Finite Machine Word Library

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

May 26, 2024

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 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

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
```

2 Dedicated operation for the most significant bit

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

class msb =
  fixes msb :: ‹'a ⇒ bool›

instantiation int :: msb
begin

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

instance ..

end

lemma msb_bin_rest [simp]: "msb (x div 2) = msb x"
  for x :: int
  by (simp add: msb_int_def)

context
  includes bit_operations_syntax
begin

lemma int_msb_and [simp]: "msb ((x :: int) AND y) ⟷ msb x ∧ msb y"
by(simp add: msb_int_def)

lemma int_msb_or [simp]: "msb ((x :: int) OR y) ⟷ msb x ∨ msb y"
by(simp add: msb_int_def)

lemma int_msb_xor [simp]: "msb ((x :: int) XOR y) ⟷ msb x ≠ msb y"
by(simp add: msb_int_def)

lemma int_msb_not [simp]: "msb (NOT (x :: int)) ⟷ ¬ msb x"
by(simp add: msb_int_def not_less)

end

lemma msb_shiftl [simp]: "msb ((x :: int) << n) ⟷ msb x"
  by (simp add: msb_int_def shiftl_def)

lemma msb_shiftr [simp]: "msb ((x :: int) >> r) ⟷ msb x"
  by (simp add: msb_int_def shiftr_def)
```

```
lemma msb_0 [simp]: "msb (0 :: int) = False"
by(simp add: msb_int_def)

lemma msb_1 [simp]: "msb (1 :: int) = False"
by(simp add: msb_int_def)

lemma msb_numeral [simp]:
  "msb (numeral n :: int) = False"
  "msb (- numeral n :: int) = True"
by(simp_all add: msb_int_def)

instantiation word :: (len) msb
begin

definition msb_word :: ‹'a word ⇒ bool›
  where msb_word_iff_bit: ‹msb w ⟷ bit w (LENGTH('a) - Suc 0)› for
w :: ‹'a::len word›

instance ..

end

lemma msb_word_eq:
  ‹msb w ⟷ bit w (LENGTH('a) - 1)› for w :: ‹'a::len word›
  by (simp add: msb_word_iff_bit)

lemma word_msb_sint: "msb w ⟷ sint w < 0"
  by (simp add: msb_word_eq bit_last_iff)

lemma msb_word_iff_sless_0:
  ‹msb w ⟷ w <s 0›
  by (simp add: word_msb_sint word_sless_alt)

lemma msb_word_of_int:
  "msb (word_of_int x::'a::len word) = bit x (LENGTH('a) - 1)"
  by (simp add: msb_word_iff_bit bit_simps)

lemma word_msb_numeral [simp]:
  "msb (numeral w::'a::len word) = bit (numeral w :: int) (LENGTH('a)
- 1)"
  unfolding word_numeral_alt by (rule msb_word_of_int)

lemma word_msb_neg_numeral [simp]:
  "msb (- numeral w::'a::len word) = bit (- numeral w :: int) (LENGTH('a)
- 1)"
  unfolding word_neg_numeral_alt by (rule msb_word_of_int)

lemma word_msb_0 [simp]: "¬ msb (0::'a::len word)"
```

```
  by (simp add: msb_word_iff_bit)

lemma word_msb_1 [simp]: "msb (1::'a::len word) ⟷ LENGTH('a) = 1"
  by (simp add: msb_word_iff_bit le_Suc_eq)

lemma word_msb_nth: "msb w = bit (uint w) (LENGTH('a) - 1)"
  for w :: "'a::len word"
  by (simp add: msb_word_iff_bit bit_simps)

lemma msb_nth: "msb w = bit w (LENGTH('a) - 1)"
  for w :: "'a::len word"
  by (fact msb_word_eq)

lemma word_msb_n1 [simp]: "msb (-1::'a::len word)"
  by (simp add: msb_word_eq not_le)

lemma msb_shift: "msb w ⟷ w >> LENGTH('a) - 1 ≠ 0"
  for w :: "'a::len word"
  by (simp add: drop_bit_eq_zero_iff_not_bit_last msb_word_eq shiftr_def)

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

lemma word_sint_msb_eq: "sint x = uint x - (if msb x then 2 ^ size x
else 0)"
  apply (cases ‹LENGTH('a)›)
  apply (simp_all add: msb_word_iff_bit word_size)
  apply transfer
  apply (simp add: signed_take_bit_eq_take_bit_minus)
  done

lemma word_sle_msb_le: "x <=s y ⟷ (msb y ⟶ msb x) ∧ ((msb x ∧ ¬
msb y) ∨ x ≤ y)"
  apply (simp add: word_sle_eq word_sint_msb_eq word_size word_le_def)
  apply safe
   apply (rule order_trans[OF _ uint_ge_0])
   apply (simp add: order_less_imp_le)
  apply (erule notE[OF leD])
  apply (rule order_less_le_trans[OF _ uint_ge_0])
  apply simp
  done

lemma word_sless_msb_less: "x <s y ⟷ (msb y ⟶ msb x) ∧ ((msb x ∧
¬ msb y) ∨ x < y)"
  by (auto simp add: word_sless_eq word_sle_msb_le)

lemma not_msb_from_less:
  "(v :: 'a word) < 2 ^ (LENGTH('a :: len) - 1) ⟹ ¬ msb v"
  apply (clarsimp simp add: msb_nth)
  apply (drule less_mask_eq)
```

```
  apply (drule word_eqD, drule(1) iffD2)
  apply (simp add: bit_simps)
  done

lemma sint_eq_uint:
  "¬ msb x ⟹ sint x = uint x"
  apply (cases ‹LENGTH('a)›)
  apply (simp_all add: msb_word_iff_bit)
  apply transfer
  apply (simp add: signed_take_bit_eq_take_bit_minus)
  done

lemma scast_eq_ucast:
  "¬ msb x ⟹ scast x = ucast x"
  apply (cases ‹LENGTH('a)›)
  apply simp
  apply (rule bit_word_eqI)
  apply (auto simp add: bit_signed_iff bit_unsigned_iff min_def msb_word_eq)
  apply (erule notE)
  apply (metis le_less_Suc_eq test_bit_bin)
  done

lemma msb_ucast_eq:
    "LENGTH('a) = LENGTH('b) ⟹
         msb (ucast x :: ('a::len) word) = msb (x :: ('b::len) word)"
  by (simp add: msb_word_eq bit_simps)

lemma msb_big:
  ‹msb a ⟷ 2 ^ (LENGTH('a) - Suc 0) ≤ a›
  for a :: ‹'a::len word›
  using bang_is_le [of a ‹LENGTH('a) - Suc 0›]
  apply (auto simp add: msb_nth word_le_not_less)
  apply (rule ccontr)
  apply (erule notE)
  apply (rule ccontr)
  apply (clarsimp simp: not_less)
  apply (subgoal_tac "a = take_bit (LENGTH('a) - Suc 0) a")
   apply (cut_tac and_mask_less' [where w=a and n="LENGTH('a) - Suc 0"])
    apply auto
  apply (simp flip: take_bit_eq_mask)
  apply (rule sym)
  apply (simp add: take_bit_eq_self_iff_drop_bit_eq_0 drop_bit_eq_zero_iff_not_bit_last)
  done

instantiation integer :: msb
begin

context
  includes integer.lifting
```

begin

lift_definition msb_integer :: ‹integer ⇒ bool› **is** msb .

instance ..

end

end

end

3 Operation variant for the least significant bit

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

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

instantiation int :: lsb
begin

definition lsb_int :: ‹int ⇒ bool›
  where ‹lsb i = bit i 0› **for** i :: int

instance
  by standard (simp add: fun_eq_iff lsb_int_def bit_0)

end

lemma bin_last_conv_lsb: "odd = (lsb :: int ⇒ bool)"
  by (simp add: lsb_odd)

lemma int_lsb_numeral [simp]:
  "lsb (0 :: int) = False"
  "lsb (1 :: int) = True"
  "lsb (Numeral1 :: int) = True"
  "lsb (- 1 :: int) = True"
  "lsb (- Numeral1 :: int) = True"
  "lsb (numeral (num.Bit0 w) :: int) = False"
  "lsb (numeral (num.Bit1 w) :: int) = True"
  "lsb (- numeral (num.Bit0 w) :: int) = False"
  "lsb (- numeral (num.Bit1 w) :: int) = True"
  by (simp_all add: lsb_int_def bit_0)

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

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

instance
  apply standard
  apply (simp add: fun_eq_iff word_lsb_def)
  apply transfer apply simp
  done

end

lemma lsb_word_eq:
  ‹lsb = (odd :: 'a word ⇒ bool)› for w :: ‹'a::len word›
  by (fact lsb_odd)

lemma word_lsb_alt: "lsb w = bit w 0"
  for w :: "'a::len word"
  by (simp add: lsb_word_eq bit_0)

lemma word_lsb_1_0 [simp]: "lsb (1::'a::len word) ∧ ¬ lsb (0::'b::len
word)"
  unfolding word_lsb_def by simp

lemma word_lsb_int: "lsb w ⟷ uint w mod 2 = 1"
  apply (simp add: lsb_odd flip: odd_iff_mod_2_eq_one)
  apply transfer
  apply simp
  done

lemmas word_ops_lsb = lsb0 [unfolded word_lsb_alt]

lemma word_lsb_numeral [simp]:
  "lsb (numeral bin :: 'a::len word) ⟷ odd (numeral bin :: int)"
  by (simp only: lsb_odd, transfer) rule

lemma word_lsb_neg_numeral [simp]:
  "lsb (- numeral bin :: 'a::len word) ⟷ odd (- numeral bin :: int)"
  by (simp only: lsb_odd, transfer) rule

lemma word_lsb_nat:"lsb w = (unat w mod 2 = 1)"
  apply (simp add: word_lsb_def Groebner_Basis.algebra(31))
  apply transfer
  apply (simp add: even_nat_iff)
  done
```

```
instantiation integer :: lsb
begin

context
  includes integer.lifting
begin

lift_definition lsb_integer :: ‹integer ⇒ bool› is lsb .

instance
  by (standard; transfer) (fact lsb_odd)

end

end

end
```

4 Operation variant for setting and unsetting bits

```
theory Generic_set_bit
  imports
    "HOL-Library.Word"
    Most_significant_bit
begin

class set_bit = semiring_bits +
  fixes set_bit :: ‹'a ⇒ nat ⇒ bool ⇒ 'a›
  assumes bit_set_bit_iff_2n:
    ‹bit (set_bit a m b) n ⟷
      (if m = n then b else bit a n) ∧ 2 ^ n ≠ 0›

lemmas bit_set_bit_iff[bit_simps] = bit_set_bit_iff_2n[simplified fold_possible_bit
simp_thms]

lemma set_bit_eq:
  ‹set_bit a n b = (if b then Bit_Operations.set_bit else unset_bit) n
a›
  for a :: ‹'a::{ring_bit_operations, set_bit}›
  by (rule bit_eqI) (simp add: bit_simps)

instantiation int :: set_bit
begin

definition set_bit_int :: ‹int ⇒ nat ⇒ bool ⇒ int›
  where ‹set_bit_int i n b = (if b then Bit_Operations.set_bit else Bit_Operations.unset_b
n i›

instance
```

```
  by standard (simp_all add: set_bit_int_def bit_simps)

end

context
  includes bit_operations_syntax
begin

lemma fixes i :: int
  shows int_set_bit_True_conv_OR [code]: "Generic_set_bit.set_bit i n
True = i OR push_bit n 1"
  and int_set_bit_False_conv_NAND [code]: "Generic_set_bit.set_bit i n
False = i AND NOT (push_bit n 1)"
  and int_set_bit_conv_ops: "Generic_set_bit.set_bit i n b = (if b then
i OR (push_bit n 1) else i AND NOT (push_bit n 1))"
  by (simp_all add: bit_eq_iff) (auto simp add: bit_simps)

end

instantiation word :: (len) set_bit
begin

definition set_bit_word :: ‹'a word ⇒ nat ⇒ bool ⇒ 'a word›
  where set_bit_unfold: ‹set_bit w n b = (if b then Bit_Operations.set_bit
n w else unset_bit n w)›
  for w :: ‹'a::len word›

instance
  by standard (auto simp add: set_bit_unfold bit_simps dest: bit_imp_le_length)

end

lemma bit_set_bit_word_iff [bit_simps]:
  ‹bit (set_bit w m b) n ⟷ (if m = n then n < LENGTH('a) ∧ b else bit
w n)›
  for w :: ‹'a::len word›
  by (auto simp add: bit_simps dest: bit_imp_le_length)

lemma test_bit_set_gen:
  "bit (set_bit w n x) m ⟷ (if m = n then n < size w ∧ x else bit w
m)"
  for w :: "'a::len word"
  by (simp add: bit_set_bit_word_iff word_size)

lemma test_bit_set:
  "bit (set_bit w n x) n ⟷ n < size w ∧ x"
  for w :: "'a::len word"
  by (auto simp add: bit_simps word_size)
```

```
lemma word_set_nth: "set_bit w n (bit w n) = w"
  for w :: "'a::len word"
  by (rule bit_word_eqI) (simp add: bit_simps)

lemma word_set_set_same [simp]: "set_bit (set_bit w n x) n y = set_bit
w n y"
  for w :: "'a::len word"
  by (rule word_eqI) (simp add : test_bit_set_gen word_size)

lemma word_set_set_diff:
  fixes w :: "'a::len word"
  assumes "m ≠ n"
  shows "set_bit (set_bit w m x) n y = set_bit (set_bit w n y) m x"
  by (rule word_eqI) (auto simp: test_bit_set_gen word_size assms)

lemma word_set_nth_iff: "set_bit w n b = w ⟷ bit w n = b ∨ n ≥ size
w"
  for w :: "'a::len word"
  apply (rule iffI)
   apply (rule disjCI)
   apply (drule word_eqD)
   apply (erule sym [THEN trans])
   apply (simp add: test_bit_set)
  apply (erule disjE)
   apply clarsimp
  apply (rule word_eqI)
   apply (clarsimp simp add : test_bit_set_gen)
   apply (auto simp add: word_size)
  apply (rule bit_eqI)
  apply (simp add: bit_simps)
  done

lemma word_clr_le: "w ≥ set_bit w n False"
  for w :: "'a::len word"
  apply (simp add: set_bit_unfold)
  apply transfer
  apply (simp add: take_bit_unset_bit_eq unset_bit_less_eq)
  done

lemma word_set_ge: "w ≤ set_bit w n True"
  for w :: "'a::len word"
  apply (simp add: set_bit_unfold)
  apply transfer
  apply (simp add: take_bit_set_bit_eq set_bit_greater_eq)
  done

lemma set_bit_beyond:
  "size x ≤ n ⟹ set_bit x n b = x" for x :: "'a :: len word"
  by (simp add: word_set_nth_iff)
```

```
lemma one_bit_shiftl: "set_bit 0 n True = (1 :: 'a :: len word) << n"
  apply (rule word_eqI)
  apply (auto simp add: word_size bit_simps)
  done

lemma one_bit_pow: "set_bit 0 n True = (2 :: 'a :: len word) ^ n"
  by (simp add: one_bit_shiftl shiftl_def)

instantiation integer :: set_bit
begin

context
  includes integer.lifting
begin

lift_definition set_bit_integer :: ‹integer ⇒ nat ⇒ bool ⇒ integer› is
set_bit .

instance
  by (standard; transfer) (simp add: bit_simps)

end

end

end
```

5 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
```

6 Bitwise Operations on integers

```
theory Bits_Int
  imports
    "Word_Lib.Most_significant_bit"
    "Word_Lib.Least_significant_bit"
    "Word_Lib.Generic_set_bit"
    "Word_Lib.Bit_Comprehension"
begin
```

6.1 Implicit bit representation of `int`

```
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
```

6.2 Bit projection

```
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)"
  apply (induct k arbitrary: n)
   apply clarsimp
  apply clarsimp
  apply (simp only: bit_Suc [symmetric] add_Suc)
  done

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)+
```

6.3 Truncating

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

lemma bin_sign_simps [simp]:
  "bin_sign 0 = 0"
  "bin_sign 1 = 0"
  "bin_sign (- 1) = - 1"
  "bin_sign (numeral k) = 0"
  "bin_sign (- numeral k) = -1"
  by (simp_all add: bin_sign_def)

lemma bin_sign_rest [simp]: "bin_sign ((λk::int. k div 2) w) = bin_sign
w"
  by (simp add: bin_sign_def)

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 sign_bintr: "bin_sign ((take_bit :: nat ⇒ int ⇒ int) n w) = 0"
  by (simp add: bin_sign_def)

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 bin_sign_lem: "(bin_sign ((signed_take_bit :: nat ⇒ int ⇒ int)
n bin) = -1) = bit bin n"
  by (simp add: bin_sign_def)

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 (cases n) simp_all

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 (cases n) simp_all

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"
  apply (rule iffI)
   apply (rule box_equals [OF _ sbintrunc_bintrunc sbintrunc_bintrunc])
   apply simp
  apply (rule box_equals [OF _ bintrunc_sbintrunc bintrunc_sbintrunc])
  apply simp
  done
```

```
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 (cases n) (simp_all add: bin_sbin_eq_iff)

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 :: nat ⇒ int ⇒ int) n = (λw.
(w + 2 ^ n) mod 2 ^ Suc n - 2 ^ n :: int)"
  by (rule ext) (simp add : sbintrunc_mod2p)

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 only: fun.set_map range_bintrunc)
    apply (auto simp add: image_iff)
    apply presburger
    done
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 sign_Pls_ge_0: "bin_sign bin = 0 ⟷ bin ≥ 0"
  for bin :: int
  by (simp add: bin_sign_def)

lemma sign_Min_lt_0: "bin_sign bin = -1 ⟷ bin < 0"
  for bin :: int
  by (simp add: bin_sign_def)

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: "f ∘ g ∘ f = g ∘ f ⟹ f ∘ (g ∘ f) ^^ n = g ^^ n ∘ f"
  apply (rule ext)
  apply (induct_tac n)
   apply (simp_all (no_asm))
  apply (drule fun_cong)
  apply (unfold o_def)
  apply (erule trans)
  apply simp
  done

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]

6.4 Splitting and concatenation

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

lemma [code]:
  "bin_split (Suc n) w = (let (w1, w2) = bin_split n (w div 2) in (w1, of_bool (odd w) + 2 * w2))"
  "bin_split 0 w = (w, 0)"
  by (simp_all add: drop_bit_Suc take_bit_Suc mod_2_eq_odd)

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_sign_cat: "bin_sign ((λk n l. concat_bit n l k) x n y) = bin_sign
x"
proof -
  have ‹0 ≤ x› if ‹0 ≤ x * 2 ^ n + y mod 2 ^ n›
  proof -
    have ‹y mod 2 ^ n < 2 ^ n›
      using pos_mod_bound [of ‹2 ^ n› y] by simp
    then have ‹¬ y mod 2 ^ n ≥ 2 ^ n›
      by (simp add: less_le)
    with that have ‹x ≠ - 1›
      by auto
    have *: ‹- 1 ≤ (- (y mod 2 ^ n)) div 2 ^ n›
      by (simp add: zdiv_zminus1_eq_if)
    from that have ‹- (y mod 2 ^ n) ≤ x * 2 ^ n›
      by simp
    then have ‹(- (y mod 2 ^ n)) div 2 ^ n ≤ (x * 2 ^ n) div 2 ^ n›
      using zdiv_mono1 zero_less_numeral zero_less_power by blast
    with * have ‹- 1 ≤ x * 2 ^ n div 2 ^ n› by simp
    with ‹x ≠ - 1› show ?thesis
      by simp
  qed
  then show ?thesis
    by (simp add: bin_sign_def not_le not_less bin_cat_eq_push_bit_add_take_bit
push_bit_eq_mult take_bit_eq_mod)
qed

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)

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

lemma bin_rcat_eq_foldl:
  ‹bin_rcat n = foldl (λu v. (λk n l. concat_bit n l k) u n v) 0›
proof
  fix ks :: ‹int list›
  show ‹bin_rcat n ks = foldl (λu v. (λk n l. concat_bit n l k) u n v)
0 ks›
    by (induction ks rule: rev_induct)
      (simp_all add: bin_rcat_def concat_bit_eq push_bit_eq_mult)
```

qed

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

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

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

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

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

lemma bin_nth_cat:
  "(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_nth_split:
  "bin_split n c = (a, b) ⟹
    (∀k. (bit :: int ⇒ nat ⇒ bool) a k = (bit :: int ⇒ nat ⇒ bool)
c (n + k)) ∧
    (∀k. (bit :: int ⇒ nat ⇒ bool) b k = (k < n ∧ (bit :: int ⇒ nat
⇒ bool) c k))"
  by (auto simp add: bin_nth_drop_bit_iff bin_nth_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 bin_eqI) (auto simp: bin_nth_cat nth_bintr)

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 split_bintrunc: "bin_split n c = (a, b) ⟹ b = (take_bit :: nat
⇒ int ⇒ int) n c"
  by simp

lemma bin_cat_split: "bin_split n w = (u, v) ⟹ w = (λk n l. concat_bit
n l k) u n v"
  by (auto simp add: bin_cat_eq_push_bit_add_take_bit bits_ident)

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_split_cat: "bin_split n ((λk n l. concat_bit n l k) v n w)
= (v, (take_bit :: nat ⇒ int ⇒ int) n w)"
  by (simp add: drop_bit_bin_cat_eq take_bit_bin_cat_eq)

lemma bin_split_zero [simp]: "bin_split n 0 = (0, 0)"
  by simp

lemma bin_split_minus1 [simp]:
  "bin_split n (- 1) = (- 1, (take_bit :: nat ⇒ int ⇒ int) n (- 1))"
  by simp
```

```
lemma bin_split_trunc:
  "bin_split (min m n) c = (a, b) ⟹
    bin_split n ((take_bit :: nat ⇒ int ⇒ int) m c) = ((take_bit ::
nat ⇒ int ⇒ int) (m - n) a, b)"
  apply (induct n arbitrary: m b c, clarsimp)
  apply (simp add: bin_rest_trunc Let_def split: prod.split_asm)
  apply (case_tac m)
   apply (auto simp: Let_def drop_bit_Suc take_bit_Suc mod_2_eq_odd split:
prod.split_asm)
  done

lemma bin_split_trunc1:
  "bin_split n c = (a, b) ⟹
    bin_split n ((take_bit :: nat ⇒ int ⇒ int) m c) = ((take_bit ::
nat ⇒ int ⇒ int) (m - n) a, (take_bit :: nat ⇒ int ⇒ int) m b)"
  apply (induct n arbitrary: m b c, clarsimp)
  apply (simp add: bin_rest_trunc Let_def split: prod.split_asm)
  apply (case_tac m)
   apply (auto simp: Let_def drop_bit_Suc take_bit_Suc mod_2_eq_odd split:
prod.split_asm)
  done

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_split_num: "bin_split n b = (b div 2 ^ n, b mod 2 ^ n)"
  by (simp add: drop_bit_eq_div take_bit_eq_mod)

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

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

lemmas rsplit_aux_simp1s = rsplit_aux_simps [THEN th_if_simp1]

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

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

lemma bin_rsplit_aux_append: "bin_rsplit_aux n m c (bs @ cs) = bin_rsplit_aux
```

```
n m c bs @ cs"
  apply (induct n m c bs rule: bin_rsplit_aux.induct)
  apply (subst bin_rsplit_aux.simps)
  apply (subst bin_rsplit_aux.simps)
  apply (clarsimp split: prod.split)
  done

lemma bin_rsplitl_aux_append: "bin_rsplitl_aux n m c (bs @ cs) = bin_rsplitl_aux
n m c bs @ cs"
  apply (induct n m c bs rule: bin_rsplitl_aux.induct)
  apply (subst bin_rsplitl_aux.simps)
  apply (subst bin_rsplitl_aux.simps)
  apply (clarsimp split: prod.split)
  done

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

lemmas rsplit_def_auxs = bin_rsplit_def bin_rsplitl_def

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

lemma bin_split_minus: "0 < n ⟹ bin_split (Suc (n - 1)) w = bin_split
n w"
  by auto

lemma bin_split_pred_simp [simp]:
  "(0::nat) < numeral bin ⟹
    bin_split (numeral bin) w =
      (let (w1, w2) = bin_split (numeral bin - 1) ((λk::int. k div 2)
w)
       in (w1, of_bool (odd w) + 2 * w2))"
  by (simp add: take_bit_rec drop_bit_rec mod_2_eq_odd)

lemma bin_rsplit_aux_simp_alt:
  "bin_rsplit_aux n m c bs =
    (if m = 0 ∨ n = 0 then bs
     else let (a, b) = bin_split n c in bin_rsplit n (m - n, a) @ b #
bs)"
  apply (simp add: bin_rsplit_aux.simps [of n m c bs])
  apply (subst rsplit_aux_alts)
  apply (simp add: bin_rsplit_def)
  done

lemmas bin_rsplit_simp_alt =
  trans [OF bin_rsplit_def bin_rsplit_aux_simp_alt]

lemmas bthrs = bin_rsplit_simp_alt [THEN [2] trans]
```

```
lemma bin_rsplit_size_sign' [rule_format]:
  "n > 0 ⟹ rev sw = bin_rsplit n (nw, w) ⟹ ∀v∈set sw. (take_bit ::
nat ⇒ int ⇒ int) n v = v"
  apply (induct sw arbitrary: nw w)
   apply clarsimp
  apply clarsimp
  apply (drule bthrs)
  apply (simp (no_asm_use) add: Let_def split: prod.split_asm if_split_asm)
  apply clarify
  apply simp
  done

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

lemma bin_nth_rsplit [rule_format] :
  "n > 0 ⟹ m < n ⟹
    ∀w k nw.
      rev sw = bin_rsplit n (nw, w) ⟶
      k < size sw ⟶ (bit :: int ⇒ nat ⇒ bool) (sw ! k) m = (bit ::
int ⇒ nat ⇒ bool) w (k * n + m)"
  apply (induct sw)
   apply clarsimp
  apply clarsimp
  apply (drule bthrs)
  apply (simp (no_asm_use) add: Let_def split: prod.split_asm if_split_asm)
  apply (erule allE, erule impE, erule exI)
  apply (case_tac k)
   apply clarsimp
   prefer 2
   apply clarsimp
   apply (erule allE)
   apply (erule (1) impE)
   apply (simp add: bit_drop_bit_eq ac_simps)
  apply (simp add: bit_take_bit_iff ac_simps)
  done

lemma bin_rsplit_all: "0 < nw ⟹ nw ≤ n ⟹ bin_rsplit n (nw, w) =
[(take_bit :: nat ⇒ int ⇒ int) n w]"
  by (auto simp: bin_rsplit_def rsplit_aux_simp2ls split: prod.split dest!:
split_bintrunc)

lemma bin_rsplit_l [rule_format]:
  "∀bin. bin_rsplitl n (m, bin) = bin_rsplit n (m, (take_bit :: nat ⇒
int ⇒ int) m bin)"
  apply (rule_tac a = "m" in wf_less_than [THEN wf_induct])
  apply (simp (no_asm) add: bin_rsplitl_def bin_rsplit_def)
  apply (rule allI)
```

```
  apply (subst bin_rsplitl_aux.simps)
  apply (subst bin_rsplit_aux.simps)
  apply (clarsimp simp: Let_def split: prod.split)
  apply (simp add: ac_simps)
  apply (subst rsplit_aux_alts(1))
  apply (subst rsplit_aux_alts(2))
  apply clarsimp
  unfolding bin_rsplit_def bin_rsplitl_def
  apply (simp add: drop_bit_take_bit)
  apply (case_tac ‹x < n›)
  apply (simp_all add: not_less min_def)
  done

lemma bin_rsplit_rcat [rule_format]:
  "n > 0 ⟶ bin_rsplit n (n * size ws, bin_rcat n ws) = map ((take_bit
:: nat ⇒ int ⇒ int) n) ws"
  apply (unfold bin_rsplit_def bin_rcat_eq_foldl)
  apply (rule_tac xs = ws in rev_induct)
   apply clarsimp
  apply clarsimp
  apply (subst rsplit_aux_alts)
  apply (simp add: drop_bit_bin_cat_eq take_bit_bin_cat_eq)
  done

lemma bin_rsplit_aux_len_le [rule_format] :
  "∀ws m. n ≠ 0 ⟶ ws = bin_rsplit_aux n nw w bs ⟶
    length ws ≤ m ⟷ nw + length bs * n ≤ m * n"
proof -
  have *: R
    if d: "i ≤ j ∨ m < j'"
    and R1: "i * k ≤ j * k ⟹ R"
    and R2: "Suc m * k' ≤ j' * k' ⟹ R"
    for i j j' k k' m :: nat and R
    using d
    apply safe
    apply (rule R1, erule mult_le_mono1)
    apply (rule R2, erule Suc_le_eq [THEN iffD2 [THEN mult_le_mono1]])
    done
  have **: "0 < sc ⟹ sc - n + (n + lb * n) ≤ m * n ⟷ sc + lb * n
≤ m * n"
    for sc m n lb :: nat
    apply safe
     apply arith
    apply (case_tac "sc ≥ n")
     apply arith
    apply (insert linorder_le_less_linear [of m lb])
    apply (erule_tac k=n and k'=n in *)
     apply arith
    apply simp
```

```
    done
  show ?thesis
    apply (induct n nw w bs rule: bin_rsplit_aux.induct)
    apply (subst bin_rsplit_aux.simps)
    apply (simp add: ** Let_def split: prod.split)
    done
qed

lemma bin_rsplit_len_le: "n ≠ 0 ⟶ ws = bin_rsplit n (nw, w) ⟶ length
ws ≤ m ⟷ nw ≤ m * n"
  by (auto simp: bin_rsplit_def bin_rsplit_aux_len_le)

lemma bin_rsplit_aux_len:
  "n ≠ 0 ⟹ length (bin_rsplit_aux n nw w cs) = (nw + n - 1) div n +
length cs"
  apply (induct n nw w cs rule: bin_rsplit_aux.induct)
  apply (subst bin_rsplit_aux.simps)
  apply (clarsimp simp: Let_def split: prod.split)
  apply (erule thin_rl)
  apply (case_tac m)
   apply simp
  apply (case_tac "m ≤ n")
   apply (auto simp add: div_add_self2)
  done

lemma bin_rsplit_len: "n ≠ 0 ⟹ length (bin_rsplit n (nw, w)) = (nw
+ n - 1) div n"
  by (auto simp: bin_rsplit_def bin_rsplit_aux_len)

lemma bin_rsplit_aux_len_indep:
  "n ≠ 0 ⟹ length bs = length cs ⟹
    length (bin_rsplit_aux n nw v bs) =
    length (bin_rsplit_aux n nw w cs)"
proof (induct n nw w cs arbitrary: v bs rule: bin_rsplit_aux.induct)
  case (1 n m w cs v bs)
  show ?case
  proof (cases "m = 0")
    case True
    with ‹length bs = length cs› show ?thesis by simp
  next
    case False
    from "1.hyps" [of ‹bin_split n w› ‹drop_bit n w› ‹take_bit n w›]
‹m ≠ 0› ‹n ≠ 0›
    have hyp: "⋀v bs. length bs = Suc (length cs) ⟹
      length (bin_rsplit_aux n (m - n) v bs) =
      length (bin_rsplit_aux n (m - n) (drop_bit n w) (take_bit n w #
cs))"
      using bin_rsplit_aux_len by fastforce
    from ‹length bs = length cs› ‹n ≠ 0› show ?thesis
```

```
    by (auto simp add: bin_rsplit_aux_simp_alt Let_def bin_rsplit_len
split: prod.split)
  qed
qed

lemma bin_rsplit_len_indep:
  "n ≠ 0 ⟹ length (bin_rsplit n (nw, v)) = length (bin_rsplit n (nw,
w))"
  apply (unfold bin_rsplit_def)
  apply (simp (no_asm))
  apply (erule bin_rsplit_aux_len_indep)
  apply (rule refl)
  done
```

6.5 Logical operations

```
abbreviation (input) bin_sc :: ‹nat ⇒ bool ⇒ int ⇒ int›
  where ‹bin_sc n b i ≡ set_bit i n b›

lemma bin_sc_0 [simp]:
  "bin_sc 0 b w = of_bool b + 2 * (λk::int. k div 2) w"
  by (simp add: set_bit_int_def)

lemma bin_sc_Suc [simp]:
  "bin_sc (Suc n) b w = of_bool (odd w) + 2 * bin_sc n b (w div 2)"
  by (simp add: set_bit_int_def set_bit_Suc unset_bit_Suc bin_last_def)

lemma bin_nth_sc [bit_simps]: "bit (bin_sc n b w) n ⟷ b"
  by (simp add: bit_simps)

lemma bin_sc_sc_same [simp]: "bin_sc n c (bin_sc n b w) = bin_sc n c
w"
  by (induction n arbitrary: w) (simp_all add: bit_Suc)

lemma bin_sc_sc_diff: "m ≠ n ⟹ bin_sc m c (bin_sc n b w) = bin_sc
n b (bin_sc m c w)"
  apply (induct n arbitrary: w m)
   apply (case_tac [!] m)
     apply auto
  done

lemma bin_nth_sc_gen: "(bit :: int ⇒ nat ⇒ bool) (bin_sc n b w) m =
(if m = n then b else (bit :: int ⇒ nat ⇒ bool) w m)"
  by (simp add: bit_simps)

lemma bin_sc_eq:
  ‹bin_sc n False = unset_bit n›
  ‹bin_sc n True = Bit_Operations.set_bit n›
   apply (simp_all add: fun_eq_iff bit_eq_iff)
```

```
  apply (simp_all add: bit_simps bin_nth_sc_gen)
  done

lemma bin_sc_nth [simp]: "bin_sc n ((bit :: int ⇒ nat ⇒ bool) w n)
w = w"
  by (rule bit_eqI) (simp add: bin_nth_sc_gen)

lemma bin_sign_sc [simp]: "bin_sign (bin_sc n b w) = bin_sign w"
proof (induction n arbitrary: w)
  case 0
  then show ?case
    by (auto simp add: bin_sign_def) (use bin_rest_ge_0 in fastforce)
next
  case (Suc n)
  from Suc [of ‹w div 2›]
  show ?case by (auto simp add: bin_sign_def split: if_splits)
qed

lemma bin_sc_bintr [simp]:
  "(take_bit :: nat ⇒ int ⇒ int) m (bin_sc n x ((take_bit :: nat ⇒
int ⇒ int) m w)) = (take_bit :: nat ⇒ int ⇒ int) m (bin_sc n x w)"
  apply (rule bit_eqI)
  apply (cases x)
   apply (auto simp add: bit_simps bin_sc_eq)
  done

lemma bin_clr_le: "bin_sc n False w ≤ w"
  by (simp add: set_bit_int_def unset_bit_less_eq)

lemma bin_set_ge: "bin_sc n True w ≥ w"
  by (simp add: set_bit_int_def set_bit_greater_eq)

lemma bintr_bin_clr_le: "(take_bit :: nat ⇒ int ⇒ int) n (bin_sc m
False w) ≤ (take_bit :: nat ⇒ int ⇒ int) n w"
  by (simp add: set_bit_int_def take_bit_unset_bit_eq unset_bit_less_eq)

lemma bintr_bin_set_ge: "(take_bit :: nat ⇒ int ⇒ int) n (bin_sc m
True w) ≥ (take_bit :: nat ⇒ int ⇒ int) n w"
  by (simp add: set_bit_int_def take_bit_set_bit_eq set_bit_greater_eq)

lemma bin_sc_FP [simp]: "bin_sc n False 0 = 0"
  by (induct n) auto

lemma bin_sc_TM [simp]: "bin_sc n True (- 1) = - 1"
  by (induct n) auto

lemmas bin_sc_simps = bin_sc_0 bin_sc_Suc bin_sc_TM bin_sc_FP

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

```
w"
  by auto

lemmas bin_sc_Suc_minus =
  trans [OF bin_sc_minus [symmetric] bin_sc_Suc]

lemma bin_sc_numeral [simp]:
  "bin_sc (numeral k) b w =
    of_bool (odd w) + 2 * bin_sc (pred_numeral k) b (w div 2)"
  by (simp add: numeral_eq_Suc)

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

lemma int_set_bit_0 [simp]: fixes x :: int shows
  "set_bit x 0 b = of_bool b + 2 * (x div 2)"
  by (fact bin_sc_0)

lemma int_set_bit_Suc: fixes x :: int shows
  "set_bit x (Suc n) b = of_bool (odd x) + 2 * set_bit (x div 2) n b"
  by (fact bin_sc_Suc)

lemma bin_last_set_bit:
  "odd (set_bit x n b :: int) = (if n > 0 then odd x else b)"
  by (cases n) (simp_all add: int_set_bit_Suc)

lemma bin_rest_set_bit:
  "(set_bit x n b :: int) div 2 = (if n > 0 then set_bit (x div 2) (n
- 1) b else x div 2)"
  by (cases n) (simp_all add: int_set_bit_Suc)

lemma int_set_bit_numeral: fixes x :: int shows
  "set_bit x (numeral w) b = of_bool (odd x) + 2 * set_bit (x div 2) (pred_numeral
w) b"
  by (fact bin_sc_numeral)

lemmas int_set_bit_numerals [simp] =
  int_set_bit_numeral[where x="numeral w'"]
  int_set_bit_numeral[where x="- numeral w'"]
  int_set_bit_numeral[where x="Numeral1"]
  int_set_bit_numeral[where x="1"]
  int_set_bit_numeral[where x="0"]
  int_set_bit_Suc[where x="numeral w'"]
  int_set_bit_Suc[where x="- numeral w'"]
  int_set_bit_Suc[where x="Numeral1"]
  int_set_bit_Suc[where x="1"]
  int_set_bit_Suc[where x="0"]
  for w'
```

```
lemma msb_set_bit [simp]:
  "msb (set_bit (x :: int) n b) ⟷ msb x"
  by (simp add: msb_int_def set_bit_int_def)

lemma word_set_bit_def:
  ‹set_bit a n x = word_of_int (bin_sc n x (uint a))›
  apply (rule bit_word_eqI)
  apply (cases x)
   apply (simp_all add: bit_simps bin_sc_eq)
  done

lemma set_bit_word_of_int:
  "set_bit (word_of_int x) n b = word_of_int (bin_sc n b x)"
  unfolding word_set_bit_def
  by (rule word_eqI) (simp add: word_size bin_nth_sc_gen nth_bintr bit_simps)

lemma word_set_numeral [simp]:
  "set_bit (numeral bin::'a::len word) n b =
    word_of_int (bin_sc n b (numeral bin))"
  unfolding word_numeral_alt by (rule set_bit_word_of_int)

lemma word_set_neg_numeral [simp]:
  "set_bit (- numeral bin::'a::len word) n b =
    word_of_int (bin_sc n b (- numeral bin))"
  unfolding word_neg_numeral_alt by (rule set_bit_word_of_int)

lemma word_set_bit_0 [simp]: "set_bit 0 n b = word_of_int (bin_sc n b
0)"
  unfolding word_0_wi by (rule set_bit_word_of_int)

lemma word_set_bit_1 [simp]: "set_bit 1 n b = word_of_int (bin_sc n b
1)"
  unfolding word_1_wi by (rule set_bit_word_of_int)

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

6.5.1 Basic simplification rules

```
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)
```

6.5.2 Binary destructors

```
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)

6.5.3 Derived properties

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
```

6.5.4 Basic properties of logical (bit-wise) operations

```
lemma bbw_ao_absorb: "x AND (y OR x) = x ∧ x OR (y AND x) = x"
  for x y :: int
  by (auto simp add: bin_eq_iff bin_nth_ops)

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: bin_eq_iff bin_nth_ops)

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: bin_eq_iff bin_nth_ops)

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

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

lemma int_xor_assoc: "(x XOR y) XOR z = x XOR (y XOR z)"
  for x y z :: int
```

```
  by (auto simp add: bin_eq_iff bin_nth_ops)

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: bin_eq_iff bin_nth_ops)

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: bin_eq_iff bin_nth_ops)

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: bin_eq_iff bin_nth_ops)

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: bin_eq_iff bin_nth_ops)
```

6.5.5 Simplification with numerals

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
```

6.5.6 Interactions with arithmetic

```
lemma le_int_or: "bin_sign y = 0 ⟹ x ≤ x OR y"
  for x y :: int
  by (simp add: bin_sign_def or_greater_eq split: if_splits)

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

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

```
lemma bin_add_not: "x + NOT x = (-1::int)"
  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)
```

6.5.7 Truncating results of bit-wise operations

```
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]
```

6.5.8 More lemmas

```
lemma not_int_cmp_0 [simp]:
  fixes i :: int shows
  "0 < NOT i ⟷ i < -1"
  "0 ≤ NOT i ⟷ i < 0"
  "NOT i < 0 ⟷ i ≥ 0"
  "NOT i ≤ 0 ⟷ i ≥ -1"
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 bin_sign_and:
  "bin_sign (i AND j) = - (bin_sign i * bin_sign j)"
by(simp add: bin_sign_def)

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)
```

6.6 Setting and clearing bits

```
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)

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(simp_all add: numeral_eq_Suc shiftl_int_def)
  (metis add_One mult_inc semiring_norm(11) semiring_norm(13) semiring_norm(2)
semiring_norm(6) semiring_norm(87))+

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)

lemma bin_nth_minus_p2:
  assumes sign: "bin_sign x = 0"
    and y: "y = push_bit n 1"
    and m: "m < n"
    and x: "x < y"
  shows "bit (x - y) m = bit x m"
proof -
  from ‹bin_sign x = 0› have ‹x ≥ 0›
    by (simp add: sign_Pls_ge_0)
  moreover from x y have ‹x < 2 ^ n›
    by simp
  ultimately have ‹q < n› if ‹bit x q› for q
    using that by (metis bit_take_bit_iff take_bit_int_eq_self)
  then have ‹bit (x + NOT (mask n)) m = bit x m›
    using ‹m < n› by (simp add: disjunctive_add bit_simps)
  also have ‹x + NOT (mask n) = x - y›
    using y by (simp flip: minus_exp_eq_not_mask)
  finally show ?thesis .
qed

lemma bin_clr_conv_NAND:
  "bin_sc n False i = i AND NOT (push_bit n 1)"
  by (rule bit_eqI) (auto simp add: bin_sc_eq bit_simps)

lemma bin_set_conv_OR:
  "bin_sc n True i = i OR (push_bit n 1)"
  by (rule bit_eqI) (auto simp add: bin_sc_eq bit_simps)

end
```

6.7 More lemmas on words

```
lemma msb_conv_bin_sign:
  "msb x ⟷ bin_sign x = -1"
  by (simp add: bin_sign_def not_le msb_int_def)
```

```
lemma msb_bin_sc:
  "msb (bin_sc n b x) ⟷ msb x"
  by (simp add: msb_conv_bin_sign)

lemma msb_word_def:
  ‹msb a ⟷ bin_sign (signed_take_bit (LENGTH('a) - 1) (uint a)) = -
1›
  for a :: ‹'a::len word›
  by (simp add: bin_sign_def bit_simps msb_word_iff_bit)

lemma word_msb_def:
  "msb a ⟷ bin_sign (sint a) = - 1"
  by (simp add: msb_word_def sint_uint)

lemma word_rcat_eq:
  ‹word_rcat ws = word_of_int (bin_rcat (LENGTH('a::len)) (map uint ws))›
  for ws :: ‹'a::len word list›
  apply (simp add: word_rcat_def bin_rcat_def rev_map)
  apply transfer
  apply (simp add: horner_sum_foldr foldr_map comp_def)
  done

lemma sign_uint_Pls [simp]: "bin_sign (uint x) = 0"
  by (simp add: sign_Pls_ge_0)

lemmas bin_log_bintrs = bin_trunc_not bin_trunc_xor bin_trunc_and bin_trunc_or

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

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

— the binary operations only
lemmas word_log_binary_defs =
  word_and_def word_or_def word_xor_def

lemma setBit_no: "Bit_Operations.set_bit n (numeral bin) = word_of_int
(bin_sc n True (numeral bin))"
  by (rule bit_word_eqI) (simp add: bit_simps)

lemma clearBit_no:
  "unset_bit n (numeral bin) = word_of_int (bin_sc n False (numeral bin))"
  by (rule bit_word_eqI) (simp add: bit_simps)

lemma eq_mod_iff: "0 < n ⟹ b = b mod n ⟷ 0 ≤ b ∧ b < n"
  for b n :: int
  using pos_mod_sign [of n b] pos_mod_bound [of n b] by (safe, auto)
```

```
lemma split_uint_lem: "bin_split n (uint w) = (a, b) ⟹
    a = take_bit (LENGTH('a) - n) a ∧ b = take_bit (LENGTH('a)) b"
  for w :: "'a::len word"
  by transfer (simp add: drop_bit_take_bit ac_simps)
```

— 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 transfer (simp add: take_bit_concat_bit_eq)

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 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 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

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_sign_mask [simp]: "bin_sign (mask n) = 0"
  by (simp add: bin_sign_def bin_mask_conv_pow2)
```

```
lemma bin_mask_p1_conv_shift: "mask n + 1 = push_bit n (1 :: int)"
  by (simp add: bin_mask_conv_pow2 shiftl_int_def)

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))"
  apply (simp_all add: image_def)
  apply (metis sbintrunc_sbintrunc)+
  done

lemma sbintrunc_If:
  "- 3 * (2 ^ n) ≤ x ∧ x < 3 * (2 ^ n)
    ⟹ (signed_take_bit :: nat ⇒ int ⇒ int) n x = (if x < - (2 ^ n)
then x + 2 * (2 ^ n)
       else if x ≥ 2 ^ n then x - 2 * (2 ^ n) else x)"
  apply (simp add: no_sbintr_alt2, safe)
   apply (simp add: mod_pos_geq)
  apply (subst mod_add_self1[symmetric], simp)
  done

lemma sint_range':
  ‹- (2 ^ (LENGTH('a) - Suc 0)) ≤ sint x ∧ sint x < 2 ^ (LENGTH('a) -
Suc 0)›
  for x :: ‹'a::len word›
  apply transfer
  using sbintr_ge sbintr_lt apply auto
  done

lemma signed_arith_eq_checks_to_ord:
  "(sint a + sint b = sint (a + b ))
    = ((a <=s a + b) = (0 <=s b))"
  "(sint a - sint b = sint (a - b ))
    = ((0 <=s a - b) = (b <=s a))"
  "(- sint a = sint (- a)) = (0 <=s (- a) = (a <=s 0))"
  using sint_range'[where x=a] sint_range'[where x=b]
  by (simp_all add: sint_word_ariths word_sle_eq word_sless_alt sbintrunc_If)

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

```
    by simp

  have abs: "!! x :: 'a word. abs (sint x) < 2 ^ (size a - 1) + 1"
    apply (cut_tac x=x in sint_range')
    apply (simp add: abs_le_iff word_size)
    done
  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)
  then show ?thesis
    using P[unfolded range_sbintrunc] abs_ab le
    apply clarsimp
    apply (transfer fixing: r s)
    apply (auto simp add: signed_take_bit_int_eq_self simp flip: signed_take_bit_eq_iff_tak
    done
qed

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)

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_sc_pos:
  "0 ≤ i ⟹ 0 ≤ bin_sc n b i"
  by (metis bin_sign_sc sign_Pls_ge_0)

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

end

7 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
  apply (simp add: is_aligned_iff_udvd)
  apply transfer
  apply auto
  done

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"
  apply (simp del: add_uminus_conv_diff add:diff_conv_add_uminus)
  apply (erule aligned_add_aligned, simp_all)
  apply (erule is_alignedE)
  apply (rule_tac k="- of_nat q" in is_alignedI)
```

```
  apply simp
  done

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"
        apply (subst unat_of_nat)
        apply (rule mod_less)
        apply (rule order_less_trans [OF qv])
        apply (rule power_strict_increasing [OF diff_less [OF sne]])
         apply (simp_all)
        done

      have uptr: "unat ptr = 2 ^ sz * q"
        apply (subst ptrq)
        apply (subst iffD1 [OF unat_mult_lem])
         apply (subst unat_power_lower [OF szv])
         apply (subst uq)
```

```
         apply (rule nat_less_power_trans [OF qv order_less_imp_le [OF
szv]])
        apply (subst uq)
        apply (subst unat_power_lower [OF szv])
        apply simp
        done

      show "unat ptr + unat off < 2 ^ LENGTH('a)" using szv
        apply (subst uptr)
        apply (subst mult.commute, rule nat_add_offset_less [OF _ qv])
        apply (rule order_less_le_trans [OF unat_mono [OF off] order_eq_refl])
         apply simp_all
        done
    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 arbitrary: n n')
  case Nil then show ?case by simp
next
  case (Cons y ys)
  from Cons.prems
  show ?case
    apply simp
    apply (cases "n = Suc (length ys)")
     apply (simp add: nth_append)
    apply (simp add: Suc_diff_le Cons.hyps nth_append)
    apply clarsimp
    apply arith
    done
qed

lemma drop_minus:
  "⟦ n < length xs; n' = length xs ⟧ ⟹ drop (n' - Suc n) xs = rev xs
! n  # drop (n' - n) xs"
  apply (subst drop_minus_lem)
     apply simp
    apply simp
   apply simp
  apply simp
  apply (cases "length xs", simp)
  apply (simp add: Suc_diff_le)
  done

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'"
  apply (erule aligned_add_aligned)
  apply (auto intro: is_alignedI [where k="x*z"])
  done

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'"
  apply (erule aligned_add_aligned)
  apply (auto intro: is_alignedI [where k="x"])
  done

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)"
  apply (drule is_aligned_no_wrap[where off="of_nat off"])
   apply (simp add: word_less_nat_alt)
   apply (erule order_le_less_trans[rotated])
  apply (simp add: take_bit_eq_mod unsigned_of_nat)
  apply (subst(asm) unat_of_nat_len)
   apply (erule order_less_trans)
   apply (erule power_strict_increasing)
   apply simp
  apply assumption
  done

lemma is_aligned_get_word_bits:
  fixes p :: "'a::len word"
  shows "⟦ is_aligned p n; ⟦ is_aligned p n; n < LENGTH('a) ⟧ ⟹ P;
          ⟦ p = 0; n ≥ LENGTH('a) ⟧ ⟹ P ⟧ ⟹ P"
  apply (cases "n < LENGTH('a)")
   apply simp
  apply simp
  apply (erule meta_mp)
  apply (simp add: is_aligned_mask power_add power_overflow not_less
    flip: take_bit_eq_mask)
  apply (metis take_bit_length_eq take_bit_of_0 take_bit_tightened)
  done

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:
  "⟦is_aligned x n; is_aligned y n; x < y⟧ ⟹ x ≤ y - 2 ^ n"
  apply (erule is_aligned_get_word_bits[where p=y])
   apply (rule ccontr)
   apply (clarsimp simp: linorder_not_le)
   apply (subgoal_tac "y - x = 0")
    apply clarsimp
   apply (rule aligned_small_is_0)
    apply (erule(1) aligned_sub_aligned)
    apply simp
   apply unat_arith
  apply simp
  done

lemma is_aligned_no_overflow'':
  "⟦is_aligned x n; x + 2 ^ n ≠ 0⟧ ⟹ x ≤ x + 2 ^ n"
  apply (frule is_aligned_no_overflow')
  apply (erule order_trans)
  apply (simp add: field_simps)
  apply (erule word_sub_1_le)
  done

lemma is_aligned_bitI:
  ‹is_aligned p m› if ‹⋀n. n < m ⟹ ¬ bit p n›
  apply (simp add: is_aligned_mask)
  apply (rule bit_word_eqI)
  using that
  apply (auto simp add: bit_simps)
  done

lemma is_aligned_nth:
  "is_aligned p m = (∀n < m. ¬ bit p n)"
  apply (auto intro: is_aligned_bitI simp add: is_aligned_mask bit_eq_iff)
   apply (auto simp: bit_simps)
  using bit_imp_le_length not_less apply blast
  done

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"
```

```
  apply (clarsimp simp only: range_inter)
  apply (erule_tac x=p in allE)
  apply simp
  apply (erule impE)
   apply (erule is_aligned_no_overflow')
  apply (erule notE)
  apply (erule is_aligned_no_overflow)
  done

lemma not_aligned_mod_nz:
  assumes al: "¬ is_aligned a n"
  shows "a mod 2 ^ n ≠ 0"
  apply (rule ccontr)
  using al apply (rule notE)
  apply simp
  apply (rule is_alignedI [of _ _ <a div 2 ^ n>])
  apply (metis add.right_neutral mult.commute word_mod_div_equality)
  done

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"
        apply (subst unat_of_nat)
        apply (rule mod_less)
        apply (rule order_less_trans [OF qv])
        apply (rule power_strict_increasing [OF diff_less [OF sne]])
        apply simp_all
        done

      have uptr: "unat ptr = 2 ^ sz * q"
        apply (subst ptrq)
        apply (subst iffD1 [OF unat_mult_lem])
        apply (subst unat_power_lower [OF szv])
        apply (subst uq)
        apply (rule nat_less_power_trans [OF qv order_less_imp_le [OF
szv]])
        apply (subst uq)
        apply (subst unat_power_lower [OF szv])
        apply simp
        done

      show "unat ptr + off ≤ 2 ^ LENGTH('a)" using szv
        apply (subst uptr)
        apply (subst mult.commute, rule nat_add_offset_le [OF off qv])
        apply simp
        done
    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›
  using that
  apply (cases ‹n < LENGTH('a)›)
  apply (simp_all add: not_less is_aligned_beyond_length)
  apply transfer
  apply (simp flip: take_bit_eq_0_iff)
  apply (subst take_bit_minus [symmetric])
  apply simp
  done

lemma add_mask_lower_bits:
  "⟦is_aligned (x :: 'a :: len word) n;
    ∀n' ≥ n. n' < LENGTH('a) ⟶ ¬ bit p n'⟧ ⟹ x + p AND NOT (mask
n) = x"
  apply (subst word_plus_and_or_coroll)
   apply (rule word_eqI)
   apply (clarsimp simp: word_size is_aligned_nth)
   apply (erule_tac x=na in allE)+
   apply (simp add: bit_simps)
  apply (rule bit_word_eqI)
  apply (auto simp add: bit_simps not_less word_size)
  apply (metis is_aligned_nth not_le)
  done

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"
  apply (rule bit_word_eqI)
  apply (auto simp add: bit_simps is_aligned_nth)
  done
```

```
lemma is_aligned_shiftr_shiftl:
  "is_aligned w n ⟹ w >> n << n = w"
  apply (rule bit_word_eqI)
  apply (auto simp add: bit_simps is_aligned_nth intro: ccontr)
  apply (subst add_diff_inverse_nat)
   apply (auto intro: ccontr)
  done

lemma aligned_shiftr_mask_shiftl:
  "is_aligned x n ⟹ ((x >> n) AND mask v) << n = x AND mask (v + n)"
  apply (rule word_eqI)
  apply (simp add: word_size bit_simps)
  apply (subgoal_tac "∀m. bit x m ⟶ m ≥ n")
   apply auto[1]
  apply (clarsimp simp: is_aligned_mask)
  apply (drule_tac x=m in word_eqD)
  apply (frule test_bit_size)
  apply (simp add: word_size bit_simps)
  done

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
    apply (subst xv)
    apply (subst unat_mult_power_lem[OF kl])
    apply (subst mult.commute, rule nat_add_offset_less)
      apply (rule less_le_trans[OF unat_mono[OF offv, simplified]])
      apply (erule eq_imp_le[OF unat_power_lower])
     apply (rule kl)
    apply simp
   done
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
```

```
    apply (simp flip: take_bit_eq_mod)
    apply (rule bit_word_eqI)
    apply (auto simp add: bit_simps)
    apply (metis assms(2) bit_or_iff is_aligned_mask is_aligned_nth leD
less_mask_eq word_and_le1 word_bw_lcs(1) word_neq_0_conv word_plus_and_or_coroll)
     apply (meson assms(2) leI less_2p_is_upper_bits_unset)
    apply (metis assms(2) bit_disjunctive_add_iff bit_imp_le_length bit_push_bit_iff
is_alignedE' less_2p_is_upper_bits_unset)
    done
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"
      apply -
      apply simp
      apply (subst (asm) add.commute, subst (asm) add.commute, drule word_plus_mcs_4)
       apply (subst add.commute, subst no_plus_overflow_uint_size)
       apply transfer
      apply simp
      apply (auto simp add: le_less power_2_ge_iff szv)
      apply (metis le_less_trans mask_eq_decr_exp mask_lt_2pn order_less_imp_le
szv)
      done

    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)"
  apply (subst (asm) is_aligned_mask)
  apply (drule less_mask_eq)
  apply (rule context_conjI)
   apply (subst word_plus_and_or_coroll)
    apply (simp_all flip: take_bit_eq_mask)
   apply (metis take_bit_eq_mask word_bw_lcs(1) word_log_esimps(1))
  apply (metis add.commute add_left_imp_eq take_bit_eq_mask word_plus_and_or_coroll2)
  done

lemmas mask_inner_mask = mask_eqs(1)

lemma mask_add_aligned:
  "is_aligned p n ⟹ (p + q) AND mask n = q AND mask n"
  apply (simp add: is_aligned_mask)
```

```
  apply (subst mask_inner_mask [symmetric])
  apply simp
  done

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"
  apply (subst disjunctive_add, simp_all)
  apply (clarsimp simp: is_aligned_nth less_2p_is_upper_bits_unset)
  subgoal for m
    apply (cases ‹m < n›)
     apply (auto simp add: not_less dest: bit_imp_possible_bit)
    done
  done

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"
    apply (rule word_plus_mono_right)
     apply (rule order_less_imp_le, rule and_mask_less_size)
     apply (simp add: word_size word_bits)
    apply (rule is_aligned_no_overflow'', simp add: is_aligned_neg_mask
word_bits)
    apply (rule not_greatest_aligned, rule b; simp add: is_aligned_neg_mask)
    done
  also have "... ≤ x AND NOT(mask n)"
    using b
    apply (subst add.commute)
```

```
    apply (rule le_plus)
     apply (rule aligned_at_least_t2n_diff; simp add: is_aligned_neg_mask)
    apply (rule ccontr, simp add: linorder_not_le)
    apply (drule aligned_small_is_0[rotated]; simp add: is_aligned_neg_mask)
    done
  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:
  "⟦is_aligned (p::'a::len word) n; d < 2^n; n ≤ m; p < 2^m⟧ ⟹ p + d
< 2^m"
  apply (case_tac "m < LENGTH('a)")
   apply (subst mask_eq_iff_w2p[symmetric])
    apply (simp add: word_size)
   apply (simp add: is_aligned_add_or word_ao_dist less_mask_eq)
```

```
  apply (subst less_mask_eq)
   apply (erule order_less_le_trans)
   apply (erule(1) two_power_increasing)
  apply simp
 apply (simp add: power_overflow)
 done

lemma aligned_offset_non_zero:
 "⟦ is_aligned x n; y < 2 ^ n; x ≠ 0 ⟧ ⟹ x + y ≠ 0"
 apply (cases "y = 0")
  apply simp
 apply (subst word_neq_0_conv)
 apply (subst gt0_iff_gem1)
 apply (erule is_aligned_get_word_bits)
  apply (subst field_simps[symmetric], subst plus_le_left_cancel_nowrap)
    apply (rule is_aligned_no_wrap')
     apply simp
    apply (rule word_leq_le_minus_one)
     apply simp
    apply assumption
   apply (erule (1) is_aligned_no_wrap')
  apply (simp add: gt0_iff_gem1 [symmetric] word_neq_0_conv)
 apply simp
 done

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:
 "⟦ n' ≤ n; p' ≤ p + mask n; is_aligned p n; is_aligned p' n' ⟧ ⟹
  (p'::'a::len word) + mask n' ≤ p + mask n"
 apply (cases "LENGTH('a) ≤ n")
  apply (frule (1) is_aligned_over_length)
  apply (drule mask_over_length)
  apply clarsimp
 apply (simp add: not_le)
 apply (simp add: word_le_nat_alt unat_plus_simple)
 apply (subst unat_plus_simple[THEN iffD1], erule is_aligned_no_overflow_mask)+
 apply (subst (asm) unat_plus_simple[THEN iffD1], erule is_aligned_no_overflow_mask)
 apply (clarsimp simp: dvd_def is_aligned_iff_dvd_nat)
 apply (rename_tac k k')
 apply (thin_tac "unat p = x" for p x)+
 apply (subst Suc_le_mono[symmetric])
 apply (simp only: Suc_2p_unat_mask)
```

```
  apply (drule le_imp_less_Suc, subst (asm) Suc_2p_unat_mask, assumption)
  apply (erule (1) power_2_mult_step_le)
  done

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

  from offv szv have offv': "unat off < 2 ^ sz"
    by (simp add: mask_2pm1 unat_less_power)

  show ?thesis using szv
    using al is_aligned_no_wrap''' offv' by blast
next
  assume "¬ sz < LENGTH('a)"
  with al have "x = 0"
    by (meson is_aligned_get_word_bits)
  thus ?thesis by simp
qed

lemma aligned_less_plus_1:
  "⟦ is_aligned x n; n > 0 ⟧ ⟹ x < x + 1"
  apply (rule plus_one_helper2)
   apply (rule order_refl)
  apply (clarsimp simp: field_simps)
  apply (drule arg_cong[where f="λx. x - 1"])
  apply (clarsimp simp: is_aligned_mask)
  done

lemma aligned_add_offset_less:
  "⟦is_aligned x n; is_aligned y n; x < y; z < 2 ^ n⟧ ⟹ x + z < y"
  apply (cases "y = 0")
   apply simp
  apply (erule is_aligned_get_word_bits[where p=y], simp_all)
  apply (cases "z = 0", simp_all)
  apply (drule(2) aligned_at_least_t2n_diff[rotated -1])
  apply (drule plus_one_helper2)
   apply (rule less_is_non_zero_p1)
   apply (rule aligned_less_plus_1)
    apply (erule aligned_sub_aligned[OF _ _ order_refl],
           simp_all add: is_aligned_triv)[1]
```

```
   apply (cases n, simp_all)[1]
  apply (simp only: trans[OF diff_add_eq diff_diff_eq2[symmetric]])
  apply (drule word_less_add_right)
   apply (rule ccontr, simp add: linorder_not_le)
   apply (drule aligned_small_is_0, erule order_less_trans)
    apply (clarsimp simp: power_overflow)
   apply simp
  apply (erule order_le_less_trans[rotated],
         rule word_plus_mono_right)
   apply (erule word_le_minus_one_leq)
  apply (simp add: is_aligned_no_wrap' is_aligned_no_overflow field_simps)
  done

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"
  apply (simp flip: not_le)
  apply (erule notE)
  apply (cases "LENGTH('a) ≤ n")
   apply (drule (1) is_aligned_over_length)+
   apply (drule mask_over_length)
   apply clarsimp
  apply (clarsimp simp: word_le_nat_alt not_less not_le)
  apply (subst (asm) unat_plus_simple[THEN iffD1], erule is_aligned_no_overflow_mask)
  apply (subst (asm) unat_add_lem' [symmetric])
   apply (simp add: is_aligned_mask_offset_unat)
  apply (metis gap_between_aligned linorder_not_less mask_eq_decr_exp
unat_arith_simps(2))
  done

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"
  and     s10: "0 < s1"
shows  "∃q. m - n = of_nat q * 2 ^ s1 ∧ q < 2 ^ (s2 - s1)"
proof -
  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"
      apply (subst unat_of_nat)
      apply (subst mod_less)
       apply (erule order_less_le_trans)
       apply (rule power_increasing)
        apply simp_all
      done

    then show "?thesis m s" using s m
      apply (subst iffD1 [OF unat_mult_lem])
      apply (simp add: nat_less_power_trans)+
      done
  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 add: of_nat_diff)
  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 from s10 have "LENGTH('a) - s1 < LENGTH('a)"
      by (rule diff_less, simp)
    ultimately show "of_nat (mq - 2 ^ sq * nq) < (2::'a word) ^ (LENGTH('a)
- s1)"
      using take_bit_nat_less_self_iff [of ‹LENGTH('a)› ‹mq - 2 ^ sq
* nq›]
      apply (auto simp add: word_less_nat_alt not_le not_less unsigned_of_nat)
      apply (metis take_bit_nat_eq_self_iff)
      done
  qed
  then have "mq - 2 ^ sq * nq < 2 ^ (s2 - s1)" using mq s2wb
    apply (simp add: word_less_nat_alt take_bit_eq_mod unsigned_of_nat)
    apply (subst (asm) mod_less)
    apply auto
     apply (rule order_le_less_trans)
      apply (rule diff_le_self)
     apply (erule order_less_le_trans)
    apply simp
    done
  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 "%x. 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:
  "⟦x < 2 ^ n; is_aligned (y :: 'a :: len word) n;n ≤ LENGTH('a)⟧
   ⟹ (x + y) >> n = y >> n"
  apply (subst word_plus_and_or_coroll; rule bit_word_eqI)
   apply (auto simp add: bit_simps is_aligned_nth)
   apply (metis less_2p_is_upper_bits_unset not_le)
  apply (metis le_add1 less_2p_is_upper_bits_unset test_bit_bin)
  done

lemma aligned_shift':
  "⟦x < 2 ^ n; is_aligned (y :: 'a :: len word) n;n ≤ LENGTH('a)⟧
   ⟹ (y + x) >> n = y >> n"
  apply (subst word_plus_and_or_coroll; rule bit_word_eqI)
   apply (auto simp add: bit_simps is_aligned_nth)
   apply (metis less_2p_is_upper_bits_unset not_le)
  apply (metis bit_imp_le_length le_add1 less_2p_is_upper_bits_unset)
  done

lemma and_neg_mask_plus_mask_mono: "(p AND NOT (mask n)) + mask n ≥
p"
  for p :: <'a::len word>
  apply (rule word_le_minus_cancel[where x = "p AND NOT (mask n)"])
   apply (clarsimp simp: subtract_mask)
   using word_and_le1[where a = "mask n" and y = p]
    apply (clarsimp simp: mask_eq_decr_exp word_le_less_eq)
  apply (rule is_aligned_no_overflow'[folded mask_2pm1])
  apply (clarsimp simp: is_aligned_neg_mask)
  done

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:
  "⟦is_aligned ptr m; m ≤ n; a < 2 ^ m⟧
   ⟹ ptr + a AND mask n = (ptr AND mask n) + a"
  apply (rule mask_eqI[where n = m])
   apply (simp add:mask_twice min_def)
    apply (simp add:is_aligned_add_helper)
    apply (subst is_aligned_add_helper[THEN conjunct1])
      apply (erule is_aligned_after_mask)
     apply simp
    apply simp
   apply simp
  apply (subgoal_tac "(ptr + a AND mask n) AND NOT (mask m)
     = (ptr + a AND NOT (mask m) ) AND mask n")
   apply (simp add:is_aligned_add_helper)
   apply (subst is_aligned_add_helper[THEN conjunct2])
     apply (simp add:is_aligned_after_mask)
    apply simp
   apply simp
  apply (simp add:word_bw_comms word_bw_lcs)
  done

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"
    apply -
    apply (rule word_power_nonzero)
    apply (erule order_le_less_trans  [OF _ kw])
    apply (rule mw)
    apply (rule less_is_non_zero_p1 [OF ak])
    done
  then have "(a * 2 ^ m) + off < ((a + 1) * 2 ^ m)" using kw mw
    apply -
    apply (simp add: distrib_right)
    apply (rule word_plus_strict_mono_right [OF off])
    apply (rule is_aligned_no_overflow'')
    apply (rule is_aligned_mult_triv2)
    apply assumption
    done
  also have "... ≤ k * 2 ^ m" using ak1 mw kw False
    apply -
    apply (erule word_mult_le_mono1)
    apply (simp add: p2_gt_0)
    apply (simp add: word_less_nat_alt)
    apply (meson nat_mult_power_less_eq zero_less_numeral)
    done
  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"
  apply (erule is_aligned_add_not_aligned)
  apply transfer
  apply (auto simp add: is_aligned_iff_udvd)
  apply (metis (no_types, lifting) le_less_trans less_not_refl2 less_or_eq_imp_le
of_nat_eq_0_iff take_bit_eq_0_iff take_bit_nat_eq_self_iff take_bit_of_nat
unat_lt2p unat_power_lower)
  done

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

8 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
```

9 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 [code]:
  ‹v sdiv w = word_of_int (sint v sdiv sint w)›
  for v w :: ‹'a::len word›
  by transfer simp

lemma smod_word_def [code]:
  ‹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 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)"
  apply transfer
  apply simp
  done

lemma signed_mod_arith:
    "sint ((a::('a::len) word) smod b) = signed_take_bit (LENGTH('a) -
1) (sint a smod sint b)"
  apply transfer
  apply simp
  done

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

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"
  apply (cases ‹LENGTH('a)›)
  apply simp_all
  apply transfer
  apply simp
  apply (case_tac nat)
   apply simp_all
  apply (simp add: take_bit_signed_take_bit)
  done

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"
  apply (auto simp: sdiv_word_def signed_divide_int_def sgn_if)
  apply transfer
```

```
  apply simp
  apply (metis Suc_pred len_gt_0 signed_take_bit_eq_iff_take_bit_eq signed_take_bit_of_0
take_bit_of_0)
  done

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›
  apply (auto simp add: sdiv_word_def signed_divide_int_def)
  apply transfer
  apply simp
  done

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)"
  using sdiv_int_range [where a="sint a" and b="sint b"]
  apply auto
  apply (cases ‹LENGTH('a)›)
   apply simp_all
  apply transfer
  apply simp
  apply (rule order_trans)
   defer
   apply assumption
  apply simp
  apply (metis abs_le_iff add.inverse_inverse le_cases le_minus_iff not_le
```

```
signed_take_bit_int_eq_self_iff signed_take_bit_minus)
  done

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
    apply (auto simp add: sgn_if not_less signed_divide_int_def split:
if_splits)
     apply (smt (verit) pos_imp_zdiv_neg_iff zero_le_power)
    apply (smt (verit) not_exp_less_eq_0_int pos_imp_zdiv_neg_iff)
    done
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)"
  apply (cases ‹sint b = 0›)
   apply (simp_all add: sint_less)
  apply (cases ‹LENGTH('a)›)
   apply simp_all
  using smod_int_range [where a="sint a" and b="sint b"]
  apply auto
  apply (rule less_le_trans)
```

```
   apply assumption
  apply (auto simp add: abs_le_iff)
   apply (metis diff_Suc_1 le_cases linorder_not_le sint_lt)
  apply (metis add.inverse_inverse diff_Suc_1 linorder_not_less neg_less_iff_less
sint_ge)
  done

lemma smod_word_min:
  "- (2 ^ (LENGTH('a::len) - Suc 0)) ≤ sint (a::'a word) smod sint (b::'a
word)"
  apply (cases ‹LENGTH('a)›)
   apply simp_all
  apply (cases ‹sint b = 0›)
   apply simp_all
   apply (metis diff_Suc_1 sint_ge)
  using smod_int_range [where a="sint a" and b="sint b"]
  apply auto
  apply (rule order_trans)
   defer
   apply assumption
  apply (auto simp add: algebra_simps abs_le_iff)
   apply (metis abs_zero add.left_neutral add_mono_thms_linordered_semiring(1)
diff_Suc_1 le_cases linorder_not_less sint_lt zabs_less_one_iff)
  apply (metis abs_zero add.inverse_inverse add.left_neutral add_mono_thms_linordered_semir
diff_Suc_1 le_cases le_minus_iff linorder_not_less sint_ge zabs_less_one_iff)
  done

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
```

10 Bit values as reversed lists of bools

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

```
    Legacy_Aliases
begin

context
  includes bit_operations_syntax
begin

lemma horner_sum_of_bool_2_concat:
  ‹horner_sum of_bool 2 (concat (map (λx. map (bit x) [0..<LENGTH('a)])
ws)) = horner_sum uint (2 ^ LENGTH('a)) ws›
  for ws :: ‹'a::len word list›
proof (induction ws)
  case Nil
  then show ?case
    by simp
next
  case (Cons w ws)
  moreover have ‹horner_sum of_bool 2 (map (bit w) [0..<LENGTH('a)])
= uint w›
  proof transfer
    fix k :: int
    have ‹map (λn. n < LENGTH('a) ∧ bit k n) [0..<LENGTH('a)]
      = map (bit k) [0..<LENGTH('a)]›
      by simp
    then show ‹horner_sum of_bool 2 (map (λn. n < LENGTH('a) ∧ bit k
n) [0..<LENGTH('a)])
      = take_bit LENGTH('a) k›
      by (simp only: horner_sum_bit_eq_take_bit)
  qed
  ultimately show ?case
    by (simp add: horner_sum_append)
qed
```

10.1 Implicit augmentation of list prefixes

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

lemma nth_takefill: "m < n ⟹ takefill fill n l ! m = (if m < length
l then l ! m else fill)"
  apply (induct n arbitrary: m l)
   apply clarsimp
  apply clarsimp
  apply (case_tac m)
```

```
   apply (simp split: list.split)
  apply (simp split: list.split)
  done

lemma takefill_alt: "takefill fill n l = take n l @ replicate (n - length
l) fill"
  by (induct n arbitrary: l) (auto split: list.split)

lemma takefill_replicate [simp]: "takefill fill n (replicate m fill)
= replicate n fill"
  by (simp add: takefill_alt replicate_add [symmetric])

lemma takefill_le': "n = m + k ⟹ takefill x m (takefill x n l) = takefill
x m l"
  by (induct m arbitrary: l n) (auto split: list.split)

lemma length_takefill [simp]: "length (takefill fill n l) = n"
  by (simp add: takefill_alt)

lemma take_takefill': "n = k + m ⟹ take k (takefill fill n w) = takefill
fill k w"
  by (induct k arbitrary: w n) (auto split: list.split)

lemma drop_takefill: "drop k (takefill fill (m + k) w) = takefill fill
m (drop k w)"
  by (induct k arbitrary: w) (auto split: list.split)

lemma takefill_le [simp]: "m ≤ n ⟹ takefill x m (takefill x n l) =
takefill x m l"
  by (auto simp: le_iff_add takefill_le')

lemma take_takefill [simp]: "m ≤ n ⟹ take m (takefill fill n w) =
takefill fill m w"
  by (auto simp: le_iff_add take_takefill')

lemma takefill_append: "takefill fill (m + length xs) (xs @ w) = xs @
(takefill fill m w)"
  by (induct xs) auto

lemma takefill_same': "l = length xs ⟹ takefill fill l xs = xs"
  by (induct xs arbitrary: l) auto

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

lemma tf_rev:
  "n + k = m + length bl ⟹ takefill x m (rev (takefill y n bl)) =
    rev (takefill y m (rev (takefill x k (rev bl))))"
  apply (rule nth_equalityI)
   apply (auto simp add: nth_takefill rev_nth)
```

```
  apply (rule_tac f = "λn. bl ! n" in arg_cong)
  apply arith
  done

lemma takefill_minus: "0 < n ⟹ takefill fill (Suc (n - 1)) w = takefill
fill n w"
  by auto

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

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

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

lemma takefill_numeral_Nil [simp]:
  "takefill fill (numeral k) [] = fill # takefill fill (pred_numeral k)
[]"
  by (simp add: numeral_eq_Suc)

lemma takefill_numeral_Cons [simp]:
  "takefill fill (numeral k) (x # xs) = x # takefill fill (pred_numeral
k) xs"
  by (simp add: numeral_eq_Suc)
```

10.2 Range projection

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

lemma bl_of_nth_simps [simp, code]:
  "bl_of_nth 0 f = []"
  "bl_of_nth (Suc n) f = f n # bl_of_nth n f"
  by (simp_all add: bl_of_nth_def)

lemma length_bl_of_nth [simp]: "length (bl_of_nth n f) = n"
  by (simp add: bl_of_nth_def)

lemma nth_bl_of_nth [simp]: "m < n ⟹ rev (bl_of_nth n f) ! m = f m"
  by (simp add: bl_of_nth_def rev_map)

lemma bl_of_nth_inj: "(⋀k. k < n ⟹ f k = g k) ⟹ bl_of_nth n f =
bl_of_nth n g"
  by (simp add: bl_of_nth_def)

lemma bl_of_nth_nth_le: "n ≤ length xs ⟹ bl_of_nth n (nth (rev xs))
= drop (length xs - n) xs"
```

```
  apply (induct n arbitrary: xs)
   apply clarsimp
  apply clarsimp
  apply (rule trans [OF _ hd_Cons_tl])
   apply (frule Suc_le_lessD)
   apply (simp add: rev_nth trans [OF drop_Suc drop_tl, symmetric])
   apply (subst hd_drop_conv_nth)
    apply force
   apply simp_all
  apply (rule_tac f = "λn. drop n xs" in arg_cong)
  apply simp
  done

lemma bl_of_nth_nth [simp]: "bl_of_nth (length xs) ((!) (rev xs)) = xs"
  by (simp add: bl_of_nth_nth_le)
```

10.3 More

```
definition rotater1 :: "'a list ⇒ 'a list"
  where "rotater1 ys =
    (case ys of [] ⇒ [] | x # xs ⇒ last ys # butlast ys)"

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

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

lemma rotate1_rl': "rotater1 (l @ [a]) = a # l"
  by (cases l) (auto simp: rotater1_def)

lemma rotate1_rl [simp] : "rotater1 (rotate1 l) = l"
  apply (unfold rotater1_def)
  apply (cases "l")
   apply (case_tac [2] "list")
    apply auto
  done

lemma rotate1_lr [simp] : "rotate1 (rotater1 l) = l"
  by (cases l) (auto simp: rotater1_def)

lemma rotater1_rev': "rotater1 (rev xs) = rev (rotate1 xs)"
  by (cases "xs") (simp add: rotater1_def, simp add: rotate1_rl')

lemma rotater_rev': "rotater n (rev xs) = rev (rotate n xs)"
  by (induct n) (auto simp: rotater_def intro: rotater1_rev')

lemma rotater_rev: "rotater n ys = rev (rotate n (rev ys))"
  using rotater_rev' [where xs = "rev ys"] by simp
```

```
lemma rotater_drop_take:
  "rotater n xs =
    drop (length xs - n mod length xs) xs @
    take (length xs - n mod length xs) xs"
  by (auto simp: rotater_rev rotate_drop_take rev_take rev_drop)

lemma rotater_Suc [simp]: "rotater (Suc n) xs = rotater1 (rotater n xs)"
  unfolding rotater_def by auto

lemma nth_rotater:
  ‹rotater m xs ! n = xs ! ((n + (length xs - m mod length xs)) mod length
xs)› if ‹n < length xs›
  using that by (simp add: rotater_drop_take nth_append not_less less_diff_conv
ac_simps le_mod_geq)

lemma nth_rotater1:
  ‹rotater1 xs ! n = xs ! ((n + (length xs - 1)) mod length xs)› if ‹n
< length xs›
  using that nth_rotater [of n xs 1] by simp

lemma rotate_inv_plus [rule_format]:
  "∀k. k = m + n ⟶ rotater k (rotate n xs) = rotater m xs ∧
    rotate k (rotater n xs) = rotate m xs ∧
    rotater n (rotate k xs) = rotate m xs ∧
    rotate n (rotater k xs) = rotater m xs"
  by (induct n) (auto simp: rotater_def rotate_def intro: funpow_swap1
[THEN trans])

lemmas rotate_inv_rel = le_add_diff_inverse2 [symmetric, THEN rotate_inv_plus]

lemmas rotate_inv_eq = order_refl [THEN rotate_inv_rel, simplified]

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

lemma rotate_gal: "rotater n xs = ys ⟷ rotate n ys = xs"
  by auto

lemma rotate_gal': "ys = rotater n xs ⟷ xs = rotate n ys"
  by auto

lemma length_rotater [simp]: "length (rotater n xs) = length xs"
  by (simp add : rotater_rev)

lemma rotate_eq_mod: "m mod length xs = n mod length xs ⟹ rotate m
xs = rotate n xs"
  apply (rule box_equals)
    defer
    apply (rule rotate_conv_mod [symmetric])+
```

```
  apply simp
  done

lemma restrict_to_left: "x = y ⟹ x = z ⟷ y = z"
  by simp

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

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

lemma butlast_map: "xs ≠ [] ⟹ butlast (map f xs) = map f (butlast
xs)"
  by (induct xs) auto

lemma rotater1_map: "rotater1 (map f xs) = map f (rotater1 xs)"
  by (cases xs) (auto simp: rotater1_def last_map butlast_map)

lemma rotater_map: "rotater n (map f xs) = map f (rotater n xs)"
  by (induct n) (auto simp: rotater_def rotater1_map)

lemma but_last_zip [rule_format] :
  "∀ys. length xs = length ys ⟶ xs ≠ [] ⟶
    last (zip xs ys) = (last xs, last ys) ∧
    butlast (zip xs ys) = zip (butlast xs) (butlast ys)"
  apply (induct xs)
   apply auto
     apply ((case_tac ys, auto simp: neq_Nil_conv)[1])+
  done

lemma but_last_map2 [rule_format] :
  "∀ys. length xs = length ys ⟶ xs ≠ [] ⟶
    last (map2 f xs ys) = f (last xs) (last ys) ∧
    butlast (map2 f xs ys) = map2 f (butlast xs) (butlast ys)"
  apply (induct xs)
   apply auto
     apply ((case_tac ys, auto simp: neq_Nil_conv)[1])+
  done

lemma rotater1_zip:
```

```
  "length xs = length ys ⟹
    rotater1 (zip xs ys) = zip (rotater1 xs) (rotater1 ys)"
  apply (unfold rotater1_def)
  apply (cases xs)
   apply auto
   apply ((case_tac ys, auto simp: neq_Nil_conv but_last_zip)[1])+
  done

lemma rotater1_map2:
  "length xs = length ys ⟹
    rotater1 (map2 f xs ys) = map2 f (rotater1 xs) (rotater1 ys)"
  by (simp add: rotater1_map rotater1_zip)

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

lemma rotater_map2:
  "length xs = length ys ⟹
    rotater n (map2 f xs ys) = map2 f (rotater n xs) (rotater n ys)"
  by (induct n) (auto intro!: lrth)

lemma rotate1_map2:
  "length xs = length ys ⟹
    rotate1 (map2 f xs ys) = map2 f (rotate1 xs) (rotate1 ys)"
  by (cases xs; cases ys) auto

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

lemma rotate_map2:
  "length xs = length ys ⟹
    rotate n (map2 f xs ys) = map2 f (rotate n xs) (rotate n ys)"
  by (induct n) (auto intro!: lth)
```

10.4 Explicit bit representation of `int`

```
primrec bl_to_bin_aux :: "bool list ⇒ int ⇒ int"
  where
    Nil: "bl_to_bin_aux [] w = w"
  | Cons: "bl_to_bin_aux (b # bs) w = bl_to_bin_aux bs (of_bool b + 2
* w)"

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

primrec bin_to_bl_aux :: "nat ⇒ int ⇒ bool list ⇒ bool list"
  where
```

```
    Z: "bin_to_bl_aux 0 w bl = bl"
  | Suc: "bin_to_bl_aux (Suc n) w bl = bin_to_bl_aux n (w div 2) (odd
w # bl)"

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

lemma bin_to_bl_aux_zero_minus_simp [simp]:
  "0 < n ⟹ bin_to_bl_aux n 0 bl = bin_to_bl_aux (n - 1) 0 (False # bl)"
  by (cases n) auto

lemma bin_to_bl_aux_minus1_minus_simp [simp]:
  "0 < n ⟹ bin_to_bl_aux n (- 1) bl = bin_to_bl_aux (n - 1) (- 1) (True
# bl)"
  by (cases n) auto

lemma bin_to_bl_aux_one_minus_simp [simp]:
  "0 < n ⟹ bin_to_bl_aux n 1 bl = bin_to_bl_aux (n - 1) 0 (True # bl)"
  by (cases n) auto

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

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

lemma bl_to_bin_aux_append: "bl_to_bin_aux (bs @ cs) w = bl_to_bin_aux
cs (bl_to_bin_aux bs w)"
  by (induct bs arbitrary: w) auto

lemma bin_to_bl_aux_append: "bin_to_bl_aux n w bs @ cs = bin_to_bl_aux
n w (bs @ cs)"
  by (induct n arbitrary: w bs) auto

lemma bl_to_bin_append: "bl_to_bin (bs @ cs) = bl_to_bin_aux cs (bl_to_bin
bs)"
  unfolding bl_to_bin_def by (rule bl_to_bin_aux_append)

lemma bin_to_bl_aux_alt: "bin_to_bl_aux n w bs = bin_to_bl n w @ bs"
  by (simp add: bin_to_bl_def bin_to_bl_aux_append)

lemma bin_to_bl_0 [simp]: "bin_to_bl 0 bs = []"
  by (auto simp: bin_to_bl_def)
```

```
lemma size_bin_to_bl_aux: "length (bin_to_bl_aux n w bs) = n + length
bs"
  by (induct n arbitrary: w bs) auto

lemma size_bin_to_bl [simp]: "length (bin_to_bl n w) = n"
  by (simp add: bin_to_bl_def size_bin_to_bl_aux)

lemma bl_bin_bl': "bin_to_bl (n + length bs) (bl_to_bin_aux bs w) = bin_to_bl_aux
n w bs"
  apply (induct bs arbitrary: w n)
   apply auto
    apply (simp_all only: add_Suc [symmetric])
    apply (auto simp add: bin_to_bl_def)
  done

lemma bl_bin_bl [simp]: "bin_to_bl (length bs) (bl_to_bin bs) = bs"
  unfolding bl_to_bin_def
  apply (rule box_equals)
    apply (rule bl_bin_bl')
   prefer 2
   apply (rule bin_to_bl_aux.Z)
  apply simp
  done

lemma bl_to_bin_inj: "bl_to_bin bs = bl_to_bin cs ⟹ length bs = length
cs ⟹ bs = cs"
  apply (rule_tac box_equals)
    defer
    apply (rule bl_bin_bl)
   apply (rule bl_bin_bl)
  apply simp
  done

lemma bl_to_bin_False [simp]: "bl_to_bin (False # bl) = bl_to_bin bl"
  by (auto simp: bl_to_bin_def)

lemma bl_to_bin_Nil [simp]: "bl_to_bin [] = 0"
  by (auto simp: bl_to_bin_def)

lemma bin_to_bl_zero_aux: "bin_to_bl_aux n 0 bl = replicate n False @
bl"
  by (induct n arbitrary: bl) (auto simp: replicate_app_Cons_same)

lemma bin_to_bl_zero: "bin_to_bl n 0 = replicate n False"
  by (simp add: bin_to_bl_def bin_to_bl_zero_aux)

lemma bin_to_bl_minus1_aux: "bin_to_bl_aux n (- 1) bl = replicate n True
@ bl"
```

```
  by (induct n arbitrary: bl) (auto simp: replicate_app_Cons_same)

lemma bin_to_bl_minus1: "bin_to_bl n (- 1) = replicate n True"
  by (simp add: bin_to_bl_def bin_to_bl_minus1_aux)
```

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

```
lemma bin_bl_bin': "bl_to_bin (bin_to_bl_aux n w bs) = bl_to_bin_aux
bs (take_bit n w)"
  by (induct n arbitrary: w bs) (auto simp: bl_to_bin_def take_bit_Suc
ac_simps mod_2_eq_odd)

lemma bin_bl_bin [simp]: "bl_to_bin (bin_to_bl n w) = take_bit n w"
  by (auto simp: bin_to_bl_def bin_bl_bin')

lemma bl_to_bin_rep_F: "bl_to_bin (replicate n False @ bl) = bl_to_bin
bl"
  by (simp add: bin_to_bl_zero_aux [symmetric] bin_bl_bin') (simp add:
bl_to_bin_def)

lemma bin_to_bl_trunc [simp]: "n ≤ m ⟹ bin_to_bl n (take_bit m w)
= bin_to_bl n w"
  by (auto intro: bl_to_bin_inj)

lemma bin_to_bl_aux_bintr:
  "bin_to_bl_aux n (take_bit m bin) bl =
    replicate (n - m) False @ bin_to_bl_aux (min n m) bin bl"
  apply (induct n arbitrary: m bin bl)
   apply clarsimp
  apply clarsimp
  apply (case_tac "m")
   apply (clarsimp simp: bin_to_bl_zero_aux)
   apply (erule thin_rl)
   apply (induct_tac n)
    apply (auto simp add: take_bit_Suc)
  done

lemma bin_to_bl_bintr:
  "bin_to_bl n (take_bit m bin) = replicate (n - m) False @ bin_to_bl
(min n m) bin"
  unfolding bin_to_bl_def by (rule bin_to_bl_aux_bintr)

lemma bl_to_bin_rep_False: "bl_to_bin (replicate n False) = 0"
  by (induct n) auto

lemma len_bin_to_bl_aux: "length (bin_to_bl_aux n w bs) = n + length
bs"
  by (fact size_bin_to_bl_aux)
```

```
lemma len_bin_to_bl: "length (bin_to_bl n w) = n"
  by (fact size_bin_to_bl)

lemma sign_bl_bin': "bin_sign (bl_to_bin_aux bs w) = bin_sign w"
  by (induction bs arbitrary: w) (simp_all add: bin_sign_def)

lemma sign_bl_bin: "bin_sign (bl_to_bin bs) = 0"
  by (simp add: bl_to_bin_def sign_bl_bin')

lemma bl_sbin_sign_aux: "hd (bin_to_bl_aux (Suc n) w bs) = (bin_sign
(signed_take_bit n w) = -1)"
  by (induction n arbitrary: w bs) (auto simp add: bin_sign_def even_iff_mod_2_eq_zero
bit_Suc)

lemma bl_sbin_sign: "hd (bin_to_bl (Suc n) w) = (bin_sign (signed_take_bit
n w) = -1)"
  unfolding bin_to_bl_def by (rule bl_sbin_sign_aux)

lemma bin_nth_of_bl_aux:
  "bit (bl_to_bin_aux bl w) n =
    (n < size bl ∧ rev bl ! n ∨ n ≥ length bl ∧ bit w (n - size bl))"
  apply (induction bl arbitrary: w)
   apply simp_all
  apply safe
                      apply (simp_all add: not_le nth_append bit_double_iff
even_bit_succ_iff split: if_splits)
  done

lemma bin_nth_of_bl: "bit (bl_to_bin bl) n = (n < length bl ∧ rev bl
! n)"
  by (simp add: bl_to_bin_def bin_nth_of_bl_aux)

lemma bin_nth_bl: "n < m ⟹ bit w n = nth (rev (bin_to_bl m w)) n"
  by (metis bin_bl_bin bin_nth_of_bl nth_bintr size_bin_to_bl)

lemma nth_bin_to_bl_aux:
  "n < m + length bl ⟹ (bin_to_bl_aux m w bl) ! n =
    (if n < m then bit w (m - 1 - n) else bl ! (n - m))"
  apply (induction bl arbitrary: w)
   apply simp_all
   apply (simp add: bin_nth_bl [of ‹m - Suc n› m] rev_nth flip: bin_to_bl_def)
   apply (metis One_nat_def Suc_pred add_diff_cancel_left'
     add_diff_cancel_right' bin_to_bl_aux_alt bin_to_bl_def
     diff_Suc_Suc diff_is_0_eq diff_zero less_Suc_eq_0_disj
     less_antisym less_imp_Suc_add list.size(3) nat_less_le nth_append
size_bin_to_bl_aux)
  done

lemma nth_bin_to_bl: "n < m ⟹ (bin_to_bl m w) ! n = bit w (m - Suc
```

```
n)"
  by (simp add: bin_to_bl_def nth_bin_to_bl_aux)

lemma takefill_bintrunc: "takefill False n bl = rev (bin_to_bl n (bl_to_bin
(rev bl)))"
  apply (rule nth_equalityI)
   apply simp
  apply (clarsimp simp: nth_takefill rev_nth nth_bin_to_bl bin_nth_of_bl)
  done

lemma bl_bin_bl_rtf: "bin_to_bl n (bl_to_bin bl) = rev (takefill False
n (rev bl))"
  by (simp add: takefill_bintrunc)

lemma bl_to_bin_lt2p_aux: "bl_to_bin_aux bs w < (w + 1) * (2 ^ length
bs)"
proof (induction bs arbitrary: w)
  case Nil
  then show ?case
    by simp
next
  case (Cons b bs)
  from Cons.IH [of <1 + 2 * w>] Cons.IH [of <2 * w>]
  show ?case
    apply (auto simp add: algebra_simps)
    apply (subst mult_2 [of <2 ^ length bs>])
    apply (simp only: add.assoc)
    apply (rule pos_add_strict)
     apply simp_all
    done
qed

lemma bl_to_bin_lt2p_drop: "bl_to_bin bs < 2 ^ length (dropWhile Not
bs)"
proof (induct bs)
  case Nil
  then show ?case by simp
next
  case (Cons b bs)
  with bl_to_bin_lt2p_aux[where w=1] show ?case
    by (simp add: bl_to_bin_def)
qed

lemma bl_to_bin_lt2p: "bl_to_bin bs < 2 ^ length bs"
  by (metis bin_bl_bin bintr_lt2p bl_bin_bl)

lemma bl_to_bin_ge2p_aux: "bl_to_bin_aux bs w ≥ w * (2 ^ length bs)"
proof (induction bs arbitrary: w)
  case Nil
```

```
  then show ?case
    by simp
next
  case (Cons b bs)
  from Cons.IH [of ‹1 + 2 * w›] Cons.IH [of ‹2 * w›]
  show ?case
    apply (auto simp add: algebra_simps)
    apply (rule add_le_imp_le_left [of ‹2 ^ length bs›])
    apply (rule add_increasing)
    apply simp_all
    done
qed

lemma bl_to_bin_ge0: "bl_to_bin bs ≥ 0"
  apply (unfold bl_to_bin_def)
  apply (rule xtrans(4))
   apply (rule bl_to_bin_ge2p_aux)
  apply simp
  done

lemma butlast_rest_bin: "butlast (bin_to_bl n w) = bin_to_bl (n - 1)
(w div 2)"
  apply (unfold bin_to_bl_def)
  apply (cases n, clarsimp)
  apply clarsimp
  apply (auto simp add: bin_to_bl_aux_alt)
  done

lemma butlast_bin_rest: "butlast bl = bin_to_bl (length bl - Suc 0) (bl_to_bin
bl div 2)"
  using butlast_rest_bin [where w="bl_to_bin bl" and n="length bl"] by
simp

lemma butlast_rest_bl2bin_aux:
  "bl ≠ [] ⟹ bl_to_bin_aux (butlast bl) w = bl_to_bin_aux bl w div 2"
  by (induct bl arbitrary: w) auto

lemma butlast_rest_bl2bin: "bl_to_bin (butlast bl) = bl_to_bin bl div
2"
  by (cases bl) (auto simp: bl_to_bin_def butlast_rest_bl2bin_aux)

lemma trunc_bl2bin_aux:
  "take_bit m (bl_to_bin_aux bl w) =
    bl_to_bin_aux (drop (length bl - m) bl) (take_bit (m - length bl)
w)"
proof (induct bl arbitrary: w)
  case Nil
  show ?case by simp
next
```

```
  case (Cons b bl)
  show ?case
  proof (cases "m - length bl")
    case 0
    then have "Suc (length bl) - m = Suc (length bl - m)" by simp
    with Cons show ?thesis by simp
  next
    case (Suc n)
    then have "m - Suc (length bl) = n" by simp
    with Cons Suc show ?thesis by (simp add: take_bit_Suc ac_simps)
  qed
qed

lemma trunc_bl2bin: "take_bit m (bl_to_bin bl) = bl_to_bin (drop (length
bl - m) bl)"
  by (simp add: bl_to_bin_def trunc_bl2bin_aux)

lemma trunc_bl2bin_len [simp]: "take_bit (length bl) (bl_to_bin bl) =
bl_to_bin bl"
  by (simp add: trunc_bl2bin)

lemma bl2bin_drop: "bl_to_bin (drop k bl) = take_bit (length bl - k)
(bl_to_bin bl)"
  apply (rule trans)
   prefer 2
   apply (rule trunc_bl2bin [symmetric])
  apply (cases "k ≤ length bl")
   apply auto
  done

lemma take_rest_power_bin: "m ≤ n ⟹ take m (bin_to_bl n w) = bin_to_bl
m (((λw. w div 2) ^^ (n - m)) w)"
  apply (rule nth_equalityI)
   apply simp
  apply (clarsimp simp add: nth_bin_to_bl nth_rest_power_bin)
  done

lemma last_bin_last': "size xs > 0 ⟹ last xs ⟷ odd (bl_to_bin_aux
xs w)"
  by (induct xs arbitrary: w) auto

lemma last_bin_last: "size xs > 0 ⟹ last xs ⟷ odd (bl_to_bin xs)"
  unfolding bl_to_bin_def by (erule last_bin_last')

lemma bin_last_last: "odd w ⟷ last (bin_to_bl (Suc n) w)"
  by (simp add: bin_to_bl_def) (auto simp: bin_to_bl_aux_alt)

lemma drop_bin2bl_aux:
  "drop m (bin_to_bl_aux n bin bs) =
```

```
    bin_to_bl_aux (n - m) bin (drop (m - n) bs)"
  apply (induction n arbitrary: m bin bs)
   apply auto
  apply (case_tac "m ≤ n")
   apply (auto simp add: not_le Suc_diff_le)
  apply (case_tac "m - n")
   apply auto
  apply (use Suc_diff_Suc in fastforce)
  done

lemma drop_bin2bl: "drop m (bin_to_bl n bin) = bin_to_bl (n - m) bin"
  by (simp add: bin_to_bl_def drop_bin2bl_aux)

lemma take_bin2bl_lem1: "take m (bin_to_bl_aux m w bs) = bin_to_bl m
w"
  apply (induct m arbitrary: w bs)
   apply clarsimp
  apply clarsimp
  apply (simp add: bin_to_bl_aux_alt)
  apply (simp add: bin_to_bl_def)
  apply (simp add: bin_to_bl_aux_alt)
  done

lemma take_bin2bl_lem: "take m (bin_to_bl_aux (m + n) w bs) = take m
(bin_to_bl (m + n) w)"
  by (induct n arbitrary: w bs) (simp_all (no_asm) add: bin_to_bl_def
take_bin2bl_lem1, simp)

lemma bin_split_take: "bin_split n c = (a, b) ⟹ bin_to_bl m a = take
m (bin_to_bl (m + n) c)"
  apply (induct n arbitrary: b c)
   apply clarsimp
  apply (clarsimp simp: Let_def split: prod.split_asm)
  apply (simp add: bin_to_bl_def)
  apply (simp add: take_bin2bl_lem drop_bit_Suc)
  done

lemma bin_to_bl_drop_bit:
  "k = m + n ⟹ bin_to_bl m (drop_bit n c) = take m (bin_to_bl k c)"
  using bin_split_take by simp

lemma bin_split_take1:
  "k = m + n ⟹ bin_split n c = (a, b) ⟹ bin_to_bl m a = take m (bin_to_bl
k c)"
  using bin_split_take by simp

lemma bl_bin_bl_rep_drop:
  "bin_to_bl n (bl_to_bin bl) =
    replicate (n - length bl) False @ drop (length bl - n) bl"
```

```
  by (simp add: bl_to_bin_inj bl_to_bin_rep_F trunc_bl2bin)

lemma bl_to_bin_aux_cat:
  "bl_to_bin_aux bs (concat_bit nv v w) =
    concat_bit (nv + length bs) (bl_to_bin_aux bs v) w"
  by (rule bit_eqI)
    (auto simp add: bin_nth_of_bl_aux bin_nth_cat algebra_simps)

lemma bin_to_bl_aux_cat:
  "bin_to_bl_aux (nv + nw) (concat_bit nw w v) bs =
    bin_to_bl_aux nv v (bin_to_bl_aux nw w bs)"
  by (induction nw arbitrary: w bs) (simp_all add: concat_bit_Suc)

lemma bl_to_bin_aux_alt: "bl_to_bin_aux bs w = concat_bit (length bs)
(bl_to_bin bs) w"
  using bl_to_bin_aux_cat [where nv = "0" and v = "0"]
  by (simp add: bl_to_bin_def [symmetric])

lemma bin_to_bl_cat:
  "bin_to_bl (nv + nw) (concat_bit nw w v) =
    bin_to_bl_aux nv v (bin_to_bl nw w)"
  by (simp add: bin_to_bl_def bin_to_bl_aux_cat)

lemmas bl_to_bin_aux_app_cat =
  trans [OF bl_to_bin_aux_append bl_to_bin_aux_alt]

lemmas bin_to_bl_aux_cat_app =
  trans [OF bin_to_bl_aux_cat bin_to_bl_aux_alt]

lemma bl_to_bin_app_cat:
  "bl_to_bin (bsa @ bs) = concat_bit (length bs) (bl_to_bin bs) (bl_to_bin
bsa)"
  by (simp only: bl_to_bin_aux_app_cat bl_to_bin_def)

lemma bin_to_bl_cat_app:
  "bin_to_bl (n + nw) (concat_bit nw wa w) = bin_to_bl n w @ bin_to_bl
nw wa"
  by (simp only: bin_to_bl_def bin_to_bl_aux_cat_app)
```

`bl_to_bin_app_cat_alt` and `bl_to_bin_app_cat` are easily interderivable.

```
lemma bl_to_bin_app_cat_alt: "concat_bit n w (bl_to_bin cs) = bl_to_bin
(cs @ bin_to_bl n w)"
  by (simp add: bl_to_bin_app_cat)

lemma mask_lem: "(bl_to_bin (True # replicate n False)) = bl_to_bin (replicate
n True) + 1"
  apply (unfold bl_to_bin_def)
  apply (induct n)
   apply simp
```

```
  apply (simp only: Suc_eq_plus1 replicate_add append_Cons [symmetric]
bl_to_bin_aux_append)
  apply simp
  done

lemma bin_exhaust:
  "(⋀x b. bin = of_bool b + 2 * x ⟹ Q) ⟹ Q" for bin :: int
  apply (cases ‹even bin›)
   apply (auto elim!: evenE oddE)
   apply fastforce
  apply fastforce
  done

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

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

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

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

lemma size_rbl_pred: "length (rbl_pred bl) = length bl"
  by (induct bl) auto

lemma size_rbl_succ: "length (rbl_succ bl) = length bl"
  by (induct bl) auto

lemma size_rbl_add: "length (rbl_add bl cl) = length bl"
  by (induct bl arbitrary: cl) (auto simp: Let_def size_rbl_succ)

lemma size_rbl_mult: "length (rbl_mult bl cl) = length bl"
```

```
  by (induct bl arbitrary: cl) (auto simp add: Let_def size_rbl_add)

lemmas rbl_sizes [simp] =
  size_rbl_pred size_rbl_succ size_rbl_add size_rbl_mult

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

lemma rbl_add_app2: "length blb ≥ length bla ⟹ rbl_add bla (blb @
blc) = rbl_add bla blb"
  apply (induct bla arbitrary: blb)
   apply simp
  apply clarsimp
  apply (case_tac blb, clarsimp)
  apply (clarsimp simp: Let_def)
  done

lemma rbl_add_take2:
  "length blb ≥ length bla ⟹ rbl_add bla (take (length bla) blb) = rbl_add
bla blb"
  apply (induct bla arbitrary: blb)
   apply simp
  apply clarsimp
  apply (case_tac blb, clarsimp)
  apply (clarsimp simp: Let_def)
  done

lemma rbl_mult_app2: "length blb ≥ length bla ⟹ rbl_mult bla (blb
@ blc) = rbl_mult bla blb"
  apply (induct bla arbitrary: blb)
   apply simp
  apply clarsimp
  apply (case_tac blb, clarsimp)
  apply (clarsimp simp: Let_def rbl_add_app2)
  done

lemma rbl_mult_take2:
  "length blb ≥ length bla ⟹ rbl_mult bla (take (length bla) blb) =
rbl_mult bla blb"
  apply (rule trans)
   apply (rule rbl_mult_app2 [symmetric])
   apply simp
  apply (rule_tac f = "rbl_mult bla" in arg_cong)
  apply (rule append_take_drop_id)
  done

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

```
      (y ⟶ ((x ⟶ P (False # rbl_succ ws)) ∧ (¬ x ⟶ P (True # ws))))
∧
      (¬ y ⟶ P (x # ws)))"
  by (cases y) (auto simp: Let_def)

lemma rbl_mult_split:
  "P (rbl_mult (y # ys) xs) =
    (∀ws. length ws = Suc (length ys) ⟶ ws = False # rbl_mult ys xs
⟶
      (y ⟶ P (rbl_add ws xs)) ∧ (¬ y ⟶ P ws))"
  by (auto simp: Let_def)

lemma rbl_pred: "rbl_pred (rev (bin_to_bl n bin)) = rev (bin_to_bl n
(bin - 1))"
proof (unfold bin_to_bl_def, induction n arbitrary: bin)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  obtain b k where ‹bin = of_bool b + 2 * k›
    using bin_exhaust by blast
  moreover have ‹(2 * k - 1) div 2 = k - 1›
    by simp
  ultimately show ?case
    using Suc [of ‹bin div 2›]
    by simp (auto simp add: bin_to_bl_aux_alt)
qed

lemma rbl_succ: "rbl_succ (rev (bin_to_bl n bin)) = rev (bin_to_bl n
(bin + 1))"
  apply (unfold bin_to_bl_def)
  apply (induction n arbitrary: bin)
   apply simp_all
  apply (case_tac bin rule: bin_exhaust)
   apply (simp_all add: bin_to_bl_aux_alt ac_simps)
  done

lemma rbl_add:
  "⋀bina binb. rbl_add (rev (bin_to_bl n bina)) (rev (bin_to_bl n binb))
=
    rev (bin_to_bl n (bina + binb))"
  apply (unfold bin_to_bl_def)
  apply (induct n)
   apply simp
  apply clarsimp
  apply (case_tac bina rule: bin_exhaust)
  apply (case_tac binb rule: bin_exhaust)
  apply (case_tac b)
```

```
   apply (case_tac [!] "ba")
     apply (auto simp: rbl_succ bin_to_bl_aux_alt Let_def ac_simps)
  done

lemma rbl_add_long:
  "m ≥ n ⟹ rbl_add (rev (bin_to_bl n bina)) (rev (bin_to_bl m binb))
=
    rev (bin_to_bl n (bina + binb))"
  apply (rule box_equals [OF _ rbl_add_take2 rbl_add])
   apply (rule_tac f = "rbl_add (rev (bin_to_bl n bina))" in arg_cong)
   apply (rule rev_swap [THEN iffD1])
   apply (simp add: rev_take drop_bin2bl)
  apply simp
  done

lemma rbl_mult_gt1:
  "m ≥ length bl ⟹
    rbl_mult bl (rev (bin_to_bl m binb)) =
    rbl_mult bl (rev (bin_to_bl (length bl) binb))"
  apply (rule trans)
   apply (rule rbl_mult_take2 [symmetric])
   apply simp_all
  apply (rule_tac f = "rbl_mult bl" in arg_cong)
  apply (rule rev_swap [THEN iffD1])
  apply (simp add: rev_take drop_bin2bl)
  done

lemma rbl_mult_gt:
  "m > n ⟹
    rbl_mult (rev (bin_to_bl n bina)) (rev (bin_to_bl m binb)) =
    rbl_mult (rev (bin_to_bl n bina)) (rev (bin_to_bl n binb))"
  by (auto intro: trans [OF rbl_mult_gt1])

lemmas rbl_mult_Suc = lessI [THEN rbl_mult_gt]

lemma rbbl_Cons: "b # rev (bin_to_bl n x) = rev (bin_to_bl (Suc n) (of_bool
b + 2 * x))"
  by (simp add: bin_to_bl_def) (simp add: bin_to_bl_aux_alt)

lemma rbl_mult:
  "rbl_mult (rev (bin_to_bl n bina)) (rev (bin_to_bl n binb)) =
    rev (bin_to_bl n (bina * binb))"
  apply (induct n arbitrary: bina binb)
   apply simp_all
  apply (unfold bin_to_bl_def)
  apply clarsimp
  apply (case_tac bina rule: bin_exhaust)
  apply (case_tac binb rule: bin_exhaust)
  apply (simp_all add: bin_to_bl_aux_alt)
```

```
  apply (simp_all add: rbbl_Cons rbl_mult_Suc rbl_add algebra_simps)
  done

lemma sclem: "size (concat (map (bin_to_bl n) xs)) = length xs * n"
  by (simp add: length_concat comp_def sum_list_triv)

lemma bin_cat_foldl_lem:
  "foldl (λu k. concat_bit n k u) x xs =
    concat_bit (size xs * n) (foldl (λu k. concat_bit n k u) y xs) x"
  apply (induct xs arbitrary: x)
   apply simp
  apply (simp (no_asm))
  apply (frule asm_rl)
  apply (drule meta_spec)
  apply (erule trans)
  apply (drule_tac x = "concat_bit n a y" in meta_spec)
  apply (simp add: bin_cat_assoc_sym)
  done

lemma bin_rcat_bl: "bin_rcat n wl = bl_to_bin (concat (map (bin_to_bl
n) wl))"
  apply (unfold bin_rcat_eq_foldl)
  apply (rule sym)
  apply (induct wl)
   apply (auto simp add: bl_to_bin_append)
  apply (simp add: bl_to_bin_aux_alt sclem)
  apply (simp add: bin_cat_foldl_lem [symmetric])
  done

lemma bin_last_bl_to_bin: "odd (bl_to_bin bs) ⟷ bs ≠ [] ∧ last bs"
by(cases "bs = []")(auto simp add: bl_to_bin_def last_bin_last'[where
w=0])

lemma bin_rest_bl_to_bin: "bl_to_bin bs div 2 = bl_to_bin (butlast bs)"
by(cases "bs = []")(simp_all add: bl_to_bin_def butlast_rest_bl2bin_aux)

lemma bl_xor_aux_bin:
  "map2 (λx y. x ≠ y) (bin_to_bl_aux n v bs) (bin_to_bl_aux n w cs) =
    bin_to_bl_aux n (v XOR w) (map2 (λx y. x ≠ y) bs cs)"
  apply (induction n arbitrary: v w bs cs)
   apply auto
  apply (case_tac v rule: bin_exhaust)
  apply (case_tac w rule: bin_exhaust)
  apply clarsimp
  done

lemma bl_or_aux_bin:
  "map2 (∨) (bin_to_bl_aux n v bs) (bin_to_bl_aux n w cs) =
    bin_to_bl_aux n (v OR w) (map2 (∨) bs cs)"
```

```
  by (induct n arbitrary: v w bs cs) simp_all

lemma bl_and_aux_bin:
  "map2 (∧) (bin_to_bl_aux n v bs) (bin_to_bl_aux n w cs) =
    bin_to_bl_aux n (v AND w) (map2 (∧) bs cs)"
  by (induction n arbitrary: v w bs cs) simp_all

lemma bl_not_aux_bin: "map Not (bin_to_bl_aux n w cs) = bin_to_bl_aux
n (NOT w) (map Not cs)"
  by (induct n arbitrary: w cs) auto

lemma bl_not_bin: "map Not (bin_to_bl n w) = bin_to_bl n (NOT w)"
  by (simp add: bin_to_bl_def bl_not_aux_bin)

lemma bl_and_bin: "map2 (∧) (bin_to_bl n v) (bin_to_bl n w) = bin_to_bl
n (v AND w)"
  by (simp add: bin_to_bl_def bl_and_aux_bin)

lemma bl_or_bin: "map2 (∨) (bin_to_bl n v) (bin_to_bl n w) = bin_to_bl
n (v OR w)"
  by (simp add: bin_to_bl_def bl_or_aux_bin)

lemma bl_xor_bin: "map2 (≠) (bin_to_bl n v) (bin_to_bl n w) = bin_to_bl
n (v XOR w)"
  using bl_xor_aux_bin by (simp add: bin_to_bl_def)
```

10.6 Type 'a word

```
lift_definition of_bl :: ‹bool list ⇒ 'a::len word›
  is bl_to_bin .

lift_definition to_bl :: ‹'a::len word ⇒ bool list›
  is ‹bin_to_bl LENGTH('a)›
  by (simp add: bl_to_bin_inj)

lemma to_bl_eq:
  ‹to_bl w = bin_to_bl (LENGTH('a)) (uint w)›
  for w :: ‹'a::len word›
  by transfer simp

lemma bit_of_bl_iff [bit_simps]:
  ‹bit (of_bl bs :: 'a word) n ⟷ rev bs ! n ∧ n < LENGTH('a::len) ∧
n < length bs›
  by transfer (simp add: bin_nth_of_bl ac_simps)

lemma rev_to_bl_eq:
  ‹rev (to_bl w) = map (bit w) [0..<LENGTH('a)]›
  for w :: ‹'a::len word›
  apply (rule nth_equalityI)
```

```
   apply (simp add: to_bl.rep_eq)
  apply (simp add: bin_nth_bl bit_word.rep_eq to_bl.rep_eq)
  done

lemma to_bl_eq_rev:
  ‹to_bl w = map (bit w) (rev [0..<LENGTH('a)])›
  for w :: ‹'a::len word›
  using rev_to_bl_eq [of w]
  apply (subst rev_is_rev_conv [symmetric])
  apply (simp add: rev_map)
  done

lemma of_bl_rev_eq:
  ‹of_bl (rev bs) = horner_sum of_bool 2 bs›
  apply (rule bit_word_eqI)
  apply (simp add: bit_of_bl_iff)
  apply transfer
  apply (simp add: bit_horner_sum_bit_iff ac_simps)
  done

lemma of_bl_eq:
  ‹of_bl bs = horner_sum of_bool 2 (rev bs)›
  using of_bl_rev_eq [of ‹rev bs›] by simp

lemma bshiftr1_eq:
  ‹bshiftr1 b w = of_bl (b # butlast (to_bl w))›
  apply (rule bit_word_eqI)
  apply (auto simp add: bit_simps to_bl_eq_rev nth_append rev_nth nth_butlast
not_less simp flip: bit_Suc)
  apply (metis Suc_pred len_gt_0 less_eq_decr_length_iff not_bit_length
verit_la_disequality)
  done

lemma length_to_bl_eq:
  ‹length (to_bl w) = LENGTH('a)›
  for w :: ‹'a::len word›
  by transfer simp

lemma word_rotr_eq:
  ‹word_rotr n w = of_bl (rotater n (to_bl w))›
  apply (rule bit_word_eqI)
  subgoal for n
    apply (cases ‹n < LENGTH('a)›)
     apply (simp_all add: bit_word_rotr_iff bit_of_bl_iff rotater_rev
length_to_bl_eq nth_rotate rev_to_bl_eq ac_simps)
    done
  done

lemma word_rotl_eq:
```

```
  ‹word_rotl n w = of_bl (rotate n (to_bl w))›
proof -
  have ‹rotate n (to_bl w) = rev (rotater n (rev (to_bl w)))›
    by (simp add: rotater_rev')
  then show ?thesis
    apply (simp add: word_rotl_eq_word_rotr bit_of_bl_iff length_to_bl_eq
rev_to_bl_eq)
    apply (rule bit_word_eqI)
    subgoal for n
      apply (cases ‹n < LENGTH('a)›)
       apply (simp_all add: bit_word_rotr_iff bit_of_bl_iff nth_rotater)
      done
    done
qed

lemma to_bl_def': "(to_bl :: 'a::len word ⇒ bool list) = bin_to_bl (LENGTH('a))
∘ uint"
  by transfer (simp add: fun_eq_iff)
```

— type definitions theorem for in terms of equivalent bool list

```
lemma td_bl:
  "type_definition
    (to_bl :: 'a::len word ⇒ bool list)
    of_bl
    {bl. length bl = LENGTH('a)}"
  apply (standard; transfer)
  apply (auto dest: sym)
  done

global_interpretation word_bl:
  type_definition
    "to_bl :: 'a::len word ⇒ bool list"
    of_bl
    "{bl. length bl = LENGTH('a::len)}"
  by (fact td_bl)

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

lemma word_size_bl: "size w = size (to_bl w)"
  by (auto simp: word_size)

lemma to_bl_use_of_bl: "to_bl w = bl ⟷ w = of_bl bl ∧ length bl =
length (to_bl w)"
  by (fastforce elim!: word_bl.Abs_inverse [unfolded mem_Collect_eq])

lemma length_bl_gt_0 [iff]: "0 < length (to_bl x)"
  for x :: "'a::len word"
  unfolding word_bl_Rep' by (rule len_gt_0)
```

```
lemma bl_not_Nil [iff]: "to_bl x ≠ []"
  for x :: "'a::len word"
  by (fact length_bl_gt_0 [unfolded length_greater_0_conv])

lemma length_bl_neq_0 [iff]: "length (to_bl x) ≠ 0"
  for x :: "'a::len word"
  by (fact length_bl_gt_0 [THEN gr_implies_not0])

lemma hd_to_bl_iff:
  ‹hd (to_bl w) ⟷ bit w (LENGTH('a) - 1)›
  for w :: ‹'a::len word›
  by (simp add: to_bl_eq_rev hd_map hd_rev)

lemma hd_bl_sign_sint: "hd (to_bl w) = (bin_sign (sint w) = -1)"
  by (simp add: hd_to_bl_iff bit_last_iff bin_sign_def)

lemma of_bl_drop':
  "lend = length bl - LENGTH('a::len) ⟹
    of_bl (drop lend bl) = (of_bl bl :: 'a word)"
  by transfer (simp flip: trunc_bl2bin)

lemma test_bit_of_bl:
  "bit (of_bl bl::'a::len word) n = (rev bl ! n ∧ n < LENGTH('a) ∧ n <
length bl)"
  by transfer (simp add: bin_nth_of_bl ac_simps)

lemma no_of_bl: "(numeral bin ::'a::len word) = of_bl (bin_to_bl (LENGTH('a))
(numeral bin))"
  by transfer simp

lemma uint_bl: "to_bl w = bin_to_bl (size w) (uint w)"
  by transfer simp

lemma to_bl_bin: "bl_to_bin (to_bl w) = uint w"
  by (simp add: uint_bl word_size)

lemma to_bl_of_bin: "to_bl (word_of_int bin::'a::len word) = bin_to_bl
(LENGTH('a)) bin"
  by (auto simp: uint_bl word_ubin.eq_norm word_size)

lemma to_bl_numeral [simp]:
  "to_bl (numeral bin::'a::len word) =
    bin_to_bl (LENGTH('a)) (numeral bin)"
  unfolding word_numeral_alt by (rule to_bl_of_bin)

lemma to_bl_neg_numeral [simp]:
  "to_bl (- numeral bin::'a::len word) =
    bin_to_bl (LENGTH('a)) (- numeral bin)"
  unfolding word_neg_numeral_alt by (rule to_bl_of_bin)
```

```
lemma to_bl_to_bin [simp] : "bl_to_bin (to_bl w) = uint w"
  by (simp add: uint_bl word_size)

lemma uint_bl_bin: "bl_to_bin (bin_to_bl (LENGTH('a)) (uint x)) = uint
x"
  for x :: "'a::len word"
  by (rule trans [OF bin_bl_bin word_ubin.norm_Rep])

lemma ucast_bl: "ucast w = of_bl (to_bl w)"
  by transfer simp

lemma ucast_down_bl:
  ‹(ucast :: 'a::len word ⇒ 'b::len word) (of_bl bl) = of_bl bl›
    if ‹is_down (ucast :: 'a::len word ⇒ 'b::len word)›
  using that by transfer simp

lemma of_bl_append_same: "of_bl (X @ to_bl w) = w"
  by transfer (simp add: bl_to_bin_app_cat)

lemma ucast_of_bl_up:
  ‹ucast (of_bl bl :: 'a::len word) = of_bl bl›
  if ‹size bl ≤ size (of_bl bl :: 'a::len word)›
  using that
  apply transfer
  apply (rule bit_eqI)
  apply (auto simp add: bit_take_bit_iff)
  apply (subst (asm) trunc_bl2bin_len [symmetric])
  apply (auto simp only: bit_take_bit_iff)
  done

lemma word_rev_tf:
  "to_bl (of_bl bl::'a::len word) =
    rev (takefill False (LENGTH('a)) (rev bl))"
  by transfer (simp add: bl_bin_bl_rtf)

lemma word_rep_drop:
  "to_bl (of_bl bl::'a::len word) =
    replicate (LENGTH('a) - length bl) False @
    drop (length bl - LENGTH('a)) bl"
  by (simp add: word_rev_tf takefill_alt rev_take)

lemma to_bl_ucast:
  "to_bl (ucast (w::'b::len word) ::'a::len word) =
    replicate (LENGTH('a) - LENGTH('b)) False @
    drop (LENGTH('b) - LENGTH('a)) (to_bl w)"
  apply (unfold ucast_bl)
  apply (rule trans)
   apply (rule word_rep_drop)
```

```
  apply simp
  done

lemma ucast_up_app:
  ‹to_bl (ucast w :: 'b::len word) = replicate n False @ (to_bl w)›
    if ‹source_size (ucast :: 'a word ⇒ 'b word) + n = target_size (ucast
:: 'a word ⇒ 'b word)›
    for w :: ‹'a::len word›
  using that
  by (auto simp add : source_size target_size to_bl_ucast)

lemma ucast_down_drop [OF refl]:
  "uc = ucast ⟹ source_size uc = target_size uc + n ⟹
    to_bl (uc w) = drop n (to_bl w)"
  by (auto simp add : source_size target_size to_bl_ucast)

lemma scast_down_drop [OF refl]:
  "sc = scast ⟹ source_size sc = target_size sc + n ⟹
    to_bl (sc w) = drop n (to_bl w)"
  apply (subgoal_tac "sc = ucast")
   apply safe
   apply simp
   apply (erule ucast_down_drop)
  apply (rule down_cast_same [symmetric])
  apply (simp add : source_size target_size is_down)
  done

lemma word_0_bl [simp]: "of_bl [] = 0"
  by transfer simp

lemma word_1_bl: "of_bl [True] = 1"
  by transfer (simp add: bl_to_bin_def)

lemma of_bl_0 [simp]: "of_bl (replicate n False) = 0"
  by transfer (simp add: bl_to_bin_rep_False)

lemma to_bl_0 [simp]: "to_bl (0::'a::len word) = replicate (LENGTH('a))
False"
  by (simp add: uint_bl word_size bin_to_bl_zero)

— links with rbl operations
lemma word_succ_rbl: "to_bl w = bl ⟹ to_bl (word_succ w) = rev (rbl_succ
(rev bl))"
  by transfer (simp add: rbl_succ)

lemma word_pred_rbl: "to_bl w = bl ⟹ to_bl (word_pred w) = rev (rbl_pred
(rev bl))"
  by transfer (simp add: rbl_pred)
```

```
lemma word_add_rbl:
  "to_bl v = vbl ⟹ to_bl w = wbl ⟹
    to_bl (v + w) = rev (rbl_add (rev vbl) (rev wbl))"
  apply transfer
  apply (drule sym)
  apply (drule sym)
  apply (simp add: rbl_add)
  done

lemma word_mult_rbl:
  "to_bl v = vbl ⟹ to_bl w = wbl ⟹
    to_bl (v * w) = rev (rbl_mult (rev vbl) (rev wbl))"
  apply transfer
  apply (drule sym)
  apply (drule sym)
  apply (simp add: rbl_mult)
  done

lemma rtb_rbl_ariths:
  "rev (to_bl w) = ys ⟹ rev (to_bl (word_succ w)) = rbl_succ ys"
  "rev (to_bl w) = ys ⟹ rev (to_bl (word_pred w)) = rbl_pred ys"
  "rev (to_bl v) = ys ⟹ rev (to_bl w) = xs ⟹ rev (to_bl (v * w)) =
rbl_mult ys xs"
  "rev (to_bl v) = ys ⟹ rev (to_bl w) = xs ⟹ rev (to_bl (v + w)) =
rbl_add ys xs"
  by (auto simp: rev_swap [symmetric] word_succ_rbl word_pred_rbl word_mult_rbl
word_add_rbl)

lemma of_bl_length_less:
  ‹(of_bl x :: 'a::len word) < 2 ^ k›
    if ‹length x = k› ‹k < LENGTH('a)›
proof -
  from that have ‹length x < LENGTH('a)›
    by simp
  then have ‹(of_bl x :: 'a::len word) < 2 ^ length x›
    apply (simp add: of_bl_eq)
    apply transfer
    apply (simp add: take_bit_horner_sum_bit_eq)
    apply (subst length_rev [symmetric])
    apply (simp only: horner_sum_of_bool_2_less)
    done
  with that show ?thesis
    by simp
qed

lemma word_eq_rbl_eq: "x = y ⟷ rev (to_bl x) = rev (to_bl y)"
  by simp

lemma bl_word_not: "to_bl (NOT w) = map Not (to_bl w)"
```

```
  by transfer (simp add: bl_not_bin)

lemma bl_word_xor: "to_bl (v XOR w) = map2 (≠) (to_bl v) (to_bl w)"
  by transfer (simp flip: bl_xor_bin)

lemma bl_word_or: "to_bl (v OR w) = map2 (∨) (to_bl v) (to_bl w)"
  by transfer (simp flip: bl_or_bin)

lemma bl_word_and: "to_bl (v AND w) = map2 (∧) (to_bl v) (to_bl w)"
  by transfer (simp flip: bl_and_bin)

lemma bin_nth_uint': "bit (uint w) n ⟷ rev (bin_to_bl (size w) (uint
w)) ! n ∧ n < size w"
  apply (unfold word_size)
  apply (safe elim!: bin_nth_uint_imp)
   apply (frule bin_nth_uint_imp)
   apply (fast dest!: bin_nth_bl)+
  done

lemmas bin_nth_uint = bin_nth_uint' [unfolded word_size]

lemma test_bit_bl: "bit w n ⟷ rev (to_bl w) ! n ∧ n < size w"
  by transfer (auto simp add: bin_nth_bl)

lemma to_bl_nth: "n < size w ⟹ to_bl w ! n = bit w (size w - Suc n)"
  by (simp add: word_size rev_nth test_bit_bl)

lemma map_bit_interval_eq:
  ‹map (bit w) [0..<n] = takefill False n (rev (to_bl w))› for w :: ‹'a::len
word›
proof (rule nth_equalityI)
  show ‹length (map (bit w) [0..<n]) =
    length (takefill False n (rev (to_bl w)))›
    by simp
  fix m
  assume ‹m < length (map (bit w) [0..<n])›
  then have ‹m < n›
    by simp
  then have ‹bit w m ⟷ takefill False n (rev (to_bl w)) ! m›
    by (auto simp add: nth_takefill not_less rev_nth to_bl_nth word_size
dest: bit_imp_le_length)
  with ‹m < n ›show ‹map (bit w) [0..<n] ! m ⟷ takefill False n (rev
(to_bl w)) ! m›
    by simp
qed

lemma to_bl_unfold:
  ‹to_bl w = rev (map (bit w) [0..<LENGTH('a)])› for w :: ‹'a::len word›
  by (simp add: map_bit_interval_eq takefill_bintrunc to_bl_def flip:
```

```
bin_to_bl_def)

lemma nth_rev_to_bl:
  ‹rev (to_bl w) ! n ⟷ bit w n›
  if ‹n < LENGTH('a)› for w :: ‹'a::len word›
  using that by (simp add: to_bl_unfold)

lemma nth_to_bl:
  ‹to_bl w ! n ⟷ bit w (LENGTH('a) - Suc n)›
  if ‹n < LENGTH('a)› for w :: ‹'a::len word›
  using that by (simp add: to_bl_unfold rev_nth)

lemma of_bl_rep_False: "of_bl (replicate n False @ bs) = of_bl bs"
  by (auto simp: of_bl_def bl_to_bin_rep_F)

lemma [code abstract]:
  ‹Word.the_int (of_bl bs :: 'a word) = horner_sum of_bool 2 (take LENGTH('a::len)
(rev bs))›
  apply (simp add: of_bl_eq flip: take_bit_horner_sum_bit_eq)
  apply transfer
  apply simp
  done

lemma [code]:
  ‹to_bl w = map (bit w) (rev [0..<LENGTH('a::len)])›
  for w :: ‹'a::len word›
  by (fact to_bl_eq_rev)

lemma word_reverse_eq_of_bl_rev_to_bl:
  ‹word_reverse w = of_bl (rev (to_bl w))›
  by (rule bit_word_eqI)
    (auto simp add: bit_word_reverse_iff bit_of_bl_iff nth_to_bl)

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

lemma to_bl_word_rev: "to_bl (word_reverse w) = rev (to_bl w)"
  by (rule nth_equalityI) (simp_all add: nth_rev_to_bl word_reverse_def
word_rep_drop flip: of_bl_eq)

lemma to_bl_n1 [simp]: "to_bl (-1::'a::len word) = replicate (LENGTH('a))
True"
  apply (rule word_bl.Abs_inverse')
   apply simp
  apply (rule word_eqI)
  apply (clarsimp simp add: word_size)
  apply (auto simp add: word_bl.Abs_inverse test_bit_bl word_size)
  done
```

```
lemma rbl_word_or: "rev (to_bl (x OR y)) = map2 (∨) (rev (to_bl x))
(rev (to_bl y))"
  by (simp add: zip_rev bl_word_or rev_map)

lemma rbl_word_and: "rev (to_bl (x AND y)) = map2 (∧) (rev (to_bl x))
(rev (to_bl y))"
  by (simp add: zip_rev bl_word_and rev_map)

lemma rbl_word_xor: "rev (to_bl (x XOR y)) = map2 (≠) (rev (to_bl x))
(rev (to_bl y))"
  by (simp add: zip_rev bl_word_xor rev_map)

lemma rbl_word_not: "rev (to_bl (NOT x)) = map Not (rev (to_bl x))"
  by (simp add: bl_word_not rev_map)

lemma bshiftr1_numeral [simp]:
  ‹bshiftr1 b (numeral w :: 'a word) = of_bl (b # butlast (bin_to_bl LENGTH('a::len)
(numeral w)))›
  by (rule bit_word_eqI) (auto simp add: bit_simps rev_nth nth_append
nth_butlast nth_bin_to_bl simp flip: bit_Suc)

lemma bshiftr1_bl: "to_bl (bshiftr1 b w) = b # butlast (to_bl w)"
  unfolding bshiftr1_eq by (rule word_bl.Abs_inverse) simp

lemma shiftl1_of_bl: "shiftl1 (of_bl bl) = of_bl (bl @ [False])"
  apply (rule bit_word_eqI)
  apply (simp add: bit_simps)
  subgoal for n
    apply (cases n)
     apply simp_all
    done
  done

lemma shiftl1_bl: "shiftl1 w = of_bl (to_bl w @ [False])"
  apply (rule bit_word_eqI)
  apply (simp add: bit_simps)
  subgoal for n
    apply (cases n)
     apply (simp_all add: nth_rev_to_bl)
    done
  done

lemma bl_shiftl1: "to_bl (shiftl1 w) = tl (to_bl w) @ [False]"
  for w :: "'a::len word"
  by (simp add: shiftl1_bl word_rep_drop drop_Suc drop_Cons') (fast intro!:
Suc_leI)

lemma to_bl_double_eq:
  ‹to_bl (2 * w) = tl (to_bl w) @ [False]›
```

```
  using bl_shiftl1 [of w] by (simp add: shiftl1_def ac_simps)

— Generalized version of bl_shiftl1. Maybe this one should replace it?
lemma bl_shiftl1': "to_bl (shiftl1 w) = tl (to_bl w @ [False])"
  by (simp add: shiftl1_bl word_rep_drop drop_Suc del: drop_append)

lemma shiftr1_bl:
  ‹shiftr1 w = of_bl (butlast (to_bl w))›
proof (rule bit_word_eqI)
  fix n
  assume ‹n < LENGTH('a)›
  show ‹bit (shiftr1 w) n ⟷ bit (of_bl (butlast (to_bl w)) :: 'a word)
n›
  proof (cases ‹n = LENGTH('a) - 1›)
    case True
    then show ?thesis
      by (simp add: bit_shiftr1_iff bit_of_bl_iff)
  next
    case False
    with ‹n < LENGTH('a)›
    have ‹n < LENGTH('a) - 1›
      by simp
    with ‹n < LENGTH('a)› show ?thesis
      by (simp add: bit_shiftr1_iff bit_of_bl_iff rev_nth nth_butlast
        word_size to_bl_nth)
  qed
qed

lemma bl_shiftr1: "to_bl (shiftr1 w) = False # butlast (to_bl w)"
  for w :: "'a::len word"
  by (simp add: shiftr1_bl word_rep_drop len_gt_0 [THEN Suc_leI])

— Generalized version of bl_shiftr1. Maybe this one should replace it?
lemma bl_shiftr1': "to_bl (shiftr1 w) = butlast (False # to_bl w)"
  apply (rule word_bl.Abs_inverse')
   apply (simp del: butlast.simps)
  apply (simp add: shiftr1_bl of_bl_def)
  done

lemma bl_sshiftr1: "to_bl (sshiftr1 w) = hd (to_bl w) # butlast (to_bl
w)"
  for w :: "'a::len word"
proof (rule nth_equalityI)
  fix n
  assume ‹n < length (to_bl (sshiftr1 w))›
  then have ‹n < LENGTH('a)›
    by simp
  then show ‹to_bl (sshiftr1 w) ! n ⟷ (hd (to_bl w) # butlast (to_bl
w)) ! n›
```

```
    apply (cases n)
     apply (simp_all add: to_bl_nth word_size hd_conv_nth bit_sshiftr1_iff
nth_butlast Suc_diff_Suc nth_to_bl)
    done
qed simp

lemma drop_shiftr: "drop n (to_bl (w >> n)) = take (size w - n) (to_bl
w)"
  for w :: "'a::len word"
  apply (rule nth_equalityI)
   apply (simp_all add: word_size to_bl_nth bit_simps)
  done

lemma drop_sshiftr: "drop n (to_bl (w >>> n)) = take (size w - n) (to_bl
w)"
  for w :: "'a::len word"
  apply (rule nth_equalityI)
   apply (simp_all add: word_size nth_to_bl bit_simps)
  done

lemma take_shiftr: "n ≤ size w ⟹ take n (to_bl (w >> n)) = replicate
n False"
  apply (rule nth_equalityI)
   apply (auto simp add: word_size to_bl_nth bit_simps dest: bit_imp_le_length)
  done

lemma take_sshiftr':
  "n ≤ size w ⟹ hd (to_bl (w >>> n)) = hd (to_bl w) ∧
    take n (to_bl (w >>> n)) = replicate n (hd (to_bl w))"
  for w :: "'a::len word"
  apply (cases n)
   apply (auto simp add: hd_to_bl_iff bit_simps not_less word_size)
  apply (rule nth_equalityI)
   apply (auto simp add: nth_to_bl bit_simps nth_Cons split: nat.split)
  done

lemmas hd_sshiftr = take_sshiftr' [THEN conjunct1]
lemmas take_sshiftr = take_sshiftr' [THEN conjunct2]

lemma atd_lem: "take n xs = t ⟹ drop n xs = d ⟹ xs = t @ d"
  by (auto intro: append_take_drop_id [symmetric])

lemmas bl_shiftr = atd_lem [OF take_shiftr drop_shiftr]
lemmas bl_sshiftr = atd_lem [OF take_sshiftr drop_sshiftr]

lemma shiftl_of_bl: "of_bl bl << n = of_bl (bl @ replicate n False)"
  apply (rule bit_word_eqI)
  apply (auto simp add: bit_simps nth_append)
  done
```

```
lemma shiftl_bl: "w << n = of_bl (to_bl w @ replicate n False)"
  for w :: "'a::len word"
  by (simp flip: shiftl_of_bl)

lemma bl_shiftl: "to_bl (w << n) = drop n (to_bl w) @ replicate (min
(size w) n) False"
  by (simp add: shiftl_bl word_rep_drop word_size)

lemma shiftr1_bl_of:
  "length bl ≤ LENGTH('a) ⟹
    shiftr1 (of_bl bl::'a::len word) = of_bl (butlast bl)"
  apply (rule bit_word_eqI)
  apply (simp add: bit_simps)
  apply (cases bl rule: rev_cases)
  apply auto
  done

lemma shiftr_bl_of:
  "length bl ≤ LENGTH('a) ⟹
     (of_bl bl::'a::len word) >> n = of_bl (take (length bl - n) bl)"
  by (rule bit_word_eqI) (auto simp add: bit_simps rev_nth)

lemma shiftr_bl: "x >> n ≡ of_bl (take (LENGTH('a) - n) (to_bl x))"
  for x :: "'a::len word"
  using shiftr_bl_of [where 'a='a, of "to_bl x"] by simp

lemma aligned_bl_add_size [OF refl]:
  "size x - n = m ⟹ n ≤ size x ⟹ drop m (to_bl x) = replicate n False
⟹
    take m (to_bl y) = replicate m False ⟹
    to_bl (x + y) = take m (to_bl x) @ drop m (to_bl y)" for x :: ‹'a::len
word›
  apply (subgoal_tac "x AND y = 0")
   prefer 2
   apply (rule word_bl.Rep_eqD)
   apply (simp add: bl_word_and)
   apply (rule align_lem_and [THEN trans])
       apply (simp_all add: word_size)[5]
   apply simp
  apply (subst word_plus_and_or [symmetric])
  apply (simp add : bl_word_or)
  apply (rule align_lem_or)
     apply (simp_all add: word_size)
  done

lemma mask_bl: "mask n = of_bl (replicate n True)"
  by (auto simp add: bit_simps intro!: word_eqI)
```

```
lemma bl_and_mask':
  "to_bl (w AND mask n :: 'a::len word) =
    replicate (LENGTH('a) - n) False @
    drop (LENGTH('a) - n) (to_bl w)"
  apply (rule nth_equalityI)
   apply simp
  apply (clarsimp simp add: to_bl_nth word_size bit_simps)
  apply (auto simp add: word_size test_bit_bl nth_append rev_nth)
  done

lemma slice1_eq_of_bl:
  ‹(slice1 n w :: 'b::len word) = of_bl (takefill False n (to_bl w))›
  for w :: ‹'a::len word›
proof (rule bit_word_eqI)
  fix m
  assume ‹m < LENGTH('b)›
  show ‹bit (slice1 n w :: 'b::len word) m ⟷ bit (of_bl (takefill False
n (to_bl w)) :: 'b word) m›
    by (cases ‹m ≥ n›; cases ‹LENGTH('a) ≥ n›)
      (auto simp add: bit_slice1_iff bit_of_bl_iff not_less rev_nth not_le
nth_takefill nth_to_bl algebra_simps)
qed

lemma slice1_no_bin [simp]:
  "slice1 n (numeral w :: 'b word) = of_bl (takefill False n (bin_to_bl
(LENGTH('b::len)) (numeral w)))"
  by (simp add: slice1_eq_of_bl)

lemma slice_no_bin [simp]:
  "slice n (numeral w :: 'b word) = of_bl (takefill False (LENGTH('b::len)
- n)
    (bin_to_bl (LENGTH('b::len)) (numeral w)))"
  by (simp add: slice_def)

lemma slice_take': "slice n w = of_bl (take (size w - n) (to_bl w))"
  by (simp add: slice_def word_size slice1_eq_of_bl takefill_alt)

lemmas slice_take = slice_take' [unfolded word_size]

— shiftr to a word of the same size is just slice, slice is just shiftr then ucast
lemmas shiftr_slice = trans [OF shiftr_bl [THEN meta_eq_to_obj_eq] slice_take
[symmetric]]

lemma slice1_down_alt':
  "sl = slice1 n w ⟹ fs = size sl ⟹ fs + k = n ⟹
    to_bl sl = takefill False fs (drop k (to_bl w))"
  apply (simp add: slice1_eq_of_bl)
  apply transfer
  apply (simp add: bl_bin_bl_rep_drop)
```

```
  using drop_takefill
  apply force
  done

lemma slice1_up_alt':
  "sl = slice1 n w ⟹ fs = size sl ⟹ fs = n + k ⟹
    to_bl sl = takefill False fs (replicate k False @ (to_bl w))"
  apply (simp add: slice1_eq_of_bl)
  apply transfer
  apply (simp add: bl_bin_bl_rep_drop flip: takefill_append)
  apply (metis diff_add_inverse)
  done

lemmas sd1 = slice1_down_alt' [OF refl refl, unfolded word_size]
lemmas su1 = slice1_up_alt' [OF refl refl, unfolded word_size]
lemmas slice1_down_alt = le_add_diff_inverse [THEN sd1]
lemmas slice1_up_alts =
  le_add_diff_inverse [symmetric, THEN su1]
  le_add_diff_inverse2 [symmetric, THEN su1]

lemma slice1_tf_tf':
  "to_bl (slice1 n w :: 'a::len word) =
    rev (takefill False (LENGTH('a)) (rev (takefill False n (to_bl w))))"
  unfolding slice1_eq_of_bl by (rule word_rev_tf)

lemmas slice1_tf_tf = slice1_tf_tf' [THEN word_bl.Rep_inverse', symmetric]

lemma revcast_eq_of_bl:
  ‹(revcast w :: 'b::len word) = of_bl (takefill False (LENGTH('b)) (to_bl
w))›
  for w :: ‹'a::len word›
  by (simp add: revcast_def slice1_eq_of_bl)

lemmas revcast_no_def [simp] = revcast_eq_of_bl [where w="numeral w",
unfolded word_size] for w

lemma to_bl_revcast:
  "to_bl (revcast w :: 'a::len word) = takefill False (LENGTH('a)) (to_bl
w)"
  apply (rule nth_equalityI)
  apply simp
  apply (cases ‹LENGTH('a) ≤ LENGTH('b)›)
   apply (auto simp add: nth_to_bl nth_takefill bit_revcast_iff)
  done

lemma word_cat_bl: "word_cat a b = of_bl (to_bl a @ to_bl b)"
  apply (rule bit_word_eqI)
  apply (simp add: bit_word_cat_iff bit_of_bl_iff nth_append not_less
nth_rev_to_bl)
```

```
  apply (meson bit_word.rep_eq less_diff_conv2 nth_rev_to_bl)
  done

lemma of_bl_append:
  "(of_bl (xs @ ys) :: 'a::len word) = of_bl xs * 2^(length ys) + of_bl
ys"
  apply transfer
  apply (simp add: bl_to_bin_app_cat bin_cat_num)
  done

lemma of_bl_False [simp]: "of_bl (False#xs) = of_bl xs"
  by (rule word_eqI) (auto simp: test_bit_of_bl nth_append)

lemma of_bl_True [simp]: "(of_bl (True # xs) :: 'a::len word) = 2^length
xs + of_bl xs"
  by (subst of_bl_append [where xs="[True]", simplified]) (simp add:
word_1_bl)

lemma of_bl_Cons: "of_bl (x#xs) = of_bool x * 2^length xs + of_bl xs"
  by (cases x) simp_all

lemma word_split_bl':
  "std = size c - size b ⟹ (word_split c = (a, b)) ⟹
    (a = of_bl (take std (to_bl c)) ∧ b = of_bl (drop std (to_bl c)))"
  apply (simp add: word_split_def)
  apply transfer
  apply (cases ‹LENGTH('b) ≤ LENGTH('a)›)
   apply (auto simp add: drop_bit_take_bit drop_bin2bl bin_to_bl_drop_bit
[symmetric, of ‹LENGTH('a)› ‹LENGTH('a) - LENGTH('b)› ‹LENGTH('b)›]
min_absorb2)
  done

lemma word_split_bl: "std = size c - size b ⟹
    (a = of_bl (take std (to_bl c)) ∧ b = of_bl (drop std (to_bl c)))
⟷
    word_split c = (a, b)"
  apply (rule iffI)
   defer
   apply (erule (1) word_split_bl')
  apply (case_tac "word_split c")
  apply (auto simp add: word_size)
  apply (frule word_split_bl' [rotated])
   apply (auto simp add: word_size)
  done

lemma word_split_bl_eq:
  "(word_split c :: ('c::len word × 'd::len word)) =
    (of_bl (take (LENGTH('a::len) - LENGTH('d::len)) (to_bl c)),
     of_bl (drop (LENGTH('a) - LENGTH('d)) (to_bl c)))"
```

```
  for c :: "'a::len word"
  apply (rule word_split_bl [THEN iffD1])
   apply (unfold word_size)
   apply (rule refl conjI)+
  done

lemma word_rcat_bl:
  ‹word_rcat wl = of_bl (concat (map to_bl wl))›
proof -
  define ws where ‹ws = rev wl›
  moreover have ‹word_rcat (rev ws) = of_bl (concat (map to_bl (rev ws)))›
    apply (simp add: word_rcat_def of_bl_eq rev_concat rev_map comp_def
rev_to_bl_eq flip: horner_sum_of_bool_2_concat)
    apply transfer
    apply simp
    done
  ultimately show ?thesis
    by simp
qed

lemma size_rcat_lem': "size (concat (map to_bl wl)) = length wl * size
(hd wl)"
  by (induct wl) (auto simp: word_size)

lemmas size_rcat_lem = size_rcat_lem' [unfolded word_size]

lemma nth_rcat_lem:
  "n < length (wl::'a word list) * LENGTH('a::len) ⟹
    rev (concat (map to_bl wl)) ! n =
    rev (to_bl (rev wl ! (n div LENGTH('a)))) ! (n mod LENGTH('a))"
  apply (induct wl)
   apply clarsimp
  apply (clarsimp simp add : nth_append size_rcat_lem)
  apply (simp flip: mult_Suc minus_div_mult_eq_mod add: less_Suc_eq_le
not_less)
  apply (metis (no_types, lifting) diff_is_0_eq div_le_mono len_not_eq_0
less_Suc_eq less_mult_imp_div_less nonzero_mult_div_cancel_right not_le
nth_Cons_0)
  done

lemma foldl_eq_foldr: "foldl (+) x xs = foldr (+) (x # xs) 0"
  for x :: "'a::comm_monoid_add"
  by (induct xs arbitrary: x) (auto simp: add.assoc)

lemmas word_cat_bl_no_bin [simp] =
  word_cat_bl [where a="numeral a" and b="numeral b", unfolded to_bl_numeral]
  for a b

lemmas word_split_bl_no_bin [simp] =
```

```
  word_split_bl_eq [where c="numeral c", unfolded to_bl_numeral] for c

lemmas word_rot_defs = word_roti_eq_word_rotr_word_rotl word_rotr_eq
word_rotl_eq

lemma to_bl_rotl: "to_bl (word_rotl n w) = rotate n (to_bl w)"
  by (simp add: word_rotl_eq to_bl_use_of_bl)

lemmas blrs0 = rotate_eqs [THEN to_bl_rotl [THEN trans]]

lemmas word_rotl_eqs =
  blrs0 [simplified word_bl_Rep' word_bl.Rep_inject to_bl_rotl [symmetric]]

lemma to_bl_rotr: "to_bl (word_rotr n w) = rotater n (to_bl w)"
  by (simp add: word_rotr_eq to_bl_use_of_bl)

lemmas brrs0 = rotater_eqs [THEN to_bl_rotr [THEN trans]]

lemmas word_rotr_eqs =
  brrs0 [simplified word_bl_Rep' word_bl.Rep_inject to_bl_rotr [symmetric]]

declare word_rotr_eqs (1) [simp]
declare word_rotl_eqs (1) [simp]

lemmas abl_cong = arg_cong [where f = "of_bl"]

end

locale word_rotate
begin

lemmas word_rot_defs' = to_bl_rotl to_bl_rotr

lemmas blwl_syms [symmetric] = bl_word_not bl_word_and bl_word_or bl_word_xor

lemmas lbl_lbl = trans [OF word_bl_Rep' word_bl_Rep' [symmetric]]

lemmas ths_map2 [OF lbl_lbl] = rotate_map2 rotater_map2

lemmas ths_map [where xs = "to_bl v"] = rotate_map rotater_map for v

lemmas th1s [simplified word_rot_defs' [symmetric]] = ths_map2 ths_map

end

lemmas bl_word_rotl_dt = trans [OF to_bl_rotl rotate_drop_take,
  simplified word_bl_Rep']

lemmas bl_word_rotr_dt = trans [OF to_bl_rotr rotater_drop_take,
```

```
  simplified word_bl_Rep']

lemma bl_word_roti_dt':
  "n = nat ((- i) mod int (size (w :: 'a::len word))) ⟹
    to_bl (word_roti i w) = drop n (to_bl w) @ take n (to_bl w)"
  apply (unfold word_roti_eq_word_rotr_word_rotl)
  apply (simp add: bl_word_rotl_dt bl_word_rotr_dt word_size)
  apply safe
   apply (simp add: zmod_zminus1_eq_if)
   apply safe
    apply (auto simp add: nat_mult_distrib nat_mod_distrib)
  using nat_0_le nat_minus_as_int zmod_int apply presburger
  done

lemmas bl_word_roti_dt = bl_word_roti_dt' [unfolded word_size]

lemmas word_rotl_dt = bl_word_rotl_dt [THEN word_bl.Rep_inverse' [symmetric]]
lemmas word_rotr_dt = bl_word_rotr_dt [THEN word_bl.Rep_inverse' [symmetric]]
lemmas word_roti_dt = bl_word_roti_dt [THEN word_bl.Rep_inverse' [symmetric]]

lemmas word_rotr_dt_no_bin' [simp] =
  word_rotr_dt [where w="numeral w", unfolded to_bl_numeral] for w


lemmas word_rotl_dt_no_bin' [simp] =
  word_rotl_dt [where w="numeral w", unfolded to_bl_numeral] for w


lemma max_word_bl: "to_bl (- 1::'a::len word) = replicate LENGTH('a)
True"
  by (fact to_bl_n1)

lemma to_bl_mask:
  "to_bl (mask n :: 'a::len word) =
  replicate (LENGTH('a) - n) False @
    replicate (min (LENGTH('a)) n) True"
  by (simp add: mask_bl word_rep_drop min_def)

lemma map_replicate_True:
  "n = length xs ⟹
    map (λ(x,y). x ∧ y) (zip xs (replicate n True)) = xs"
  by (induct xs arbitrary: n) auto

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

context
```

```
  includes bit_operations_syntax
begin

lemma bl_and_mask:
  fixes w :: "'a::len word"
    and n :: nat
  defines "n' ≡ LENGTH('a) - n"
  shows "to_bl (w AND mask n) = replicate n' False @ drop n' (to_bl w)"
proof -
  note [simp] = map_replicate_True map_replicate_False
  have "to_bl (w AND mask n) = map2 (∧) (to_bl w) (to_bl (mask n::'a::len
word))"
    by (simp add: bl_word_and)
  also have "to_bl w = take n' (to_bl w) @ drop n' (to_bl w)"
    by simp
  also have "map2 (∧) ... (to_bl (mask n::'a::len word)) =
      replicate n' False @ drop n' (to_bl w)"
    unfolding to_bl_mask n'_def by (subst zip_append) auto
  finally show ?thesis .
qed

lemma drop_rev_takefill:
  "length xs ≤ n ⟹
    drop (n - length xs) (rev (takefill False n (rev xs))) = xs"
  by (simp add: takefill_alt rev_take)

declare bin_to_bl_def [simp]

lemmas of_bl_reasoning = to_bl_use_of_bl of_bl_append

lemma uint_of_bl_is_bl_to_bin_drop:
  "length (dropWhile Not l) ≤ LENGTH('a) ⟹ uint (of_bl l :: 'a::len
word) = bl_to_bin l"
  apply transfer
  apply (simp add: take_bit_eq_mod)
  apply (rule Divides.mod_less)
   apply (rule bl_to_bin_ge0)
  using bl_to_bin_lt2p_drop apply (rule order.strict_trans2)
  apply simp
  done

corollary uint_of_bl_is_bl_to_bin:
  "length l≤LENGTH('a) ⟹ uint ((of_bl::bool list⇒ ('a :: len) word)
l) = bl_to_bin l"
  apply(rule uint_of_bl_is_bl_to_bin_drop)
  using le_trans length_dropWhile_le by blast

lemma bin_to_bl_or:
  "bin_to_bl n (a OR b) = map2 (∨) (bin_to_bl n a) (bin_to_bl n b)"
```

```
  using bl_or_aux_bin[where n=n and v=a and w=b and bs="[]" and cs="[]"]
  by simp

lemma word_and_1_bl:
  fixes x::"'a::len word"
  shows "(x AND 1) = of_bl [bit x 0]"
  by (simp add: word_and_1)

lemma word_1_and_bl:
  fixes x::"'a::len word"
  shows "(1 AND x) = of_bl [bit x 0]"
  using word_and_1_bl [of x] by (simp add: ac_simps)

lemma of_bl_drop:
  "of_bl (drop n xs) = (of_bl xs AND mask (length xs - n))"
  apply (rule bit_word_eqI)
  apply (auto simp: rev_nth bit_simps cong: rev_conj_cong)
  done

lemma to_bl_1:
  "to_bl (1::'a::len word) = replicate (LENGTH('a) - 1) False @ [True]"
  by (rule nth_equalityI) (auto simp add: to_bl_unfold nth_append rev_nth
bit_1_iff not_less not_le)

lemma eq_zero_set_bl:
  "(w = 0) = (True ∉ set (to_bl w))"
  apply (auto simp add: to_bl_unfold)
  apply (rule bit_word_eqI)
  apply auto
  done

lemma of_drop_to_bl:
  "of_bl (drop n (to_bl x)) = (x AND mask (size x - n))"
  by (simp add: of_bl_drop word_size_bl)

lemma unat_of_bl_length:
  "unat (of_bl xs :: 'a::len word) < 2 ^ (length xs)"
proof (cases "length xs < LENGTH('a)")
  case True
  then have "(of_bl xs::'a::len word) < 2 ^ length xs"
    by (simp add: of_bl_length_less)
  with True
  show ?thesis
    by (simp add: word_less_nat_alt unat_of_nat)
next
  case False
  have "unat (of_bl xs::'a::len word) < 2 ^ LENGTH('a)"
    by (simp split: unat_split)
  also
```

```
  from False
  have "LENGTH('a) ≤ length xs" by simp
  then have "2 ^ LENGTH('a) ≤ (2::nat) ^ length xs"
    by (rule power_increasing) simp
  finally
  show ?thesis .
qed

lemma word_msb_alt: "msb w ⟷ hd (to_bl w)"
  for w :: "'a::len word"
  apply (simp add: msb_word_eq)
  apply (subst hd_conv_nth)
   apply simp
  apply (subst nth_to_bl)
   apply simp
  apply simp
  done

lemma word_lsb_last:
  ‹lsb w ⟷ last (to_bl w)›
  for w :: ‹'a::len word›
  using nth_to_bl [of ‹LENGTH('a) - Suc 0› w]
  by (simp add: last_conv_nth bit_0 lsb_odd)

lemma is_aligned_to_bl:
  "is_aligned (w :: 'a :: len word) n = (True ∉ set (drop (size w - n)
(to_bl w)))"
  by (simp add: is_aligned_mask eq_zero_set_bl bl_and_mask word_size)

lemma is_aligned_replicate:
  fixes w::"'a::len word"
  assumes aligned: "is_aligned w n"
  and          nv: "n ≤ LENGTH('a)"
  shows   "to_bl w = (take (LENGTH('a) - n) (to_bl w)) @ replicate n False"
  apply (rule nth_equalityI)
  using assms apply (simp_all add: nth_append not_less word_size to_bl_nth
is_aligned_imp_not_bit)
  done

lemma is_aligned_drop:
  fixes w::"'a::len word"
  assumes "is_aligned w n" "n ≤ LENGTH('a)"
  shows "drop (LENGTH('a) - n) (to_bl w) = replicate n False"
proof -
  have "to_bl w = take (LENGTH('a) - n) (to_bl w) @ replicate n False"
    by (rule is_aligned_replicate) fact+
  then have "drop (LENGTH('a) - n) (to_bl w) = drop (LENGTH('a) - n)
..." by simp
  also have "... = replicate n False" by simp
```

```
  finally show ?thesis .
qed

lemma less_is_drop_replicate:
  fixes x::"'a::len word"
  assumes lt: "x < 2 ^ n"
  shows   "to_bl x = replicate (LENGTH('a) - n) False @ drop (LENGTH('a)
- n) (to_bl x)"
  by (metis assms bl_and_mask' less_mask_eq)

lemma is_aligned_add_conv:
  fixes off::"'a::len word"
  assumes aligned: "is_aligned w n"
  and        offv: "off < 2 ^ n"
  shows    "to_bl (w + off) =
   (take (LENGTH('a) - n) (to_bl w)) @ (drop (LENGTH('a) - n) (to_bl off))"
proof cases
  assume nv: "n ≤ LENGTH('a)"
  show ?thesis
  proof (subst aligned_bl_add_size, simp_all only: word_size)
    show "drop (LENGTH('a) - n) (to_bl w) = replicate n False"
      by (subst is_aligned_replicate [OF aligned nv]) (simp add: word_size)

    from offv show "take (LENGTH('a) - n) (to_bl off) =
                    replicate (LENGTH('a) - n) False"
      by (subst less_is_drop_replicate, assumption) simp
  qed fact
next
  assume "¬ n ≤ LENGTH('a)"
  with offv show ?thesis by (simp add: power_overflow)
qed

lemma is_aligned_replicateI:
  "to_bl p = addr @ replicate n False ⟹ is_aligned (p::'a::len word)
n"
  apply (simp add: is_aligned_to_bl word_size)
  apply (subgoal_tac "length addr = LENGTH('a) - n")
   apply (simp add: replicate_not_True)
  apply (drule arg_cong [where f=length])
  apply simp
  done

lemma to_bl_2p:
  "n < LENGTH('a) ⟹
   to_bl ((2::'a::len word) ^ n) =
   replicate (LENGTH('a) - Suc n) False @ True # replicate n False"
  apply (rule nth_equalityI)
   apply (auto simp add: nth_append to_bl_nth word_size bit_simps not_less
nth_Cons le_diff_conv)
```

```
  subgoal for i
  apply (cases <Suc (i + n) - LENGTH('a)>)
  apply simp_all
    done
  done

lemma xor_2p_to_bl:
  fixes x::"'a::len word"
  shows "to_bl (x XOR 2^n) =
  (if n < LENGTH('a)
   then take (LENGTH('a)-Suc n) (to_bl x) @ (¬rev (to_bl x)!n) # drop
(LENGTH('a)-n) (to_bl x)
   else to_bl x)"
  apply (auto simp add: to_bl_eq_rev take_map drop_map take_rev drop_rev
bit_simps)
  apply (rule nth_equalityI)
   apply (auto simp add: bit_simps rev_nth nth_append Suc_diff_Suc)
  done

lemma is_aligned_replicateD:
  "⟦ is_aligned (w::'a::len word) n; n ≤ LENGTH('a) ⟧
     ⟹ ∃xs. to_bl w = xs @ replicate n False
               ∧ length xs = size w - n"
  apply (subst is_aligned_replicate, assumption+)
  apply (rule exI, rule conjI, rule refl)
  apply (simp add: word_size)
  done
```

right-padding a word to a certain length

```
definition
  "bl_pad_to bl sz ≡ bl @ (replicate (sz - length bl) False)"

lemma bl_pad_to_length:
  assumes lbl: "length bl ≤ sz"
  shows   "length (bl_pad_to bl sz) = sz"
  using lbl by (simp add: bl_pad_to_def)

lemma bl_pad_to_prefix:
  "prefix bl (bl_pad_to bl sz)"
  by (simp add: bl_pad_to_def)


lemma of_bl_length:
  "length xs < LENGTH('a) ⟹ of_bl xs < (2 :: 'a::len word) ^ length
xs"
  by (simp add: of_bl_length_less)

lemma of_bl_mult_and_not_mask_eq:
  "⟦is_aligned (a :: 'a::len word) n; length b + m ≤ n⟧
```

```
   ⟹ a + of_bl b * (2^m) AND NOT(mask n) = a"
  apply (simp flip: push_bit_eq_mult subtract_mask(1) take_bit_eq_mask)
  apply (subst disjunctive_add)
   apply (auto simp add: bit_simps not_le not_less)
   apply (meson is_aligned_imp_not_bit is_aligned_weaken less_diff_conv2)
  apply (erule is_alignedE')
  apply (simp add: take_bit_push_bit)
  apply (rule bit_word_eqI)
  apply (auto simp add: bit_simps)
  done

lemma bin_to_bl_of_bl_eq:
  "⟦is_aligned (a::'a::len word) n; length b + c ≤ n; length b + c < LENGTH('a)⟧
  ⟹ bin_to_bl (length b) (uint ((a + of_bl b * 2^c) >> c)) = b"
  apply (simp flip: push_bit_eq_mult take_bit_eq_mask)
  apply (subst disjunctive_add)
   apply (auto simp add: bit_simps not_le not_less unsigned_or_eq unsigned_drop_bit_eq
     unsigned_push_bit_eq bin_to_bl_or simp flip: bin_to_bl_def)
   apply (meson is_aligned_imp_not_bit is_aligned_weaken less_diff_conv2)
  apply (erule is_alignedE')
  apply (rule nth_equalityI)
   apply (auto simp add: nth_bin_to_bl bit_simps rev_nth simp flip: bin_to_bl_def)
  done


lemma bl_cast_long_short_long_ingoreLeadingZero_generic:
  "⟦ length (dropWhile Not (to_bl w)) ≤ LENGTH('s); LENGTH('s) ≤ LENGTH('l)
⟧ ⟹
   (of_bl :: _ ⇒ 'l::len word) (to_bl ((of_bl::_ ⇒ 's::len word) (to_bl
w))) = w"
  by (rule word_uint_eqI) (simp add: uint_of_bl_is_bl_to_bin uint_of_bl_is_bl_to_bin_drop)


corollary ucast_short_ucast_long_ingoreLeadingZero:
  "⟦ length (dropWhile Not (to_bl w)) ≤ LENGTH('s); LENGTH('s) ≤ LENGTH('l)
⟧ ⟹
   (ucast:: 's::len word ⇒ 'l::len word) ((ucast:: 'l::len word ⇒ 's::len
word) w) = w"
  apply (subst ucast_bl)+
  apply (rule bl_cast_long_short_long_ingoreLeadingZero_generic; simp)
  done

lemma length_drop_mask:
  fixes w::"'a::len word"
  shows "length (dropWhile Not (to_bl (w AND mask n))) ≤ n"
proof -
  have "length (takeWhile Not (replicate n False @ ls)) = n + length (takeWhile
Not ls)"
    for ls n by(subst takeWhile_append2) simp+
```

```
  then show ?thesis
    unfolding bl_and_mask by (simp add: dropWhile_eq_drop)
qed

lemma map_bits_rev_to_bl:
  "map (bit x) [0..<size x] = rev (to_bl x)"
  by (auto simp: list_eq_iff_nth_eq test_bit_bl word_size)

lemma of_bl_length2:
  "length xs + c < LENGTH('a) ⟹ of_bl xs * 2^c < (2::'a::len word) ^
(length xs + c)"
  by (simp add: of_bl_length word_less_power_trans2)

lemma of_bl_max:
  "(of_bl xs :: 'a::len word) ≤ mask (length xs)"
proof -
  define ys where ‹ys = rev xs›
  have ‹take_bit (length ys) (horner_sum of_bool 2 ys :: 'a word) = horner_sum
of_bool 2 ys›
    by transfer (simp add: take_bit_horner_sum_bit_eq min_def)
  then have ‹(of_bl (rev ys) :: 'a word) ≤ mask (length ys)›
    by (simp only: of_bl_rev_eq less_eq_mask_iff_take_bit_eq_self)
  with ys_def show ?thesis
    by simp
qed
```

Some auxiliaries for sign-shifting by the entire word length or more

```
lemma sshiftr_clamp_pos:
  assumes
    "LENGTH('a) ≤ n"
    "0 ≤ sint x"
  shows "(x::'a::len word) >>> n = 0"
  apply (rule bit_word_eqI)
  using assms
  apply (auto simp add: bit_simps bit_last_iff)
  done

lemma sshiftr_clamp_neg:
  assumes
    "LENGTH('a) ≤ n"
    "sint x < 0"
  shows "(x::'a::len word) >>> n = -1"
  apply (rule bit_word_eqI)
  using assms
  apply (auto simp add: bit_simps bit_last_iff)
  done

lemma sshiftr_clamp:
  assumes "LENGTH('a) ≤ n"
```

```
  shows "(x::'a::len word) >>> n = x >>> LENGTH('a)"
  apply (rule bit_word_eqI)
  using assms
  apply (auto simp add: bit_simps bit_last_iff)
  done
```

Like `length ?bl ≤ LENGTH(?'a) ⟹ shiftr1 (of_bl ?bl) = of_bl (butlast ?bl)`, but the precondition is stronger because we need to pick the msb out of the list.

```
lemma sshiftr1_bl_of:
  "length bl = LENGTH('a) ⟹
    sshiftr1 (of_bl bl::'a::len word) = of_bl (hd bl # butlast bl)"
  apply (rule word_bl.Rep_eqD)
  apply (subst bl_sshiftr1[of "of_bl bl :: 'a word"])
  by (simp add: word_bl.Abs_inverse)
```

Like `length ?bl = LENGTH(?'a) ⟹ sshiftr1 (of_bl ?bl) = of_bl (hd ?bl # butlast ?bl)`, with a weaker precondition. We still get a direct equation for `sshiftr1 (of_bl bl)`, it's just uglier.

```
lemma sshiftr1_bl_of':
  "LENGTH('a) ≤ length bl ⟹
    sshiftr1 (of_bl bl::'a::len word) =
    of_bl (hd (drop (length bl - LENGTH('a)) bl) # butlast (drop (length
bl - LENGTH('a)) bl))"
  apply (subst of_bl_drop'[symmetric, of "length bl - LENGTH('a)"])
  using sshiftr1_bl_of[of "drop (length bl - LENGTH('a)) bl"]
  by auto
```

Like `length ?bl ≤ LENGTH(?'a) ⟹ of_bl ?bl >> ?n = of_bl (take (length ?bl - ?n) ?bl)`.

```
lemma sshiftr_bl_of:
  assumes "length bl = LENGTH('a)"
  shows "(of_bl bl::'a::len word) >>> n = of_bl (replicate n (hd bl) @
take (length bl - n) bl)"
proof -
  from assms obtain b bs where ‹bl = b # bs›
    by (cases bl) simp_all
  then have *: ‹bl ! 0 ⟷ b› ‹hd bl ⟷ b›
    by simp_all
  show ?thesis
  apply (rule bit_word_eqI)
    using assms * by (auto simp add: bit_simps nth_append rev_nth not_less)
qed
```

Like `?x >> ?n ≡ of_bl (take (LENGTH(?'a) - ?n) (to_bl ?x))`

```
lemma sshiftr_bl: "x >>> n ≡ of_bl (replicate n (msb x) @ take (LENGTH('a)
- n) (to_bl x))"
  for x :: "'a::len word"
```

```
  unfolding word_msb_alt
  by (smt (verit) length_to_bl_eq sshiftr_bl_of word_bl.Rep_inverse)

end

lemma of_bl_drop_eq_take_bit:
  ‹of_bl (drop n xs) = take_bit (length xs - n) (of_bl xs)›
  by (simp add: of_bl_drop take_bit_eq_mask)

lemma of_bl_take_to_bl_eq_drop_bit:
  ‹of_bl (take n (to_bl w)) = drop_bit (LENGTH('a) - n) w›
    if ‹n ≤ LENGTH('a)›
    for w :: ‹'a::len word›
  using that shiftr_bl [of w ‹LENGTH('a) - n›] by (simp add: shiftr_def)

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]"
  apply (rule_tac s="rev (to_bl (word_succ (0 :: 'a word)))" in trans)
   apply simp
  apply (simp only: rtb_rbl_ariths(1)[OF refl])
  apply simp
  apply (case_tac "LENGTH('a)")
   apply simp
  apply (simp add: takefill_alt)
  done

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"
  apply (erule list_induct2)
   apply simp
  apply (simp only: rbl_add_carry_Cons)
  apply simp
  done

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)"]
  apply (simp add: word_add_rbl[OF refl refl])
  apply (drule_tac x=False in spec)
  apply (simp add: zip_rev)
  done
```

```
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)))"
  apply (simp add: slice_take word_rev_tf rev_take)
  apply (cases "n < LENGTH('b)", simp_all)
  done

lemma rbl_word_ucast:
  "rev (to_bl (ucast x :: 'a::len word)) = takefill False (LENGTH('a))
(rev (to_bl x))"
  apply (simp add: to_bl_ucast takefill_alt)
```

```
  apply (simp add: rev_drop)
  apply (cases "LENGTH('a) < LENGTH('b)")
   apply simp_all
  done

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)))"
  apply (cases "n < size w")
   apply (simp add: bl_sshiftr takefill_last_def word_size
      takefill_alt rev_take last_rev
      drop_nonempty_def)
  apply (subgoal_tac "(w >>> n) = of_bl (replicate (size w) (msb w))")
   apply (simp add: word_size takefill_last_def takefill_alt
      last_rev word_msb_alt word_rev_tf
      drop_nonempty_def take_Cons')
   apply (case_tac "LENGTH('a)", simp_all)
  apply (rule word_eqI)
  apply (simp add: bit_simps word_size test_bit_of_bl
      msb_nth)
  done
```

```
lemma nth_word_of_int:
  "bit (word_of_int x :: 'a::len word) n = (n < LENGTH('a) ∧ bit x n)"
  apply (simp add: test_bit_bl word_size to_bl_of_bin)
  apply (subst conj_cong[OF refl], erule bin_nth_bl)
  apply auto
  done

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)))"
  apply transfer
  apply (auto simp add: bit_signed_take_bit_iff min_def)
  done

lemma rbl_word_scast:
  "rev (to_bl (scast x :: 'a::len word)) = takefill_last False (LENGTH('a))
(rev (to_bl x))"
  apply (rule nth_equalityI)
   apply (simp add: word_size takefill_last_def)
  apply (clarsimp simp: nth_scast takefill_last_def
      nth_takefill word_size rev_nth to_bl_nth)
  apply (cases "LENGTH('b)")
   apply simp
  apply (clarsimp simp: less_Suc_eq_le linorder_not_less
      last_rev word_msb_alt[symmetric]
      msb_nth)
  done

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)"
  apply (simp add: rbl_plus_def take_zip[symmetric])
  apply (rule_tac list="zip xs ys" in list.induct)
   apply simp
```

```
  apply (clarsimp simp: split_def)
  apply (case_tac n, simp_all)
  done

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 show ?case
    apply (simp add: trans [OF of_bl_append add.commute]
        rbl_mul_simps rbl_word_plus' distrib_right mult_bit shiftl rbl_shiftl)
    apply (simp add: takefill_alt word_size rev_map take_rbl_plus min_def)
    apply (simp add: rbl_plus_def)
    apply (simp add: zip_take_triv)
    apply (simp only: mult.commute [of _ 2] to_bl_double_eq)
    apply (simp flip: butlast_rev add: take_butlast)
    done
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)
  apply (rule conj_cong[OF refl])
  apply (cases "xs = ys")
   apply (simp add: nth_Cons')
   apply blast
  apply (simp add: nth_Cons')
  apply safe
   apply (rule_tac x="n - 1" in exI)
   apply simp
  apply (rule_tac x="Suc n" in exI)
  apply simp
  done

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))"
  apply (induct xs ys rule: list_induct2)
   apply (simp_all add: rev_bl_order_simps bl_to_bin_app_cat concat_bit_Suc)
  apply (auto simp add: bl_to_bin_def add1_zle_eq)
  done

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)))"
  using word_msb_alt[where w=x] word_msb_alt[where w=y]
  apply (simp add: word_sle_msb_le word_le_rbl)
  apply (subgoal_tac "length (to_bl x) = length (to_bl y)")
   apply (cases "to_bl x", simp)
   apply (cases "to_bl y", simp)
   apply (clarsimp simp: map_last_def rev_bl_order_rev_simp)
   apply auto
  done

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)))"
  using word_msb_alt[where w=x] word_msb_alt[where w=y]
  apply (simp add: word_sless_msb_less word_less_rbl)
  apply (subgoal_tac "length (to_bl x) = length (to_bl y)")
   apply (cases "to_bl x", simp)
   apply (cases "to_bl y", simp)
   apply (clarsimp simp: map_last_def rev_bl_order_rev_simp)
   apply auto
  done
```

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",
    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
```

11 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
  then have ***: ‹∃n. ∀n'≥n. bit k n' ⟷ bit k n›
    by meson
  have l: ‹(LEAST q. ∀m≥q. bit k m ⟷ bit k q) = n›
    apply (rule Least_equality)
```

```
    using * apply blast
    apply (metis "**" One_nat_def Suc_pred le_cases le0 neq0_conv not_less_eq_eq)
    done
  show ‹set_bits (bit k) = k›
    apply (simp only: *** set_bits_int_def horner_sum_bit_eq_take_bit
l)
    apply simp
    apply (rule bit_eqI)
    apply (simp add: bit_signed_take_bit_iff min_def)
    apply (auto simp add: not_le bit_take_bit_iff dest: *)
    done
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
    apply (simp add: set_bits_int_def n_def [symmetric] ** del: upt.upt_Suc)
    apply (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)
    done
  next
    case True
    with n have *: ‹∃n. ∀n'≥n. f n'›
      by blast
    have ***: ‹¬ (∃n. ∀n'≥n. ¬ f n')›
      apply (rule ccontr)
      using * nat_le_linear by auto
    have **: ‹(LEAST n. ∀n'≥n. f n') = n›
      using True n_eq by simp
    from * *** True show ?thesis
    apply (simp add: set_bits_int_def n_def [symmetric] ** del: upt.upt_Suc)
    apply (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)
    done
  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
```

12 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_nop1 [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_nop2 [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_nop1 scast_nop2 scast_id

type_synonym 'a sword = "'a signed word"

end
```

13 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
```

14 Enumeration Instances for Words

```
theory Enumeration_Word
  imports
    "HOL-Library.Word"
    More_Word
    Enumeration
    Even_More_List
begin

lemma length_word_enum: "length (enum :: 'a :: len word list) = 2 ^ LENGTH('a)"
  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" ("[_ , _ .e. _]")
where
  "upto_enum_step a b c ≡
```

```
      if c < a then [] else map (λx. a + x * (b - a)) [0 .e. (c - a) div
(b - a)]"


lemma maxBound_word:
  "(maxBound::'a::len word) = -1"
  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)

  show ?thesis
    apply (subst upto_enum_red)
    apply (simp del: upt.simps)
    apply (subst Suc_unat_diff_1 [OF lt])
    apply (rule map_cong [OF refl])
    apply (rule toEnum_of_nat)
    apply simp
    apply (erule order_less_trans [OF _ lt'])
    done
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
  apply (subst unat_power_lower[OF szv, symmetric])
  apply (rule upto_enum_red')
  apply (subst word_le_nat_alt, simp)
  done

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)]" using szv
  unfolding upto_enum_step_def
  apply (subst if_not_P)
   apply (rule leD)
   apply (subst word_le_nat_alt)
   apply (subst unat_minus_one)
    apply simp
   apply simp
  apply simp
  apply (subst upto_enum_red)
  apply (simp del: upt.simps)
  apply (subst Suc_div_unat_helper [where 'a = 'a, OF szv usszv, symmetric])
  apply clarsimp
  apply (subst toEnum_of_nat)
   apply (erule order_less_trans)
   using szv
   apply simp
  apply simp
  done

lemma upto_enum_word:
  "[x .e. y] = map of_nat [unat x ..< Suc (unat y)]"
  apply (subst upto_enum_red)
  apply clarsimp
  apply (subst toEnum_of_nat)
   prefer 2
   apply (rule refl)
  apply (erule disjE, simp)
  apply clarsimp
  apply (erule order_less_trans)
  apply simp
  done

lemma word_upto_Cons_eq:
  "x < y ⟹ [x::'a::len word .e. y] = x # [x + 1 .e. y]"
  apply (subst upto_enum_red)
  apply (subst upt_conv_Cons)
  apply simp_all
   apply unat_arith
  apply (simp only: list.map list.inject upto_enum_red to_from_enum simp_thms)
  apply simp_all
  apply unat_arith
  done

lemma distinct_enum_upto:
  "distinct [(0 :: 'a::len word) .e. b]"
proof -
```

```
  have "⋀(b::'a word). [0 .e. b] = nths enum {..< Suc (fromEnum b)}"
    apply (subst upto_enum_red)
    apply (subst nths_upt_eq_take)
    apply (subst enum_word_def)
    apply (subst take_map)
    apply (subst take_upt)
     apply (simp only: add_0 fromEnum_unat)
     apply (rule order_trans [OF _ order_eq_refl])
      apply (rule Suc_leI [OF unat_lt2p])
     apply simp
    apply clarsimp
    apply (rule toEnum_of_nat)
    apply (erule order_less_trans [OF _ unat_lt2p])
    done

  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}"
  apply (subst upto_enum_red)
  apply (subst set_map)
  apply safe
    apply simp
    apply clarsimp
    apply (erule disjE)
     apply simp
     apply (erule iffD2 [OF word_le_nat_alt])
    apply clarsimp
  apply simp_all
    apply (metis le_unat_uoi nat_less_le toEnum_of_nat unsigned_less word_le_nat_alt)
   apply (metis le_unat_uoi less_or_eq_imp_le toEnum_of_nat unsigned_less
word_le_nat_alt)
    apply (rule_tac x="fromEnum x" in image_eqI)
   apply clarsimp
  apply clarsimp
  apply transfer
  apply auto
  done

lemma upto_enum_less:
  assumes xin: "x ∈ set [(a::'a::len word).e.2 ^ n - 1]"
  and     nv:  "n < LENGTH('a::len)"
  shows   "x < 2 ^ n"
proof (cases n)
  case 0
  then show ?thesis using xin by simp
```

```
next
  case (Suc m)
  show ?thesis using xin nv le_m1_iff_lt p2_gt_0 by auto
qed

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 (subst word_upto_Cons_eq, auto, fact+)

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"
  apply (simp add: word_le_nat_alt upto_enum_red)
  apply (clarsimp simp: Suc_diff_le)
  done

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)"
  apply (case_tac "a ≤ b")
   apply (clarsimp)
  apply (clarsimp simp: upto_enum_def)
  apply unat_arith
  done

lemma length_upto_enum_less_one:
  "⟦a ≤ b; b ≠ 0⟧
  ⟹ length [a .e. b - 1] = unat (b - a)"
```

```
  apply clarsimp
  apply (subst unat_sub[symmetric], assumption)
  apply clarsimp
  done

lemma drop_upto_enum:
  "drop (unat n) [0 .e. m] = [n .e. m]"
  apply (clarsimp simp: upto_enum_def)
  apply (induct m, simp)
  by (metis drop_map drop_upt plus_nat.add_0)

lemma distinct_enum_upto' [simp]:
  "distinct [a::'a::len word .e. b]"
  apply (subst drop_upto_enum [symmetric])
  apply (rule distinct_drop)
  apply (rule distinct_enum_upto)
  done

lemma length_interval:
  "⟦set xs = {x. (a::'a::len word) ≤ x ∧ x ≤ b}; distinct xs⟧
  ⟹ length xs = Suc (unat b) - unat a"
  apply (frule distinct_card)
  apply (subgoal_tac "set xs = set [a .e. b]")
   apply (cut_tac distinct_card [where xs="[a .e. b]"])
    apply (subst (asm) length_upto_enum)
    apply clarsimp
   apply (rule distinct_enum_upto')
  apply simp
  done

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)"
  apply (simp only: enum_word_def)
  apply (subst upt_add_eq_append'[where j="unat v"])
    apply simp
   apply (rule order_less_imp_le, simp)
  apply (simp add: upt_conv_Cons)
  apply (intro exI conjI)
    apply fastforce
   apply clarsimp
   apply (drule of_nat_mono_maybe[rotated, where 'a='a])
    apply simp
   apply simp
  apply (clarsimp simp: Suc_le_eq)
  apply (drule of_nat_mono_maybe[rotated, where 'a='a])
   apply simp
```

```
  apply simp
  done

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
```

15 Print Words in Hex

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

Print words in hex.

```
typed_print_translation ‹
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

16 Normalising Word Numerals

theory Norm_Words
  imports Signed_Words
begin

Normalise word numerals, including negative ones apart from `- (1::'a)`, to the interval `[0..2^len_of 'a)`. Only for concrete word lengths.

lemma 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::'a)` untouched by default, because it is often useful and its normal form can be large. To include it in the normalisation, add `minus_one_norm_num`. The additional normalisation is restricted to concrete numeral word lengths, like the rest.

```
context
  notes minus_one_norm_num [simp]
begin

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
```

17 Splitting words into lists

```
theory Rsplit
  imports "HOL-Library.Word" More_Word Bits_Int
begin

definition word_rsplit :: "'a::len word ⇒ 'b::len word list"
  where "word_rsplit w = map word_of_int (bin_rsplit LENGTH('b) (LENGTH('a),
uint w))"

lemma word_rsplit_no:
  "(word_rsplit (numeral bin :: 'b::len word) :: 'a word list) =
    map word_of_int (bin_rsplit (LENGTH('a::len))
      (LENGTH('b), take_bit (LENGTH('b)) (numeral bin)))"
  by (simp add: word_rsplit_def of_nat_take_bit)

lemmas word_rsplit_no_cl [simp] = word_rsplit_no
  [unfolded bin_rsplitl_def bin_rsplit_l [symmetric]]
```

This odd result arises from the fact that the statement of the result implies that the decoded words are of the same type, and therefore of the same length, as the original word.

```
lemma word_rsplit_same: "word_rsplit w = [w]"
  apply (simp add: word_rsplit_def bin_rsplit_all)
  apply transfer
  apply simp
  done

lemma word_rsplit_empty_iff_size: "word_rsplit w = [] ⟷ size w = 0"
  by (simp add: word_rsplit_def bin_rsplit_def word_size bin_rsplit_aux_simp_alt
Let_def
      split: prod.split)

lemma test_bit_rsplit:
  "sw = word_rsplit w ⟹ m < size (hd sw) ⟹
    k < length sw ⟹ bit (rev sw ! k) m = bit w (k * size (hd sw) + m)"
  for sw :: "'a::len word list"
  apply (unfold word_rsplit_def word_test_bit_def)
  apply (rule trans)
   apply (rule_tac f = "λx. bit x m" in arg_cong)
   apply (rule nth_map [symmetric])
   apply simp
  apply (rule bin_nth_rsplit)
     apply simp_all
  apply (simp add : word_size rev_map)
  apply (rule trans)
   defer
   apply (rule map_ident [THEN fun_cong])
  apply (rule refl [THEN map_cong])
```

```
  apply (simp add: unsigned_of_int take_bit_int_eq_self_iff)
  using bin_rsplit_size_sign take_bit_int_eq_self_iff by blast

lemma test_bit_rsplit_alt:
  ‹bit ((word_rsplit w  :: 'b::len word list) ! i) m ⟷
    bit w ((length (word_rsplit w :: 'b::len word list) - Suc i) * size
(hd (word_rsplit w :: 'b::len word list)) + m)›
  if ‹i < length (word_rsplit w :: 'b::len word list)› ‹m < size (hd (word_rsplit
w :: 'b::len word list))› ‹0 < length (word_rsplit w :: 'b::len word
list)›
  for w :: ‹'a::len word›
  apply (rule trans)
   apply (rule test_bit_cong)
   apply (rule rev_nth [of _ ‹rev (word_rsplit w)›, simplified rev_rev_ident])
  apply simp
   apply (rule that(1))
  apply simp
  apply (rule test_bit_rsplit)
    apply (rule refl)
  apply (rule asm_rl)
   apply (rule that(2))
  apply (rule diff_Suc_less)
  apply (rule that(3))
  done

lemma word_rsplit_len_indep [OF refl refl refl refl]:
  "[u,v] = p ⟹ [su,sv] = q ⟹ word_rsplit u = su ⟹
    word_rsplit v = sv ⟹ length su = length sv"
  by (auto simp: word_rsplit_def bin_rsplit_len_indep)

lemma length_word_rsplit_size:
  "n = LENGTH('a::len) ⟹
    length (word_rsplit w :: 'a word list) ≤ m ⟷ size w ≤ m * n"
  by (auto simp: word_rsplit_def word_size bin_rsplit_len_le)

lemmas length_word_rsplit_lt_size =
  length_word_rsplit_size [unfolded Not_eq_iff linorder_not_less [symmetric]]

lemma length_word_rsplit_exp_size:
  "n = LENGTH('a::len) ⟹
    length (word_rsplit w :: 'a word list) = (size w + n - 1) div n"
  by (auto simp: word_rsplit_def word_size bin_rsplit_len)

lemma length_word_rsplit_even_size:
  "n = LENGTH('a::len) ⟹ size w = m * n ⟹
    length (word_rsplit w :: 'a word list) = m"
  by (cases ‹LENGTH('a)›) (simp_all add: length_word_rsplit_exp_size
div_nat_eqI)
```

```
lemmas length_word_rsplit_exp_size' = refl [THEN length_word_rsplit_exp_size]

— alternative proof of word_rcat_rsplit
lemmas tdle = times_div_less_eq_dividend
lemmas dtle = xtrans(4) [OF tdle mult.commute]

lemma word_rcat_rsplit: "word_rcat (word_rsplit w) = w"
  apply (rule word_eqI)
  apply (clarsimp simp: test_bit_rcat word_size)
  apply (subst refl [THEN test_bit_rsplit])
    apply (simp_all add: word_size
      refl [THEN length_word_rsplit_size [simplified not_less [symmetric],
simplified]])
   apply safe
   apply (erule xtrans(7), rule dtle)+
  done

lemma size_word_rsplit_rcat_size:
  "word_rcat ws = frcw ⟹ size frcw = length ws * LENGTH('a)
    ⟹ length (word_rsplit frcw::'a word list) = length ws"
  for ws :: "'a::len word list" and frcw :: "'b::len word"
  by (cases <LENGTH('a)>) (simp_all add: word_size length_word_rsplit_exp_size'
div_nat_eqI)

lemma word_rsplit_rcat_size [OF refl]:
  "word_rcat ws = frcw ⟹
    size frcw = length ws * LENGTH('a) ⟹ word_rsplit frcw = ws"
  for ws :: "'a::len word list"
  apply (frule size_word_rsplit_rcat_size, assumption)
  apply (clarsimp simp add : word_size)
  apply (rule nth_equalityI, assumption)
  apply clarsimp
  apply (rule word_eqI [rule_format])
  apply (rule trans)
   apply (rule test_bit_rsplit_alt)
     apply (clarsimp simp: word_size)+
  apply (rule trans)
   apply (rule test_bit_rcat [OF refl refl])
  apply (simp add: word_size)
  apply (subst rev_nth)
   apply arith
  apply (simp add: le0 [THEN [2] xtrans(7), THEN diff_Suc_less])
  apply safe
  apply (simp add: diff_mult_distrib)
   apply (cases "size ws")
    apply simp_all
  done

lemma word_rsplit_upt:
```

```
  "⟦ size x = LENGTH('a :: len) * n; n ≠ 0 ⟧
    ⟹ word_rsplit x = map (λi. ucast (x >> (i * LENGTH('a))) :: 'a word)
(rev [0 ..< n])"
  apply (subgoal_tac "length (word_rsplit x :: 'a word list) = n")
   apply (rule nth_equalityI, simp)
   apply (intro allI word_eqI impI)
   apply (simp add: test_bit_rsplit_alt word_size)
   apply (simp add: nth_ucast bit_simps rev_nth field_simps)
  apply (simp add: length_word_rsplit_exp_size word_size)
  apply (subst diff_add_assoc)
   apply (simp flip: less_eq_Suc_le)
  apply simp
  done

end
```

18 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
```

19 `sgn` and `abs` for `'a word`

19.1 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]
```

19.2 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
```

20 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" ("(1UCAST/(1'(_ → _')))")
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" ("(1SCAST/(1'(_ → _')))")
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" ("(1REVCAST/(1'(_ → _')))")
translations
  "REVCAST('s → 't)" => "CONST revcast :: ('s word ⇒ 't word)"
typed_print_translation
  ‹ [(@{const_syntax revcast}, Word_Syntax.tr' @{syntax_const "_Revcast"})]
›



end
```

21 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
```

22 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::'a) ≤ g x y`, `f x y ≤ - (1::'a)`, 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

23 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)"
  apply (rule iffI)
   apply (cases "LENGTH('b) < LENGTH('a)")
    apply (subst less_mask_eq[OF x, symmetric])
    apply (subst less_mask_eq[OF y, symmetric])
    apply (unfold n)
    apply (subst ucast_ucast_mask[symmetric])+
    apply (simp add: ucast_le_ucast)+
  apply (erule ucast_mono_le[OF _ y[unfolded n]])
  done

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))"
  apply (simp flip: take_bit_eq_mask)
  apply transfer
  apply (simp add: ac_simps)
  done

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
    apply (simp add: unsigned_ucast_eq take_bit_word_eq_self_iff)
    apply (meson ‹x ≤ 2 ^ LENGTH('b) - 1› not_le word_less_sub_le)
    done
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)))"
  apply (auto simp add: bit_simps word_size ac_simps not_less)
  apply (meson bit_imp_le_length bit_shiftr_word_iff leD)
```

done

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"
  apply (unfold 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›
  apply safe
   apply (rule word_le_0_iff [THEN iffD1])
   apply (rule xtrans(3))
    apply (erule_tac [2] le_shiftr)
   apply simp
  apply (rule word_leI)
  apply (rename_tac n')
  apply (drule_tac x = "n' - n" in word_eqD)
  apply (simp add : nth_shiftr word_size bit_simps)
  apply (case_tac "n <= n'")
  by auto

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:
  "⟦ (x::'a::len word) ≤ 2 ^ m; m + n < LENGTH('a::len); x ≠ 0⟧
   ⟹ x << n ≠ 0"
  apply (simp only: word_neq_0_conv word_less_nat_alt
```

```
                  shiftl_t2n mod_0 unat_word_ariths
                  unat_power_lower word_le_nat_alt)
  apply (subst mod_less)
   apply (rule order_le_less_trans)
    apply (erule mult_le_mono2)
   apply (subst power_add[symmetric])
   apply (rule power_strict_increasing)
    apply simp
   apply simp
  apply simp
  done

lemma word_shiftr_lt:
  fixes w :: "'a::len word"
  shows "unat (w >> n) < (2 ^ (LENGTH('a) - n))"
  apply (subst shiftr_div_2n')
  apply transfer
  apply (simp flip: drop_bit_eq_div add: drop_bit_nat_eq drop_bit_take_bit)
  done

lemma shiftr_less_t2n':
  "⟦ x AND mask (n + m) = x; m < LENGTH('a) ⟧ ⟹ x >> n < 2 ^ m" for x
:: "'a :: len word"
  apply (simp add: word_size mask_eq_iff_w2p [symmetric] flip: take_bit_eq_mask)
  apply transfer
  apply (simp add: take_bit_drop_bit ac_simps)
  done

lemma shiftr_less_t2n:
  "x < 2 ^ (n + m) ⟹ x >> n < 2 ^ m" for x :: "'a :: len word"
  apply (rule shiftr_less_t2n')
   apply (erule less_mask_eq)
  apply (rule ccontr)
  apply (simp add: not_less)
  apply (subst (asm) p2_eq_0[symmetric])
  apply (simp add: power_add)
  done

lemma shiftr_eq_0:
  "n ≥ LENGTH('a) ⟹ ((w::'a::len word) >> n) = 0"
  apply (cut_tac shiftr_less_t2n'[of w n 0], simp)
   apply (simp add: mask_eq_iff)
  apply (simp add: lt2p_lem)
  apply simp
  done

lemma shiftl_less_t2n:
  fixes x :: "'a :: len word"
  shows "⟦ x < (2 ^ (m - n)); m < LENGTH('a) ⟧ ⟹ (x << n) < 2 ^ m"
```

```
  apply (simp add: word_size mask_eq_iff_w2p [symmetric] flip: take_bit_eq_mask)
  apply transfer
  apply (simp add: take_bit_push_bit)
  done

lemma shiftl_less_t2n':
  "(x::'a::len word) < 2 ^ m ⟹ m+n < LENGTH('a) ⟹ x << n < 2 ^ (m
+ n)"
  by (rule shiftl_less_t2n) simp_all

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 add: 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›
  using shiftl_shiftr2[where a=a and b=0 and c="size a - b"]
  apply (cases "b < size a")
   apply simp
  apply (simp add: linorder_not_less mask_eq_decr_exp word_size
                   p2_eq_0[THEN iffD2])
  done
```

```
lemma shiftl_shiftr3:
  "b ≤ c ⟹ a << b >> c = (a >> c - b) AND mask (size a - c)"
  for a :: ‹'a::len word›
  apply (cases "b = c")
   apply (simp add: shiftl_shiftr1)
  apply (simp add: shiftl_shiftr2)
  done

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"
  apply (rule ccontr)
  apply (drule (2) word_shift_nonzero)
  apply simp
  done

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"
  apply (subst shiftr1_is_div_2)
  apply (rule div_less_dividend_word)
   apply simp+
  done

lemma shiftr1_0_or_1:"(x::('a::len) word) >> 1 = 0 ⟹ x = 0 ∨ x = 1"
  apply (subst (asm) shiftr1_is_div_2)
  apply (drule word_less_div)
  apply (case_tac "LENGTH('a) = 1")
   apply (simp add:degenerate_word)
  apply (erule disjE)
   apply (subgoal_tac "(2::'a word) ≠ 0")
    apply simp
   apply (rule not_degenerate_imp_2_neq_0)
   apply (subgoal_tac "LENGTH('a) ≠ 0")
    apply arith
   apply simp
  apply (rule x_less_2_0_1', simp+)
  done

lemma shiftr1_irrelevant_lsb: "bit (x::('a::len) word) 0 ∨ x >> 1 =
(x + 1) >> 1"
  by (auto simp add: 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:
  "⟦n = LENGTH('a::len); m = LENGTH('b::len); n * 2 = m⟧ ⟹
  (((ucast ((ucast (x::'b word))::'a word))::'b word) OR (((ucast ((ucast
(x >> n))::'a word))::'b word) << n)) = x"
  apply (subgoal_tac "((ucast ((ucast (x >> n))::'a word))::'b word) =
x >> n")
   apply clarsimp
   apply (subst and_not_mask[symmetric])
   apply (subst ucast_ucast_mask)
   apply (subst word_ao_dist2[symmetric])
   apply clarsimp
  apply (rule ucast_ucast_len)
  apply (rule shiftr_less_t2n')
   apply (subst and_mask_eq_iff_le_mask)
```

```
   apply (simp_all add: mask_eq_decr_exp flip: mult_2_right)
  apply (metis add_diff_cancel_left' len_gt_0 mult_2_right zero_less_diff)
  done

lemma cast_chunk_scast_assemble_id:
  "⟦n = LENGTH('a::len); m = LENGTH('b::len); n * 2 = m⟧ ⟹
  (((ucast ((scast (x::'b word))::'a word))::'b word) OR
   (((ucast ((scast (x >> n))::'a word))::'b word) << n)) = x"
  apply (subgoal_tac "((scast x)::'a word) = ((ucast x)::'a word)")
   apply (subgoal_tac "((scast (x >> n))::'a word) = ((ucast (x >> n))::'a
word)")
    apply (simp add:cast_chunk_assemble_id)
   apply (subst down_cast_same[symmetric], subst is_down, arith, simp)+
  done

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)"
  apply (erule order_less_le_trans[rotated])
  using ucast_less[where x=x and 'a='a]
  apply (simp only: shiftl_t2n field_simps)
  apply (rule word_less_power_trans2; simp)
  done


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 add: 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)}"
    apply(simp add: word_and_le1)
    apply(metis helper3 le_word_or2 word_bw_comms(1) word_bw_comms(2))
  done
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"
  apply (simp add: less_eq_mask_iff_take_bit_eq_self)
  apply transfer
  apply (simp add: ac_simps)
  done

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 [])"
  apply (induct as arbitrary: x; clarsimp)
   apply (case_tac "x < y")
    prefer 2
    apply (case_tac "x = y", simp)
    apply (simp add: not_less)
    apply (drule (1) dual_order.not_eq_order_implies_strict)
    apply (simp add: word_upto_Nil)
   apply (simp add: word_upto_Cons_eq)
  apply (case_tac "x < y")
   prefer 2
   apply (case_tac "x = y", simp)
   apply (simp add: not_less)
   apply (drule (1) dual_order.not_eq_order_implies_strict)
   apply (simp add: word_upto_Nil)
  apply (clarsimp simp: word_upto_Cons_eq)
  apply (frule word_enum_decomp_elem)
  apply clarsimp
  apply (rule conjI)
   prefer 2
   apply (subst word_Suc_le[symmetric]; clarsimp)
  apply (drule meta_spec)
  apply (drule (1) meta_mp)
  apply clarsimp
  apply (rule conjI; clarsimp)
  apply (subst (2) word_upto_Cons_eq)
   apply unat_arith
  apply simp
  done

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:
  "⟦ i ≤ j; k ≤ unat (j - i) ⟧ ⟹ [i .e. j] ! k = i + of_nat k"
  apply (clarsimp simp: upto_enum_def nth_append)
  apply (clarsimp simp: word_le_nat_alt[symmetric])
  apply (rule conjI, clarsimp)
   apply (subst toEnum_of_nat, unat_arith)
   apply unat_arith
  apply (clarsimp simp: not_less unat_sub[symmetric])
  apply unat_arith
  done
```

```
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:
  "a < - 1 ⟹ [(0 :: 'a :: len word) .e. 1 + a] = [0 .e. a] @ [1 + a]"
  apply (simp add: upto_enum_word)
  apply (subgoal_tac "unat (1+a) = 1 + unat a")
   apply simp
  apply (subst unat_plus_simple[THEN iffD1])
   apply (metis add.commute no_plus_overflow_neg olen_add_eqv)
  apply unat_arith
  done

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:
  "⟦ x AND mask n = 0; y AND mask n = 0; x ≤ x + y ⟧
   ⟹ (x + y :: ('a :: len) word) >> n = (x >> n) + (y >> n)"
  apply (simp add: no_olen_add_nat is_aligned_mask[symmetric])
  apply (simp add: unat_arith_simps shiftr_div_2n' split del: if_split)
  apply (subst if_P)
   apply (erule order_le_less_trans[rotated])
   apply (simp add: add_mono)
  apply (simp add: shiftr_div_2n' is_aligned_iff_dvd_nat)
  done

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)"
  using add_right_shift[where x="x - y" and y=y and n=n]
  by (simp add: aligned_sub_aligned is_aligned_mask[symmetric] word_sub_le)

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 [simp]:
```

```
  "(x :: 'a :: len word) AND mask LENGTH('a) = x"
  using uint_lt2p [of x] by (simp add: mask_eq_iff)

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"
  apply (drule sym)
  apply simp
  apply (rule bit_word_eqI)
  apply (auto simp add: bit_simps)
  done

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 (meson and_mask_eq_iff_le_mask eq_ucast_ucast_eq not_le_imp_less
shiftr_less_t2n'
            ucast_ucast_len)

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"
  apply (induct x arbitrary: n)
    apply simp
  by (simp add: 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))"
  apply (insert ucast_add[where a=a and b="-b"])
  apply (metis (no_types, opaque_lifting) add_diff_eq diff_add_cancel
ucast_add)
  done

lemma scast_ucast_add_one [simp]:
  "scast (ucast (x :: 'a::len word) + (1 :: 'a signed word)) = x + 1"
  apply (subst ucast_1[symmetric])
  apply (subst ucast_add[symmetric])
  apply clarsimp
  done

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"
  apply transfer
  apply (simp add: take_bit_drop_bit)
  apply (simp add: drop_bit_take_bit)
  apply (rule order_trans)
   defer
   apply assumption
  apply (simp add: nat_le_iff of_nat_diff)
  done

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"
  apply (rule bit_word_eqI)
  apply (auto simp add: bit_simps dest: bit_imp_le_length)
  done

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 (metis diff_diff_cancel le_m1_iff_lt less_imp_diff_less p2_gt_0 two_power_increasing
           word_1_le_power word_le_minus_mono_left word_less_sub_1)

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
    apply (transfer fixing: m n)
    apply (simp add: not_le take_bit_push_bit)
    apply (metis diff_le_self order_le_less_trans push_bit_of_0 take_bit_0
take_bit_int_eq_self
      take_bit_int_less_exp take_bit_nonnegative take_bit_tightened)
    done
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
    apply (transfer fixing: m n q r)
    apply (simp add: not_le take_bit_push_bit)
    apply (simp add: push_bit_eq_mult power_add)
    using take_bit_tightened_less_eq_int [of ‹r + q› ‹r + q + n›]
    apply (rule le_less_trans)
     apply simp_all
    done
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.right_neutral and_mask_eq_iff_le_mask and_not_mask le_mask_iff
mask_add_aligned
           mask_out_add_aligned mult_zero_right shiftl_t2n shiftr_le_0)

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:
  "2 ^ m ≠ (0 :: 'a word) ⟹ 2 ^ n ≤ - (2 ^ m :: 'a :: len word)"
  apply (cases "n < LENGTH('a)"; simp add: power_overflow)
  apply (cases "m < LENGTH('a)"; simp add: power_overflow)
  apply (simp add: word_le_nat_alt unat_minus word_size)
  apply (cases "LENGTH('a)"; simp)
  apply (simp add: less_Suc_eq_le)
  apply (drule power_increasing[where a=2 and n=n] power_increasing[where
a=2 and n=m], simp)+
  apply (drule(1) add_le_mono)
  apply simp
  done

lemma unat_shiftl_absorb:
  "⟦ x ≤ 2 ^ p; p + k < LENGTH('a) ⟧ ⟹ unat (x :: 'a :: len word) *
```

```
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:
  "⟦ unat ((x :: 'a :: len word) AND mask sz) + unat z < 2 ^ sz; sz <
LENGTH('a) ⟧
   ⟹ x ≤ x + z"
  apply (subgoal_tac "(x AND NOT(mask sz)) + (x AND mask sz) ≤ (x AND
NOT(mask sz)) + ((x AND mask sz) + z)")
   apply (simp add:word_plus_and_or_coroll2 field_simps)
  apply (rule word_plus_mono_right)
   apply (simp add: less_le_trans no_olen_add_nat)
  using of_nat_power is_aligned_no_wrap' by force

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 semiring_normalization_rules(7)
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_sub_mask mask_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 (split if_split, simp_all add: power_overflow)

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›
  apply (subst word_plus_and_or_coroll; word_eqI; fastforce?)
  apply (erule allE, drule (1) iffD2)
  apply clarsimp
  done

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 (metis add.commute le_mask_iff shiftr_mask_le 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)"
  apply (simp add: shiftl_def shiftr_def flip: push_bit_eq_mult take_bit_eq_mask
word_unat_eq_iff)
  apply (subst disjunctive_add)
   apply (auto simp add: bit_simps)
  apply (subst disjunctive_add)
   apply (auto simp add: bit_simps)
    apply (rule bit_word_eqI)
  apply (auto simp add: bit_simps)
```

done

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]"
apply (simp flip: take_bit_eq_mask)
apply (rule image_eqI [of _ _ ‹unat (take_bit n x)›])
using len **apply** simp_all
apply transfer
apply simp
done

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 (smt (verit) eq_or_less_helperD le_add2 le_eq_less_or_eq le_trans power_add unat_mult_lem unat_pow_le_intro unat_power_lower word_eq_unatI)

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 add: 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 add: 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 add: 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 fixing: m q)
    apply (simp add: signed_take_bit_take_bit)
    apply (rule impI)
    apply (subst bit_eq_iff)
    apply (simp add: bit_take_bit_iff bit_signed_take_bit_iff min_def)
    apply (auto simp add: Suc_le_eq)
    using less_imp_le_nat apply blast
    using less_imp_le_nat apply blast
    done
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 (clarsimp simp: le_mask_high_bits neg_mask_le_high_bits scast_ucast_high_bits
word_size)
  apply (rule iffI; clarsimp)
   apply (rename_tac i j; case_tac "i = LENGTH('b) - 1"; case_tac "j =
LENGTH('b) - 1")
  by auto

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)"
  apply (erule order_less_le_trans[rotated])
  using ucast_less[where x=x and 'a='a]
  apply (simp only: shiftl_t2n field_simps)
  apply (rule word_less_power_trans2; simp)
  done

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:
  "is_aligned p n ⟹ ([p , p + 2 ^ m .e. p + 2 ^ n - 1]) = map ((+) p)
[0, 2 ^ m .e. 2 ^ n - 1]"
  apply (erule is_aligned_get_word_bits)
   prefer 2
   apply (simp add: map_idI)
  apply (clarsimp simp: upto_enum_step_def)
  apply (frule is_aligned_no_overflow)
  apply (simp add: linorder_not_le [symmetric])
  done

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)]"
  apply (subst upto_enum_step_shift, assumption)
  apply (simp add: upto_enum_step_red)
  done

lemma upto_enum_step_subset:
  "set [x, y .e. z] ⊆ {x .. z}"
  apply (clarsimp simp: upto_enum_step_def linorder_not_less)
  apply (drule div_to_mult_word_lt)
  apply (rule conjI)
   apply (erule word_random[rotated])
   apply simp
  apply (rule order_trans)
   apply (erule word_plus_mono_right)
   apply simp
  apply simp
  done
```

```
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)"
  apply (simp only: ucast_eq)
  apply (subst lift_M)
  apply (subst of_int_uint [symmetric], subst lift_M')
  apply (metis local.distrib local.is_down take_bit_eq_mod ucast_down_wi
uint_word_of_int_eq word_of_int_uint)
  done

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 (rule ucast_distrib [where L="(+)"], (clarsimp simp: uint_word_ariths)+,
presburger, simp)

lemma ucast_down_minus:
    "is_down (ucast:: 'a word ⇒ 'b word) ⟹  ucast ((a :: 'a::len word)
- b) = (ucast a - ucast b :: 'b::len word)"
  apply (rule ucast_distrib [where L="(-)"], (clarsimp simp: uint_word_ariths)+)
  apply (metis mod_diff_left_eq mod_diff_right_eq)
  apply simp
  done

lemma ucast_down_mult:
    "is_down (ucast:: 'a word ⇒ 'b word) ⟹  ucast ((a :: 'a::len word)
* b) = (ucast a * ucast b :: 'b::len word)"
  apply (rule ucast_distrib [where L="(*)"], (clarsimp simp: uint_word_ariths)+)
  apply (metis mod_mult_eq)
  apply simp
  done

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)"
  apply (subst (1 2 3) down_cast_same [symmetric])
   apply (insert is_down)
   apply (clarsimp simp: is_down_def target_size source_size is_down)
  apply (rule ucast_distrib [where L=L, OF lift_M lift_M' distrib])
  apply (insert is_down)
  apply (clarsimp simp: is_down_def target_size source_size is_down)
  done

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 (rule scast_distrib [where L="(+)"], (clarsimp simp: uint_word_ariths)+,
presburger, simp)

lemma scast_down_minus:
    "is_down (scast:: 'a word ⇒ 'b word) ⟹  scast ((a :: 'a::len word)
- b) = (scast a - scast b :: 'b::len word)"
  apply (rule scast_distrib [where L="(-)"], (clarsimp simp: uint_word_ariths)+)
  apply (metis mod_diff_left_eq mod_diff_right_eq)
  apply simp
  done

lemma scast_down_mult:
    "is_down (scast:: 'a word ⇒ 'b word) ⟹  scast ((a :: 'a::len word)
* b) = (scast a * scast b :: 'b::len word)"
  apply (rule scast_distrib [where L="(*)"], (clarsimp simp: uint_word_ariths)+)
  apply (metis mod_mult_eq)
  apply simp
  done

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"
  apply (simp only: scast_eq)
  apply (metis down_cast_same is_up_down scast_eq ucast_down_wi)
  done

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 (clarsimp)

  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)"
    apply (rule iffI [rotated])
     apply (clarsimp simp: signed_divide_int_def sgn_if word_size sint_int_min)
    apply (rule classical)
    apply (case_tac "?a_int_min")
     apply (clarsimp simp: word_size sint_int_min)
     apply (metis diff_0_right
              int_sdiv_negated_is_minus1 minus_diff_eq minus_int_code(2)
```

```
            power_eq_0_iff sint_minus1 zero_neq_numeral)
    apply (subgoal_tac "abs (sint a) < 2 ^ (size a - 1)")
     apply (insert sdiv_int_range [where a="sint a" and b="sint b"])[1]
     apply (clarsimp simp: word_size)
    apply (insert sdiv_int_range [where a="sint a" and b="sint b"])[1]
    apply auto
    apply (cases ‹size a›)
     apply simp_all
    apply (smt (verit) One_nat_def diff_Suc_1 signed_word_eqI sint_int_min
sint_range_size wsst_TYs(3))
    done

  have result_range_simple: "(sint a sdiv sint b ∈ ?range) ⟹ ?thesis"
    apply (insert sdiv_int_range [where a="sint a" and b="sint b"])
    apply (clarsimp simp: word_size sint_int_min)
    done

  show ?thesis
    apply (rule UnE [OF result_range result_range_simple])
     apply simp
    apply (clarsimp simp: word_size)
    using result_range_overflow
    apply (clarsimp simp: word_size)
    done
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 add: 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

lemma nasty_split_less:
  "⟦m ≤ n; n ≤ nm; nm < LENGTH('a::len); x < 2 ^ (nm - n)⟧
   ⟹ (x :: 'a word) * 2 ^ n + (2 ^ m - 1) < 2 ^ nm"
  apply (simp only: word_less_sub_le[symmetric])
  apply (rule order_trans [OF _ nasty_split_lt])
     apply (rule word_plus_mono_right)
      apply (rule word_sub_mono)
         apply (simp add: word_le_nat_alt)
```

```
       apply simp
      apply (simp add: word_sub_1_le[OF power_not_zero])
     apply (simp add: word_sub_1_le[OF power_not_zero])
    apply (rule is_aligned_no_wrap')
     apply (rule is_aligned_mult_triv2)
    apply simp
   apply (erule order_le_less_trans, simp)
  apply simp+
 done

end

end
```

24 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
```

25 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
```

26 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"      ("~~ _" [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
```

27 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
```

28 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› ..
  from ‹bit w n› show ?thesis
    apply (simp add: word_ctz_unfold)
    apply (subst Min_eq_length_takeWhile [symmetric])
      apply (auto simp add: bit_imp_le_length)
    apply (subst Min_insert)
      apply auto
    apply (subst min.absorb2)
     apply (subst Min_le_iff)
       apply auto
    apply (meson bit_imp_le_length order_less_le)
    done
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)"
  apply (clarsimp simp: word_ctz_def)
  using length_takeWhile_le apply (rule order_trans)
  apply simp
  done

lemma word_ctz_less:
  "w ≠ 0 ⟹ word_ctz (w :: ('a::len word)) < LENGTH('a)"
  apply (clarsimp simp: word_ctz_def eq_zero_set_bl)
  using length_takeWhile_less apply (rule less_le_trans)
  apply auto
  done

lemma take_bit_word_ctz_eq [simp]:
  ‹take_bit LENGTH('a) (word_ctz w) = word_ctz w›
  for w :: ‹'a::len word›
  apply (simp add: take_bit_nat_eq_self_iff word_ctz_def to_bl_unfold)
  using length_takeWhile_le apply (rule le_less_trans)
  apply simp
  done

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 (auto simp add: bit_signed_take_bit_iff
        sign_extend_def bit_and_iff bit_or_iff bit_not_iff bit_mask_iff
not_less
        exp_eq_0_imp_not_bit not_le min_def)
  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)"
  apply (rule iffI)
   apply (clarsimp simp:pop_count_def)
   apply (subst (asm) filter_empty_conv)
   apply (clarsimp simp:eq_zero_set_bl)
   apply fast
  apply simp
  done

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_same:
  "w ≠ 0 ⟹ bit w (word_log2 w)"
  by (drule bit_word_log2) simp

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_highest:
  assumes a: "bit w i"
  shows "i ≤ word_log2 w"
  using a by (simp add: word_log2_maximum)

lemma word_log2_max:
  "word_log2 w < size w"
  apply (cases ‹w = 0›)
   apply (simp_all add: word_size)
  apply (drule bit_word_log2)
  apply (fact bit_imp_le_length)
  done

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
  with take_bit_int_less_exp [of n]
    have *: ‹take_bit n k < 2 ^ LENGTH('a)› for k :: int
    by (rule less_trans)
  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"
    apply (transfer fixing: n) using sz
    apply (simp flip: take_bit_eq_mod add: div_eq_0_iff)
    apply (subst take_bit_int_eq_self)
    using *
     apply (auto simp add: diff_less_eq intro: less_imp_le)
    apply (simp add: less_le)
    done
  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 show "alignUp a n = (a div 2 ^ n + 1) * 2 ^ n" using sz
    unfolding alignUp_def
    apply (subst mask_eq_decr_exp [symmetric])
    apply (erule ssubst)
    apply (subst neg_mask_is_div)
    apply (simp add: word_arith_nat_div)
    apply (subst unat_word_ariths(1) unat_word_ariths(2))+
    apply (subst uno_simps)
    apply (subst unat_1)
    apply (subst mod_add_right_eq)
    apply simp
    apply (subst power_mod_div)
    apply (subst div_mult_self1)
     apply simp
    apply (subst um)
    apply simp
    apply (subst mod_mod_power)
    apply simp
    apply (subst word_unat_power, subst Abs_fnat_hom_mult)
    apply (subst mult_mod_left)
    apply (subst power_add [symmetric])
    apply simp
    apply (subst Abs_fnat_hom_1)
    apply (subst Abs_fnat_hom_add)
    apply (subst word_unat_power, subst Abs_fnat_hom_mult)
    apply (subst word_unat.Rep_inverse[symmetric], subst Abs_fnat_hom_mult)
    apply simp
    done
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"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
    apply -
    apply (erule contrapos_nn)
    apply (subst alignUp_not_aligned_eq [OF False sz])
    apply (subst unat_arith_simps)
    apply (subst unat_word_ariths)
    apply (subst unat_word_ariths)
    apply simp
    apply (subst mult_mod_left)
    apply (simp add: unat_div field_simps power_add[symmetric] mod_mod_power)
    done
  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
    apply (simp add: field_simps)
    apply (rule word_add_less_mono1)
    apply (rule word_mod_less_divisor)
    apply (simp add: word_less_nat_alt)
    apply (subst unat_word_ariths)
    apply (simp add: unat_div)
    done
  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
    apply (subst unat_of_nat_eq)
     apply (erule order_less_le_trans)
     apply simp
    apply (subst mult.commute)
    apply simp
    apply (rule nat_less_power_trans)
     apply simp
    apply simp
    done

  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:
  "is_aligned p sz ⟹ (p - q AND NOT (mask sz)) = (p - ((alignUp q sz)
AND NOT (mask sz)))"
  apply (clarsimp simp only:word_and_le2 diff_conv_add_uminus)
  apply (subst mask_out_add_aligned[symmetric]; simp)
  apply (simp add: eq_neg_iff_add_eq_0)
  apply (subst add.commute)
  apply (simp add: alignUp_distance is_aligned_neg_mask_eq mask_out_add_aligned
and_mask_eq_iff_le_mask flip: mask_eq_x_eq_0)
```

```
  done

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"
  apply auto
   apply (auto simp add: bit_eq_iff)
    apply (simp_all add: bit_simps sign_extend_eq_signed_take_bit not_le
min_def sign_extended_def word_size split: if_splits)
  using le_imp_less_or_eq apply auto
  done

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
     apply (simp add: sign_extended_iff_sign_extend sign_extend_def i)
     apply (simp add: and_or word_bw_comms[of p f])
     apply (clarsimp simp: word_ao_dist fm word_bw_assocs split: if_splits)
    done
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
  apply (subst arg_cong2[where f="(AND)"])
    apply (erule sym)+
  apply (simp(no_asm) add: word_bw_assocs word_bw_comms word_bw_lcs)
  done

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"
  by (simp add: from_bool_def limited_and_def split: bool.split)

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:
  "is_aligned ptr bits ⟹ (ptr' AND NOT(mask bits) = ptr) = (ptr' ∈ mask_range
ptr bits)"
  apply (erule is_aligned_get_word_bits)
   apply (rule iffI)
    apply (drule sym)
    apply (simp add: word_and_le2)
    apply (subst word_plus_and_or_coroll, word_eqI_solve)
    apply (metis bit.disj_ac(2) bit.disj_conj_distrib2 le_word_or2 word_and_max
word_or_not)
   apply clarsimp
   apply (smt (verit) add.right_neutral eq_iff is_aligned_neg_mask_eq
mask_out_add_aligned neg_mask_mono_le
              word_and_not)
  apply (simp add: power_overflow mask_eq_decr_exp)
  done

lemma aligned_offset_in_range:
  "⟦ is_aligned (x :: 'a :: len word) m; y < 2 ^ m; is_aligned p n; n
≥ m; n < LENGTH('a) ⟧
   ⟹ (x + y ∈ {p .. p + mask n}) = (x ∈ mask_range p n)"
  apply (subst disjunctive_add)
   apply (simp add: bit_simps)
  apply (erule is_alignedE')
   apply (auto simp add: bit_simps not_le)[1]
   apply (metis less_2p_is_upper_bits_unset)
  apply (simp only: is_aligned_add_or word_ao_dist flip: neg_mask_in_mask_range)
  apply (subgoal_tac ‹y AND NOT (mask n) = 0›)
   apply simp
  apply (metis (full_types) is_aligned_mask is_aligned_neg_mask less_mask_eq
word_bw_comms(1) word_bw_lcs(1))
  done

lemma mask_range_to_bl':
  "⟦ is_aligned (ptr :: 'a :: len word) bits; bits < LENGTH('a) ⟧
   ⟹ mask_range ptr bits
       = {x. take (LENGTH('a) - bits) (to_bl x) = take (LENGTH('a) - bits)
(to_bl ptr)}"
  apply (rule set_eqI, rule iffI)
   apply clarsimp
```

```
   apply (subgoal_tac "∃y. x = ptr + y ∧ y < 2 ^ bits")
    apply clarsimp
    apply (subst is_aligned_add_conv)
       apply assumption
      apply simp
    apply simp
   apply (rule_tac x="x - ptr" in exI)
   apply (simp add: add_diff_eq[symmetric])
   apply (simp only: word_less_sub_le[symmetric])
   apply (rule word_diff_ls')
    apply (simp add: field_simps mask_eq_decr_exp)
   apply assumption
  apply simp
  apply (subgoal_tac "∃y. y < 2 ^ bits ∧ to_bl (ptr + y) = to_bl x")
   apply clarsimp
   apply (rule conjI)
    apply (erule(1) is_aligned_no_wrap')
   apply (simp only: add_diff_eq[symmetric] mask_eq_decr_exp)
   apply (rule word_plus_mono_right)
    apply simp
   apply (erule is_aligned_no_wrap')
   apply simp
  apply (rule_tac x="of_bl (drop (LENGTH('a) - bits) (to_bl x))" in exI)
  apply (rule context_conjI)
   apply (rule order_less_le_trans [OF of_bl_length])
    apply simp
   apply simp
  apply (subst is_aligned_add_conv)
     apply assumption
    apply simp
  apply (drule sym)
  apply (simp add: word_rep_drop)
  done

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 (erule is_aligned_get_word_bits)
   apply (erule(1) mask_range_to_bl')
  apply (rule set_eqI)
  apply (simp add: power_overflow mask_eq_decr_exp)
  done

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 (simp add: mask_range_to_bl)
  apply (rule Meson.disj_comm, rule disjCI)
  apply auto
  apply (subgoal_tac "(∃n''. LENGTH('a) - n = (LENGTH('a) - n') + n'')
                   ∨ (∃n''. LENGTH('a) - n' = (LENGTH('a) - n) + n'')")
   apply (fastforce simp: take_add)
  apply arith
  done

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 using szv szv'
    apply auto
    using al assms(4) is_aligned_no_wrap' apply blast
    apply (simp only: flip: add_diff_eq add_mask_fold)
    apply (subst add.assoc, rule word_plus_mono_right)
     using al' is_aligned_add_less_t2n xsz
     apply fastforce
    apply (simp add: field_simps szv al is_aligned_no_overflow)
    done
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:
  "⟦ is_aligned p n; is_aligned ptr bits; n ≥ m; n < size p; m ≤ bits;
```

```
    (∀y < 2 ^ (n - m). p + (y << m) ∉ mask_range ptr bits) ⟧
   ⟹ mask_range p n ∩ mask_range ptr bits = {}"
  apply safe
  apply (simp only: flip: neg_mask_in_mask_range)
  apply (drule_tac x="x AND mask n >> m" in spec)
  apply (erule notE[OF mp])
   apply (simp flip: take_bit_eq_mask add: shiftr_def drop_bit_take_bit)
   apply transfer
  apply simp
   apply (simp add: word_size and_mask_less_size)
  apply (subst disjunctive_add)
   apply (auto simp add: bit_simps word_size intro!: bit_eqI)
  done

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]
  apply (simp add: word_size signed_of_nat)
  apply (subst signed_take_bit_int_eq_self)
    apply simp_all
   apply (metis negative_zle of_nat_numeral semiring_1_class.of_nat_power)
  apply (drule small_powers_of_2)
  apply (erule le_less_trans)
  apply simp
  done

lemma word_clz_sint_lower[simp]:
  "LENGTH('a) ≥ 3
   ⟹ - sint (of_nat (word_clz (w :: 'a :: len word)) :: 'a signed word)
≤ int (LENGTH('a))"
  apply (subst sint_eq_uint)
  using word_clz_max [of w]
   apply (simp_all add: word_size unsigned_of_nat)
  apply (rule not_msb_from_less)
  apply (simp add: word_less_nat_alt unsigned_of_nat)
  apply (subst take_bit_nat_eq_self)
   apply (simp add: le_less_trans)
  apply (drule small_powers_of_2)
  apply (erule le_less_trans)
  apply simp
  done

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 < 2 ^ (LENGTH('a) - Suc 0)›
    using word_ctz_le apply (rule le_less_trans)
    using that small_powers_of_2 [of ‹LENGTH('a)›] apply simp
    done
  have ‹int (word_ctz x) div 2 ^ (LENGTH('a) - Suc 0) = 0›
    apply (rule div_pos_pos_trivial)
     apply (simp_all add: *)
    done
  then show ?P by (simp add: signed_of_nat bit_iff_odd)
  show ?Q
    apply (auto simp add: signed_of_nat)
    apply (subst signed_take_bit_int_eq_self)
      apply (auto simp add: word_ctz_le * minus_le_iff [of _ ‹int (word_ctz
x)›])
    apply (rule order.trans [of _ 0])
     apply simp_all
    done
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"
  apply (frule_tac p' = ptr in aligned_mask_range_cases, assumption)
  apply (elim disjE)
```

```
    apply (drule_tac is_aligned_no_overflow_mask, simp)+
    apply (simp add: algebra_split_simps word_le_not_less)
   apply (drule is_aligned_no_overflow_mask; fastforce)
  apply (simp add: is_aligned_weaken algebra_split_simps)
  apply (auto simp add: not_le)
  using is_aligned_no_overflow_mask leD apply blast
  apply (meson aligned_add_mask_less_eq is_aligned_weaken le_less_trans)
  done

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
```

29 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

30 Ancient comprehensive Word Library

theory Word_Lib_Sumo
imports
  "HOL-Library.Word"
  Aligned
  Bit_Comprehension
  Bit_Comprehension_Int
  Bit_Shifts_Infix_Syntax
  Bits_Int
  Bitwise_Signed
  Bitwise
  Enumeration_Word
  Generic_set_bit
  Hex_Words
  Least_significant_bit
  More_Arithmetic
  More_Divides
  More_Sublist
  Even_More_List
  More_Misc
  Strict_part_mono
  Legacy_Aliases
  Most_significant_bit
  Next_and_Prev
  Norm_Words
  Reversed_Bit_Lists
  Rsplit
  Signed_Words
  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
```

31 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
```

32 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
```

33 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
```

34 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
```

35 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
```

36 A short overview over bit operations and word types

36.1 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` $\longleftrightarrow$ `(`$\forall$`n. 2^n` $\neq$ `0` $\longrightarrow$ `bit a n` $\longleftrightarrow$ `bit b n)`
  - Induction rule:

    ```
    ⟦⋀a. a div 2 = a ⟹ P a;
     ⋀a b. ⟦P a; (of_bool b + 2 * a) div 2 = a⟧ ⟹ P (of_bool b
    + 2 * a)⟧
    ⟹ P a
    ```
- Characteristic properties `bit (f x) n` $\longleftrightarrow$ `P x n` are available in fact collection `bit_simps`.

On top of this, the following generic operations are provided:

- Singleton `n`th bit: 2^n
- Bit mask upto bit `n`: `mask n = 2^n - 1`
- Left shift: `push_bit n a = a * 2^n`
- Right shift: `drop_bit n a = a div 2^n`
- Truncation: `take_bit n a = a mod 2^n`
- Bitwise negation: `bit (NOT a) n` $\longleftrightarrow$ `2^n` $\neq$ `0` $\wedge$ $\neg$ `bit a n`
- Bitwise conjunction: `bit (a AND b) n` $\longleftrightarrow$ `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: `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).

36.2 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.

36.3 More library theories

Note: currently, most theories listed here are hardly separate entities since they import each other in various ways. Always inspect them to understand what you pull in if you want to import one.

Syntax `Word_Lib.Syntax_Bundles` Bundles to provide alternative syntax for various bit operations.

`Word_Lib.Hex_Words` Printing word numerals as hexadecimal numerals.

`Word_Lib.Type_Syntax` Pretty type-sensitive syntax for cast operations.

`Word_Lib.Word_Syntax` Specific ASCII syntax for prominent bit operations on word.

Proof tools `Word_Lib.Norm_Words` Rewriting word numerals to normal forms.

`Word_Lib.Bitwise` Method `word_bitwise` decomposes word equalities and inequalities into bit propositions.

`Word_Lib.Bitwise_Signed` Method `word_bitwise_signed` decomposes word equalities and inequalities into bit propositions.

`Word_Lib.Word_EqI` Method `word_eqI_solve` decomposes word equalities and inequalities into bit propositions.

Operations `Word_Lib.Signed_Division_Word` Signed division on word:

- `(sdiv) :: 'a::len word ⇒ 'a word ⇒ 'a word`
- `(smod) :: 'a::len word ⇒ 'a word ⇒ 'a word`

`Word_Lib.Aligned`

- `is_aligned w n ⟷ 2^n udvd w`

`Word_Lib.Least_significant_bit` The least significant bit as an alias: `lsb = odd`

`Word_Lib.Most_significant_bit` The most significant bit:

- `msb k ⟷ k < 0`
- `msb w ⟷ sint w < 0`
- `msb w ⟷ w <s 0`
- `msb w ⟷ bit w (LENGTH('a) - Suc 0)`

`Word_Lib.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.

36.4 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.

36.5 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` Kind of an alias: `Generic_set_bit.set_bit a n b = (if b then set_bit else unset_bit) n a`

`Word_Lib.Typedef_Morphisms` A low-level extension to HOL typedef providing conversions along type morphisms. The `transfer` method seems to be sufficient for most applications though.

`Word_Lib.Bit_Comprehension` Comprehension syntax for bit values over predicates `nat ⇒ bool`, for `'a word`; straightforward alternatives exist.

`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.

37 Changelog

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`

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` ≡ λ`w. w div 2`, `bintrunc` ≡ `take_bit`, `sbintrunc` ≡ `signed_take_bit`, `norm_sint` ≡ λ`n. signed_take_bit (n - 1)`, `bin_cat` ≡ λ`k 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. 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.