2 len_signed negative_zle order_le_less_trans
    **by** (smt (verit, best) int_eq_sint word_clz_max wsst_TYs(3))
**qed**

**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"
  **using** aligned_mask_step **by** fastforce

**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"
  **by** (simp add: is_aligned_no_wrap' mask_2pm1 nasty_split_lt word_less_power_trans2
               word_plus_mono_right)

**lemma** from_to_bool_last_bit:
  "from_bool (to_bool (x AND 1)) = x AND 1"
  **by** (metis from_bool_to_bool_iff word_and_1)

**lemma** sint_ctz:
  ‹0 ≤ sint (of_nat (word_ctz (x :: 'a :: len word)) :: 'a signed word)
     ∧ sint (of_nat (word_ctz x) :: 'a signed word) ≤ int (LENGTH('a))›
(**is** ‹?P ∧ ?Q›)
  **if** ‹LENGTH('a) > 2›
**proof**
  **have** *: ‹word_ctz x ≤ LENGTH('a)›
    **by** (simp add: word_ctz_le)

```
  also have ‹... < 2 ^ (LENGTH('a) - Suc 0)›
    using that small_powers_of_2 by simp
  finally have ‹int (word_ctz x) div 2 ^ (LENGTH('a) - Suc 0) = 0›
    by simp
  then show ?P by (simp add: signed_of_nat bit_iff_odd)
  show ?Q
    by (smt (verit, best) "*" id_apply of_int_eq_id of_nat_0_le_iff of_nat_mono
signed_of_nat
        signed_take_bit_int_eq_self sint_range')
qed

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"
  by (metis less_trans unat_of_nat_eq word_log2_max word_size)

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"
  by (simp add: add.commute aligned_add_mask_less_eq is_aligned_weaken)

lemma Suc_mask_eq_mask:
  "¬bit a n ⟹ a AND mask (Suc n) = a AND mask n" for a::"'a::len word"
  by (metis sign_extend_def sign_extend_def')

lemma word_less_high_bits:
  fixes a::"'a::len word"
  assumes high_bits: "∀i > n. bit a i = bit b i"
  assumes less: "a AND mask (Suc n) < b AND mask (Suc n)"
  shows "a < b"
proof -
  let ?masked = "λx. x AND NOT (mask (Suc n))"
  from high_bits
  have "?masked a = ?masked b"
    by - word_eqI_solve
  then
  have "?masked a + (a AND mask (Suc n)) < ?masked b + (b AND mask (Suc
n))"
    by (metis AND_NOT_mask_plus_AND_mask_eq less word_and_le2 word_plus_strict_mono_right)
  then
  show ?thesis
    by (simp add: AND_NOT_mask_plus_AND_mask_eq)
qed

lemma word_less_bitI:
  fixes a :: "'a::len word"
```

```
  assumes hi_bits: "∀i > n. bit a i = bit b i"
  assumes a_bits: "¬bit a n"
  assumes b_bits: "bit b n" "n < LENGTH('a)"
  shows "a < b"
proof -
  from b_bits
  have "a AND mask n < b AND mask (Suc n)"
    by (metis bit_mask_iff impossible_bit le2p_bits_unset leI lessI less_Suc_eq_le
mask_eq_decr_exp
              word_and_less' word_ao_nth)
  with a_bits
  have "a AND mask (Suc n) < b AND mask (Suc n)"
    by (simp add: Suc_mask_eq_mask)
  with hi_bits
  show ?thesis
    by (rule word_less_high_bits)
qed

lemma word_less_bitD:
  fixes a::"'a::len word"
  assumes less: "a < b"
  shows "∃n. (∀i > n. bit a i = bit b i) ∧ ¬bit a n ∧ bit b n"
proof -
  define xs where "xs ≡ zip (to_bl a) (to_bl b)"
  define tk where "tk ≡ length (takeWhile (λ(x,y). x = y) xs)"
  define  n where  "n ≡ LENGTH('a) - Suc tk"
  have n_less: "n < LENGTH('a)"
    by (simp add: n_def)
  moreover
  { fix i
    have "¬ i < LENGTH('a) ⟹ bit a i = bit b i"
      using bit_imp_le_length by blast
    moreover
    assume "i > n"
    with n_less
    have "i < LENGTH('a) ⟹ LENGTH('a) - Suc i < tk"
      unfolding n_def by arith
    hence "i < LENGTH('a) ⟹ bit a i = bit b i"
      unfolding n_def tk_def xs_def
      by (fastforce dest: takeWhile_take_has_property_nth simp: rev_nth
simp flip: nth_rev_to_bl)
    ultimately
    have "bit a i = bit b i"
      by blast
  }
  note all = this
  moreover
  from less
  have "a ≠ b" by simp
```

```
  then
  obtain i where "to_bl a ! i ≠ to_bl b ! i"
    using nth_equalityI word_bl.Rep_eqD word_rotate.lbl_lbl by blast
  then
  have "tk ≠ length xs"
    unfolding tk_def xs_def
    by (metis length_takeWhile_less list_eq_iff_zip_eq nat_neq_iff word_rotate.lbl_lbl)
  then
  have "tk < length xs"
    using length_takeWhile_le order_le_neq_trans tk_def by blast
  from nth_length_takeWhile[OF this[unfolded tk_def]]
  have "fst (xs ! tk) ≠ snd (xs ! tk)"
    by (clarsimp simp: tk_def)
  with `tk < length xs`
  have "bit a n ≠ bit b n"
    by (clarsimp simp: xs_def n_def tk_def nth_rev simp flip: nth_rev_to_bl)
  with less all
  have "¬bit a n ∧ bit b n"
    by (metis n_less order.asym word_less_bitI)
  ultimately
  show ?thesis by blast
qed

lemma word_less_bit_eq:
  "(a < b) = (∃n < LENGTH('a). (∀i > n. bit a i = bit b i) ∧ ¬bit a n
∧ bit b n)" for a::"'a::len word"
  by (meson bit_imp_le_length word_less_bitD word_less_bitI)

end

end
```

# 25 Words of Length 32

```
theory Word_32
  imports
    Word_Lemmas
    Word_Syntax
    Word_Names
    Rsplit
    More_Word_Operations
    Bitwise
begin

context
  includes bit_operations_syntax
begin

type_synonym word32 = "32 word"
```

```
lemma len32: "len_of (x :: 32 itself) = 32" by simp

type_synonym sword32 = "32 sword"

lemma ucast_8_32_inj:
  "inj (ucast ::  8 word ⇒ 32 word)"
  by (rule down_ucast_inj) (clarsimp simp: is_down_def target_size source_size)

lemmas unat_power_lower32' = unat_power_lower[where 'a=32]

lemmas word32_less_sub_le' = word_less_sub_le[where 'a = 32]

lemmas word32_power_less_1' = word_power_less_1[where 'a = 32]

lemmas unat_of_nat32' = unat_of_nat_eq[where 'a=32]

lemmas unat_mask_word32' = unat_mask[where 'a=32]

lemmas word32_minus_one_le' = word_minus_one_le[where 'a=32]
lemmas word32_minus_one_le = word32_minus_one_le'[simplified]

lemma unat_ucast_8_32:
  fixes x :: "8 word"
  shows "unat (ucast x :: word32) = unat x"
  by transfer simp

lemma ucast_le_ucast_8_32:
  "(ucast x ≤ (ucast y :: word32)) = (x ≤ (y :: 8 word))"
  by (simp add: ucast_le_ucast)

lemma eq_2_32_0:
  "(2 ^ 32 :: word32) = 0"
  by simp

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"
  by (rule of_nat_n_less_equal_power_2, clarsimp simp: word_size)

lemma unat_ucast_10_32 :
  fixes x :: "10 word"
  shows "unat (ucast x :: word32) = unat x"
  by transfer simp

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)"
  by (simp_all add: word_size)

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)"
  by (simp_all add: of_nat_take_bit take_bit_word_eq_self unsigned_of_nat)

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)"
  by auto

lemmas signed_shift_guard_to_word_32
    = signed_shift_guard_to_word[OF word32_31_less(1-2)]
    signed_shift_guard_to_word[OF word32_31_less(3-4)]
```

```
lemma has_zero_byte:
  "~~ (((((v::word32) && 0x7f7f7f7f) + 0x7f7f7f7f) || v) || 0x7f7f7f7f)
≠ 0
    ⟹ v && 0xff000000 = 0 ∨ v && 0xff0000 = 0 ∨ v && 0xff00 = 0 ∨ v
&& 0xff = 0"
  by word_bitwise auto

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 -
  from ‹b && 1 = 1› have ‹odd b›
    by (auto simp add: mod_2_eq_odd and_one_eq)
  then have ‹b mod 2 = 1›
    using odd_iff_mod_2_eq_one by blast
  from ‹∃x. x < 32 ∧ mask x = b >> 1› obtain x where ‹x < 32› ‹mask
x = b >> 1› by blast
  then have ‹mask x = b div 2›
    using shiftr1_is_div_2 [of b] by simp
  with ‹b mod 2 = 1› have ‹2 * mask x + 1 = 2 * (b div 2) + b mod 2›
    by (simp only:)
  also have ‹... = b›
    by (simp add: mult_div_mod_eq)
  finally have ‹2 * mask x + 1 = b› .
  moreover have ‹mask (Suc x) = 2 * mask x + (1 :: 'a::len word)›
    by (simp add: mask_Suc_rec)
  ultimately show ?thesis
    by auto
qed

lemma unat_of_int_32:
  "⟦i ≥ 0; i ≤2 ^ 31⟧ ⟹ (unat ((of_int i)::sword32)) = nat i"
  by (simp add: unsigned_of_int nat_take_bit_eq take_bit_nat_eq_self)

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"
  by (simp add:cast_chunk_assemble_id)


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"
  by (simp add:cast_chunk_scast_assemble_id)
```

**lemma** cast_down_u64: "(scast::64 word ⇒ 32 word) = (ucast::64 word ⇒ 32 word)"
  **by** (subst down_cast_same[symmetric]; simp add:is_down)+

**lemma** cast_down_s64: "(scast::64 sword ⇒ 32 word) = (ucast::64 sword ⇒ 32 word)"
  **by** (subst down_cast_same[symmetric]; simp add:is_down)

**lemma** word32_and_max_simp:
  ‹x AND 0xFFFFFFFF = x› **for** x :: ‹32 word›
  **using** word_and_full_mask_simp [of x]
  **by** (simp add: numeral_eq_Suc mask_Suc_exp)

**end**

**end**

# 26 Ancient comprehensive Word Library

**theory** Word_Lib_Sumo
**imports**
  "HOL-Library.Word"
  Aligned
  Bit_Comprehension
  Bit_Comprehension_Int
  Bit_Shifts_Infix_Syntax
  Bitwise_Signed
  Bitwise
  Enumeration_Word
  Generic_set_bit
  Hex_Words
  Least_significant_bit
  More_Arithmetic
  More_Divides
  More_Sublist
  More_Int
  Bin_sign
  Even_More_List
  More_Misc
  Legacy_Aliases
  Most_significant_bit
  Next_and_Prev
  Norm_Words
  Reversed_Bit_Lists
  Rsplit
  Signed_Words
  Syntax_Bundles
  Sgn_Abs
  Typedef_Morphisms

```
  Type_Syntax
  Word_EqI
  Word_Lemmas
  Word_8
  Word_16
  Word_32
  Word_Syntax
  Signed_Division_Word
  Singleton_Bit_Shifts
  More_Word_Operations
  Many_More
begin

unbundle bit_operations_syntax
unbundle bit_projection_infix_syntax

declare word_induct2[induct type]
declare word_nat_cases[cases type]

declare signed_take_bit_Suc [simp]


lemmas of_int_and_nat = unsigned_of_nat unsigned_of_int signed_of_int
signed_of_nat

bundle no_take_bit
begin
declare of_int_and_nat[simp del]
end

lemmas bshiftr1_def = bshiftr1_eq
lemmas is_down_def = is_down_eq
lemmas is_up_def = is_up_eq
lemmas mask_def = mask_eq
lemmas scast_def = scast_eq
lemmas shiftl1_def = shiftl1_eq
lemmas shiftr1_def = shiftr1_eq
lemmas sshiftr1_def = sshiftr1_eq
lemmas sshiftr_def = sshiftr_eq_funpow_sshiftr1
lemmas to_bl_def = to_bl_eq
lemmas ucast_def = ucast_eq
lemmas unat_def = unat_eq_nat_uint
lemmas word_cat_def = word_cat_eq
lemmas word_reverse_def = word_reverse_eq_of_bl_rev_to_bl
lemmas word_roti_def = word_roti_eq_word_rotr_word_rotl
lemmas word_rotl_def = word_rotl_eq
lemmas word_rotr_def = word_rotr_eq
lemmas word_sle_def = word_sle_eq
lemmas word_sless_def = word_sless_eq
```

```
lemmas uint_0 = uint_nonnegative
lemmas uint_lt = uint_bounded
lemmas uint_mod_same = uint_idem
lemmas of_nth_def = word_set_bits_def

lemmas of_nat_word_eq_iff = word_of_nat_eq_iff
lemmas of_nat_word_eq_0_iff = word_of_nat_eq_0_iff
lemmas of_int_word_eq_iff = word_of_int_eq_iff
lemmas of_int_word_eq_0_iff = word_of_int_eq_0_iff

lemmas word_next_def = word_next_unfold

lemmas word_prev_def = word_prev_unfold

lemmas is_aligned_def = is_aligned_iff_dvd_nat

lemmas word_and_max_simps =
  word8_and_max_simp
  word16_and_max_simp
  word32_and_max_simp

lemma distinct_lemma: "f x ≠ f y ⟹ x ≠ y" by auto

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"
  by (auto simp: sint_word_ariths)

declare of_nat_diff [simp]


notation (input)
  bit (<testBit>)

lemmas cast_simps = cast_simps ucast_down_bl

end
```

# 27 32 bit standard platform-specific word size and alignment.

```
theory Machine_Word_32_Basics
imports "HOL-Library.Word" Word_32
begin

type_synonym machine_word_len = 32

definition word_bits :: nat
where
  ‹word_bits = LENGTH(machine_word_len)›

lemma word_bits_conv [code]:
  ‹word_bits = 32›
  by (simp add: word_bits_def)
```

The following two are numerals so they can be used as nats and words.

```
definition word_size_bits :: ‹'a :: numeral›
where
  ‹word_size_bits = 2›

definition word_size :: ‹'a :: numeral›
where
  ‹word_size = 4›

lemma n_less_word_bits:
  "(n < word_bits) = (n < 32)"
  by (simp add: word_bits_def word_size_def)

lemmas upper_bits_unset_is_l2p_32 = upper_bits_unset_is_l2p [where 'a=32,
folded word_bits_def]

lemmas le_2p_upper_bits_32 = le_2p_upper_bits [where 'a=32, folded word_bits_def]
lemmas le2p_bits_unset_32 = le2p_bits_unset[where 'a=32, folded word_bits_def]

lemmas unat_power_lower32 [simp] = unat_power_lower32'[folded word_bits_def]

lemmas word32_less_sub_le[simp] = word32_less_sub_le' [folded word_bits_def]

lemmas word32_power_less_1[simp] = word32_power_less_1'[folded word_bits_def]

lemma of_nat32_0:
  "⟦of_nat n = (0::word32); n < 2 ^ word_bits⟧ ⟹ n = 0"
  by (erule of_nat_0, simp add: word_bits_def)

lemmas unat_of_nat32 = unat_of_nat32'[folded word_bits_def]

lemmas word_power_nonzero_32 = word_power_nonzero [where 'a=32, folded
```

```
word_bits_def]

lemmas div_power_helper_32 = div_power_helper [where 'a=32, folded word_bits_def]

lemmas of_nat_less_pow_32 = of_nat_power [where 'a=32, folded word_bits_def]

lemmas unat_mask_word32 = unat_mask_word32'[folded word_bits_def]

end
```

# 28 32-Bit Machine Word Setup

```
theory Machine_Word_32
  imports Machine_Word_32_Basics More_Word Bit_Shifts_Infix_Syntax Rsplit
begin

context
  includes bit_operations_syntax
begin

type_synonym machine_word = ‹machine_word_len word›

lemma word_bits_len_of:
  ‹LENGTH(machine_word_len) = word_bits›
  by (simp add: word_bits_conv)

lemma word_bits_size:
  "size (w :: machine_word) = word_bits"
  by (simp add: word_bits_def word_size)

lemma word_bits_word_size_conv:
  ‹word_bits = word_size * 8›
  by (simp add: word_bits_def word_size_def)

lemma word_size_word_size_bits:
  ‹word_size = (2 :: 'a :: semiring_1) ^ word_size_bits›
  by (simp add: word_size_def word_size_bits_def)

lemma lt_word_bits_lt_pow:
  "sz < word_bits ⟹ sz < 2 ^ word_bits"
  by (simp add: word_bits_conv)

lemma if_then_1_else_0:
  "((if P then 1 else 0) = (0 :: machine_word)) = (¬ P)"
  by simp

lemma if_then_0_else_1:
  "((if P then 0 else 1) = (0 :: machine_word)) = (P)"
  by simp
```

```
lemmas if_then_simps = if_then_0_else_1 if_then_1_else_0

lemma bool_mask [simp]:
  ‹0 < x AND 1 ⟷ x AND 1 = 1› for x :: machine_word
  by (rule bool_mask') auto

lemma in_16_range:
  "0 ∈ S ⟹ r ∈ (λx. r + x * (16 :: machine_word)) ‘ S"
  "n - 1 ∈ S ⟹ (r + (16 * n - 16)) ∈ (λx :: machine_word. r + x * 16)
‘ S"
  by (clarsimp simp: image_def elim!: bexI[rotated])+

lemma le_step_down_word_3:
  fixes x :: machine_word
  shows "⟦x ≤ y; x ≠ y⟧ ⟹ x ≤ y - 1"
  by (fact le_step_down_word_2)

lemma shiftr_1:
  "(x::machine_word) >> 1 = 0 ⟹ x < 2"
  apply transfer
  apply (simp add: take_bit_drop_bit)
  apply (simp add: drop_bit_Suc)
  done

lemma Suc_unat_mask_div:
  "Suc (unat (mask sz div word_size :: machine_word)) = 2 ^ (min sz word_bits
- word_size_bits)"
  by (simp add: word_size_word_size_bits unat_drop_bit_eq unat_mask_eq
drop_bit_mask_eq Suc_mask_eq_exp
    flip: drop_bit_eq_div word_bits_conv)

lemma ucast_not_helper:
  fixes a::"8 word"
  assumes a: "a ≠ 0xFF"
  shows "ucast a ≠ (0xFF::machine_word)"
proof
  assume "ucast a = (0xFF::machine_word)"
  also
  have "(0xFF::machine_word) = ucast (0xFF::8 word)" by simp
  finally
  show False using a
    apply -
    apply (drule up_ucast_inj, simp)
    apply simp
    done
qed

lemma unat_less_2p_word_bits:
```

```
  "unat (x :: machine_word) < 2 ^ word_bits"
  apply (simp only: word_bits_def)
  apply (rule unat_lt2p)
  done

lemma unat_less_word_bits:
  fixes y :: machine_word
  shows "x < unat y ⟹ x < 2 ^ word_bits"
  unfolding word_bits_def
  by (rule order_less_trans [OF _ unat_lt2p])

lemma unat_mask_2_less_4:
  "unat (p AND mask 2 :: machine_word) < 4"
  by (rule unat_less_helper) (simp only: take_bit_eq_mod word_mod_less_divisor
flip: take_bit_eq_mask, simp add: word_mod_less_divisor)

lemma unat_mult_simple:
  ‹unat (x * y) = unat x * unat y›
    if ‹unat x * unat y < 2 ^ LENGTH(machine_word_len)›
    for x y :: machine_word
  using that by (simp flip: unat_mult_lem)

lemma upto_2_helper:
  "{0..<2 :: machine_word} = {0, 1}"
  by (safe; simp) unat_arith

lemma word_ge_min:
  ‹- (2 ^ (word_bits - 1)) ≤ sint x› for x :: machine_word
  using sint_ge [of x] by (simp add: word_bits_def)

lemma word_rsplit_0:
  "word_rsplit (0 :: machine_word) = replicate (word_bits div 8) (0 ::
8 word)"
  by (simp add: word_rsplit_def bin_rsplit_def word_bits_def word_size_def
Cons_replicate_eq)

lemma x_less_2_0_1:
  fixes x :: machine_word
  shows "x < 2 ⟹ x = 0 ∨ x = 1"
  by (rule x_less_2_0_1') auto

end

end
```

# 29 Words of Length 64

```
theory Word_64
  imports
```

```
    Word_Lemmas
    Word_Names
    Word_Syntax
    Rsplit
    More_Word_Operations
begin

context
  includes bit_operations_syntax
begin

lemma len64: "len_of (x :: 64 itself) = 64" by simp

lemma ucast_8_64_inj:
  "inj (ucast ::  8 word ⇒ 64 word)"
  by (rule down_ucast_inj) (clarsimp simp: is_down_def target_size source_size)

lemmas unat_power_lower64' = unat_power_lower[where 'a=64]

lemmas word64_less_sub_le' = word_less_sub_le[where 'a = 64]

lemmas word64_power_less_1' = word_power_less_1[where 'a = 64]

lemmas unat_of_nat64' = unat_of_nat_eq[where 'a=64]

lemmas unat_mask_word64' = unat_mask[where 'a=64]

lemmas word64_minus_one_le' = word_minus_one_le[where 'a=64]
lemmas word64_minus_one_le = word64_minus_one_le'[simplified]

lemma less_4_cases:
  "(x::word64) < 4 ⟹ x=0 ∨ x=1 ∨ x=2 ∨ x=3"
  apply clarsimp
  apply (drule word_less_cases, erule disjE, simp, simp)+
  done

lemma ucast_le_ucast_8_64:
  "(ucast x ≤ (ucast y :: word64)) = (x ≤ (y :: 8 word))"
  by (simp add: ucast_le_ucast)

lemma eq_2_64_0:
  "(2 ^ 64 :: word64) = 0"
  by simp

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"
  by (rule of_nat_n_less_equal_power_2, clarsimp simp: word_size)
```

```
lemma unat_ucast_10_64 :
  fixes x :: "10 word"
  shows "unat (ucast x :: word64) = unat x"
  by transfer simp

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)"
  by (simp_all add: word_size)

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)"
  by (simp_all add: of_nat_take_bit take_bit_word_eq_self unsigned_of_nat)

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)"
  by auto

lemmas signed_shift_guard_to_word_64
    = signed_shift_guard_to_word[OF word64_31_less(1-2)]
    signed_shift_guard_to_word[OF word64_31_less(3-4)]

lemma mask_step_down_64:
  ‹∃x. mask x = b› if ‹b && 1 = 1›
    and ‹∃x. x < 64 ∧ mask x = b >> 1› for b :: ‹64word›
proof -
  from ‹b && 1 = 1› have ‹odd b›
    by (auto simp add: mod_2_eq_odd and_one_eq)
  then have ‹b mod 2 = 1›
    using odd_iff_mod_2_eq_one by blast
  from ‹∃x. x < 64 ∧ mask x = b >> 1› obtain x where ‹x < 64› ‹mask
x = b >> 1› by blast
  then have ‹mask x = b div 2›
    using shiftr1_is_div_2 [of b] by simp
  with ‹b mod 2 = 1› have ‹2 * mask x + 1 = 2 * (b div 2) + b mod 2›
    by (simp only:)
  also have ‹... = b›
    by (simp add: mult_div_mod_eq)
  finally have ‹2 * mask x + 1 = b› .
  moreover have ‹mask (Suc x) = 2 * mask x + (1 :: 'a::len word)›
    by (simp add: mask_Suc_rec)
  ultimately show ?thesis
    by auto
qed

lemma unat_of_int_64:
  "⟦i ≥ 0; i ≤ 2 ^ 63⟧ ⟹ (unat ((of_int i)::sword64)) = nat i"
  by (simp add: unsigned_of_int nat_take_bit_eq take_bit_nat_eq_self)

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›
  using word_and_full_mask_simp [of x]
  by (simp add: numeral_eq_Suc mask_Suc_exp)

end

end
```

# 30 64 bit standard platform-specific word size and alignment.

```
theory Machine_Word_64_Basics
imports "HOL-Library.Word" Word_64
begin

type_synonym machine_word_len = 64

definition word_bits :: nat
where
  ‹word_bits = LENGTH(machine_word_len)›

lemma word_bits_conv [code]:
  ‹word_bits = 64›
  by (simp add: word_bits_def)
```

The following two are numerals so they can be used as nats and words.

```
definition word_size_bits :: ‹'a :: numeral›
where
  ‹word_size_bits = 3›

definition word_size :: ‹'a :: numeral›
where
  ‹word_size = 8›

lemma n_less_word_bits:
  "(n < word_bits) = (n < 64)"
  by (simp add: word_bits_def word_size_def)

lemmas upper_bits_unset_is_l2p_64 = upper_bits_unset_is_l2p [where 'a=64,
folded word_bits_def]

lemmas le_2p_upper_bits_64 = le_2p_upper_bits [where 'a=64, folded word_bits_def]
lemmas le2p_bits_unset_64 = le2p_bits_unset[where 'a=64, folded word_bits_def]

lemmas unat_power_lower64 [simp] = unat_power_lower64'[folded word_bits_def]

lemmas word64_less_sub_le[simp] = word64_less_sub_le' [folded word_bits_def]

lemmas word64_power_less_1[simp] = word64_power_less_1'[folded word_bits_def]

lemma of_nat64_0:
  "⟦of_nat n = (0::word64); n < 2 ^ word_bits⟧ ⟹ n = 0"
  by (erule of_nat_0, simp add: word_bits_def)

lemmas unat_of_nat64 = unat_of_nat64'[folded word_bits_def]

lemmas word_power_nonzero_64 = word_power_nonzero [where 'a=64, folded
```

```
word_bits_def]

lemmas div_power_helper_64 = div_power_helper [where 'a=64, folded word_bits_def]

lemmas of_nat_less_pow_64 = of_nat_power [where 'a=64, folded word_bits_def]

lemmas unat_mask_word64 = unat_mask_word64'[folded word_bits_def]

end
```

# 31 64-Bit Machine Word Setup

```
theory Machine_Word_64
imports Machine_Word_64_Basics More_Word Bit_Shifts_Infix_Syntax Rsplit
begin

context
  includes bit_operations_syntax
begin

type_synonym machine_word = ‹machine_word_len word›

lemma word_bits_len_of:
  ‹LENGTH(machine_word_len) = word_bits›
  by (simp add: word_bits_conv)

lemma word_bits_size:
  "size (w :: machine_word) = word_bits"
  by (simp add: word_bits_def word_size)

lemma word_bits_word_size_conv:
  ‹word_bits = word_size * 8›
  by (simp add: word_bits_def word_size_def)

lemma word_size_word_size_bits:
  ‹word_size = (2 :: 'a :: semiring_1) ^ word_size_bits›
  by (simp add: word_size_def word_size_bits_def)

lemma lt_word_bits_lt_pow:
  "sz < word_bits ⟹ sz < 2 ^ word_bits"
  by (simp add: word_bits_conv)

lemma if_then_1_else_0:
  "((if P then 1 else 0) = (0 :: machine_word)) = (¬ P)"
  by simp

lemma if_then_0_else_1:
  "((if P then 0 else 1) = (0 :: machine_word)) = (P)"
  by simp
```

```
lemmas if_then_simps = if_then_0_else_1 if_then_1_else_0

lemma bool_mask [simp]:
  ‹0 < x AND 1 ⟷ x AND 1 = 1› for x :: machine_word
  by (rule bool_mask') auto

lemma in_16_range:
  "0 ∈ S ⟹ r ∈ (λx. r + x * (16 :: machine_word)) ‘ S"
  "n - 1 ∈ S ⟹ (r + (16 * n - 16)) ∈ (λx :: machine_word. r + x * 16)
‘ S"
  by (clarsimp simp: image_def elim!: bexI[rotated])+

lemma le_step_down_word_3:
  fixes x :: machine_word
  shows "⟦x ≤ y; x ≠ y⟧ ⟹ x ≤ y - 1"
  by (fact le_step_down_word_2)

lemma shiftr_1:
  "(x::machine_word) >> 1 = 0 ⟹ x < 2"
  apply transfer
  apply (simp add: take_bit_drop_bit)
  apply (simp add: drop_bit_Suc)
  done

lemma Suc_unat_mask_div:
  "Suc (unat (mask sz div word_size :: machine_word)) = 2 ^ (min sz word_bits
- word_size_bits)"
  by (simp add: word_size_word_size_bits unat_drop_bit_eq unat_mask_eq
drop_bit_mask_eq Suc_mask_eq_exp
    flip: drop_bit_eq_div word_bits_conv)

lemma ucast_not_helper:
  fixes a::"8 word"
  assumes a: "a ≠ 0xFF"
  shows "ucast a ≠ (0xFF::machine_word)"
proof
  assume "ucast a = (0xFF::machine_word)"
  also
  have "(0xFF::machine_word) = ucast (0xFF::8 word)" by simp
  finally
  show False using a
    apply -
    apply (drule up_ucast_inj, simp)
    apply simp
    done
qed

lemma unat_less_2p_word_bits:
```

```
  "unat (x :: machine_word) < 2 ^ word_bits"
  apply (simp only: word_bits_def)
  apply (rule unat_lt2p)
  done

lemma unat_less_word_bits:
  fixes y :: machine_word
  shows "x < unat y ⟹ x < 2 ^ word_bits"
  unfolding word_bits_def
  by (rule order_less_trans [OF _ unat_lt2p])

lemma unat_mask_2_less_4:
  "unat (p AND mask 2 :: machine_word) < 4"
  by (rule unat_less_helper) (simp only: take_bit_eq_mod word_mod_less_divisor
flip: take_bit_eq_mask, simp add: word_mod_less_divisor)

lemma unat_mult_simple:
  ‹unat (x * y) = unat x * unat y›
    if ‹unat x * unat y < 2 ^ LENGTH(machine_word_len)›
    for x y :: machine_word
  using that by (simp flip: unat_mult_lem)

lemma upto_2_helper:
  "{0..<2 :: machine_word} = {0, 1}"
  by (safe; simp) unat_arith

lemma word_ge_min:
  ‹- (2 ^ (word_bits - 1)) ≤ sint x› for x :: machine_word
  using sint_ge [of x] by (simp add: word_bits_def)

lemma word_rsplit_0:
  "word_rsplit (0 :: machine_word) = replicate (word_bits div 8) (0 ::
8 word)"
  by (simp add: word_rsplit_def bin_rsplit_def word_bits_def word_size_def
Cons_replicate_eq)

lemma x_less_2_0_1:
  fixes x :: machine_word
  shows "x < 2 ⟹ x = 0 ∨ x = 1"
  by (rule x_less_2_0_1') auto

end

end
```

# 32 A short overview over bit operations and word types

## 32.1 Key principles

When formalizing bit operations, it is tempting to represent bit values as explicit lists over a binary type. This however is a bad idea, mainly due to the inherent ambiguities in representation concerning repeating leading bits.
Hence this approach avoids such explicit lists altogether following an algebraic path:

- Bit values are represented by numeric types: idealized unbounded bit values can be represented by type `int`, bounded bit values by quotient types over `int`, aka `'a word`.
- (A special case are idealized unbounded bit values ending in `0` which can be represented by type `nat` but only support a restricted set of operations).

The fundamental principles are developed in theory `HOL.Bit_Operations` (which is part of `Main`):

- Multiplication by `2` is a bit shift to the left and
- Division by `2` is a bit shift to the right.
- Concerning bounded bit values, iterated shifts to the left may result in eliminating all bits by shifting them all beyond the boundary. The property $2^{n} \neq 0$ represents that `n` is *not* beyond that boundary.
- The projection on a single bit is then `bit a n` $\longleftrightarrow$ `odd (a div` $2^{n}$`)`.
- This leads to the most fundamental properties of bit values:
  - Equality rule:

    ```
    a = b ⟷ (∀n. 2^n ≠ 0 ⟶ bit a n ⟷ bit b n)
    ```

  - Induction rule:

    ```
    ⟦⋀a. a div 2 = a ⟹ P a;
     ⋀a b. ⟦P a; (of_bool b + 2 * a) div 2 = a⟧ ⟹ P (of_bool b
    + 2 * a)⟧
    ⟹ P a
    ```

- Characteristic properties `bit (f x) n` $\longleftrightarrow$ `P x n` are available in fact collection `bit_simps`.

On top of this, the following generic operations are provided:

- Singleton nth bit: $2^n$
- Bit mask upto bit n: mask n = $2^n$ - 1
- Left shift: push_bit n a = a * $2^n$
- Right shift: drop_bit n a = a div $2^n$
- Truncation: take_bit n a = a mod $2^n$
- Bitwise negation: bit (NOT a) n $\longleftrightarrow$ $2^n \neq 0 \wedge \neg$ bit a n
- Bitwise conjunction: bit (a AND b) n $\longleftrightarrow$ bit a n $\wedge$ bit b n
- Bitwise disjunction: bit (a OR b) n $\longleftrightarrow$ bit a n $\vee$ bit b n
- Bitwise exclusive disjunction: bit (a XOR b) n $\longleftrightarrow$ bit a n $\neq$ bit b n
- Setting a single bit: semiring_bit_operations_class.set_bit n a = a OR push_bit n 1
- Unsetting a single bit: unset_bit n a = a AND NOT (push_bit n 1)
- Flipping a single bit: flip_bit n a = a XOR push_bit n 1
- Signed truncation, or modulus centered around 0:

  signed_take_bit n a = take_bit n a OR of_bool (bit a n) * NOT (mask n)

- (Bounded) conversion from and to a list of bits:

  horner_sum of_bool 2 (map (bit a) [0..<n]) = take_bit n a

Bit concatenation on int as given by

concat_bit n k l = take_bit n k OR push_bit n l

appears quite technical but is the logical foundation for the quite natural bit concatenation on 'a word (see below).

## 32.2 Core word theory

Proper word types are introduced in theory `HOL-Library.Word`, with the following specific operations:

- Standard arithmetic: `(+)`, `uminus`, `(-)`, `(*)`, `0`, `1`, numerals etc.
- Standard bit operations: see above.
- Conversion with unsigned interpretation of words:
  - `unsigned :: 'a::len word ⇒ 'b::semiring_1`
  - Important special cases as abbreviations:
    - `unat :: 'a::len word ⇒ nat`
    - `uint :: 'a::len word ⇒ int`
    - `ucast :: 'a::len word ⇒ 'b::len word`
- Conversion with signed interpretation of words:
  - `signed :: 'a::len word ⇒ 'b::ring_1`
  - Important special cases as abbreviations:
    - `sint :: 'a::len word ⇒ int`
    - `scast :: 'a::len word ⇒ 'b::len word`
- Operations with unsigned interpretation of words:
  - `a ≤ b ⟷ unat a ≤ unat b`
  - `a < b ⟷ unat a < unat b`
  - `unat (v div w) = unat v div unat w`
  - `unat (drop_bit n w) = drop_bit n (unat w)`
  - `unat (v mod w) = unat v mod unat w`
  - `x udvd y ⟷ unat x dvd unat y`
- Operations with signed interpretation of words:
  - `a ≤s b ⟷ sint a ≤ sint b`
  - `a <s b ⟷ sint a < sint b`
  - `sint (signed_drop_bit n w) = drop_bit n (sint w)`
- Rotation and reversal:
  - `word_rotl :: nat ⇒ 'a::len word ⇒ 'a word`

- `word_rotr :: nat ⇒ 'a::len word ⇒ 'a word`
- `word_roti :: int ⇒ 'a::len word ⇒ 'a word`
- `word_reverse :: 'a::len word ⇒ 'a word`

- Concatenation:

  `word_cat :: 'a::len word ⇒ 'b::len word ⇒ 'c::len word`

For proofs about words the following default strategies are applicable:

- Using bit extensionality (facts `bit_eq_iff`, `bit_word_eqI`; fact collection `bit_simps`).
- Using the `transfer` method.

## 32.3 More library theories

Note: currently, most theories listed here are hardly separate entities since they import each other in various ways. Always inspect them to understand what you pull in if you want to import one.

**Syntax** `Word_Lib.Syntax_Bundles` Bundles to provide alternative syntax for various bit operations.

`Word_Lib.Hex_Words` Printing word numerals as hexadecimal numerals.

`Word_Lib.Type_Syntax` Pretty type-sensitive syntax for cast operations.

`Word_Lib.Word_Syntax` Specific ASCII syntax for prominent bit operations on word.

**Proof tools** `Word_Lib.Norm_Words` Rewriting word numerals to normal forms.

`Word_Lib.Bitwise` Method `word_bitwise` decomposes word equalities and inequalities into bit propositions.

`Word_Lib.Bitwise_Signed` Method `word_bitwise_signed` decomposes word equalities and inequalities into bit propositions.

`Word_Lib.Word_EqI` Method `word_eqI_solve` decomposes word equalities and inequalities into bit propositions.

**Operations** `Word_Lib.Signed_Division_Word` Signed division on word:

- `(sdiv) :: 'a::len word ⇒ 'a word ⇒ 'a word`
- `(smod) :: 'a::len word ⇒ 'a word ⇒ 'a word`

`Word_Lib.Aligned`

- `is_aligned w n` $\longleftrightarrow$ $2^{\texttt{n}}$ `udvd w`

`Word_Lib.Least_significant_bit` The least significant bit as abbreviation `lsb` $\equiv$ $\lambda$`a. bit a 0`.

`Word_Lib.Most_significant_bit` The most significant bit:

- `msb k` $\longleftrightarrow$ `k < 0`
- `msb w` $\longleftrightarrow$ `sint w < 0`
- `msb w` $\longleftrightarrow$ `w <s 0`
- `msb w` $\longleftrightarrow$ `bit w (LENGTH('a) - Suc 0)`

`Word_Lib.Bit_Shifts_Infix_Syntax` Bit shifts decorated with infix syntax:

- `a << n = push_bit n a`
- `a >> n = drop_bit n a`
- `w >>> n = signed_drop_bit n w`

`Word_Lib.Next_and_Prev`

- `word_next w = (if w = - 1 then - 1 else w + 1)`
- `word_prev w = (if w = 0 then 0 else w - 1)`

`Word_Lib.Enumeration_Word` More on explicit enumeration of word types.

`Word_Lib.More_Word_Operations` Even more operations on word.

**Types** `Word_Lib.Signed_Words` Formal tagging of word types with a `signed` marker.

**Lemmas** `Word_Lib.More_Word` More lemmas on words.

`Word_Lib.Word_Lemmas` More lemmas on words, covering many other theories mentioned here.

**Words of popular lengths** .

`Word_Lib.Word_8` for 8-bit words.

`Word_Lib.Word_16` for 16-bit words.

`Word_Lib.Word_32` for 32-bit words.

`Word_Lib.Word_64` for 64-bit words. This theory is not part of `Word_Lib_Sumo`, because it shadows names from `Word_Lib.Word_32`. They can be used together, but then will have to use qualified names in applications.

`Word_Lib.Machine_Word_32` **and** `Word_Lib.Machine_Word_64` provide lemmas for 32-bit words and 64-bit words under the same name, which can help to organize applications relying on some form of genericity.

## 32.4 More library sessions

`Native_Word` Makes machine words and machine arithmetic available for code generation. It provides a common abstraction that hides the differences between the different target languages. The code generator maps these operations to the APIs of the target languages.

## 32.5 Legacy theories

The following theories contain material which has been factored out since it is not recommended to use it in new applications, mostly because matters can be expressed succinctly using already existing operations.

This section gives some indication how to migrate away from those theories. However theorem coverage may still be terse in some cases.

`Word_Lib.Word_Lib_Sumo` An entry point importing any relevant theory in that session. Intended for backward compatibility: start importing this theory when migrating applications to Isabelle2021, and later sort out what you really need. You may need to include `Word_Lib.Word_64` separately.

`Word_Lib.Generic_set_bit` A variant of a singleton bit operation: `Generic_set_bit.set_bit a n b = (if b then semiring_bit_operations_class.set_bit else unset_bit) n a`

`Word_Lib.Typedef_Morphisms` A low-level extension to HOL typedef providing conversions along type morphisms. The `transfer` method seems to be sufficient for most applications though.

`Word_Lib.Bit_Comprehension` Comprehension syntax for bit values over predicates `nat ⇒ bool`, for `'a word`; straightforward alternatives exist.

`Word_Lib.Bit_Comprehension_Int` Comprehension syntax for bit values over predicates `nat ⇒ bool`, for `int`; inherently non-computational.

`Word_Lib.Reversed_Bit_Lists` Representation of bit values as explicit list in *reversed* order.

This should rarely be necessary: the `bit` projection should be sufficient in most cases. In case explicit lists are needed, existing operations can be used:

```
horner_sum of_bool 2 (map (bit a) [0..<n]) = take_bit n a
```

`Word_Lib.Many_More` Collection of operations and theorems which are kept for backward compatibility and not used in other theories in session `Word_Lib`. They are used in applications of `Word_Lib`, but should be migrated to there.

# 33 Changelog

### Changes since AFP 2025

- `Generic_set_bit.set_bit` is now a regular derived operation without any special treatment.

### Changes since AFP 2024

- Theory `Strict_part_mono` is not part of text`Word_Lib_Sumo` any longer.
- Session `Native_Word`: Fact aliases `word_sdiv_def` and `word_smod_def` are gone, use `sdiv_word_def` and `smod_word_def` instead.
- Session `Native_Word`: Removed abbreviation `word_of_integer`.

### Changes since AFP 2022

- Theory `Word_Lib.Ancient_Numeral` has been removed from session.
- Bit comprehension syntax for `int` moved to separate theory `Word_Lib.Bit_Comprehension_In`
- Operation `lsb` ≡ `λa. bit a 0` turned into abbreviation or `bit _ 0`.

### Changes since AFP 2021

- Theory `Word_Lib.Ancient_Numeral` is not part of `Word_Lib.Word_Lib_Sumo` any longer.
- Infix syntax for `(AND)`, `(OR)`, `(XOR)` organized in syntax bundle `bit_operations_syntax`.
- Abbreviation `max_word` ≡ `- 1` moved from distribution into theory `Word_Lib.Legacy_Aliases`.
- Operation `test_bit` replaced by input abbreviation `test_bit` ≡ `bit`.
- Abbreviations `bin_nth` ≡ `bit`, `bin_last` ≡ `odd`, `bin_rest` ≡ `λw. w div 2`, `bintrunc` ≡ `take_bit`, `sbintrunc` ≡ `signed_take_bit`, `norm_sint` ≡ `λn. signed_take_bit (n - 1)`, `bin_cat` ≡ `λk n l. concat_bit n l k` moved into theory `Word_Lib.Legacy_Aliases`.

- Operations `bshiftr1` ≡ λ`b w. w div 2 OR push_bit (LENGTH('a) - Suc 0) (of_bool b)`, `setBit` ≡ λ`w n. semiring_bit_operations_class.set_bit n w`, `clearBit` ≡ λ`w n. unset_bit n w` moved from distribution into theory `Word_Lib.Legacy_Aliases` and replaced by input abbreviations.
- Operations `shiftl1`, `shiftr1`, `sshiftr1` moved here from distribution.
- Operation `complement` replaced by input abbreviation `complement` ≡ `NOT`.

## References

[1] D. Leijen. Division and modulus for computer scientists. 2001.