Machine Words in Isabelle/HOL

Jeremy Dawson, Paul Graunke, Brian Huffman, Gerwin Klein, and John Matthews

October 8, 2017

Abstract

A formalisation of generic, fixed size machine words in Isabelle/HOL.
An earlier version of this formalisation is described in [1].

Contents

1 Syntactic classes for bitwise operations **4**

2 Bit operations in $\mathcal{Z}_\in$ **4**

3 Useful Numerical Lemmas **6**

4 Integers as implict bit strings **7**
4.1 Constructors and destructors for binary integers 7
4.2 Truncating binary integers 12
4.3 Simplifications for (s)bintrunc 13
4.4 Splitting and concatenation 21

5 Bitwise Operations on Binary Integers **22**
5.1 Logical operations 22
5.1.1 Basic simplification rules 22
5.1.2 Binary destructors 23
5.1.3 Derived properties 24
5.1.4 Simplification with numerals 26
5.1.5 Interactions with arithmetic 29
5.1.6 Truncating results of bit-wise operations 30
5.2 Setting and clearing bits 30
5.3 Splitting and concatenation 32
5.4 Miscellaneous lemmas 35

6 Bool lists and integers **36**
6.1 Operations on lists of booleans . . . 36
6.2 Arithmetic in terms of bool lists . . . 37
6.3 Links between bit-wise operations and operations on bool lists 44
6.4 Links with function *bl-to-bin* . . . 48
6.5 Function *bl-of-nth* . . . 49
6.6 Repeated splitting or concatenation . . . 53

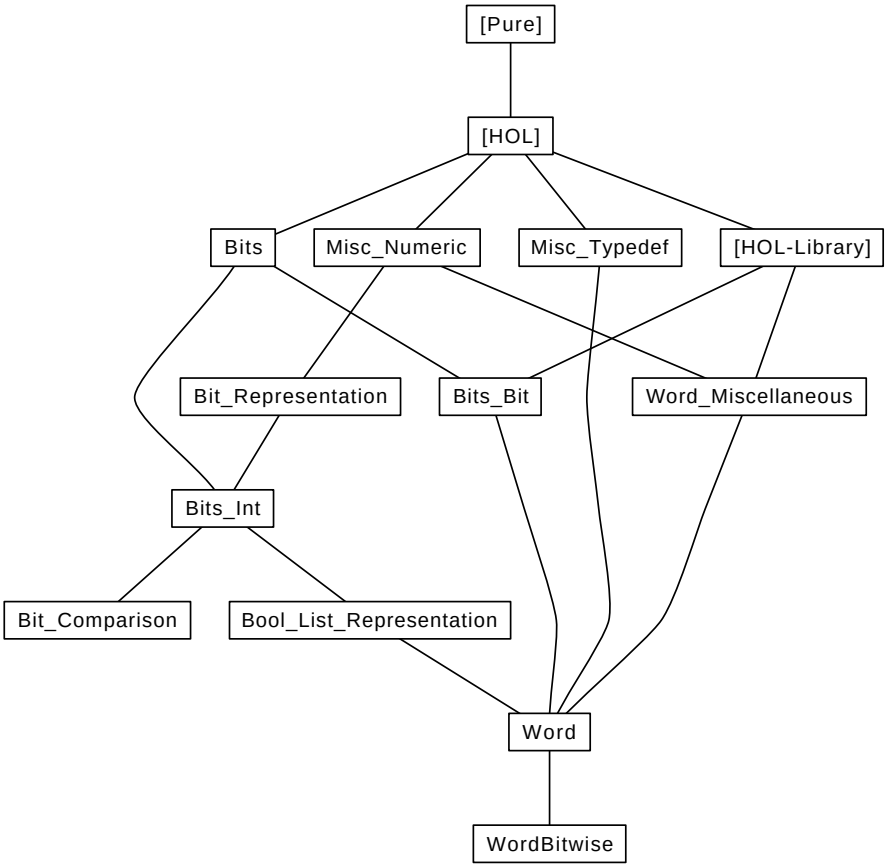
7 Type Definition Theorems **58**

8 More lemmas about normal type definitions **59**
8.1 Extended form of type definition predicate . . . 60

9 Miscellaneous lemmas, of at least doubtful value **63**

10 A type of finite bit strings **71**
10.1 Type definition . . . 71
10.2 Type conversions and casting . . . 72
10.3 Correspondence relation for theorem transfer . . . 74
10.4 Basic code generation setup . . . 75
10.5 Type-definition locale instantiations . . . 75
10.6 Arithmetic operations . . . 76
10.7 Ordering . . . 78
10.8 Bit-wise operations . . . 78
10.9 Shift operations . . . 80
10.10 Rotation . . . 80
10.11 Split and cat operations . . . 80
10.12 Theorems about typedefs . . . 81
10.13 Testing bits . . . 86
10.14 Word Arithmetic . . . 94
10.15 Transferring goals from words to ints . . . 95
10.16 Order on fixed-length words . . . 97
10.17 Conditions for the addition (etc) of two words to overflow . . . 99
10.18 Definition of *uint-arith* . . . 100
10.19 More on overflows and monotonicity . . . 101
10.20 Arithmetic type class instantiations . . . 106
10.21 Word and nat . . . 106
10.22 Definition of *unat-arith* tactic . . . 110
10.23 Cardinality, finiteness of set of words . . . 114
10.24 Bitwise Operations on Words . . . 114
10.25 Shifting, Rotating, and Splitting Words . . . 125
10.25.1 shift functions in terms of lists of bools . . . 128
10.25.2 Mask . . . 134
10.25.3 Revcast . . . 136

10.25.4 Slices . . . 139
10.26 Split and cat . . . 142
10.26.1 Split and slice . . . 145
10.27 Rotation . . . 149
10.27.1 Rotation of list to right . . . 149
10.27.2 map, map2, commuting with rotate(r) . . . 151
10.27.3 "Word rotation commutes with bit-wise operations . . . 154
10.28 Maximum machine word . . . 155
10.29 Recursion combinator for words . . . 161
10.30 Tactic definition . . . 172



1 Syntactic classes for bitwise operations

theory *Bits*
imports *Main*
begin

class *bit* =
fixes *bitNOT* :: *'a* ⇒ *'a* (*NOT - [70] 71*)
and *bitAND* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** *AND 64*)
and *bitOR* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** *OR 59*)
and *bitXOR* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** *XOR 59*)

We want the bitwise operations to bind slightly weaker than + and −, but ~~ to bind slightly stronger than ∗.

Testing and shifting operations.

class *bits* = *bit* +
fixes *test-bit* :: *'a* ⇒ *nat* ⇒ *bool* (**infixl** !! *100*)
and *lsb* :: *'a* ⇒ *bool*
and *set-bit* :: *'a* ⇒ *nat* ⇒ *bool* ⇒ *'a*
and *set-bits* :: (*nat* ⇒ *bool*) ⇒ *'a* (**binder** *BITS 10*)
and *shiftl* :: *'a* ⇒ *nat* ⇒ *'a* (**infixl** << *55*)
and *shiftr* :: *'a* ⇒ *nat* ⇒ *'a* (**infixl** >> *55*)

class *bitss* = *bits* +
fixes *msb* :: *'a* ⇒ *bool*

end

2 Bit operations in $\mathcal{Z}_\in$

theory *Bits-Bit*
imports *Bits HOL−Library.Bit*
begin

instantiation *bit* :: *bit*
begin

primrec *bitNOT-bit*
where
NOT 0 = (*1*::*bit*)
| *NOT 1* = (*0*::*bit*)

primrec *bitAND-bit*
where
0 AND y = (*0*::*bit*)
| *1 AND y* = (*y*::*bit*)

primrec *bitOR-bit*

where
0 OR y = (*y*::*bit*)
| *1 OR y* = (*1*::*bit*)

primrec *bitXOR-bit*
where
0 XOR y = (*y*::*bit*)
| *1 XOR y* = (*NOT y* :: *bit*)

instance ..

end

lemmas *bit-simps* =
bitNOT-bit.simps bitAND-bit.simps bitOR-bit.simps bitXOR-bit.simps

lemma *bit-extra-simps* [*simp*]:
x AND 0 = 0
x AND 1 = x
x OR 1 = 1
x OR 0 = x
x XOR 1 = NOT x
x XOR 0 = x
for *x* :: *bit*
by (*cases x*, *auto*)+

lemma *bit-ops-comm*:
x AND y = y AND x
x OR y = y OR x
x XOR y = y XOR x
for *x* :: *bit*
by (*cases y*, *auto*)+

lemma *bit-ops-same* [*simp*]:
x AND x = x
x OR x = x
x XOR x = 0
for *x* :: *bit*
by (*cases x*, *auto*)+

lemma *bit-not-not* [*simp*]: *NOT* (*NOT x*) = *x*
for *x* :: *bit*
by (*cases x*) *auto*

lemma *bit-or-def*: *b OR c* = *NOT* (*NOT b AND NOT c*)
for *b c* :: *bit*
by (*induct b*) *simp-all*

lemma *bit-xor-def*: *b XOR c* = (*b AND NOT c*) *OR* (*NOT b AND c*)

for *b c* :: *bit*
by (*induct b*) *simp-all*

lemma *bit-NOT-eq-1-iff* [*simp*]: *NOT b = 1* ⟷ *b = 0*
for *b* :: *bit*
by (*induct b*) *simp-all*

lemma *bit-AND-eq-1-iff* [*simp*]: *a AND b = 1* ⟷ *a = 1* ∧ *b = 1*
for *a b* :: *bit*
by (*induct a*) *simp-all*

end

3 Useful Numerical Lemmas

theory *Misc-Numeric*
imports *Main*
begin

lemma *one-mod-exp-eq-one* [*simp*]: *1 mod* (*2* ∗ *2* ^ *n*) = (*1*::*int*)
by (*smt mod-pos-pos-trivial zero-less-power*)

lemma *mod-2-neq-1-eq-eq-0*: *k mod 2* ≠ *1* ⟷ *k mod 2 = 0*
for *k* :: *int*
by (*fact not-mod-2-eq-1-eq-0*)

lemma *z1pmod2*: (*2* ∗ *b* + *1*) *mod 2* = (*1*::*int*)
for *b* :: *int*
by *arith*

lemma *diff-le-eq'*: *a* − *b* ≤ *c* ⟷ *a* ≤ *b* + *c*
for *a b c* :: *int*
by *arith*

lemma *emep1*: *even n* ⟹ *even d* ⟹ *0* ≤ *d* ⟹ (*n* + *1*) *mod d* = (*n mod d*) + *1*
for *n d* :: *int*
by (*auto simp add*: *pos-zmod-mult-2 add.commute dvd-def*)

lemma *int-mod-ge*: *a* < *n* ⟹ *0* < *n* ⟹ *a* ≤ *a mod n*
for *a n* :: *int*
by (*metis dual-order.trans le-cases mod-pos-pos-trivial pos-mod-conj*)

lemma *int-mod-ge'*: *b* < *0* ⟹ *0* < *n* ⟹ *b* + *n* ≤ *b mod n*
for *b n* :: *int*
by (*metis add-less-same-cancel2 int-mod-ge mod-add-self2*)

lemma *int-mod-le'*: *0* ≤ *b* − *n* ⟹ *b mod n* ≤ *b* − *n*
for *b n* :: *int*

by (*metis minus-mod-self2 zmod-le-nonneg-dividend*)

lemma *zless2*: $0 < (2 :: int)$
by (*fact zero-less-numeral*)

lemma *zless2p*: $0 < (2 \hat{} n :: int)$
by *arith*

lemma *zle2p*: $0 \leq (2 \hat{} n :: int)$
by *arith*

lemma *m1mod2k*: $-1 \bmod 2 \hat{} n = (2 \hat{} n - 1 :: int)$
using *zless2p* **by** (*rule zmod-minus1*)

lemma *p1mod22k'*: $(1 + 2 * b) \bmod (2 * 2 \hat{} n) = 1 + 2 * (b \bmod 2 \hat{} n)$
for $b :: int$
using *zle2p* **by** (*rule pos-zmod-mult-2*)

lemma *p1mod22k*: $(2 * b + 1) \bmod (2 * 2 \hat{} n) = 2 * (b \bmod 2 \hat{} n) + 1$
for $b :: int$
by (*simp add*: *p1mod22k' add.commute*)

lemma *int-mod-lem*: $0 < n \Longrightarrow 0 \leq b \wedge b < n \longleftrightarrow b \bmod n = b$
for $b\ n :: int$
apply *safe*
apply (*erule* (*1*) *mod-pos-pos-trivial*)
apply (*erule-tac* [!] *subst*)
apply *auto*
done

end

4 Integers as implict bit strings

theory *Bit-Representation*
imports *Misc-Numeric*
begin

4.1 Constructors and destructors for binary integers

definition *Bit* :: $int \Rightarrow bool \Rightarrow int$ (**infixl** *BIT 90*)
where $k\ BIT\ b = (if\ b\ then\ 1\ else\ 0) + k + k$

lemma *Bit-B0*: $k\ BIT\ False = k + k$
by (*simp add*: *Bit-def*)

lemma *Bit-B1*: $k\ BIT\ True = k + k + 1$
by (*simp add*: *Bit-def*)

lemma *Bit-B0-2t*: *k BIT False = 2 * k*
by (*rule trans*, *rule Bit-B0*) *simp*

lemma *Bit-B1-2t*: *k BIT True = 2 * k + 1*
by (*rule trans*, *rule Bit-B1*) *simp*

definition *bin-last* :: *int ⇒ bool*
where *bin-last w ⟷ w mod 2 = 1*

lemma *bin-last-odd*: *bin-last = odd*
by (*rule ext*) (*simp add*: *bin-last-def even-iff-mod-2-eq-zero*)

definition *bin-rest* :: *int ⇒ int*
where *bin-rest w = w div 2*

lemma *bin-rl-simp* [*simp*]: *bin-rest w BIT bin-last w = w*
unfolding *bin-rest-def bin-last-def Bit-def*
by (*cases w mod 2 = 0*) (*use div-mult-mod-eq* [*of w 2*] **in** *simp-all*)

lemma *bin-rest-BIT* [*simp*]: *bin-rest (x BIT b) = x*
unfolding *bin-rest-def Bit-def*
by (*cases b*) *simp-all*

lemma *bin-last-BIT* [*simp*]: *bin-last (x BIT b) = b*
unfolding *bin-last-def Bit-def*
by (*cases b*) *simp-all*

lemma *BIT-eq-iff* [*iff*]: *u BIT b = v BIT c ⟷ u = v ∧ b = c*
by (*auto simp*: *Bit-def*) *arith+*

lemma *BIT-bin-simps* [*simp*]:
numeral k BIT False = numeral (Num.Bit0 k)
numeral k BIT True = numeral (Num.Bit1 k)
(− numeral k) BIT False = − numeral (Num.Bit0 k)
(− numeral k) BIT True = − numeral (Num.BitM k)
unfolding *numeral.simps numeral-BitM*
by (*simp-all add*: *Bit-def del*: *arith-simps add-numeral-special diff-numeral-special*)

lemma *BIT-special-simps* [*simp*]:
shows *0 BIT False = 0*
and *0 BIT True = 1*
and *1 BIT False = 2*
and *1 BIT True = 3*
and *(− 1) BIT False = − 2*
and *(− 1) BIT True = − 1*
by (*simp-all add*: *Bit-def*)

lemma *Bit-eq-0-iff*: *w BIT b = 0 ⟷ w = 0 ∧ ¬ b*
by (*auto simp*: *Bit-def*) *arith*

lemma *Bit-eq-m1-iff*: $w\ BIT\ b = -1 \longleftrightarrow w = -1 \wedge b$
by (*auto simp*: *Bit-def*) *arith*

lemma *BitM-inc*: $Num.BitM\ (Num.inc\ w) = Num.Bit1\ w$
by (*induct w*) *simp-all*

lemma *expand-BIT*:
$numeral\ (Num.Bit0\ w) = numeral\ w\ BIT\ False$
$numeral\ (Num.Bit1\ w) = numeral\ w\ BIT\ True$
$-\ numeral\ (Num.Bit0\ w) = (-\ numeral\ w)\ BIT\ False$
$-\ numeral\ (Num.Bit1\ w) = (-\ numeral\ (w + Num.One))\ BIT\ True$
by (*simp-all add*: *add-One BitM-inc*)

lemma *bin-last-numeral-simps* [*simp*]:
$\neg\ bin\text{-}last\ 0$
$bin\text{-}last\ 1$
$bin\text{-}last\ (-\ 1)$
$bin\text{-}last\ Numeral1$
$\neg\ bin\text{-}last\ (numeral\ (Num.Bit0\ w))$
$bin\text{-}last\ (numeral\ (Num.Bit1\ w))$
$\neg\ bin\text{-}last\ (-\ numeral\ (Num.Bit0\ w))$
$bin\text{-}last\ (-\ numeral\ (Num.Bit1\ w))$
by (*simp-all add*: *bin-last-def zmod-zminus1-eq-if*) (*auto simp add*: *divmod-def*)

lemma *bin-rest-numeral-simps* [*simp*]:
$bin\text{-}rest\ 0 = 0$
$bin\text{-}rest\ 1 = 0$
$bin\text{-}rest\ (-\ 1) = -\ 1$
$bin\text{-}rest\ Numeral1 = 0$
$bin\text{-}rest\ (numeral\ (Num.Bit0\ w)) = numeral\ w$
$bin\text{-}rest\ (numeral\ (Num.Bit1\ w)) = numeral\ w$
$bin\text{-}rest\ (-\ numeral\ (Num.Bit0\ w)) = -\ numeral\ w$
$bin\text{-}rest\ (-\ numeral\ (Num.Bit1\ w)) = -\ numeral\ (w + Num.One)$
by (*simp-all add*: *bin-rest-def zdiv-zminus1-eq-if*) (*auto simp add*: *divmod-def*)

lemma *less-Bits*: $v\ BIT\ b < w\ BIT\ c \longleftrightarrow v < w \vee v \leq w \wedge \neg\ b \wedge c$
by (*auto simp*: *Bit-def*)

lemma *le-Bits*: $v\ BIT\ b \leq w\ BIT\ c \longleftrightarrow v < w \vee v \leq w \wedge (\neg\ b \vee c)$
by (*auto simp*: *Bit-def*)

lemma *pred-BIT-simps* [*simp*]:
$x\ BIT\ False - 1 = (x - 1)\ BIT\ True$
$x\ BIT\ True - 1 = x\ BIT\ False$
by (*simp-all add*: *Bit-B0-2t Bit-B1-2t*)

lemma *succ-BIT-simps* [*simp*]:
$x\ BIT\ False + 1 = x\ BIT\ True$

x BIT True + 1 = (x + 1) BIT False
by (*simp-all add*: *Bit-B0-2t Bit-B1-2t*)

lemma *add-BIT-simps* [*simp*]:
x BIT False + y BIT False = (x + y) BIT False
x BIT False + y BIT True = (x + y) BIT True
x BIT True + y BIT False = (x + y) BIT True
x BIT True + y BIT True = (x + y + 1) BIT False
by (*simp-all add*: *Bit-B0-2t Bit-B1-2t*)

lemma *mult-BIT-simps* [*simp*]:
*x BIT False * y = (x * y) BIT False*
*x * y BIT False = (x * y) BIT False*
*x BIT True * y = (x * y) BIT False + y*
by (*simp-all add*: *Bit-B0-2t Bit-B1-2t algebra-simps*)

lemma *B-mod-2'*: $X = 2 \Longrightarrow (w\ BIT\ True)\ mod\ X = 1 \wedge (w\ BIT\ False)\ mod\ X = 0$
by (*simp add*: *Bit-B0 Bit-B1*)

lemma *bin-ex-rl*: $\exists\, w\ b.\ w\ BIT\ b = bin$
by (*metis bin-rl-simp*)

lemma *bin-exhaust*:
assumes *that*: $\bigwedge x\ b.\ bin = x\ BIT\ b \Longrightarrow Q$
shows *Q*
apply (*insert bin-ex-rl* [*of bin*])
apply (*erule exE*)+
apply (*rule that*)
apply *force*
done

primrec *bin-nth*
where
Z: *bin-nth w 0* $\longleftrightarrow$ *bin-last w*
| *Suc*: *bin-nth w* (*Suc n*) $\longleftrightarrow$ *bin-nth* (*bin-rest w*) *n*

lemma *bin-abs-lem*: $bin = (w\ BIT\ b) \Longrightarrow bin \neq -1 \longrightarrow bin \neq 0 \longrightarrow nat\ |w| < nat\ |bin|$
apply *clarsimp*
apply (*unfold Bit-def*)
apply (*cases b*)
apply (*clarsimp*, *arith*)
apply (*clarsimp*, *arith*)
done

lemma *bin-induct*:
assumes *PPls*: *P 0*
and *PMin*: *P* (− *1*)

```
    and PBit: ⋀bin bit. P bin ⟹ P (bin BIT bit)
  shows P bin
   apply (rule-tac P=P and a=bin and f1=nat ∘ abs in wf-measure [THEN
wf-induct])
  apply (simp add: measure-def inv-image-def)
  apply (case-tac x rule: bin-exhaust)
  apply (frule bin-abs-lem)
  apply (auto simp add : PPls PMin PBit)
  done

lemma Bit-div2 [simp]: (w BIT b) div 2 = w
  unfolding bin-rest-def [symmetric] by (rule bin-rest-BIT)

lemma bin-nth-eq-iff: bin-nth x = bin-nth y ⟷ x = y
proof −
  have bin-nth-lem [rule-format]: ∀ y. bin-nth x = bin-nth y ⟶ x = y
    apply (induct x rule: bin-induct)
      apply safe
      apply (erule rev-mp)
      apply (induct-tac y rule: bin-induct)
        apply safe
        apply (drule-tac x=0 in fun-cong, force)
       apply (erule notE, rule ext, drule-tac x=Suc x in fun-cong, force)
      apply (drule-tac x=0 in fun-cong, force)
     apply (erule rev-mp)
     apply (induct-tac y rule: bin-induct)
       apply safe
       apply (drule-tac x=0 in fun-cong, force)
      apply (erule notE, rule ext, drule-tac x=Suc x in fun-cong, force)
     apply (metis Bit-eq-m1-iff Z bin-last-BIT)
    apply (case-tac y rule: bin-exhaust)
    apply clarify
    apply (erule allE)
    apply (erule impE)
     prefer 2
     apply (erule conjI)
     apply (drule-tac x=0 in fun-cong, force)
    apply (rule ext)
    apply (drule-tac x=Suc x for x in fun-cong, force)
    done
  show ?thesis
    by (auto elim: bin-nth-lem)
qed

lemmas bin-eqI = ext [THEN bin-nth-eq-iff [THEN iffD1]]

lemma bin-eq-iff: x = y ⟷ (∀ n. bin-nth x n = bin-nth y n)
  using bin-nth-eq-iff by auto
```

lemma *bin-nth-zero* [*simp*]: ¬ *bin-nth 0 n*
by (*induct n*) *auto*

lemma *bin-nth-1* [*simp*]: *bin-nth 1 n* ⟷ *n = 0*
by (*cases n*) *simp-all*

lemma *bin-nth-minus1* [*simp*]: *bin-nth* (− *1*) *n*
by (*induct n*) *auto*

lemma *bin-nth-0-BIT*: *bin-nth* (*w BIT b*) *0* ⟷ *b*
by *auto*

lemma *bin-nth-Suc-BIT*: *bin-nth* (*w BIT b*) (*Suc n*) = *bin-nth w n*
by *auto*

lemma *bin-nth-minus* [*simp*]: *0 < n* ⟹ *bin-nth* (*w BIT b*) *n* = *bin-nth w* (*n* − *1*)
by (*cases n*) *auto*

lemma *bin-nth-numeral*: *bin-rest x = y* ⟹ *bin-nth x* (*numeral n*) = *bin-nth y* (*pred-numeral n*)
by (*simp add*: *numeral-eq-Suc*)

lemmas *bin-nth-numeral-simps* [*simp*] =
bin-nth-numeral [*OF bin-rest-numeral-simps*(*2*)]
bin-nth-numeral [*OF bin-rest-numeral-simps*(*5*)]
bin-nth-numeral [*OF bin-rest-numeral-simps*(*6*)]
bin-nth-numeral [*OF bin-rest-numeral-simps*(*7*)]
bin-nth-numeral [*OF bin-rest-numeral-simps*(*8*)]

lemmas *bin-nth-simps* =
bin-nth.Z bin-nth.Suc bin-nth-zero bin-nth-minus1
bin-nth-numeral-simps

4.2 Truncating binary integers

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*
bin-sign (*w BIT b*) = *bin-sign w*
by (*simp-all add*: *bin-sign-def Bit-def*)

lemma *bin-sign-rest* [*simp*]: *bin-sign* (*bin-rest w*) = *bin-sign w*

by (*cases w rule*: *bin-exhaust*) *auto*

primrec *bintrunc* :: *nat* ⇒ *int* ⇒ *int*
where
Z : *bintrunc 0 bin* = *0*
| *Suc* : *bintrunc* (*Suc n*) *bin* = *bintrunc n* (*bin-rest bin*) *BIT* (*bin-last bin*)

primrec *sbintrunc* :: *nat* ⇒ *int* ⇒ *int*
where
Z : *sbintrunc 0 bin* = (*if bin-last bin then* −*1 else 0*)
| *Suc* : *sbintrunc* (*Suc n*) *bin* = *sbintrunc n* (*bin-rest bin*) *BIT* (*bin-last bin*)

lemma *sign-bintr*: *bin-sign* (*bintrunc n w*) = *0*
by (*induct n arbitrary*: *w*) *auto*

lemma *bintrunc-mod2p*: *bintrunc n w* = *w mod 2* ^ *n*
by (*induct n arbitrary*: *w*) (*auto simp add*: *bin-last-def bin-rest-def Bit-def zmod-zmult2-eq*)

lemma *sbintrunc-mod2p*: *sbintrunc n w* = (*w* + *2* ^ *n*) *mod 2* ^ (*Suc n*) − *2* ^ *n*
apply (*induct n arbitrary*: *w*)
apply (*auto simp add*: *bin-last-odd bin-rest-def Bit-def elim*!: *evenE oddE*)
apply (*smt pos-zmod-mult-2 zle2p*)
apply (*simp add*: *mult-mod-right*)
done

4.3 Simplifications for (s)bintrunc

lemma *bintrunc-n-0* [*simp*]: *bintrunc n 0* = *0*
by (*induct n*) *auto*

lemma *sbintrunc-n-0* [*simp*]: *sbintrunc n 0* = *0*
by (*induct n*) *auto*

lemma *sbintrunc-n-minus1* [*simp*]: *sbintrunc n* (− *1*) = −*1*
by (*induct n*) *auto*

lemma *bintrunc-Suc-numeral*:
bintrunc (*Suc n*) *1* = *1*
bintrunc (*Suc n*) (− *1*) = *bintrunc n* (− *1*) *BIT True*
bintrunc (*Suc n*) (*numeral* (*Num.Bit0 w*)) = *bintrunc n* (*numeral w*) *BIT False*
bintrunc (*Suc n*) (*numeral* (*Num.Bit1 w*)) = *bintrunc n* (*numeral w*) *BIT True*
bintrunc (*Suc n*) (− *numeral* (*Num.Bit0 w*)) = *bintrunc n* (− *numeral w*) *BIT False*
bintrunc (*Suc n*) (− *numeral* (*Num.Bit1 w*)) = *bintrunc n* (− *numeral* (*w* + *Num.One*)) *BIT True*
by *simp-all*

lemma *sbintrunc-0-numeral* [*simp*]:
sbintrunc 0 1 = −*1*

sbintrunc 0 (numeral (Num.Bit0 w)) = 0
sbintrunc 0 (numeral (Num.Bit1 w)) = −1
sbintrunc 0 (− numeral (Num.Bit0 w)) = 0
sbintrunc 0 (− numeral (Num.Bit1 w)) = −1
by *simp-all*

lemma *sbintrunc-Suc-numeral*:
sbintrunc (Suc n) 1 = 1
sbintrunc (Suc n) (numeral (Num.Bit0 w)) = sbintrunc n (numeral w) BIT False
sbintrunc (Suc n) (numeral (Num.Bit1 w)) = sbintrunc n (numeral w) BIT True
sbintrunc (Suc n) (− numeral (Num.Bit0 w)) = sbintrunc n (− numeral w) BIT False
sbintrunc (Suc n) (− numeral (Num.Bit1 w)) = sbintrunc n (− numeral (w + Num.One)) BIT True
by *simp-all*

lemma *bin-sign-lem*: *(bin-sign (sbintrunc n bin) = −1) = bin-nth bin n*
apply (*induct n arbitrary*: *bin*)
apply (*case-tac bin rule*: *bin-exhaust*, *case-tac b*, *auto*)
done

lemma *nth-bintr*: *bin-nth (bintrunc m w) n ⟷ n < m ∧ bin-nth w n*
apply (*induct n arbitrary*: *w m*)
apply (*case-tac m*, *auto*)[*1*]
apply (*case-tac m*, *auto*)[*1*]
done

lemma *nth-sbintr*: *bin-nth (sbintrunc m w) n = (if n < m then bin-nth w n else bin-nth w m)*
apply (*induct n arbitrary*: *w m*)
apply (*case-tac m*)
apply *simp-all*
apply (*case-tac m*)
apply *simp-all*
done

lemma *bin-nth-Bit*: *bin-nth (w BIT b) n ⟷ n = 0 ∧ b ∨ (∃ m. n = Suc m ∧ bin-nth w m)*
by (*cases n*) *auto*

lemma *bin-nth-Bit0*:
bin-nth (numeral (Num.Bit0 w)) n ⟷
(∃ m. n = Suc m ∧ bin-nth (numeral w) m)
using *bin-nth-Bit* [**where** *w=numeral w* **and** *b=False*] **by** *simp*

lemma *bin-nth-Bit1*:
bin-nth (numeral (Num.Bit1 w)) n ⟷
n = 0 ∨ (∃ m. n = Suc m ∧ bin-nth (numeral w) m)
using *bin-nth-Bit* [**where** *w=numeral w* **and** *b=True*] **by** *simp*

lemma *bintrunc-bintrunc-l*: $n \leq m \Longrightarrow$ *bintrunc m* (*bintrunc n w*) = *bintrunc n w*
by (*rule bin-eqI*) (*auto simp*: *nth-bintr*)

lemma *sbintrunc-sbintrunc-l*: $n \leq m \Longrightarrow$ *sbintrunc m* (*sbintrunc n w*) = *sbintrunc n w*
by (*rule bin-eqI*) (*auto simp*: *nth-sbintr*)

lemma *bintrunc-bintrunc-ge*: $n \leq m \Longrightarrow$ *bintrunc n* (*bintrunc m w*) = *bintrunc n w*
by (*rule bin-eqI*) (*auto simp*: *nth-bintr*)

lemma *bintrunc-bintrunc-min* [*simp*]: *bintrunc m* (*bintrunc n w*) = *bintrunc* (*min m n*) *w*
by (*rule bin-eqI*) (*auto simp*: *nth-bintr*)

lemma *sbintrunc-sbintrunc-min* [*simp*]: *sbintrunc m* (*sbintrunc n w*) = *sbintrunc* (*min m n*) *w*
by (*rule bin-eqI*) (*auto simp*: *nth-sbintr min.absorb1 min.absorb2*)

lemmas *bintrunc-Pls* =
bintrunc.Suc [**where** *bin=0*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *bintrunc-Min* [*simp*] =
bintrunc.Suc [**where** *bin=−1*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *bintrunc-BIT* [*simp*] =
bintrunc.Suc [**where** *bin=w BIT b*, *simplified bin-last-BIT bin-rest-BIT*] **for** *w b*

lemmas *bintrunc-Sucs* = *bintrunc-Pls bintrunc-Min bintrunc-BIT*
bintrunc-Suc-numeral

lemmas *sbintrunc-Suc-Pls* =
sbintrunc.Suc [**where** *bin=0*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *sbintrunc-Suc-Min* =
sbintrunc.Suc [**where** *bin=−1*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *sbintrunc-Suc-BIT* [*simp*] =
sbintrunc.Suc [**where** *bin=w BIT b*, *simplified bin-last-BIT bin-rest-BIT*] **for** *w b*

lemmas *sbintrunc-Sucs* = *sbintrunc-Suc-Pls sbintrunc-Suc-Min sbintrunc-Suc-BIT*
sbintrunc-Suc-numeral

lemmas *sbintrunc-Pls* =
sbintrunc.Z [**where** *bin=0*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *sbintrunc-Min* =
sbintrunc.Z [**where** *bin*=−*1*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]

lemmas *sbintrunc-0-BIT-B0* [*simp*] =
sbintrunc.Z [**where** *bin*=*w BIT False*, *simplified bin-last-numeral-simps bin-rest-numeral-simps*]
for *w*

lemmas *sbintrunc-0-BIT-B1* [*simp*] =
sbintrunc.Z [**where** *bin*=*w BIT True*, *simplified bin-last-BIT bin-rest-numeral-simps*]
for *w*

lemmas *sbintrunc-0-simps* =
sbintrunc-Pls sbintrunc-Min sbintrunc-0-BIT-B0 sbintrunc-0-BIT-B1

lemmas *bintrunc-simps* = *bintrunc.Z bintrunc-Sucs*
lemmas *sbintrunc-simps* = *sbintrunc-0-simps sbintrunc-Sucs*

lemma *bintrunc-minus*: *0* < *n* ⟹ *bintrunc* (*Suc* (*n* − *1*)) *w* = *bintrunc n w*
by *auto*

lemma *sbintrunc-minus*: *0* < *n* ⟹ *sbintrunc* (*Suc* (*n* − *1*)) *w* = *sbintrunc n w*
by *auto*

lemmas *bintrunc-minus-simps* =
bintrunc-Sucs [*THEN* [*2*] *bintrunc-minus* [*symmetric*, *THEN trans*]]
lemmas *sbintrunc-minus-simps* =
sbintrunc-Sucs [*THEN* [*2*] *sbintrunc-minus* [*symmetric*, *THEN trans*]]

lemmas *thobini1* = *arg-cong* [**where** *f* = λ*w*. *w BIT b*] **for** *b*

lemmas *bintrunc-BIT-I* = *trans* [*OF bintrunc-BIT thobini1*]
lemmas *bintrunc-Min-I* = *trans* [*OF bintrunc-Min thobini1*]

lemmas *bmsts* = *bintrunc-minus-simps*(*1*−*3*) [*THEN thobini1* [*THEN* [*2*] *trans*]]
lemmas *bintrunc-Pls-minus-I* = *bmsts*(*1*)
lemmas *bintrunc-Min-minus-I* = *bmsts*(*2*)
lemmas *bintrunc-BIT-minus-I* = *bmsts*(*3*)

lemma *bintrunc-Suc-lem*: *bintrunc* (*Suc n*) *x* = *y* ⟹ *m* = *Suc n* ⟹ *bintrunc*
m x = *y*
by *auto*

lemmas *bintrunc-Suc-Ialts* =
bintrunc-Min-I [*THEN bintrunc-Suc-lem*]
bintrunc-BIT-I [*THEN bintrunc-Suc-lem*]

lemmas *sbintrunc-BIT-I* = *trans* [*OF sbintrunc-Suc-BIT thobini1*]

lemmas *sbintrunc-Suc-Is* =

sbintrunc-Sucs(*1−3*) [*THEN thobini1* [*THEN* [*2*] *trans*]]

lemmas *sbintrunc-Suc-minus-Is* =
sbintrunc-minus-simps(*1−3*) [*THEN thobini1* [*THEN* [*2*] *trans*]]

lemma *sbintrunc-Suc-lem*: *sbintrunc* (*Suc n*) *x* = *y* ⟹ *m* = *Suc n* ⟹ *sbintrunc m x* = *y*
by *auto*

lemmas *sbintrunc-Suc-Ialts* =
sbintrunc-Suc-Is [*THEN sbintrunc-Suc-lem*]

lemma *sbintrunc-bintrunc-lt*: *m* > *n* ⟹ *sbintrunc n* (*bintrunc m w*) = *sbintrunc n w*
by (*rule bin-eqI*) (*auto simp*: *nth-sbintr nth-bintr*)

lemma *bintrunc-sbintrunc-le*: *m* ≤ *Suc n* ⟹ *bintrunc m* (*sbintrunc n w*) = *bintrunc m w*
apply (*rule bin-eqI*)
apply (*auto simp*: *nth-sbintr nth-bintr*)
apply (*subgoal-tac x=n*, *safe*, *arith+*)[*1*]
apply (*subgoal-tac x=n*, *safe*, *arith+*)[*1*]
done

lemmas *bintrunc-sbintrunc* [*simp*] = *order-refl* [*THEN bintrunc-sbintrunc-le*]
lemmas *sbintrunc-bintrunc* [*simp*] = *lessI* [*THEN sbintrunc-bintrunc-lt*]
lemmas *bintrunc-bintrunc* [*simp*] = *order-refl* [*THEN bintrunc-bintrunc-l*]
lemmas *sbintrunc-sbintrunc* [*simp*] = *order-refl* [*THEN sbintrunc-sbintrunc-l*]

lemma *bintrunc-sbintrunc'* [*simp*]: *0* < *n* ⟹ *bintrunc n* (*sbintrunc* (*n* − *1*) *w*) = *bintrunc n w*
by (*cases n*) (*auto simp del*: *bintrunc.Suc*)

lemma *sbintrunc-bintrunc'* [*simp*]: *0* < *n* ⟹ *sbintrunc* (*n* − *1*) (*bintrunc n w*) = *sbintrunc* (*n* − *1*) *w*
by (*cases n*) (*auto simp del*: *bintrunc.Suc*)

lemma *bin-sbin-eq-iff*: *bintrunc* (*Suc n*) *x* = *bintrunc* (*Suc n*) *y* ⟷ *sbintrunc n x* = *sbintrunc 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* ⟹ *bintrunc n x* = *bintrunc n y* ⟷ *sbintrunc* (*n* − *1*) *x* = *sbintrunc* (*n* − *1*) *y*

by (*cases n*) (*simp-all add*: *bin-sbin-eq-iff del*: *bintrunc.Suc*)

lemmas *bintrunc-sbintruncS0* [*simp*] = *bintrunc-sbintrunc'* [*unfolded One-nat-def*]
lemmas *sbintrunc-bintruncS0* [*simp*] = *sbintrunc-bintrunc'* [*unfolded One-nat-def*]

lemmas *bintrunc-bintrunc-l'* = *le-add1* [*THEN bintrunc-bintrunc-l*]
lemmas *sbintrunc-sbintrunc-l'* = *le-add1* [*THEN sbintrunc-sbintrunc-l*]

lemmas *nat-non0-gr* =
trans [*OF iszero-def* [*THEN Not-eq-iff* [*THEN iffD2*]] *refl*]

lemma *bintrunc-numeral*:
bintrunc (*numeral k*) *x* = *bintrunc* (*pred-numeral k*) (*bin-rest x*) *BIT bin-last x*
by (*simp add*: *numeral-eq-Suc*)

lemma *sbintrunc-numeral*:
sbintrunc (*numeral k*) *x* = *sbintrunc* (*pred-numeral k*) (*bin-rest x*) *BIT bin-last x*
by (*simp add*: *numeral-eq-Suc*)

lemma *bintrunc-numeral-simps* [*simp*]:
bintrunc (*numeral k*) (*numeral* (*Num.Bit0 w*)) = *bintrunc* (*pred-numeral k*) (*numeral w*) *BIT False*
bintrunc (*numeral k*) (*numeral* (*Num.Bit1 w*)) = *bintrunc* (*pred-numeral k*) (*numeral w*) *BIT True*
bintrunc (*numeral k*) (− *numeral* (*Num.Bit0 w*)) = *bintrunc* (*pred-numeral k*) (− *numeral w*) *BIT False*
bintrunc (*numeral k*) (− *numeral* (*Num.Bit1 w*)) =
bintrunc (*pred-numeral k*) (− *numeral* (*w* + *Num.One*)) *BIT True*
bintrunc (*numeral k*) *1* = *1*
by (*simp-all add*: *bintrunc-numeral*)

lemma *sbintrunc-numeral-simps* [*simp*]:
sbintrunc (*numeral k*) (*numeral* (*Num.Bit0 w*)) = *sbintrunc* (*pred-numeral k*) (*numeral w*) *BIT False*
sbintrunc (*numeral k*) (*numeral* (*Num.Bit1 w*)) = *sbintrunc* (*pred-numeral k*) (*numeral w*) *BIT True*
sbintrunc (*numeral k*) (− *numeral* (*Num.Bit0 w*)) =
sbintrunc (*pred-numeral k*) (− *numeral w*) *BIT False*
sbintrunc (*numeral k*) (− *numeral* (*Num.Bit1 w*)) =
sbintrunc (*pred-numeral k*) (− *numeral* (*w* + *Num.One*)) *BIT True*
sbintrunc (*numeral k*) *1* = *1*
by (*simp-all add*: *sbintrunc-numeral*)

lemma *no-bintr-alt1*: *bintrunc n* = (λ*w*. *w mod 2* ^ *n* :: *int*)
by (*rule ext*) (*rule bintrunc-mod2p*)

lemma *range-bintrunc*: *range* (*bintrunc n*) = {*i*. *0* ≤ *i* ∧ *i* < *2* ^ *n*}
apply (*unfold no-bintr-alt1*)
apply (*auto simp add*: *image-iff*)
apply (*rule exI*)
apply (*auto intro*: *int-mod-lem* [*THEN iffD1*, *symmetric*])
done

lemma *no-sbintr-alt2*: *sbintrunc n* = (λ*w*. (*w* + *2* ^ *n*) *mod* *2* ^ *Suc n* − *2* ^ *n* :: *int*)
by (*rule ext*) (*simp add* : *sbintrunc-mod2p*)

lemma *range-sbintrunc*: *range* (*sbintrunc n*) = {*i*. − (*2* ^ *n*) ≤ *i* ∧ *i* < *2* ^ *n*}
apply (*unfold no-sbintr-alt2*)
apply (*auto simp add*: *image-iff eq-diff-eq*)
apply (*rule exI*)
apply (*auto intro*: *int-mod-lem* [*THEN iffD1*, *symmetric*])
done

lemma *sb-inc-lem*: *a* + *2*^*k* < *0* ⟹ *a* + *2*^*k* + *2*^(*Suc k*) ≤ (*a* + *2*^*k*) *mod* *2*^(*Suc k*)
for *a* :: *int*
apply (*erule int-mod-ge'* [**where** *n* = *2* ^ (*Suc k*) **and** *b* = *a* + *2* ^ *k*, *simplified zless2p*])
apply (*rule TrueI*)
done

lemma *sb-inc-lem'*: *a* < − (*2*^*k*) ⟹ *a* + *2*^*k* + *2*^(*Suc k*) ≤ (*a* + *2*^*k*) *mod* *2*^(*Suc k*)
for *a* :: *int*
by (*rule sb-inc-lem*) *simp*

lemma *sbintrunc-inc*: *x* < − (*2*^*n*) ⟹ *x* + *2*^(*Suc n*) ≤ *sbintrunc n x*
unfolding *no-sbintr-alt2* **by** (*drule sb-inc-lem'*) *simp*

lemma *sb-dec-lem*: *0* ≤ − (*2* ^ *k*) + *a* ⟹ (*a* + *2* ^ *k*) *mod* (*2* ∗ *2* ^ *k*) ≤ − (*2* ^ *k*) + *a*
for *a* :: *int*
using *int-mod-le'*[**where** *n* = *2* ^ (*Suc k*) **and** *b* = *a* + *2* ^ *k*] **by** *simp*

lemma *sb-dec-lem'*: *2* ^ *k* ≤ *a* ⟹ (*a* + *2* ^ *k*) *mod* (*2* ∗ *2* ^ *k*) ≤ − (*2* ^ *k*) + *a*
for *a* :: *int*
by (*rule sb-dec-lem*) *simp*

lemma *sbintrunc-dec*: *x* ≥ (*2* ^ *n*) ⟹ *x* − *2* ^ (*Suc n*) >= *sbintrunc n x*
unfolding *no-sbintr-alt2* **by** (*drule sb-dec-lem'*) *simp*

lemmas *m2pths* = *pos-mod-sign pos-mod-bound* [*OF zless2p*]

lemma *bintr-ge0*: *0* ≤ *bintrunc n w*

by (*simp add*: *bintrunc-mod2p*)

lemma *bintr-lt2p*: *bintrunc n w* < *2* ^ *n*
by (*simp add*: *bintrunc-mod2p*)

lemma *bintr-Min*: *bintrunc n* (− *1*) = *2* ^ *n* − *1*
by (*simp add*: *bintrunc-mod2p m1mod2k*)

lemma *sbintr-ge*: − (*2* ^ *n*) ≤ *sbintrunc n w*
by (*simp add*: *sbintrunc-mod2p*)

lemma *sbintr-lt*: *sbintrunc n w* < *2* ^ *n*
by (*simp add*: *sbintrunc-mod2p*)

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*: *bin-rest* (*bintrunc n bin*) = *bintrunc* (*n* − *1*) (*bin-rest bin*)
by (*induct n arbitrary*: *bin*) *auto*

lemma *bin-rest-power-trunc*:
(*bin-rest* ^^ *k*) (*bintrunc n bin*) = *bintrunc* (*n* − *k*) ((*bin-rest* ^^ *k*) *bin*)
by (*induct k*) (*auto simp*: *bin-rest-trunc*)

lemma *bin-rest-trunc-i*: *bintrunc n* (*bin-rest bin*) = *bin-rest* (*bintrunc* (*Suc n*) *bin*)
by *auto*

lemma *bin-rest-strunc*: *bin-rest* (*sbintrunc* (*Suc n*) *bin*) = *sbintrunc n* (*bin-rest bin*)
by (*induct n arbitrary*: *bin*) *auto*

lemma *bintrunc-rest* [*simp*]: *bintrunc n* (*bin-rest* (*bintrunc n bin*)) = *bin-rest* (*bintrunc n bin*)
apply (*induct n arbitrary*: *bin*)
apply *simp*
apply (*case-tac bin rule*: *bin-exhaust*)
apply (*auto simp*: *bintrunc-bintrunc-l*)
done

lemma *sbintrunc-rest* [*simp*]: *sbintrunc n* (*bin-rest* (*sbintrunc n bin*)) = *bin-rest* (*sbintrunc n bin*)
apply (*induct n arbitrary*: *bin*)
apply *simp*

apply (*case-tac bin rule*: *bin-exhaust*)
apply (*auto simp*: *bintrunc-bintrunc-l split*: *bool.splits*)
done

lemma *bintrunc-rest'*: *bintrunc n ∘ bin-rest ∘ bintrunc n = bin-rest ∘ bintrunc n*
by (*rule ext*) *auto*

lemma *sbintrunc-rest'*: *sbintrunc n ∘ bin-rest ∘ sbintrunc n = bin-rest ∘ sbintrunc 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*]

4.4 Splitting and concatenation

primrec *bin-split* :: *nat ⇒ int ⇒ int × int*
where
Z: *bin-split 0 w = (w, 0)*
| *Suc*: *bin-split* (*Suc n*) *w* =
(*let* (*w1*, *w2*) = *bin-split n* (*bin-rest w*)
in (*w1*, *w2 BIT bin-last w*))

lemma [*code*]:
bin-split (*Suc n*) *w* = (*let* (*w1*, *w2*) = *bin-split n* (*bin-rest w*) *in* (*w1*, *w2 BIT bin-last w*))
bin-split 0 w = (*w*, *0*)
by *simp-all*

primrec *bin-cat* :: *int ⇒ nat ⇒ int ⇒ int*
where
Z: *bin-cat w 0 v = w*
| *Suc*: *bin-cat w* (*Suc n*) *v = bin-cat w n* (*bin-rest v*) *BIT bin-last v*

end

5 Bitwise Operations on Binary Integers

theory *Bits-Int*
imports *Bits Bit-Representation*
begin

5.1 Logical operations

bit-wise logical operations on the int type

instantiation *int* :: *bit*
begin

definition *int-not-def*: *bitNOT* = (λ*x*::*int*. − *x* − *1*)

function *bitAND-int*
where *bitAND-int x y* =
(*if x* = *0 then 0 else if x* = −*1 then y*
else (*bin-rest x AND bin-rest y*) *BIT* (*bin-last x* ∧ *bin-last y*))
by *pat-completeness simp*

termination
by (*relation measure* (*nat* ∘ *abs* ∘ *fst*), *simp-all add*: *bin-rest-def*)

declare *bitAND-int.simps* [*simp del*]

definition *int-or-def*:
bitOR = (λ*x y*::*int*. *NOT* (*NOT x AND NOT y*))

definition *int-xor-def*:
bitXOR = (λ*x y*::*int*. (*x AND NOT y*) *OR* (*NOT x AND y*))

instance ..

end

5.1.1 Basic simplification rules

lemma *int-not-BIT* [*simp*]:
NOT (*w BIT b*) = (*NOT w*) *BIT* (¬ *b*)
unfolding *int-not-def Bit-def* **by** (*cases b*, *simp-all*)

lemma *int-not-simps* [*simp*]:
NOT (*0*::*int*) = −*1*
NOT (*1*::*int*) = −*2*
NOT (− *1*::*int*) = *0*
NOT (*numeral w*::*int*) = − *numeral* (*w* + *Num.One*)
NOT (− *numeral* (*Num.Bit0 w*)::*int*) = *numeral* (*Num.BitM w*)
NOT (− *numeral* (*Num.Bit1 w*)::*int*) = *numeral* (*Num.Bit0 w*)
unfolding *int-not-def* **by** *simp-all*

lemma *int-not-not* [*simp*]: $NOT\ (NOT\ (x::int)) = x$
unfolding *int-not-def* **by** *simp*

lemma *int-and-0* [*simp*]: $(0::int)\ AND\ x = 0$
by (*simp add*: *bitAND-int.simps*)

lemma *int-and-m1* [*simp*]: $(-1::int)\ AND\ x = x$
by (*simp add*: *bitAND-int.simps*)

lemma *int-and-Bits* [*simp*]:
$(x\ BIT\ b)\ AND\ (y\ BIT\ c) = (x\ AND\ y)\ BIT\ (b \wedge c)$
by (*subst bitAND-int.simps*, *simp add*: *Bit-eq-0-iff Bit-eq-m1-iff*)

lemma *int-or-zero* [*simp*]: $(0::int)\ OR\ x = x$
unfolding *int-or-def* **by** *simp*

lemma *int-or-minus1* [*simp*]: $(-1::int)\ OR\ x = -1$
unfolding *int-or-def* **by** *simp*

lemma *int-or-Bits* [*simp*]:
$(x\ BIT\ b)\ OR\ (y\ BIT\ c) = (x\ OR\ y)\ BIT\ (b \vee c)$
unfolding *int-or-def* **by** *simp*

lemma *int-xor-zero* [*simp*]: $(0::int)\ XOR\ x = x$
unfolding *int-xor-def* **by** *simp*

lemma *int-xor-Bits* [*simp*]:
$(x\ BIT\ b)\ XOR\ (y\ BIT\ c) = (x\ XOR\ y)\ BIT\ ((b \vee c) \wedge \neg\ (b \wedge c))$
unfolding *int-xor-def* **by** *auto*

5.1.2 Binary destructors

lemma *bin-rest-NOT* [*simp*]: $bin\text{-}rest\ (NOT\ x) = NOT\ (bin\text{-}rest\ x)$
by (*cases x rule*: *bin-exhaust*, *simp*)

lemma *bin-last-NOT* [*simp*]: $bin\text{-}last\ (NOT\ x) \longleftrightarrow \neg\ bin\text{-}last\ x$
by (*cases x rule*: *bin-exhaust*, *simp*)

lemma *bin-rest-AND* [*simp*]: $bin\text{-}rest\ (x\ AND\ y) = bin\text{-}rest\ x\ AND\ bin\text{-}rest\ y$
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-last-AND* [*simp*]: $bin\text{-}last\ (x\ AND\ y) \longleftrightarrow bin\text{-}last\ x \wedge bin\text{-}last\ y$
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-rest-OR* [*simp*]: $bin\text{-}rest\ (x\ OR\ y) = bin\text{-}rest\ x\ OR\ bin\text{-}rest\ y$
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-last-OR* [*simp*]: $bin\text{-}last\ (x\ OR\ y) \longleftrightarrow bin\text{-}last\ x \vee bin\text{-}last\ y$

by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-rest-XOR* [*simp*]: *bin-rest* (*x XOR y*) = *bin-rest x XOR bin-rest y*
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-last-XOR* [*simp*]: *bin-last* (*x XOR y*) ⟷ (*bin-last x* ∨ *bin-last y*) ∧ ¬ (*bin-last x* ∧ *bin-last y*)
by (*cases x rule*: *bin-exhaust*, *cases y rule*: *bin-exhaust*, *simp*)

lemma *bin-nth-ops*:
!!*x y*. *bin-nth* (*x AND y*) *n* = (*bin-nth x n* & *bin-nth y n*)
!!*x y*. *bin-nth* (*x OR y*) *n* = (*bin-nth x n* | *bin-nth y n*)
!!*x y*. *bin-nth* (*x XOR y*) *n* = (*bin-nth x n* ~= *bin-nth y n*)
!!*x*. *bin-nth* (*NOT x*) *n* = (~ *bin-nth x n*)
by (*induct n*) *auto*

5.1.3 Derived properties

lemma *int-xor-minus1* [*simp*]: (−1::*int*) *XOR x* = *NOT x*
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-xor-extra-simps* [*simp*]:
w XOR (0::*int*) = *w*
w XOR (−1::*int*) = *NOT w*
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-or-extra-simps* [*simp*]:
w OR (0::*int*) = *w*
w OR (−1::*int*) = −1
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-and-extra-simps* [*simp*]:
w AND (0::*int*) = 0
w AND (−1::*int*) = *w*
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bin-ops-comm*:
shows
int-and-comm: !!*y*::*int*. *x AND y* = *y AND x* **and**
int-or-comm: !!*y*::*int*. *x OR y* = *y OR x* **and**
int-xor-comm: !!*y*::*int*. *x XOR y* = *y XOR x*
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bin-ops-same* [*simp*]:
(*x*::*int*) *AND x* = *x*
(*x*::*int*) *OR x* = *x*
(*x*::*int*) *XOR x* = 0
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemmas *bin-log-esimps* =
int-and-extra-simps int-or-extra-simps int-xor-extra-simps
int-and-0 int-and-m1 int-or-zero int-or-minus1 int-xor-zero int-xor-minus1

lemma *bbw-ao-absorb*:
!!*y*::*int*. *x AND* (*y OR x*) = *x* & *x OR* (*y AND x*) = *x*
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*::*int*)
(*y OR x*) *AND x* = *x* ∧ *x OR* (*x AND y*) = (*x*::*int*)
(*x OR y*) *AND x* = *x* ∧ (*x AND y*) *OR x* = (*x*::*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*:
!!*y*::*int*. (*NOT x*) *XOR y* = *NOT* (*x XOR y*) &
x XOR (*NOT y*) = *NOT* (*x XOR y*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-and-assoc*:
(*x AND y*) *AND* (*z*::*int*) = *x AND* (*y AND z*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-or-assoc*:
(*x OR y*) *OR* (*z*::*int*) = *x OR* (*y OR z*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *int-xor-assoc*:
(*x XOR y*) *XOR* (*z*::*int*) = *x XOR* (*y XOR z*)
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*::*int*) *AND* (*x AND z*) = *x AND* (*y AND z*)
(*y*::*int*) *OR* (*x OR z*) = *x OR* (*y OR z*)
(*y*::*int*) *XOR* (*x XOR z*) = *x XOR* (*y XOR z*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bbw-not-dist*:
!!*y*::*int*. *NOT* (*x OR y*) = (*NOT x*) *AND* (*NOT y*)
!!*y*::*int*. *NOT* (*x AND y*) = (*NOT x*) *OR* (*NOT y*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bbw-oa-dist*:
!!*y z*::*int*. (*x AND y*) *OR z* =
(*x OR z*) *AND* (*y OR z*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

lemma *bbw-ao-dist*:
!!*y z*::*int*. (*x OR y*) *AND z* =
(*x AND z*) *OR* (*y AND z*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops*)

5.1.4 Simplification with numerals

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

lemma *bin-rl-eqI*: ⟦*bin-rest x* = *bin-rest y*; *bin-last x* = *bin-last y*⟧ ⟹ *x* = *y*
by (*metis* (*mono-tags*) *BIT-eq-iff bin-ex-rl bin-last-BIT bin-rest-BIT*)

lemma *bin-rest-neg-numeral-BitM* [*simp*]:
bin-rest (− *numeral* (*Num.BitM w*)) = − *numeral w*
by (*simp only*: *BIT-bin-simps* [*symmetric*] *bin-rest-BIT*)

lemma *bin-last-neg-numeral-BitM* [*simp*]:
bin-last (− *numeral* (*Num.BitM w*))
by (*simp only*: *BIT-bin-simps* [*symmetric*] *bin-last-BIT*)

FIXME: The rule sets below are very large (24 rules for each operator). Is there a simpler way to do this?

lemma *int-and-numerals* [*simp*]:
numeral (*Num.Bit0 x*) *AND numeral* (*Num.Bit0 y*) = (*numeral x AND numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *AND numeral* (*Num.Bit1 y*) = (*numeral x AND numeral y*) *BIT False*
numeral (*Num.Bit1 x*) *AND numeral* (*Num.Bit0 y*) = (*numeral x AND numeral y*) *BIT False*
numeral (*Num.Bit1 x*) *AND numeral* (*Num.Bit1 y*) = (*numeral x AND numeral y*) *BIT True*
numeral (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit0 y*) = (*numeral x AND* − *numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit1 y*) = (*numeral x AND* − *numeral* (*y* + *Num.One*)) *BIT False*
numeral (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit0 y*) = (*numeral x AND* − *numeral y*) *BIT False*
numeral (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit1 y*) = (*numeral x AND* − *numeral* (*y* + *Num.One*)) *BIT True*
− *numeral* (*Num.Bit0 x*) *AND numeral* (*Num.Bit0 y*) = (− *numeral x AND numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *AND numeral* (*Num.Bit1 y*) = (− *numeral x AND numeral y*) *BIT False*

− numeral (Num.Bit1 x) AND numeral (Num.Bit0 y) = (− numeral (x + Num.One) AND numeral y) BIT False
− numeral (Num.Bit1 x) AND numeral (Num.Bit1 y) = (− numeral (x + Num.One) AND numeral y) BIT True
− numeral (Num.Bit0 x) AND − numeral (Num.Bit0 y) = (− numeral x AND − numeral y) BIT False
− numeral (Num.Bit0 x) AND − numeral (Num.Bit1 y) = (− numeral x AND − numeral (y + Num.One)) BIT False
− numeral (Num.Bit1 x) AND − numeral (Num.Bit0 y) = (− numeral (x + Num.One) AND − numeral y) BIT False
− numeral (Num.Bit1 x) AND − numeral (Num.Bit1 y) = (− numeral (x + Num.One) AND − numeral (y + Num.One)) BIT True
(1::int) AND numeral (Num.Bit0 y) = 0
(1::int) AND numeral (Num.Bit1 y) = 1
(1::int) AND − numeral (Num.Bit0 y) = 0
(1::int) AND − numeral (Num.Bit1 y) = 1
numeral (Num.Bit0 x) AND (1::int) = 0
numeral (Num.Bit1 x) AND (1::int) = 1
− numeral (Num.Bit0 x) AND (1::int) = 0
− numeral (Num.Bit1 x) AND (1::int) = 1
by (*rule bin-rl-eqI*, *simp*, *simp*)+

lemma *int-or-numerals* [*simp*]:
numeral (Num.Bit0 x) OR numeral (Num.Bit0 y) = (numeral x OR numeral y) BIT False
numeral (Num.Bit0 x) OR numeral (Num.Bit1 y) = (numeral x OR numeral y) BIT True
numeral (Num.Bit1 x) OR numeral (Num.Bit0 y) = (numeral x OR numeral y) BIT True
numeral (Num.Bit1 x) OR numeral (Num.Bit1 y) = (numeral x OR numeral y) BIT True
numeral (Num.Bit0 x) OR − numeral (Num.Bit0 y) = (numeral x OR − numeral y) BIT False
numeral (Num.Bit0 x) OR − numeral (Num.Bit1 y) = (numeral x OR − numeral (y + Num.One)) BIT True
numeral (Num.Bit1 x) OR − numeral (Num.Bit0 y) = (numeral x OR − numeral y) BIT True
numeral (Num.Bit1 x) OR − numeral (Num.Bit1 y) = (numeral x OR − numeral (y + Num.One)) BIT True
− numeral (Num.Bit0 x) OR numeral (Num.Bit0 y) = (− numeral x OR numeral y) BIT False
− numeral (Num.Bit0 x) OR numeral (Num.Bit1 y) = (− numeral x OR numeral y) BIT True
− numeral (Num.Bit1 x) OR numeral (Num.Bit0 y) = (− numeral (x + Num.One) OR numeral y) BIT True
− numeral (Num.Bit1 x) OR numeral (Num.Bit1 y) = (− numeral (x + Num.One) OR numeral y) BIT True
− numeral (Num.Bit0 x) OR − numeral (Num.Bit0 y) = (− numeral x OR − numeral y) BIT False

− numeral (Num.Bit0 x) OR − numeral (Num.Bit1 y) = (− numeral x OR − numeral (y + Num.One)) BIT True
− numeral (Num.Bit1 x) OR − numeral (Num.Bit0 y) = (− numeral (x + Num.One) OR − numeral y) BIT True
− numeral (Num.Bit1 x) OR − numeral (Num.Bit1 y) = (− numeral (x + Num.One) OR − numeral (y + Num.One)) BIT True
(1::int) OR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)
(1::int) OR numeral (Num.Bit1 y) = numeral (Num.Bit1 y)
(1::int) OR − numeral (Num.Bit0 y) = − numeral (Num.BitM y)
(1::int) OR − numeral (Num.Bit1 y) = − numeral (Num.Bit1 y)
numeral (Num.Bit0 x) OR (1::int) = numeral (Num.Bit1 x)
numeral (Num.Bit1 x) OR (1::int) = numeral (Num.Bit1 x)
− numeral (Num.Bit0 x) OR (1::int) = − numeral (Num.BitM x)
− numeral (Num.Bit1 x) OR (1::int) = − numeral (Num.Bit1 x)
by (*rule bin-rl-eqI*, *simp*, *simp*)+

lemma *int-xor-numerals* [*simp*]:
numeral (Num.Bit0 x) XOR numeral (Num.Bit0 y) = (numeral x XOR numeral y) BIT False
numeral (Num.Bit0 x) XOR numeral (Num.Bit1 y) = (numeral x XOR numeral y) BIT True
numeral (Num.Bit1 x) XOR numeral (Num.Bit0 y) = (numeral x XOR numeral y) BIT True
numeral (Num.Bit1 x) XOR numeral (Num.Bit1 y) = (numeral x XOR numeral y) BIT False
numeral (Num.Bit0 x) XOR − numeral (Num.Bit0 y) = (numeral x XOR − numeral y) BIT False
numeral (Num.Bit0 x) XOR − numeral (Num.Bit1 y) = (numeral x XOR − numeral (y + Num.One)) BIT True
numeral (Num.Bit1 x) XOR − numeral (Num.Bit0 y) = (numeral x XOR − numeral y) BIT True
numeral (Num.Bit1 x) XOR − numeral (Num.Bit1 y) = (numeral x XOR − numeral (y + Num.One)) BIT False
− numeral (Num.Bit0 x) XOR numeral (Num.Bit0 y) = (− numeral x XOR numeral y) BIT False
− numeral (Num.Bit0 x) XOR numeral (Num.Bit1 y) = (− numeral x XOR numeral y) BIT True
− numeral (Num.Bit1 x) XOR numeral (Num.Bit0 y) = (− numeral (x + Num.One) XOR numeral y) BIT True
− numeral (Num.Bit1 x) XOR numeral (Num.Bit1 y) = (− numeral (x + Num.One) XOR numeral y) BIT False
− numeral (Num.Bit0 x) XOR − numeral (Num.Bit0 y) = (− numeral x XOR − numeral y) BIT False
− numeral (Num.Bit0 x) XOR − numeral (Num.Bit1 y) = (− numeral x XOR − numeral (y + Num.One)) BIT True
− numeral (Num.Bit1 x) XOR − numeral (Num.Bit0 y) = (− numeral (x + Num.One) XOR − numeral y) BIT True
− numeral (Num.Bit1 x) XOR − numeral (Num.Bit1 y) = (− numeral (x + Num.One) XOR − numeral (y + Num.One)) BIT False

(1::int) XOR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)
(1::int) XOR numeral (Num.Bit1 y) = numeral (Num.Bit0 y)
(1::int) XOR − numeral (Num.Bit0 y) = − numeral (Num.BitM y)
(1::int) XOR − numeral (Num.Bit1 y) = − numeral (Num.Bit0 (y + Num.One))
numeral (Num.Bit0 x) XOR (1::int) = numeral (Num.Bit1 x)
numeral (Num.Bit1 x) XOR (1::int) = numeral (Num.Bit0 x)
− numeral (Num.Bit0 x) XOR (1::int) = − numeral (Num.BitM x)
− numeral (Num.Bit1 x) XOR (1::int) = − numeral (Num.Bit0 (x + Num.One))
by (*rule bin-rl-eqI*, *simp*, *simp*)+

5.1.5 Interactions with arithmetic

lemma *plus-and-or* [*rule-format*]:
ALL y::int. (x AND y) + (x OR y) = x + y
apply (*induct x rule*: *bin-induct*)
apply *clarsimp*
apply *clarsimp*
apply *clarsimp*
apply (*case-tac y rule*: *bin-exhaust*)
apply *clarsimp*
apply (*unfold Bit-def*)
apply *clarsimp*
apply (*erule-tac x = x* **in** *allE*)
apply *simp*
done

lemma *le-int-or*:
bin-sign (y::int) = 0 ==> x <= x OR y
apply (*induct y arbitrary*: *x rule*: *bin-induct*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac x rule*: *bin-exhaust*)
apply (*case-tac b*)
apply (*case-tac* [!] *bit*)
apply (*auto simp*: *le-Bits*)
done

lemmas *int-and-le* =
xtrans(3) [*OF bbw-ao-absorbs* (*2*) [*THEN conjunct2*, *symmetric*] *le-int-or*]

lemma *bin-add-not*: *x + NOT x = (−1::int)*
apply (*induct x rule*: *bin-induct*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac bit*, *auto*)
done

5.1.6 Truncating results of bit-wise operations

lemma *bin-trunc-ao*:
!!*x y*. (*bintrunc n x*) *AND* (*bintrunc n y*) = *bintrunc n* (*x AND y*)
!!*x y*. (*bintrunc n x*) *OR* (*bintrunc n y*) = *bintrunc n* (*x OR y*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops nth-bintr*)

lemma *bin-trunc-xor*:
!!*x y*. *bintrunc n* (*bintrunc n x XOR bintrunc n y*) =
bintrunc n (*x XOR y*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops nth-bintr*)

lemma *bin-trunc-not*:
!!*x*. *bintrunc n* (*NOT* (*bintrunc n x*)) = *bintrunc n* (*NOT x*)
by (*auto simp add*: *bin-eq-iff bin-nth-ops nth-bintr*)

lemma *bintr-bintr-i*:
x = *bintrunc n y* ==> *bintrunc n x* = *bintrunc 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*]

5.2 Setting and clearing bits

primrec
bin-sc :: *nat* => *bool* => *int* => *int*
where
Z: *bin-sc 0 b w* = *bin-rest w BIT b*
| *Suc*: *bin-sc* (*Suc n*) *b w* = *bin-sc n b* (*bin-rest w*) *BIT bin-last w*

lemma *bin-nth-sc* [*simp*]:
bin-nth (*bin-sc n b w*) *n* ⟷ *b*
by (*induct n arbitrary*: *w*) *auto*

lemma *bin-sc-sc-same* [*simp*]:
bin-sc n c (*bin-sc n b w*) = *bin-sc n c w*
by (*induct n arbitrary*: *w*) *auto*

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*:
bin-nth (*bin-sc n b w*) *m* = (*if m* = *n then b else bin-nth w m*)

by (*induct n arbitrary*: *w m*) (*case-tac* [!] *m*, *auto*)

lemma *bin-sc-nth* [*simp*]:
(*bin-sc n* (*bin-nth w n*) *w*) = *w*
by (*induct n arbitrary*: *w*) *auto*

lemma *bin-sign-sc* [*simp*]:
bin-sign (*bin-sc n b w*) = *bin-sign w*
by (*induct n arbitrary*: *w*) *auto*

lemma *bin-sc-bintr* [*simp*]:
bintrunc m (*bin-sc n x* (*bintrunc m* (*w*))) = *bintrunc m* (*bin-sc n x w*)
apply (*induct n arbitrary*: *w m*)
apply (*case-tac* [!] *w rule*: *bin-exhaust*)
apply (*case-tac* [!] *m*, *auto*)
done

lemma *bin-clr-le*:
bin-sc n False w <= *w*
apply (*induct n arbitrary*: *w*)
apply (*case-tac* [!] *w rule*: *bin-exhaust*)
apply (*auto simp*: *le-Bits*)
done

lemma *bin-set-ge*:
bin-sc n True w >= *w*
apply (*induct n arbitrary*: *w*)
apply (*case-tac* [!] *w rule*: *bin-exhaust*)
apply (*auto simp*: *le-Bits*)
done

lemma *bintr-bin-clr-le*:
bintrunc n (*bin-sc m False w*) <= *bintrunc n w*
apply (*induct n arbitrary*: *w m*)
apply *simp*
apply (*case-tac w rule*: *bin-exhaust*)
apply (*case-tac m*)
apply (*auto simp*: *le-Bits*)
done

lemma *bintr-bin-set-ge*:
bintrunc n (*bin-sc m True w*) >= *bintrunc n w*
apply (*induct n arbitrary*: *w m*)
apply *simp*
apply (*case-tac w rule*: *bin-exhaust*)
apply (*case-tac m*)
apply (*auto simp*: *le-Bits*)
done

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.Z bin-sc.Suc bin-sc-TM bin-sc-FP*

lemma *bin-sc-minus*:
0 < n ==> bin-sc (Suc (n − 1)) b w = bin-sc n b w
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 =
bin-sc (pred-numeral k) b (bin-rest w) BIT bin-last w
by (*simp add*: *numeral-eq-Suc*)

5.3 Splitting and concatenation

definition *bin-rcat* :: *nat ⇒ int list ⇒ int*
where
bin-rcat n = foldl (λu v. bin-cat u n v) 0

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-sign-cat*:
bin-sign (*bin-cat x n y*) = *bin-sign x*
by (*induct n arbitrary*: *y*) *auto*

lemma *bin-cat-Suc-Bit*:
bin-cat w (*Suc n*) (*v BIT b*) = *bin-cat w n v BIT b*
by *auto*

lemma *bin-nth-cat*:
bin-nth (*bin-cat x k y*) *n* =
(*if n* < *k then bin-nth y n else bin-nth x* (*n* − *k*))
apply (*induct k arbitrary*: *n y*)
apply *clarsimp*
apply (*case-tac n*, *auto*)
done

lemma *bin-nth-split*:
bin-split n c = (*a*, *b*) ==>
(*ALL k*. *bin-nth a k* = *bin-nth c* (*n* + *k*)) &
(*ALL k*. *bin-nth b k* = (*k* < *n* & *bin-nth c k*))
apply (*induct n arbitrary*: *b c*)
apply *clarsimp*
apply (*clarsimp simp*: *Let-def split*: *prod.split-asm*)
apply (*case-tac k*)
apply *auto*
done

lemma *bin-cat-assoc*:
bin-cat (*bin-cat x m y*) *n z* = *bin-cat x* (*m* + *n*) (*bin-cat y n z*)
by (*induct n arbitrary*: *z*) *auto*

lemma *bin-cat-assoc-sym*:
bin-cat x m (*bin-cat y n z*) = *bin-cat* (*bin-cat x* (*m* − *n*) *y*) (*min m n*) *z*
apply (*induct n arbitrary*: *z m*, *clarsimp*)
apply (*case-tac m*, *auto*)
done

lemma *bin-cat-zero* [*simp*]: *bin-cat 0 n w* = *bintrunc n w*
by (*induct n arbitrary*: *w*) *auto*

lemma *bintr-cat1*:
bintrunc (*k* + *n*) (*bin-cat a n b*) = *bin-cat* (*bintrunc k a*) *n b*
by (*induct n arbitrary*: *b*) *auto*

lemma *bintr-cat*: *bintrunc m* (*bin-cat a n b*) =
bin-cat (*bintrunc* (*m* − *n*) *a*) *n* (*bintrunc* (*min m n*) *b*)
by (*rule bin-eqI*) (*auto simp*: *bin-nth-cat nth-bintr*)

lemma *bintr-cat-same* [*simp*]:
bintrunc n (*bin-cat a n b*) = *bintrunc n b*
by (*auto simp add* : *bintr-cat*)

lemma *cat-bintr* [*simp*]:
bin-cat a n (*bintrunc n b*) = *bin-cat a n b*
by (*induct n arbitrary*: *b*) *auto*

lemma *split-bintrunc*:
bin-split n c = (*a*, *b*) ==> *b* = *bintrunc n c*
by (*induct n arbitrary*: *b c*) (*auto simp*: *Let-def split*: *prod.split-asm*)

lemma *bin-cat-split*:
bin-split n w = (*u*, *v*) ==> *w* = *bin-cat u n v*
by (*induct n arbitrary*: *v w*) (*auto simp*: *Let-def split*: *prod.split-asm*)

lemma *bin-split-cat*:
bin-split n (*bin-cat v n w*) = (*v*, *bintrunc n w*)
by (*induct n arbitrary*: *w*) *auto*

lemma *bin-split-zero* [*simp*]: *bin-split n 0* = (*0*, *0*)
by (*induct n*) *auto*

lemma *bin-split-minus1* [*simp*]:
bin-split n (− *1*) = (− *1*, *bintrunc n* (− *1*))
by (*induct n*) *auto*

lemma *bin-split-trunc*:
bin-split (*min m n*) *c* = (*a*, *b*) ==>
bin-split n (*bintrunc m c*) = (*bintrunc* (*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 split*: *prod.split-asm*)
done

lemma *bin-split-trunc1*:
bin-split n c = (*a*, *b*) ==>
bin-split n (*bintrunc m c*) = (*bintrunc* (*m* − *n*) *a*, *bintrunc 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 split*: *prod.split-asm*)
done

lemma *bin-cat-num*:
bin-cat a n b = *a* ∗ *2* ^ *n* + *bintrunc n b*
apply (*induct n arbitrary*: *b*, *clarsimp*)

apply (*simp add*: *Bit-def*)
done

lemma *bin-split-num*:
bin-split n b = (*b div 2 ^ n*, *b mod 2 ^ n*)
apply (*induct n arbitrary*: *b*, *simp*)
apply (*simp add*: *bin-rest-def zdiv-zmult2-eq*)
apply (*case-tac b rule*: *bin-exhaust*)
apply *simp*
apply (*simp add*: *Bit-def mod-mult-mult1 p1mod22k*)
done

5.4 Miscellaneous lemmas

lemma *nth-2p-bin*:
bin-nth (*2 ^ n*) *m* = (*m* = *n*)
apply (*induct n arbitrary*: *m*)
apply *clarsimp*
apply *safe*
apply (*case-tac m*)
apply (*auto simp*: *Bit-B0-2t* [*symmetric*])
done

lemma *ex-eq-or*:
(*EX m*. *n* = *Suc m* & (*m* = *k* | *P m*)) = (*n* = *Suc k* | (*EX m*. *n* = *Suc m* & *P m*))
by *auto*

lemma *power-BIT*: *2 ^ (Suc n) − 1* = (*2 ^ n − 1*) *BIT True*
unfolding *Bit-B1*
by (*induct n*) *simp-all*

lemma *mod-BIT*:
bin BIT bit mod 2 ^ Suc n = (*bin mod 2 ^ n*) *BIT bit*
proof −
have *2 ∗* (*bin mod 2 ^ n*) + *1* = (*2 ∗ bin mod 2 ^ Suc n*) + *1*
by (*simp add*: *mod-mult-mult1*)
also have ... = ((*2 ∗ bin mod 2 ^ Suc n*) + *1*) *mod 2 ^ Suc n*
by (*simp add*: *ac-simps p1mod22k′*)
also have ... = (*2 ∗ bin* + *1*) *mod 2 ^ Suc n*
by (*simp only*: *mod-simps*)
finally show *?thesis*
by (*auto simp add*: *Bit-def*)
qed

lemma *AND-mod*:
fixes *x* :: *int*

shows *x AND 2 ^ n − 1 = x mod 2 ^ n*
proof (*induct x arbitrary*: *n rule*: *bin-induct*)
case *1*
then show *?case*
by *simp*
next
case *2*
then show *?case*
by (*simp*, *simp add*: *m1mod2k*)
next
case (*3 bin bit*)
show *?case*
proof (*cases n*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc m*)
with *3* **show** *?thesis*
by (*simp only*: *power-BIT mod-BIT int-and-Bits*) *simp*
qed
qed

end

6 Bool lists and integers

theory *Bool-List-Representation*
imports *Bits-Int*
begin

definition *map2* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list*
where *map2 f as bs* = *map* (*case-prod f*) (*zip as bs*)

lemma *map2-Nil* [*simp*, *code*]: *map2 f* [] *ys* = []
by (*auto simp*: *map2-def*)

lemma *map2-Nil2* [*simp*, *code*]: *map2 f xs* [] = []
by (*auto simp*: *map2-def*)

lemma *map2-Cons* [*simp*, *code*]: *map2 f* (*x* # *xs*) (*y* # *ys*) = *f x y* # *map2 f xs ys*
by (*auto simp*: *map2-def*)

6.1 Operations on lists of booleans

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* (*w BIT b*)

definition *bl-to-bin* :: *bool list* ⇒ *int*
where *bl-to-bin bs* = *bl-to-bin-aux bs 0*

primrec *bin-to-bl-aux* :: *nat* ⇒ *int* ⇒ *bool list* ⇒ *bool list*
where
Z: *bin-to-bl-aux 0 w bl* = *bl*
| *Suc*: *bin-to-bl-aux* (*Suc n*) *w bl* = *bin-to-bl-aux n* (*bin-rest w*) ((*bin-last w*) # *bl*)

definition *bin-to-bl* :: *nat* ⇒ *int* ⇒ *bool list*
where *bin-to-bl n w* = *bin-to-bl-aux n w* []

primrec *bl-of-nth* :: *nat* ⇒ (*nat* ⇒ *bool*) ⇒ *bool list*
where
Suc: *bl-of-nth* (*Suc n*) *f* = *f n* # *bl-of-nth n f*
| *Z*: *bl-of-nth 0 f* = []

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*)

6.2 Arithmetic in terms of bool lists

Arithmetic operations in terms of the reversed bool list, assuming input list(s) the same length, and don't extend them.

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 *butlast-power*: *(butlast ^^ n) bl = take (length bl − n) bl*
by *(induct n) (auto simp*: *butlast-take)*

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-Bit-minus-simp* [*simp*]:
0 < n ⟹ bin-to-bl-aux n (w BIT b) bl = bin-to-bl-aux (n − 1) w (b # 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) auto*

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) auto*

Link between bin and bool list.

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*: *size* (*bin-to-bl-aux n w bs*) = *n* + *length bs*
by (*induct n arbitrary*: *w bs*) *auto*

lemma *size-bin-to-bl* [*simp*]: *size* (*bin-to-bl n w*) = *n*
by (*simp add*: *bin-to-bl-def size-bin-to-bl-aux*)

lemma *bin-bl-bin'*: *bl-to-bin* (*bin-to-bl-aux n w bs*) = *bl-to-bin-aux bs* (*bintrunc n w*)
by (*induct n arbitrary*: *w bs*) (*auto simp*: *bl-to-bin-def*)

lemma *bin-bl-bin* [*simp*]: *bl-to-bin* (*bin-to-bl n w*) = *bintrunc n w*
by (*auto simp*: *bin-to-bl-def bin-bl-bin'*)

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*)

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 (bintrunc 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 (bintrunc 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*
done

lemma *bin-to-bl-bintr*:
bin-to-bl n (bintrunc 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 (*induct bs arbitrary*: *w*) *auto*

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* (*sbintrunc n w*) = *−1*)
apply (*induct n arbitrary*: *w bs*)
apply *clarsimp*
apply (*cases w rule*: *bin-exhaust*)
apply *simp*
done

lemma *bl-sbin-sign*: *hd* (*bin-to-bl* (*Suc n*) *w*) = (*bin-sign* (*sbintrunc n w*) = *−1*)
unfolding *bin-to-bl-def* **by** (*rule bl-sbin-sign-aux*)

lemma *bin-nth-of-bl-aux*:
bin-nth (*bl-to-bin-aux bl w*) *n* =
(*n* < *size bl* ∧ *rev bl* ! *n* | *n* ≥ *length bl* ∧ *bin-nth w* (*n* − *size bl*))
apply (*induct bl arbitrary*: *w*)
apply *clarsimp*
apply *clarsimp*
apply (*cut-tac x*=*n* **and** *y*=*size bl* **in** *linorder-less-linear*)
apply (*erule disjE*, *simp add*: *nth-append*)+
apply *auto*
done

lemma *bin-nth-of-bl*: *bin-nth* (*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* ⟹ *bin-nth w n* = *nth* (*rev* (*bin-to-bl m w*)) *n*
apply (*induct n arbitrary*: *m w*)
apply *clarsimp*
apply (*case-tac m*, *clarsimp*)
apply (*clarsimp simp*: *bin-to-bl-def*)
apply (*simp add*: *bin-to-bl-aux-alt*)
apply *clarsimp*
apply (*case-tac m*, *clarsimp*)
apply (*clarsimp simp*: *bin-to-bl-def*)
apply (*simp add*: *bin-to-bl-aux-alt*)
done

lemma *nth-rev*: *n* < *length xs* ⟹ *rev xs* ! *n* = *xs* ! (*length xs* − *1* − *n*)
apply (*induct xs*)
apply *simp*
apply (*clarsimp simp add*: *nth-append nth.simps split*: *nat.split*)
apply (*rule-tac f* = *λn*. *xs* ! *n* **in** *arg-cong*)
apply *arith*
done

lemma *nth-rev-alt*: $n < length\ ys \Longrightarrow ys\ !\ n = rev\ ys\ !\ (length\ ys - Suc\ n)$
by (*simp add*: *nth-rev*)

lemma *nth-bin-to-bl-aux*:
$n < m + length\ bl \Longrightarrow (bin\text{-}to\text{-}bl\text{-}aux\ m\ w\ bl)\ !\ n =$
$(if\ n < m\ then\ bin\text{-}nth\ w\ (m - 1 - n)\ else\ bl\ !\ (n - m))$
apply (*induct m arbitrary*: *w n bl*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac w rule*: *bin-exhaust*)
apply *simp*
done

lemma *nth-bin-to-bl*: $n < m \Longrightarrow (bin\text{-}to\text{-}bl\ m\ w)\ !\ n = bin\text{-}nth\ w\ (m - Suc\ n)$
by (*simp add*: *bin-to-bl-def nth-bin-to-bl-aux*)

lemma *bl-to-bin-lt2p-aux*: $bl\text{-}to\text{-}bin\text{-}aux\ bs\ w < (w + 1) * (2\ \hat{}\ length\ bs)$
apply (*induct bs arbitrary*: *w*)
apply *clarsimp*
apply *clarsimp*
apply (*drule meta-spec*, *erule xtrans*(*8*) [*rotated*], *simp add*: *Bit-def*)+
done

lemma *bl-to-bin-lt2p-drop*: $bl\text{-}to\text{-}bin\ bs < 2\ \hat{}\ length\ (dropWhile\ Not\ bs)$
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* **unfolding** *bl-to-bin-def* **by** *simp*
qed

lemma *bl-to-bin-lt2p*: $bl\text{-}to\text{-}bin\ bs < 2\ \hat{}\ length\ bs$
by (*metis bin-bl-bin bintr-lt2p bl-bin-bl*)

lemma *bl-to-bin-ge2p-aux*: $bl\text{-}to\text{-}bin\text{-}aux\ bs\ w \geq w * (2\ \hat{}\ length\ bs)$
apply (*induct bs arbitrary*: *w*)
apply *clarsimp*
apply *clarsimp*
apply (*drule meta-spec*, *erule order-trans* [*rotated*],
simp add: *Bit-B0-2t Bit-B1-2t algebra-simps*)+
apply (*simp add*: *Bit-def*)
done

lemma *bl-to-bin-ge0*: $bl\text{-}to\text{-}bin\ bs \geq 0$
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*) (*bin-rest w*)
apply (*unfold bin-to-bl-def*)
apply (*cases w rule*: *bin-exhaust*)
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*) (*bin-rest* (*bl-to-bin bl*))
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* = *bin-rest* (*bl-to-bin-aux bl w*)
by (*induct bl arbitrary*: *w*) *auto*

lemma *butlast-rest-bl2bin*: *bl-to-bin* (*butlast bl*) = *bin-rest* (*bl-to-bin bl*)
by (*cases bl*) (*auto simp*: *bl-to-bin-def butlast-rest-bl2bin-aux*)

lemma *trunc-bl2bin-aux*:
bintrunc m (*bl-to-bin-aux bl w*) =
bl-to-bin-aux (*drop* (*length bl* − *m*) *bl*) (*bintrunc* (*m* − *length bl*) *w*)
proof (*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*
qed
qed

lemma *trunc-bl2bin*: *bintrunc 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*]: *bintrunc* (*length bl*) (*bl-to-bin bl*) = *bl-to-bin bl*
by (*simp add*: *trunc-bl2bin*)

lemma *bl2bin-drop*: *bl-to-bin* (*drop k bl*) = *bintrunc* (*length bl* − *k*) (*bl-to-bin bl*)

apply (*rule trans*)
prefer *2*
apply (*rule trunc-bl2bin* [*symmetric*])
apply (*cases* $k \leq$ *length bl*)
apply *auto*
done

lemma *nth-rest-power-bin*: *bin-nth* ((*bin-rest* ^^ *k*) *w*) *n* = *bin-nth w* (*n* + *k*)
apply (*induct k arbitrary*: *n*)
apply *clarsimp*
apply *clarsimp*
apply (*simp only*: *bin-nth.Suc* [*symmetric*] *add-Suc*)
done

lemma *take-rest-power-bin*: *m* ≤ *n* ⟹ *take m* (*bin-to-bl n w*) = *bin-to-bl m* ((*bin-rest* ^^ (*n* − *m*)) *w*)
apply (*rule nth-equalityI*)
apply *simp*
apply (*clarsimp simp add*: *nth-bin-to-bl nth-rest-power-bin*)
done

lemma *hd-butlast*: *size xs* > *1* ⟹ *hd* (*butlast xs*) = *hd xs*
by (*cases xs*) *auto*

lemma *last-bin-last'*: *size xs* > *0* ⟹ *last xs* ⟷ *bin-last* (*bl-to-bin-aux xs w*)
by (*induct xs arbitrary*: *w*) *auto*

lemma *last-bin-last*: *size xs* > *0* ⟹ *last xs* ⟷ *bin-last* (*bl-to-bin xs*)
unfolding *bl-to-bin-def* **by** (*erule last-bin-last'*)

lemma *bin-last-last*: *bin-last w* ⟷ *last* (*bin-to-bl* (*Suc n*) *w*)
by (*simp add*: *bin-to-bl-def*) (*auto simp*: *bin-to-bl-aux-alt*)

6.3 Links between bit-wise operations and operations on bool lists

lemma *bl-xor-aux-bin*:
map2 (λ*x 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* (λ*x y*. *x* ≠ *y*) *bs cs*)
apply (*induct n arbitrary*: *v w bs cs*)
apply *simp*
apply (*case-tac v rule*: *bin-exhaust*)
apply (*case-tac w rule*: *bin-exhaust*)
apply *clarsimp*
apply (*case-tac b*)
apply *auto*
done

lemma *bl-or-aux-bin*:

map2 (*op* ∨) (*bin-to-bl-aux n v bs*) (*bin-to-bl-aux n w cs*) =
bin-to-bl-aux n (*v OR w*) (*map2* (*op* ∨) *bs cs*)
apply (*induct n arbitrary*: *v w bs cs*)
apply *simp*
apply (*case-tac v rule*: *bin-exhaust*)
apply (*case-tac w rule*: *bin-exhaust*)
apply *clarsimp*
done

lemma *bl-and-aux-bin*:
map2 (*op* ∧) (*bin-to-bl-aux n v bs*) (*bin-to-bl-aux n w cs*) =
bin-to-bl-aux n (*v AND w*) (*map2* (*op* ∧) *bs cs*)
apply (*induct n arbitrary*: *v w bs cs*)
apply *simp*
apply (*case-tac v rule*: *bin-exhaust*)
apply (*case-tac w rule*: *bin-exhaust*)
apply *clarsimp*
done

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* (*op* ∧) (*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* (*op* ∨) (*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* (λ*x y*. *x* ≠ *y*) (*bin-to-bl n v*) (*bin-to-bl n w*) = *bin-to-bl n* (*v XOR w*)
by (*simp only*: *bin-to-bl-def bl-xor-aux-bin map2-Nil*)

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 (*induct n arbitrary*: *m bin bs*, *clarsimp*)
apply *clarsimp*
apply (*case-tac bin rule*: *bin-exhaust*)
apply (*case-tac m* ≤ *n*, *simp*)
apply (*case-tac m* − *n*, *simp*)
apply *simp*
apply (*rule-tac f* = λ*nat*. *drop nat bs* **in** *arg-cong*)
apply *simp*

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*)
done

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*)
by (*auto elim*: *bin-split-take*)

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'*: ⋀*w n*. *n* = *k* + *m* ⟹ *take k* (*takefill fill n w*) = *takefill fill k w*
by (*induct k*) (*auto split*: *list.split*)

lemma *drop-takefill*: ⋀*w*. *drop k* (*takefill fill* (*m* + *k*) *w*) = *takefill fill m* (*drop k w*)
by (*induct k*) (*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 *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 nth-rev 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-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-bin-bl-rtf takefill-alt rev-take*)

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 nth-rev*)
apply (*rule-tac f* = λ*n*. *bl* ! *n* **in** *arg-cong*)
apply *arith*

done

lemma *takefill-minus*: $0 < n \Longrightarrow$ *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*)

6.4 Links with function *bl-to-bin*

lemma *bl-to-bin-aux-cat*:
$\bigwedge$*nv v*. *bl-to-bin-aux bs* (*bin-cat w nv v*) =
bin-cat w (*nv* + *length bs*) (*bl-to-bin-aux bs v*)
by (*induct bs*) (*simp*, *simp add*: *bin-cat-Suc-Bit* [*symmetric*] *del*: *bin-cat.simps*)

lemma *bin-to-bl-aux-cat*:
$\bigwedge$*w bs*. *bin-to-bl-aux* (*nv* + *nw*) (*bin-cat v nw w*) *bs* =
bin-to-bl-aux nv v (*bin-to-bl-aux nw w bs*)
by (*induct nw*) *auto*

lemma *bl-to-bin-aux-alt*: *bl-to-bin-aux bs w* = *bin-cat w* (*length bs*) (*bl-to-bin bs*)
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*) (*bin-cat v nw w*) =
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*) = *bin-cat* (*bl-to-bin bsa*) (*length bs*) (*bl-to-bin bs*)
by (*simp only*: *bl-to-bin-aux-app-cat bl-to-bin-def*)

lemma *bin-to-bl-cat-app*:
bin-to-bl (*n* + *nw*) (*bin-cat w nw wa*) = *bin-to-bl n w* @ *bin-to-bl nw wa*
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*: *bin-cat* (*bl-to-bin cs*) *n w* = *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 add*: *Bit-B0-2t Bit-B1-2t*)
done

6.5 Function *bl-of-nth*

lemma *length-bl-of-nth* [*simp*]: *length* (*bl-of-nth n f*) = *n*
by (*induct n*) *auto*

lemma *nth-bl-of-nth* [*simp*]: *m* < *n* ⟹ *rev* (*bl-of-nth n f*) ! *m* = *f m*
apply (*induct n*)
apply *simp*
apply (*clarsimp simp add*: *nth-append*)
apply (*rule-tac f* = *f* **in** *arg-cong*)
apply *simp*
done

lemma *bl-of-nth-inj*: (⋀*k*. *k* < *n* ⟹ *f k* = *g k*) ⟹ *bl-of-nth n f* = *bl-of-nth n g*
by (*induct n*) *auto*

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*: *nth-rev trans* [*OF drop-Suc drop-tl*, *symmetric*])
apply (*subst hd-drop-conv-nth*)
apply *force*
apply *simp-all*

apply (*rule-tac f* = λ*n*. *drop n xs* **in** *arg-cong*)
apply *simp*
done

lemma *bl-of-nth-nth* [*simp*]: *bl-of-nth* (*length xs*) (*op* ! (*rev xs*)) = *xs*
by (*simp add*: *bl-of-nth-nth-le*)

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-pred*: *rbl-pred* (*rev* (*bin-to-bl n bin*)) = *rev* (*bin-to-bl n* (*bin* − *1*))
apply (*unfold bin-to-bl-def*)
apply (*induct n arbitrary*: *bin*)
apply *simp*
apply *clarsimp*
apply (*case-tac bin rule*: *bin-exhaust*)
apply (*case-tac b*)
apply (*clarsimp simp*: *bin-to-bl-aux-alt*)+
done

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 (*induct n arbitrary*: *bin*)
apply *simp*
apply *clarsimp*
apply (*case-tac bin rule*: *bin-exhaust*)
apply (*case-tac b*)
apply (*clarsimp simp*: *bin-to-bl-aux-alt*)+
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-app2*: *length blb* $\geq$ *length bla* $\Longrightarrow$ *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-add-long*:
m $\geq$ *n* $\Longrightarrow$ *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-app2*: *length blb* $\geq$ *length bla* $\Longrightarrow$ *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 $\geq$ *length bla* $\Longrightarrow$ *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-mult-gt1*:
m $\geq$ *length bl* $\Longrightarrow$
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* $\Longrightarrow$
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*) (*x BIT b*))
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*
apply (*unfold bin-to-bl-def*)
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*: *bin-to-bl-aux-alt Let-def*)
apply (*auto simp*: *rbbl-Cons rbl-mult-Suc rbl-add*)
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*)

6.6 Repeated splitting or concatenation

lemma *sclem*: *size* (*concat* (*map* (*bin-to-bl n*) *xs*)) = *length xs* ∗ *n*
by (*induct xs*) *auto*

lemma *bin-cat-foldl-lem*:
foldl (λ*u*. *bin-cat u n*) *x xs* =
bin-cat x (*size xs* ∗ *n*) (*foldl* (λ*u*. *bin-cat u n*) *y xs*)
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* = *bin-cat y n a* **in** *meta-spec*)
apply (*simp add*: *bin-cat-assoc-sym min.absorb2*)
done

lemma *bin-rcat-bl*: *bin-rcat n wl* = *bl-to-bin* (*concat* (*map* (*bin-to-bl n*) *wl*))
apply (*unfold bin-rcat-def*)
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

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* = *op* = *l*, *THEN iffD1*, *THEN conjunct1*, *THEN mp*] **for** *l*
lemmas *th-if-simp2* = *if-split* [**where** *P* = *op* = *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*]

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*

lemmas *bin-split-minus-simp* =
bin-split.Suc [*THEN* [*2*] *bin-split-minus* [*symmetric*, *THEN trans*]]

lemma *bin-split-pred-simp* [*simp*]:
(*0*::*nat*) < *numeral bin* ⟹
bin-split (*numeral bin*) *w* =
(*let* (*w1*, *w2*) = *bin-split* (*numeral bin* − *1*) (*bin-rest w*)
in (*w1*, *w2 BIT bin-last w*))
by (*simp only*: *bin-split-minus-simp*)

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*]:
$n > 0 \Longrightarrow rev\ sw = bin\text{-}rsplit\ n\ (nw,\ w) \Longrightarrow \forall v{\in}set\ sw.\ bintrunc\ n\ v = v$
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 (*drule split-bintrunc*)
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*] :
$n > 0 \Longrightarrow m < n \Longrightarrow$
$\forall w\ k\ nw.$
$rev\ sw = bin\text{-}rsplit\ n\ (nw,\ w) \longrightarrow$
$k < size\ sw \longrightarrow bin\text{-}nth\ (sw\ !\ k)\ m = bin\text{-}nth\ w\ (k * n + m)$
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 *clarify*
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 (*drule bin-nth-split*, *erule conjE*, *erule allE*, *erule trans*, *simp add*: *ac-simps*)+
done

lemma *bin-rsplit-all*: $0 < nw \Longrightarrow nw \leq n \Longrightarrow bin\text{-}rsplit\ n\ (nw,\ w) = [bintrunc\ n\ w]$
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*, *bintrunc 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 (*drule bin-split-trunc*)
 apply (*drule sym* [*THEN trans*], *assumption*)
 apply (*subst rsplit-aux-alts*(*1*))
 apply (*subst rsplit-aux-alts*(*2*))
 apply *clarsimp*
 unfolding *bin-rsplit-def bin-rsplitl-def*
 apply *simp*
 done

lemma *bin-rsplit-rcat* [*rule-format*]:
 n > *0* ⟶ *bin-rsplit n* (*n* ∗ *size ws*, *bin-rcat n ws*) = *map* (*bintrunc n*) *ws*
 apply (*unfold bin-rsplit-def bin-rcat-def*)
 apply (*rule-tac xs* = *ws* **in** *rev-induct*)
 apply *clarsimp*
 apply *clarsimp*
 apply (*subst rsplit-aux-alts*)
 unfolding *bin-split-cat*
 apply *simp*
 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 ‹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) (fst (bin-split n w)) (snd (bin-split n w) #
cs))
      by auto
```

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

Even more bit operations

instantiation *int* :: *bitss*
begin

definition [*iff*]: *i* !! *n* ⟷ *bin-nth i n*

definition *lsb i* = *i* !! *0* **for** *i* :: *int*

definition *set-bit i n b* = *bin-sc n b i*

definition
set-bits f =
(*if* ∃ *n*. ∀ *n'*≥*n*. ¬ *f n'* *then*
let n = *LEAST n*. ∀ *n'*≥*n*. ¬ *f n'*
in bl-to-bin (*rev* (*map f* [*0*..<*n*]))
else if ∃ *n*. ∀ *n'*≥*n*. *f n'* *then*
let n = *LEAST n*. ∀ *n'*≥*n*. *f n'*
in sbintrunc n (*bl-to-bin* (*True* # *rev* (*map f* [*0*..<*n*])))
else 0 :: *int*)

definition *shiftl x n* = *x* ∗ *2* ^ *n* **for** *x* :: *int*

definition *shiftr x n* = *x div 2* ^ *n* **for** *x* :: *int*

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

instance ..

end

end

7 Type Definition Theorems

theory *Misc-Typedef*
imports *Main*

begin

8 More lemmas about normal type definitions

lemma
tdD1: *type-definition Rep Abs A* $\Longrightarrow \forall x.$ *Rep* $x \in A$ **and**
tdD2: *type-definition Rep Abs A* $\Longrightarrow \forall x.$ *Abs* (*Rep* x) $= x$ **and**
tdD3: *type-definition Rep Abs A* $\Longrightarrow \forall y.$ $y \in A \longrightarrow$ *Rep* (*Abs* y) $= y$
by (*auto simp*: *type-definition-def*)

lemma *td-nat-int*:
type-definition int nat (*Collect* (*op* <= *0*))
unfolding *type-definition-def* **by** *auto*

context *type-definition*
begin

declare *Rep* [*iff*] *Rep-inverse* [*simp*] *Rep-inject* [*simp*]

lemma *Abs-eqD*: *Abs* x = *Abs* y ==> $x \in A$ ==> $y \in A$ ==> $x = y$
by (*simp add*: *Abs-inject*)

lemma *Abs-inverse'*:
r : A ==> *Abs* r = a ==> *Rep* a = r
by (*safe elim*!: *Abs-inverse*)

lemma *Rep-comp-inverse*:
Rep $\circ$ f = g ==> *Abs* $\circ$ g = f
using *Rep-inverse* **by** *auto*

lemma *Rep-eqD* [*elim*!]: *Rep* x = *Rep* y ==> $x = y$
by *simp*

lemma *Rep-inverse'*: *Rep* a = r ==> *Abs* r = a
by (*safe intro*!: *Rep-inverse*)

lemma *comp-Abs-inverse*:
f $\circ$ *Abs* = g ==> g $\circ$ *Rep* = f
using *Rep-inverse* **by** *auto*

lemma *set-Rep*:
A = *range Rep*
proof (*rule set-eqI*)
fix x
show ($x \in A$) = ($x \in$ *range Rep*)
by (*auto dest*: *Abs-inverse* [*of* x, *symmetric*])
qed

lemma *set-Rep-Abs*: A = *range* (*Rep* $\circ$ *Abs*)

proof (*rule set-eqI*)
fix *x*
show (*x* ∈ *A*) = (*x* ∈ *range* (*Rep* ∘ *Abs*))
by (*auto dest*: *Abs-inverse* [*of x*, *symmetric*])
qed

lemma *Abs-inj-on*: *inj-on Abs A*
unfolding *inj-on-def*
by (*auto dest*: *Abs-inject* [*THEN iffD1*])

lemma *image*: *Abs ' A = UNIV*
by (*auto intro*!: *image-eqI*)

lemmas *td-thm* = *type-definition-axioms*

lemma *fns1*:
Rep ∘ *fa* = *fr* ∘ *Rep* | *fa* ∘ *Abs* = *Abs* ∘ *fr* ==> *Abs* ∘ *fr* ∘ *Rep* = *fa*
by (*auto dest*: *Rep-comp-inverse elim*: *comp-Abs-inverse simp*: *o-assoc*)

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

lemma *fns4*:
Rep ∘ *fa* ∘ *Abs* = *fr* ==>
Rep ∘ *fa* = *fr* ∘ *Rep* & *fa* ∘ *Abs* = *Abs* ∘ *fr*
by *auto*

end

interpretation *nat-int*: *type-definition int nat Collect* (*op* <= *0*)
by (*rule td-nat-int*)

declare
nat-int.Rep-cases [*cases del*]
nat-int.Abs-cases [*cases del*]
nat-int.Rep-induct [*induct del*]
nat-int.Abs-induct [*induct del*]

8.1 Extended form of type definition predicate

lemma *td-conds*:
norm ∘ *norm* = *norm* ==> (*fr* ∘ *norm* = *norm* ∘ *fr*) =
(*norm* ∘ *fr* ∘ *norm* = *fr* ∘ *norm* & *norm* ∘ *fr* ∘ *norm* = *norm* ∘ *fr*)
apply *safe*
apply (*simp-all add*: *comp-assoc*)
apply (*simp-all add*: *o-assoc*)
done

lemma *fn-comm-power*:

fa ∘ *tr* = *tr* ∘ *fr* ==> *fa* ^^ *n* ∘ *tr* = *tr* ∘ *fr* ^^ *n*
apply (*rule ext*)
apply (*induct n*)
apply (*auto dest*: *fun-cong*)
done

lemmas *fn-comm-power'* =
ext [*THEN fn-comm-power*, *THEN fun-cong*, *unfolded o-def*]

locale *td-ext* = *type-definition* +
fixes *norm*
assumes *eq-norm*: ⋀*x*. *Rep* (*Abs x*) = *norm x*
begin

lemma *Abs-norm* [*simp*]:
Abs (*norm x*) = *Abs x*
using *eq-norm* [*of x*] **by** (*auto elim*: *Rep-inverse'*)

lemma *td-th*:
g ∘ *Abs* = *f* ==> *f* (*Rep x*) = *g x*
by (*drule comp-Abs-inverse* [*symmetric*]) *simp*

lemma *eq-norm'*: *Rep* ∘ *Abs* = *norm*
by (*auto simp*: *eq-norm*)

lemma *norm-Rep* [*simp*]: *norm* (*Rep x*) = *Rep x*
by (*auto simp*: *eq-norm' intro*: *td-th*)

lemmas *td* = *td-thm*

lemma *set-iff-norm*: *w* : *A* ⟷ *w* = *norm w*
by (*auto simp*: *set-Rep-Abs eq-norm' eq-norm* [*symmetric*])

lemma *inverse-norm*:
(*Abs n* = *w*) = (*Rep w* = *norm n*)
apply (*rule iffI*)
apply (*clarsimp simp add*: *eq-norm*)
apply (*simp add*: *eq-norm'* [*symmetric*])
done

lemma *norm-eq-iff*:
(*norm x* = *norm y*) = (*Abs x* = *Abs y*)
by (*simp add*: *eq-norm'* [*symmetric*])

lemma *norm-comps*:
Abs ∘ *norm* = *Abs*
norm ∘ *Rep* = *Rep*
norm ∘ *norm* = *norm*

by (*auto simp*: *eq-norm'* [*symmetric*] *o-def*)

lemmas *norm-norm* [*simp*] = *norm-comps*

lemma *fns5*:
Rep ∘ *fa* ∘ *Abs* = *fr* ==>
fr ∘ *norm* = *fr* & *norm* ∘ *fr* = *fr*
by (*fold eq-norm'*) *auto*

lemma *fns2*:
Abs ∘ *fr* ∘ *Rep* = *fa* ==>
(*norm* ∘ *fr* ∘ *norm* = *fr* ∘ *norm*) = (*Rep* ∘ *fa* = *fr* ∘ *Rep*)
apply (*fold eq-norm'*)
apply *safe*
prefer *2*
apply (*simp add*: *o-assoc*)
apply (*rule ext*)
apply (*drule-tac x*=*Rep x* **in** *fun-cong*)
apply *auto*
done

lemma *fns3*:
Abs ∘ *fr* ∘ *Rep* = *fa* ==>
(*norm* ∘ *fr* ∘ *norm* = *norm* ∘ *fr*) = (*fa* ∘ *Abs* = *Abs* ∘ *fr*)
apply (*fold eq-norm'*)
apply *safe*
prefer *2*
apply (*simp add*: *comp-assoc*)
apply (*rule ext*)
apply (*drule-tac f*=*a* ∘ *b* **for** *a b* **in** *fun-cong*)
apply *simp*
done

lemma *fns*:
fr ∘ *norm* = *norm* ∘ *fr* ==>
(*fa* ∘ *Abs* = *Abs* ∘ *fr*) = (*Rep* ∘ *fa* = *fr* ∘ *Rep*)
apply *safe*
apply (*frule fns1b*)
prefer *2*
apply (*frule fns1a*)
apply (*rule fns3* [*THEN iffD1*])
prefer *3*
apply (*rule fns2* [*THEN iffD1*])
apply (*simp-all add*: *comp-assoc*)
apply (*simp-all add*: *o-assoc*)
done

lemma *range-norm*:

range (*Rep* ∘ *Abs*) = *A*
by (*simp add*: *set-Rep-Abs*)

end

lemmas *td-ext-def'* =
td-ext-def [*unfolded type-definition-def td-ext-axioms-def*]

end

9 Miscellaneous lemmas, of at least doubtful value

theory *Word-Miscellaneous*
imports *HOL−Library.Bit Misc-Numeric*
begin

lemma *power-minus-simp*: *0* < *n* ⟹ *a* ^ *n* = *a* ∗ *a* ^ (*n* − *1*)
by (*auto dest*: *gr0-implies-Suc*)

lemma *funpow-minus-simp*: *0* < *n* ⟹ *f* ^^ *n* = *f* ∘ *f* ^^ (*n* − *1*)
by (*auto dest*: *gr0-implies-Suc*)

lemma *power-numeral*: *a* ^ *numeral k* = *a* ∗ *a* ^ (*pred-numeral k*)
by (*simp add*: *numeral-eq-Suc*)

lemma *funpow-numeral* [*simp*]: *f* ^^ *numeral k* = *f* ∘ *f* ^^ (*pred-numeral k*)
by (*simp add*: *numeral-eq-Suc*)

lemma *replicate-numeral* [*simp*]: *replicate* (*numeral k*) *x* = *x* # *replicate* (*pred-numeral k*) *x*
by (*simp add*: *numeral-eq-Suc*)

lemma *rco-alt*: (*f* ∘ *g*) ^^ *n* ∘ *f* = *f* ∘ (*g* ∘ *f*) ^^ *n*
apply (*rule ext*)
apply (*induct n*)
apply (*simp-all add*: *o-def*)
done

lemma *list-exhaust-size-gt0*:
assumes *y*: ⋀*a list*. *y* = *a* # *list* ⟹ *P*
shows *0* < *length y* ⟹ *P*
apply (*cases y*)
apply *simp*
apply (*rule y*)
apply *fastforce*
done

lemma *list-exhaust-size-eq0*:
assumes *y*: *y* = [] ⟹ *P*

shows *length y = 0 ⟹ P*
apply (*cases y*)
apply (*rule y*)
apply *simp*
apply *simp*
done

lemma *size-Cons-lem-eq*: *y = xa # list ⟹ size y = Suc k ⟹ size list = k*
by *auto*

lemmas *ls-splits = prod.split prod.split-asm if-split-asm*

lemma *not-B1-is-B0*: *y ≠ 1 ⟹ y = 0*
for *y* :: *bit*
by (*cases y*) *auto*

lemma *B1-ass-B0*:
fixes *y* :: *bit*
assumes *y*: *y = 0 ⟹ y = 1*
shows *y = 1*
apply (*rule classical*)
apply (*drule not-B1-is-B0*)
apply (*erule y*)
done

— simplifications for specific word lengths
lemmas *n2s-ths* [*THEN eq-reflection*] *= add-2-eq-Suc add-2-eq-Suc'*

lemmas *s2n-ths = n2s-ths* [*symmetric*]

lemma *and-len*: *xs = ys ⟹ xs = ys ∧ length xs = length ys*
by *auto*

lemma *size-if*: *size* (*if p then xs else ys*) = (*if p then size xs else size ys*)
by *auto*

lemma *tl-if*: *tl* (*if p then xs else ys*) = (*if p then tl xs else tl ys*)
by *auto*

lemma *hd-if*: *hd* (*if p then xs else ys*) = (*if p then hd xs else hd ys*)
by *auto*

lemma *if-Not-x*: (*if p then ¬ x else x*) = (*p* = (*¬ x*))
by *auto*

lemma *if-x-Not*: (*if p then x else ¬ x*) = (*p* = *x*)
by *auto*

lemma *if-same-and*: (*If p x y ∧ If p u v*) = (*if p then x ∧ u else y ∧ v*)

by *auto*

lemma *if-same-eq*: $(If\ p\ x\ y\ = (If\ p\ u\ v)) = (if\ p\ then\ x = u\ else\ y = v)$
by *auto*

lemma *if-same-eq-not*: $(If\ p\ x\ y = (\neg\ If\ p\ u\ v)) = (if\ p\ then\ x = (\neg\ u)\ else\ y = (\neg\ v))$
by *auto*

lemma *if-Cons*: $(if\ p\ then\ x\ \#\ xs\ else\ y\ \#\ ys) = If\ p\ x\ y\ \#\ If\ p\ xs\ ys$
by *auto*

lemma *if-single*: $(if\ xc\ then\ [xab]\ else\ [an]) = [if\ xc\ then\ xab\ else\ an]$
by *auto*

lemma *if-bool-simps*:
$If\ p\ True\ y = (p \vee y) \wedge If\ p\ False\ y = (\neg\ p \wedge y) \wedge$
$If\ p\ y\ True = (p \longrightarrow y) \wedge If\ p\ y\ False = (p \wedge y)$
by *auto*

lemmas *if-simps* =
if-x-Not if-Not-x if-cancel if-True if-False if-bool-simps

lemmas *seqr* = *eq-reflection* [**where** x = *size* w] **for** w

lemma *the-elemI*: $y = \{x\} \Longrightarrow the\text{-}elem\ y = x$
by *simp*

lemma *nonemptyE*: $S \neq \{\} \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow R) \Longrightarrow R$
by *auto*

lemma *gt-or-eq-0*: $0 < y \vee 0 = y$
for y :: *nat*
by *arith*

lemmas *xtr1* = *xtrans*(1)
lemmas *xtr2* = *xtrans*(2)
lemmas *xtr3* = *xtrans*(3)
lemmas *xtr4* = *xtrans*(4)
lemmas *xtr5* = *xtrans*(5)
lemmas *xtr6* = *xtrans*(6)
lemmas *xtr7* = *xtrans*(7)
lemmas *xtr8* = *xtrans*(8)

lemmas *nat-simps* = *diff-add-inverse2 diff-add-inverse*
lemmas *nat-iffs* = *le-add1 le-add2*

lemma *sum-imp-diff*: $j = k + i \Longrightarrow j - i = k$

for k :: *nat*
by *arith*

lemmas *pos-mod-sign2* = *zless2* [*THEN pos-mod-sign* [**where** b = *2*::*int*]]
lemmas *pos-mod-bound2* = *zless2* [*THEN pos-mod-bound* [**where** b = *2*::*int*]]

lemma *nmod2*: $n \bmod 2 = 0 \vee n \bmod 2 = 1$
for n :: *int*
by *arith*

lemmas *eme1p* = *emep1* [*simplified add.commute*]

lemma *le-diff-eq'*: $a \leq c - b \longleftrightarrow b + a \leq c$
for a b c :: *int*
by *arith*

lemma *less-diff-eq'*: $a < c - b \longleftrightarrow b + a < c$
for a b c :: *int*
by *arith*

lemma *diff-less-eq'*: $a - b < c \longleftrightarrow a < b + c$
for a b c :: *int*
by *arith*

lemmas *m1mod22k* = *mult-pos-pos* [*OF zless2 zless2p*, *THEN zmod-minus1*]

lemma *z1pdiv2*: $(2 * b + 1) \operatorname{div} 2 = b$
for b :: *int*
by *arith*

lemmas *zdiv-le-dividend* = *xtr3* [*OF div-by-1* [*symmetric*] *zdiv-mono2*,
simplified int-one-le-iff-zero-less, *simplified*]

lemma *axxbyy*: $a + m + m = b + n + n \Longrightarrow a = 0 \vee a = 1 \Longrightarrow b = 0 \vee b = 1 \Longrightarrow a = b \wedge m = n$
for a b m n :: *int*
by *arith*

lemma *axxmod2*: $(1 + x + x) \bmod 2 = 1 \wedge (0 + x + x) \bmod 2 = 0$
for x :: *int*
by *arith*

lemma *axxdiv2*: $(1 + x + x) \operatorname{div} 2 = x \wedge (0 + x + x) \operatorname{div} 2 = x$
for x :: *int*
by *arith*

lemmas *iszero-minus* =
trans [*THEN trans*, *OF iszero-def neg-equal-0-iff-equal iszero-def* [*symmetric*]]

lemmas *zadd-diff-inverse* =
trans [*OF diff-add-cancel* [*symmetric*] *add.commute*]

lemmas *add-diff-cancel2* =
add.commute [*THEN diff-eq-eq* [*THEN iffD2*]]

lemmas *rdmods* [*symmetric*] = *mod-minus-eq*
mod-diff-left-eq mod-diff-right-eq mod-add-left-eq
mod-add-right-eq mod-mult-right-eq mod-mult-left-eq

lemma *mod-plus-right*: $(a + x)\ mod\ m = (b + x)\ mod\ m \longleftrightarrow a\ mod\ m = b\ mod\ m$
for *a b m x* :: *nat*
by (*induct x*) (*simp-all add*: *mod-Suc*, *arith*)

lemma *nat-minus-mod*: $(n - n\ mod\ m)\ mod\ m = 0$
for *m n* :: *nat*
by (*induct n*) (*simp-all add*: *mod-Suc*)

lemmas *nat-minus-mod-plus-right* =
trans [*OF nat-minus-mod mod-0* [*symmetric*],
THEN mod-plus-right [*THEN iffD2*], *simplified*]

lemmas *push-mods'* = *mod-add-eq*
mod-mult-eq mod-diff-eq
mod-minus-eq

lemmas *push-mods* = *push-mods'* [*THEN eq-reflection*]
lemmas *pull-mods* = *push-mods* [*symmetric*] *rdmods* [*THEN eq-reflection*]

lemma *nat-mod-eq*: $b < n \Longrightarrow a\ mod\ n = b\ mod\ n \Longrightarrow a\ mod\ n = b$
for *a b n* :: *nat*
by (*induct a*) *auto*

lemmas *nat-mod-eq'* = *refl* [*THEN* [*2*] *nat-mod-eq*]

lemma *nat-mod-lem*: $0 < n \Longrightarrow b < n \longleftrightarrow b\ mod\ n = b$
for *b n* :: *nat*
apply *safe*
apply (*erule nat-mod-eq'*)
apply (*erule subst*)
apply (*erule mod-less-divisor*)
done

lemma *mod-nat-add*: $x < z \Longrightarrow y < z \Longrightarrow (x + y)\ mod\ z = (if\ x + y < z\ then\ x + y\ else\ x + y - z)$
for *x y z* :: *nat*
apply (*rule nat-mod-eq*)
apply *auto*

apply (*rule trans*)
apply (*rule le-mod-geq*)
apply *simp*
apply (*rule nat-mod-eq'*)
apply *arith*
done

lemma *mod-nat-sub*: $x < z \Longrightarrow (x - y)\ mod\ z = x - y$
for *x y* :: *nat*
by (*rule nat-mod-eq'*) *arith*

lemma *int-mod-eq*: $0 \leq b \Longrightarrow b < n \Longrightarrow a\ mod\ n = b\ mod\ n \Longrightarrow a\ mod\ n = b$
for *a b n* :: *int*
by (*metis mod-pos-pos-trivial*)

lemmas *int-mod-eq'* = *mod-pos-pos-trivial*

lemma *int-mod-le*: $0 \leq a \Longrightarrow a\ mod\ n \leq a$
for *a* :: *int*
by (*fact Divides.semiring-numeral-div-class.mod-less-eq-dividend*)

lemma *mod-add-if-z*:
$x < z \Longrightarrow y < z \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x \Longrightarrow 0 \leq z \Longrightarrow$
$(x + y)\ mod\ z = (if\ x + y < z\ then\ x + y\ else\ x + y - z)$
for *x y z* :: *int*
by (*auto intro*: *int-mod-eq*)

lemma *mod-sub-if-z*:
$x < z \Longrightarrow y < z \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x \Longrightarrow 0 \leq z \Longrightarrow$
$(x - y)\ mod\ z = (if\ y \leq x\ then\ x - y\ else\ x - y + z)$
for *x y z* :: *int*
by (*auto intro*: *int-mod-eq*)

lemmas *zmde* = *mult-div-mod-eq* [*symmetric*, *THEN diff-eq-eq* [*THEN iffD2*], *symmetric*]
lemmas *mcl* = *mult-cancel-left* [*THEN iffD1*, *THEN make-pos-rule*]

lemma *zdiv-mult-self*: $m \neq 0 \Longrightarrow (a + m * n)\ div\ m = a\ div\ m + n$
for *a m n* :: *int*
apply (*rule mcl*)
prefer *2*
apply (*erule asm-rl*)
apply (*simp add*: *zmde ring-distribs*)
done

lemma *mod-power-lem*: $a > 1 \Longrightarrow a\ \hat{}\ n\ mod\ a\ \hat{}\ m = (if\ m \leq n\ then\ 0\ else\ a\ \hat{}\ n)$
for *a* :: *int*

apply *clarsimp*
apply *safe*
apply (*simp add*: *dvd-eq-mod-eq-0* [*symmetric*])
apply (*drule le-iff-add* [*THEN iffD1*])
apply (*force simp*: *power-add*)
apply (*rule mod-pos-pos-trivial*)
apply (*simp*)
apply (*rule power-strict-increasing*)
apply *auto*
done

lemma *pl-pl-rels*: $a + b = c + d \Longrightarrow a \geq c \wedge b \leq d \vee a \leq c \wedge b \geq d$
for *a b c d* :: *nat*
by *arith*

lemmas *pl-pl-rels'* = *add.commute* [*THEN* [*2*] *trans*, *THEN pl-pl-rels*]

lemma *minus-eq*: $m - k = m \longleftrightarrow k = 0 \vee m = 0$
for *k m* :: *nat*
by *arith*

lemma *pl-pl-mm*: $a + b = c + d \Longrightarrow a - c = d - b$
for *a b c d* :: *nat*
by *arith*

lemmas *pl-pl-mm'* = *add.commute* [*THEN* [*2*] *trans*, *THEN pl-pl-mm*]

lemmas *dme* = *div-mult-mod-eq*
lemmas *dtle* = *xtr3* [*OF dme* [*symmetric*] *le-add1*]
lemmas *th2* = *order-trans* [*OF order-refl* [*THEN* [*2*] *mult-le-mono*] *dtle*]

lemma *td-gal*: $0 < c \Longrightarrow a \geq b * c \longleftrightarrow a \mathbin{div} c \geq b$
for *a b c* :: *nat*
apply *safe*
apply (*erule* (*1*) *xtr4* [*OF div-le-mono div-mult-self-is-m*])
apply (*erule th2*)
done

lemmas *td-gal-lt* = *td-gal* [*simplified not-less* [*symmetric*], *simplified*]

lemma *div-mult-le*: $a \mathbin{div} b * b \leq a$
for *a b* :: *nat*
by (*fact dtle*)

lemmas *sdl* = *split-div-lemma* [*THEN iffD1*, *symmetric*]

lemma *given-quot*: $f > 0 \Longrightarrow (f * l + (f - 1)) \mathbin{div} f = l$
for *f l* :: *nat*
by (*rule sdl*, *assumption*) (*simp* (*no-asm*))

lemma *given-quot-alt*: $f > 0 \Longrightarrow (l * f + f - Suc\ 0)\ div\ f = l$
for $f\ l$ :: *nat*
apply (*frule given-quot*)
apply (*rule trans*)
prefer *2*
apply (*erule asm-rl*)
apply (*rule-tac* $f{=}\lambda n.\ n\ div\ f$ **in** *arg-cong*)
apply (*simp add* : *ac-simps*)
done

lemma *diff-mod-le*: $a < d \Longrightarrow b\ dvd\ d \Longrightarrow a - a\ mod\ b \leq d - b$
for $a\ b\ d$ :: *nat*
apply (*unfold dvd-def*)
apply *clarify*
apply (*case-tac k*)
apply *clarsimp*
apply *clarify*
apply (*cases* $b > 0$)
apply (*drule mult.commute* [*THEN xtr1*])
apply (*frule* (*1*) *td-gal-lt* [*THEN iffD1*])
apply (*clarsimp simp*: *le-simps*)
apply (*rule minus-mod-eq-mult-div* [*symmetric*, *THEN* [*2*] *xtr4*])
apply (*rule mult-mono*)
apply *auto*
done

lemma *less-le-mult'*: $w * c < b * c \Longrightarrow 0 \leq c \Longrightarrow (w + 1) * c \leq b * c$
for $b\ c\ w$ :: *int*
apply (*rule mult-right-mono*)
apply (*rule zless-imp-add1-zle*)
apply (*erule* (*1*) *mult-right-less-imp-less*)
apply *assumption*
done

lemma *less-le-mult*: $w * c < b * c \Longrightarrow 0 \leq c \Longrightarrow w * c + c \leq b * c$
for $b\ c\ w$ :: *int*
using *less-le-mult'* [*of w c b*] **by** (*simp add*: *algebra-simps*)

lemmas *less-le-mult-minus* = *iffD2* [*OF le-diff-eq less-le-mult*,
simplified left-diff-distrib]

lemma *gen-minus*: $0 < n \Longrightarrow f\ n = f\ (Suc\ (n - 1))$
by *auto*

lemma *mpl-lem*: $j \leq i \Longrightarrow k < j \Longrightarrow i - j + k < i$
for $i\ j\ k$ :: *nat*
by *arith*

lemma *nonneg-mod-div*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq (a \bmod b) \wedge 0 \leq a \operatorname{div} b$
for $a\ b :: int$
by (*cases* $b = 0$) (*auto intro*: *pos-imp-zdiv-nonneg-iff* [*THEN iffD2*])

declare *iszero-0* [*intro*]

lemma *min-pm* [*simp*]: $min\ a\ b + (a - b) = a$
for $a\ b :: nat$
by *arith*

lemma *min-pm1* [*simp*]: $a - b + min\ a\ b = a$
for $a\ b :: nat$
by *arith*

lemma *rev-min-pm* [*simp*]: $min\ b\ a + (a - b) = a$
for $a\ b :: nat$
by *arith*

lemma *rev-min-pm1* [*simp*]: $a - b + min\ b\ a = a$
for $a\ b :: nat$
by *arith*

lemma *min-minus* [*simp*]: $min\ m\ (m - k) = m - k$
for $m\ k :: nat$
by *arith*

lemma *min-minus'* [*simp*]: $min\ (m - k)\ m = m - k$
for $m\ k :: nat$
by *arith*

end

10 A type of finite bit strings

theory *Word*
imports
HOL−Library.Type-Length
HOL−Library.Boolean-Algebra
Bits-Bit
Bool-List-Representation
Misc-Typedef
Word-Miscellaneous
begin

See `Examples/WordExamples.thy` for examples.

10.1 Type definition

typedef (**overloaded**) $'a\ word = \{(0::int)\ ..<\ 2\ \hat{}\ len\text{-}of\ TYPE('a::len0)\}$

morphisms *uint Abs-word* **by** *auto*

lemma *uint-nonnegative*: $0 \leq$ *uint w*
using *word.uint* [*of w*] **by** *simp*

lemma *uint-bounded*: *uint w* $<$ *2* ^ *len-of TYPE*('*a*)
for *w* :: '*a*::*len0 word*
using *word.uint* [*of w*] **by** *simp*

lemma *uint-idem*: *uint w mod 2* ^ *len-of TYPE*('*a*) = *uint w*
for *w* :: '*a*::*len0 word*
using *uint-nonnegative uint-bounded* **by** (*rule mod-pos-pos-trivial*)

lemma *word-uint-eq-iff*: *a* = *b* $\longleftrightarrow$ *uint a* = *uint b*
by (*simp add*: *uint-inject*)

lemma *word-uint-eqI*: *uint a* = *uint b* $\Longrightarrow$ *a* = *b*
by (*simp add*: *word-uint-eq-iff*)

definition *word-of-int* :: *int* $\Rightarrow$ '*a*::*len0 word*
— representation of words using unsigned or signed bins, only difference in these is the type class
where *word-of-int k* = *Abs-word* (*k mod 2* ^ *len-of TYPE*('*a*))

lemma *uint-word-of-int*: *uint* (*word-of-int k* :: '*a*::*len0 word*) = *k mod 2* ^ *len-of TYPE*('*a*)
by (*auto simp add*: *word-of-int-def intro*: *Abs-word-inverse*)

lemma *word-of-int-uint*: *word-of-int* (*uint w*) = *w*
by (*simp add*: *word-of-int-def uint-idem uint-inverse*)

lemma *split-word-all*: ($\bigwedge$*x*::'*a*::*len0 word*. *PROP P x*) $\equiv$ ($\bigwedge$*x*. *PROP P* (*word-of-int x*))
proof
fix *x* :: '*a word*
assume $\bigwedge$*x*. *PROP P* (*word-of-int x*)
then have *PROP P* (*word-of-int* (*uint x*)) **.**
then show *PROP P x* **by** (*simp add*: *word-of-int-uint*)
qed

10.2 Type conversions and casting

definition *sint* :: '*a*::*len word* $\Rightarrow$ *int*
— treats the most-significant-bit as a sign bit
where *sint-uint*: *sint w* = *sbintrunc* (*len-of TYPE*('*a*) − *1*) (*uint w*)

definition *unat* :: '*a*::*len0 word* $\Rightarrow$ *nat*
where *unat w* = *nat* (*uint w*)

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

definition *sints* :: *nat* ⇒ *int set*
where *sints n* = *range* (*sbintrunc* (*n* − *1*))

lemma *uints-num*: *uints n* = {*i*. *0* ≤ *i* ∧ *i* < *2* ^ *n*}
by (*simp add*: *uints-def range-bintrunc*)

lemma *sints-num*: *sints n* = {*i*. − (*2* ^ (*n* − *1*)) ≤ *i* ∧ *i* < *2* ^ (*n* − *1*)}
by (*simp add*: *sints-def range-sbintrunc*)

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

definition *norm-sint* :: *nat* ⇒ *int* ⇒ *int*
where *norm-sint n w* = (*w* + *2* ^ (*n* − *1*)) *mod 2* ^ *n* − *2* ^ (*n* − *1*)

definition *scast* :: *'a*::*len word* ⇒ *'b*::*len word*
— cast a word to a different length
where *scast w* = *word-of-int* (*sint w*)

definition *ucast* :: *'a*::*len0 word* ⇒ *'b*::*len0 word*
where *ucast w* = *word-of-int* (*uint w*)

instantiation *word* :: (*len0*) *size*
begin

definition *word-size*: *size* (*w* :: *'a word*) = *len-of TYPE*(*'a*)

instance ..

end

lemma *word-size-gt-0* [*iff*]: *0* < *size w*
for *w* :: *'a*::*len word*
by (*simp add*: *word-size*)

lemmas *lens-gt-0* = *word-size-gt-0 len-gt-0*

lemma *lens-not-0* [*iff*]:
fixes *w* :: *'a*::*len word*
shows *size w* ≠ *0*
and *len-of TYPE*(*'a*) ≠ *0*
by *auto*

definition *source-size* :: (*'a*::*len0 word* ⇒ *'b*) ⇒ *nat*
— whether a cast (or other) function is to a longer or shorter length

where [*code del*]: *source-size c* = (*let arb* = *undefined*; *x* = *c arb in size arb*)

definition *target-size* :: (*'a* ⇒ *'b*::*len0 word*) ⇒ *nat*
where [*code del*]: *target-size c* = *size* (*c undefined*)

definition *is-up* :: (*'a*::*len0 word* ⇒ *'b*::*len0 word*) ⇒ *bool*
where *is-up c* ⟷ *source-size c* ≤ *target-size c*

definition *is-down* :: (*'a*::*len0 word* ⇒ *'b*::*len0 word*) ⇒ *bool*
where *is-down c* ⟷ *target-size c* ≤ *source-size c*

definition *of-bl* :: *bool list* ⇒ *'a*::*len0 word*
where *of-bl bl* = *word-of-int* (*bl-to-bin bl*)

definition *to-bl* :: *'a*::*len0 word* ⇒ *bool list*
where *to-bl w* = *bin-to-bl* (*len-of TYPE*(*'a*)) (*uint w*)

definition *word-reverse* :: *'a*::*len0 word* ⇒ *'a word*
where *word-reverse w* = *of-bl* (*rev* (*to-bl w*))

definition *word-int-case* :: (*int* ⇒ *'b*) ⇒ *'a*::*len0 word* ⇒ *'b*
where *word-int-case f w* = *f* (*uint w*)

translations
case x of XCONST of-int y ⇒ *b* ⇌ *CONST word-int-case* (*λy*. *b*) *x*
case x of (*XCONST of-int* :: *'a*) *y* ⇒ *b* ↽ *CONST word-int-case* (*λy*. *b*) *x*

10.3 Correspondence relation for theorem transfer

definition *cr-word* :: *int* ⇒ *'a*::*len0 word* ⇒ *bool*
where *cr-word* = (*λx y*. *word-of-int x* = *y*)

lemma *Quotient-word*:
Quotient (*λx y*. *bintrunc* (*len-of TYPE*(*'a*)) *x* = *bintrunc* (*len-of TYPE*(*'a*)) *y*)
word-of-int uint (*cr-word* :: - ⇒ *'a*::*len0 word* ⇒ *bool*)
unfolding *Quotient-alt-def cr-word-def*
by (*simp add*: *no-bintr-alt1 word-of-int-uint*) (*simp add*: *word-of-int-def Abs-word-inject*)

lemma *reflp-word*:
reflp (*λx y*. *bintrunc* (*len-of TYPE*(*'a*::*len0*)) *x* = *bintrunc* (*len-of TYPE*(*'a*)) *y*)
by (*simp add*: *reflp-def*)

setup-lifting *Quotient-word reflp-word*

TODO: The next lemma could be generated automatically.

lemma *uint-transfer* [*transfer-rule*]:
(*rel-fun pcr-word op* =) (*bintrunc* (*len-of TYPE*(*'a*))) (*uint* :: *'a*::*len0 word* ⇒ *int*)

unfolding *rel-fun-def word.pcr-cr-eq cr-word-def*
by (*simp add*: *no-bintr-alt1 uint-word-of-int*)

10.4 Basic code generation setup

definition *Word* :: *int* ⇒ *'a*::*len0 word*
where [*code-post*]: *Word* = *word-of-int*

lemma [*code abstype*]: *Word* (*uint w*) = *w*
by (*simp add*: *Word-def word-of-int-uint*)

declare *uint-word-of-int* [*code abstract*]

instantiation *word* :: (*len0*) *equal*
begin

definition *equal-word* :: *'a word* ⇒ *'a word* ⇒ *bool*
where *equal-word k l* ⟷ *HOL.equal* (*uint k*) (*uint l*)

instance
by *standard* (*simp add*: *equal equal-word-def word-uint-eq-iff*)

end

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

instantiation *word* :: ({*len0*, *typerep*}) *random*
begin

definition
random-word i = *Random.range i* ∘→ (λ*k*. *Pair* (
let j = *word-of-int* (*int-of-integer* (*integer-of-natural k*)) :: *'a word*
in (*j*, λ-::*unit*. *Code-Evaluation.term-of j*)))

instance ..

end

no-notation *fcomp* (**infixl** ∘> *60*)
no-notation *scomp* (**infixl** ∘→ *60*)

10.5 Type-definition locale instantiations

lemmas *uint-0* = *uint-nonnegative*
lemmas *uint-lt* = *uint-bounded*
lemmas *uint-mod-same* = *uint-idem*

lemma *td-ext-uint*:
td-ext (*uint* :: *'a word* ⇒ *int*) *word-of-int* (*uints* (*len-of TYPE*(*'a*::*len0*)))

```
    (λw::int. w mod 2 ^ len-of TYPE('a))
  apply (unfold td-ext-def')
  apply (simp add: uints-num word-of-int-def bintrunc-mod2p)
  apply (simp add: uint-mod-same uint-0 uint-lt
                   word.uint-inverse word.Abs-word-inverse int-mod-lem)
  done

interpretation word-uint:
  td-ext
    uint::'a::len0 word ⇒ int
    word-of-int
    uints (len-of TYPE('a::len0))
    λw. w mod 2 ^ len-of TYPE('a::len0)
  by (fact td-ext-uint)

lemmas td-uint = word-uint.td-thm
lemmas int-word-uint = word-uint.eq-norm

lemma td-ext-ubin:
  td-ext (uint :: 'a word ⇒ int) word-of-int (uints (len-of TYPE('a::len0)))
    (bintrunc (len-of TYPE('a)))
  by (unfold no-bintr-alt1) (fact td-ext-uint)

interpretation word-ubin:
  td-ext
    uint::'a::len0 word ⇒ int
    word-of-int
    uints (len-of TYPE('a::len0))
    bintrunc (len-of TYPE('a::len0))
  by (fact td-ext-ubin)
```

10.6 Arithmetic operations

```
lift-definition word-succ :: 'a::len0 word ⇒ 'a word is λx. x + 1
  by (auto simp add: bintrunc-mod2p intro: mod-add-cong)

lift-definition word-pred :: 'a::len0 word ⇒ 'a word is λx. x - 1
  by (auto simp add: bintrunc-mod2p intro: mod-diff-cong)

instantiation word :: (len0) {neg-numeral, modulo, comm-monoid-mult, comm-ring}
begin

lift-definition zero-word :: 'a word is 0 .

lift-definition one-word :: 'a word is 1 .

lift-definition plus-word :: 'a word ⇒ 'a word ⇒ 'a word is op +
  by (auto simp add: bintrunc-mod2p intro: mod-add-cong)
```

lift-definition *minus-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op −*
by (*auto simp add*: *bintrunc-mod2p intro*: *mod-diff-cong*)

lift-definition *uminus-word* :: *'a word* ⇒ *'a word* **is** *uminus*
by (*auto simp add*: *bintrunc-mod2p intro*: *mod-minus-cong*)

lift-definition *times-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op ∗*
by (*auto simp add*: *bintrunc-mod2p intro*: *mod-mult-cong*)

definition *word-div-def*: *a div b* = *word-of-int* (*uint a div uint b*)

definition *word-mod-def*: *a mod b* = *word-of-int* (*uint a mod uint b*)

instance
by *standard* (*transfer*, *simp add*: *algebra-simps*)+

end

Legacy theorems:

lemma *word-arith-wis* [*code*]:
shows *word-add-def*: *a* + *b* = *word-of-int* (*uint a* + *uint b*)
and *word-sub-wi*: *a* − *b* = *word-of-int* (*uint a* − *uint b*)
and *word-mult-def*: *a* ∗ *b* = *word-of-int* (*uint a* ∗ *uint b*)
and *word-minus-def*: − *a* = *word-of-int* (− *uint a*)
and *word-succ-alt*: *word-succ a* = *word-of-int* (*uint a* + *1*)
and *word-pred-alt*: *word-pred a* = *word-of-int* (*uint a* − *1*)
and *word-0-wi*: *0* = *word-of-int 0*
and *word-1-wi*: *1* = *word-of-int 1*
unfolding *plus-word-def minus-word-def times-word-def uminus-word-def*
unfolding *word-succ-def word-pred-def zero-word-def one-word-def*
by *simp-all*

lemma *wi-homs*:
shows *wi-hom-add*: *word-of-int a* + *word-of-int b* = *word-of-int* (*a* + *b*)
and *wi-hom-sub*: *word-of-int a* − *word-of-int b* = *word-of-int* (*a* − *b*)
and *wi-hom-mult*: *word-of-int a* ∗ *word-of-int b* = *word-of-int* (*a* ∗ *b*)
and *wi-hom-neg*: − *word-of-int a* = *word-of-int* (− *a*)
and *wi-hom-succ*: *word-succ* (*word-of-int a*) = *word-of-int* (*a* + *1*)
and *wi-hom-pred*: *word-pred* (*word-of-int a*) = *word-of-int* (*a* − *1*)
by (*transfer*, *simp*)+

lemmas *wi-hom-syms* = *wi-homs* [*symmetric*]

lemmas *word-of-int-homs* = *wi-homs word-0-wi word-1-wi*

lemmas *word-of-int-hom-syms* = *word-of-int-homs* [*symmetric*]

instance *word* :: (*len*) *comm-ring-1*
proof

have $*$: *0 < len-of TYPE('a)* **by** (*rule len-gt-0*)
show (*0::'a word*) $\neq$ *1*
by *transfer* (*use* $*$ **in** ‹*auto simp add*: *gr0-conv-Suc*›)
qed

lemma *word-of-nat*: *of-nat n = word-of-int* (*int n*)
by (*induct n*) (*auto simp add* : *word-of-int-hom-syms*)

lemma *word-of-int*: *of-int = word-of-int*
apply (*rule ext*)
apply (*case-tac x rule*: *int-diff-cases*)
apply (*simp add*: *word-of-nat wi-hom-sub*)
done

definition *udvd* :: *'a::len word* $\Rightarrow$ *'a::len word* $\Rightarrow$ *bool* (**infixl** *udvd 50*)
where *a udvd b* = ($\exists$ *n*$\geq$*0*. *uint b = n * uint a*)

10.7 Ordering

instantiation *word* :: (*len0*) *linorder*
begin

definition *word-le-def*: *a* $\leq$ *b* $\longleftrightarrow$ *uint a* $\leq$ *uint b*

definition *word-less-def*: *a < b* $\longleftrightarrow$ *uint a < uint b*

instance
by *standard* (*auto simp*: *word-less-def word-le-def*)

end

definition *word-sle* :: *'a::len word* $\Rightarrow$ *'a word* $\Rightarrow$ *bool* ((-/ <=*s* -) [*50*, *51*] *50*)
where *a <=s b* $\longleftrightarrow$ *sint a* $\leq$ *sint b*

definition *word-sless* :: *'a::len word* $\Rightarrow$ *'a word* $\Rightarrow$ *bool* ((-/ <*s* -) [*50*, *51*] *50*)
where *x <s y* $\longleftrightarrow$ *x <=s y* $\wedge$ *x* $\neq$ *y*

10.8 Bit-wise operations

instantiation *word* :: (*len0*) *bits*
begin

lift-definition *bitNOT-word* :: *'a word* $\Rightarrow$ *'a word* **is** *bitNOT*
by (*metis bin-trunc-not*)

lift-definition *bitAND-word* :: *'a word* $\Rightarrow$ *'a word* $\Rightarrow$ *'a word* **is** *bitAND*
by (*metis bin-trunc-and*)

lift-definition *bitOR-word* :: *'a word* $\Rightarrow$ *'a word* $\Rightarrow$ *'a word* **is** *bitOR*
by (*metis bin-trunc-or*)

lift-definition *bitXOR-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitXOR*
by (*metis bin-trunc-xor*)

definition *word-test-bit-def*: *test-bit a* = *bin-nth* (*uint a*)

definition *word-set-bit-def*: *set-bit a n x* = *word-of-int* (*bin-sc n x* (*uint a*))

definition *word-set-bits-def*: (*BITS n. f n*) = *of-bl* (*bl-of-nth* (*len-of TYPE*(*'a*))
f)

definition *word-lsb-def*: *lsb a* ⟷ *bin-last* (*uint a*)

definition *shiftl1* :: *'a word* ⇒ *'a word*
where *shiftl1 w* = *word-of-int* (*uint w BIT False*)

definition *shiftr1* :: *'a word* ⇒ *'a word*
— shift right as unsigned, ie logical or arithmetic
where *shiftr1 w* = *word-of-int* (*bin-rest* (*uint w*))

definition *shiftl-def*: *w* << *n* = (*shiftl1* ^^ *n*) *w*

definition *shiftr-def*: *w* >> *n* = (*shiftr1* ^^ *n*) *w*

instance ..

end

lemma [*code*]:
shows *word-not-def*: *NOT* (*a*::*'a*::*len0 word*) = *word-of-int* (*NOT* (*uint a*))
and *word-and-def*: (*a*::*'a word*) *AND b* = *word-of-int* (*uint a AND uint b*)
and *word-or-def*: (*a*::*'a word*) *OR b* = *word-of-int* (*uint a OR uint b*)
and *word-xor-def*: (*a*::*'a word*) *XOR b* = *word-of-int* (*uint a XOR uint b*)
by (*simp-all add*: *bitNOT-word-def bitAND-word-def bitOR-word-def bitXOR-word-def*)

instantiation *word* :: (*len*) *bitss*
begin

definition *word-msb-def*: *msb a* ⟷ *bin-sign* (*sint a*) = −*1*

instance ..

end

definition *setBit* :: *'a*::*len0 word* ⇒ *nat* ⇒ *'a word*
where *setBit w n* = *set-bit w n True*

definition *clearBit* :: *'a*::*len0 word* ⇒ *nat* ⇒ *'a word*
where *clearBit w n* = *set-bit w n False*

10.9 Shift operations

definition *sshiftr1* :: *'a::len word ⇒ 'a word*
where *sshiftr1 w = word-of-int (bin-rest (sint w))*

definition *bshiftr1* :: *bool ⇒ 'a::len word ⇒ 'a word*
where *bshiftr1 b w = of-bl (b # butlast (to-bl w))*

definition *sshiftr* :: *'a::len word ⇒ nat ⇒ 'a word* (**infixl** *>>>* *55*)
where *w >>> n = (sshiftr1 ^^ n) w*

definition *mask* :: *nat ⇒ 'a::len word*
where *mask n = (1 << n) − 1*

definition *revcast* :: *'a::len0 word ⇒ 'b::len0 word*
where *revcast w = of-bl (takefill False (len-of TYPE('b)) (to-bl w))*

definition *slice1* :: *nat ⇒ 'a::len0 word ⇒ 'b::len0 word*
where *slice1 n w = of-bl (takefill False n (to-bl w))*

definition *slice* :: *nat ⇒ 'a::len0 word ⇒ 'b::len0 word*
where *slice n w = slice1 (size w − n) w*

10.10 Rotation

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*

definition *word-rotr* :: *nat ⇒ 'a::len0 word ⇒ 'a::len0 word*
where *word-rotr n w = of-bl (rotater n (to-bl w))*

definition *word-rotl* :: *nat ⇒ 'a::len0 word ⇒ 'a::len0 word*
where *word-rotl n w = of-bl (rotate n (to-bl w))*

definition *word-roti* :: *int ⇒ 'a::len0 word ⇒ 'a::len0 word*
where *word-roti i w =*
(if i ≥ 0 then word-rotr (nat i) w else word-rotl (nat (− i)) w)

10.11 Split and cat operations

definition *word-cat* :: *'a::len0 word ⇒ 'b::len0 word ⇒ 'c::len0 word*
where *word-cat a b = word-of-int (bin-cat (uint a) (len-of TYPE('b)) (uint b))*

definition *word-split* :: *'a::len0 word ⇒ 'b::len0 word × 'c::len0 word*
where *word-split a =*
(case bin-split (len-of TYPE('c)) (uint a) of

(*u*, *v*) ⇒ (*word-of-int u*, *word-of-int v*))

definition *word-rcat* :: *'a*::*len0 word list* ⇒ *'b*::*len0 word*
where *word-rcat ws* = *word-of-int* (*bin-rcat* (*len-of TYPE*(*'a*)) (*map uint ws*))

definition *word-rsplit* :: *'a*::*len0 word* ⇒ *'b*::*len word list*
where *word-rsplit w* = *map word-of-int* (*bin-rsplit* (*len-of TYPE*(*'b*)) (*len-of TYPE*(*'a*), *uint w*))

definition *max-word* :: *'a*::*len word*
— Largest representable machine integer.
where *max-word* = *word-of-int* (*2* ^ *len-of TYPE*(*'a*) − *1*)

lemmas *of-nth-def* = *word-set-bits-def*

10.12 Theorems about typedefs

lemma *sint-sbintrunc'*: *sint* (*word-of-int bin* :: *'a word*) = *sbintrunc* (*len-of TYPE*(*'a*::*len*) − *1*) *bin*
by (*auto simp*: *sint-uint word-ubin.eq-norm sbintrunc-bintrunc-lt*)

lemma *uint-sint*: *uint w* = *bintrunc* (*len-of TYPE*(*'a*)) (*sint w*)
for *w* :: *'a*::*len word*
by (*auto simp*: *sint-uint bintrunc-sbintrunc-le*)

lemma *bintr-uint*: *len-of TYPE*(*'a*) ≤ *n* ⟹ *bintrunc n* (*uint w*) = *uint w*
for *w* :: *'a*::*len0 word*
apply (*subst word-ubin.norm-Rep* [*symmetric*])
apply (*simp only*: *bintrunc-bintrunc-min word-size*)
apply (*simp add*: *min.absorb2*)
done

lemma *wi-bintr*:
len-of TYPE(*'a*::*len0*) ≤ *n* ⟹
word-of-int (*bintrunc n w*) = (*word-of-int w* :: *'a word*)
by (*auto simp*: *word-ubin.norm-eq-iff* [*symmetric*] *min.absorb1*)

lemma *td-ext-sbin*:
td-ext (*sint* :: *'a word* ⇒ *int*) *word-of-int* (*sints* (*len-of TYPE*(*'a*::*len*)))
(*sbintrunc* (*len-of TYPE*(*'a*) − *1*))
apply (*unfold td-ext-def' sint-uint*)
apply (*simp add* : *word-ubin.eq-norm*)
apply (*cases len-of TYPE*(*'a*))
apply (*auto simp add* : *sints-def*)
apply (*rule sym* [*THEN trans*])
apply (*rule word-ubin.Abs-norm*)
apply (*simp only*: *bintrunc-sbintrunc*)
apply (*drule sym*)
apply *simp*

done

lemma *td-ext-sint*:
td-ext (*sint* :: *'a word* ⇒ *int*) *word-of-int* (*sints* (*len-of TYPE*(*'a*::*len*)))
(λ*w*. (*w* + *2* ^ (*len-of TYPE*(*'a*) − *1*)) *mod 2* ^ *len-of TYPE*(*'a*) −
2 ^ (*len-of TYPE*(*'a*) − *1*))
using *td-ext-sbin* [**where** *?'a* = *'a*] **by** (*simp add*: *no-sbintr-alt2*)

interpretation *word-sint*:
td-ext
sint ::*'a*::*len word* ⇒ *int*
word-of-int
sints (*len-of TYPE*(*'a*::*len*))
λ*w*. (*w* + *2*^(*len-of TYPE*(*'a*::*len*) − *1*)) *mod 2*^*len-of TYPE*(*'a*::*len*) −
2 ^ (*len-of TYPE*(*'a*::*len*) − *1*)
by (*rule td-ext-sint*)

interpretation *word-sbin*:
td-ext
sint ::*'a*::*len word* ⇒ *int*
word-of-int
sints (*len-of TYPE*(*'a*::*len*))
sbintrunc (*len-of TYPE*(*'a*::*len*) − *1*)
by (*rule td-ext-sbin*)

lemmas *int-word-sint* = *td-ext-sint* [*THEN td-ext.eq-norm*]

lemmas *td-sint* = *word-sint.td*

lemma *to-bl-def'*: (*to-bl* :: *'a*::*len0 word* ⇒ *bool list*) = *bin-to-bl* (*len-of TYPE*(*'a*))
∘ *uint*
by (*auto simp*: *to-bl-def*)

lemmas *word-reverse-no-def* [*simp*] =
word-reverse-def [*of numeral w*] **for** *w*

lemma *uints-mod*: *uints n* = *range* (λ*w*. *w mod 2* ^ *n*)
by (*fact uints-def* [*unfolded no-bintr-alt1*])

lemma *word-numeral-alt*: *numeral b* = *word-of-int* (*numeral b*)
by (*induct b*, *simp-all only*: *numeral.simps word-of-int-homs*)

declare *word-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *word-neg-numeral-alt*: − *numeral b* = *word-of-int* (− *numeral b*)
by (*simp only*: *word-numeral-alt wi-hom-neg*)

declare *word-neg-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *word-numeral-transfer* [*transfer-rule*]:
(rel-fun op = pcr-word) numeral numeral
(rel-fun op = pcr-word) (− numeral) (− numeral)
apply (*simp-all add*: *rel-fun-def word.pcr-cr-eq cr-word-def*)
using *word-numeral-alt* [*symmetric*] *word-neg-numeral-alt* [*symmetric*] **by** *auto*

lemma *uint-bintrunc* [*simp*]:
uint (numeral bin :: 'a word) =
bintrunc (len-of TYPE('a::len0)) (numeral bin)
unfolding *word-numeral-alt* **by** (*rule word-ubin.eq-norm*)

lemma *uint-bintrunc-neg* [*simp*]:
uint (− numeral bin :: 'a word) = bintrunc (len-of TYPE('a::len0)) (− numeral bin)
by (*simp only*: *word-neg-numeral-alt word-ubin.eq-norm*)

lemma *sint-sbintrunc* [*simp*]:
sint (numeral bin :: 'a word) = sbintrunc (len-of TYPE('a::len) − 1) (numeral bin)
by (*simp only*: *word-numeral-alt word-sbin.eq-norm*)

lemma *sint-sbintrunc-neg* [*simp*]:
sint (− numeral bin :: 'a word) = sbintrunc (len-of TYPE('a::len) − 1) (− numeral bin)
by (*simp only*: *word-neg-numeral-alt word-sbin.eq-norm*)

lemma *unat-bintrunc* [*simp*]:
unat (numeral bin :: 'a::len0 word) = nat (bintrunc (len-of TYPE('a)) (numeral bin))
by (*simp only*: *unat-def uint-bintrunc*)

lemma *unat-bintrunc-neg* [*simp*]:
unat (− numeral bin :: 'a::len0 word) = nat (bintrunc (len-of TYPE('a)) (− numeral bin))
by (*simp only*: *unat-def uint-bintrunc-neg*)

lemma *size-0-eq*: *size w = 0 ⟹ v = w*
for *v w :: 'a::len0 word*
apply (*unfold word-size*)
apply (*rule word-uint.Rep-eqD*)
apply (*rule box-equals*)
defer
apply (*rule word-ubin.norm-Rep*)+
apply *simp*
done

lemma *uint-ge-0* [*iff*]: *0 ≤ uint x*
for *x :: 'a::len0 word*

using *word-uint.Rep* [*of x*] **by** (*simp add*: *uints-num*)

lemma *uint-lt2p* [*iff*]: *uint x < 2 ^ len-of TYPE('a)*
for *x* :: *'a::len0 word*
using *word-uint.Rep* [*of x*] **by** (*simp add*: *uints-num*)

lemma *sint-ge*: *- (2 ^ (len-of TYPE('a) - 1)) ≤ sint x*
for *x* :: *'a::len word*
using *word-sint.Rep* [*of x*] **by** (*simp add*: *sints-num*)

lemma *sint-lt*: *sint x < 2 ^ (len-of TYPE('a) - 1)*
for *x* :: *'a::len word*
using *word-sint.Rep* [*of x*] **by** (*simp add*: *sints-num*)

lemma *sign-uint-Pls* [*simp*]: *bin-sign (uint x) = 0*
by (*simp add*: *sign-Pls-ge-0*)

lemma *uint-m2p-neg*: *uint x - 2 ^ len-of TYPE('a) < 0*
for *x* :: *'a::len0 word*
by (*simp only*: *diff-less-0-iff-less uint-lt2p*)

lemma *uint-m2p-not-non-neg*: *¬ 0 ≤ uint x - 2 ^ len-of TYPE('a)*
for *x* :: *'a::len0 word*
by (*simp only*: *not-le uint-m2p-neg*)

lemma *lt2p-lem*: *len-of TYPE('a) ≤ n ⟹ uint w < 2 ^ n*
for *w* :: *'a::len0 word*
by (*metis bintr-uint bintrunc-mod2p int-mod-lem zless2p*)

lemma *uint-le-0-iff* [*simp*]: *uint x ≤ 0 ⟷ uint x = 0*
by (*fact uint-ge-0* [*THEN leD*, *THEN linorder-antisym-conv1*])

lemma *uint-nat*: *uint w = int (unat w)*
by (*auto simp*: *unat-def*)

lemma *uint-numeral*: *uint (numeral b :: 'a::len0 word) = numeral b mod 2 ^ len-of TYPE('a)*
by (*simp only*: *word-numeral-alt int-word-uint*)

lemma *uint-neg-numeral*: *uint (- numeral b :: 'a::len0 word) = - numeral b mod 2 ^ len-of TYPE('a)*
by (*simp only*: *word-neg-numeral-alt int-word-uint*)

lemma *unat-numeral*: *unat (numeral b :: 'a::len0 word) = numeral b mod 2 ^ len-of TYPE('a)*
apply (*unfold unat-def*)
apply (*clarsimp simp only*: *uint-numeral*)
apply (*rule nat-mod-distrib* [*THEN trans*])
apply (*rule zero-le-numeral*)

apply (*simp-all add*: *nat-power-eq*)
done

lemma *sint-numeral*:
sint (*numeral b* :: *'a*::*len word*) =
(*numeral b* +
2 ^ (*len-of TYPE*(*'a*) − *1*)) *mod 2* ^ *len-of TYPE*(*'a*) −
2 ^ (*len-of TYPE*(*'a*) − *1*)
unfolding *word-numeral-alt* **by** (*rule int-word-sint*)

lemma *word-of-int-0* [*simp*, *code-post*]: *word-of-int 0* = *0*
unfolding *word-0-wi* **..**

lemma *word-of-int-1* [*simp*, *code-post*]: *word-of-int 1* = *1*
unfolding *word-1-wi* **..**

lemma *word-of-int-neg-1* [*simp*]: *word-of-int* (− *1*) = − *1*
by (*simp add*: *wi-hom-syms*)

lemma *word-of-int-numeral* [*simp*] : (*word-of-int* (*numeral bin*) :: *'a*::*len0 word*)
= *numeral bin*
by (*simp only*: *word-numeral-alt*)

lemma *word-of-int-neg-numeral* [*simp*]:
(*word-of-int* (− *numeral bin*) :: *'a*::*len0 word*) = − *numeral bin*
by (*simp only*: *word-numeral-alt wi-hom-syms*)

lemma *word-int-case-wi*:
word-int-case f (*word-of-int i* :: *'b word*) = *f* (*i mod 2* ^ *len-of TYPE*(*'b*::*len0*))
by (*simp add*: *word-int-case-def word-uint.eq-norm*)

lemma *word-int-split*:
P (*word-int-case f x*) =
(∀ *i*. *x* = (*word-of-int i* :: *'b*::*len0 word*) ∧ *0* ≤ *i* ∧ *i* < *2* ^ *len-of TYPE*(*'b*)
⟶ *P* (*f i*))
by (*auto simp*: *word-int-case-def word-uint.eq-norm mod-pos-pos-trivial*)

lemma *word-int-split-asm*:
P (*word-int-case f x*) =
(∄ *n*. *x* = (*word-of-int n* :: *'b*::*len0 word*) ∧ *0* ≤ *n* ∧ *n* < *2* ^ *len-of TYPE*(*'b*::*len0*)
∧ ¬ *P* (*f n*))
by (*auto simp*: *word-int-case-def word-uint.eq-norm mod-pos-pos-trivial*)

lemmas *uint-range'* = *word-uint.Rep* [*unfolded uints-num mem-Collect-eq*]
lemmas *sint-range'* = *word-sint.Rep* [*unfolded One-nat-def sints-num mem-Collect-eq*]

lemma *uint-range-size*: *0* ≤ *uint w* ∧ *uint w* < *2* ^ *size w*
unfolding *word-size* **by** (*rule uint-range'*)

lemma *sint-range-size*: $- (2 \hat{} (size\ w - Suc\ 0)) \leq sint\ w \wedge sint\ w < 2 \hat{} (size\ w - Suc\ 0)$
unfolding *word-size* **by** (*rule sint-range'*)

lemma *sint-above-size*: $2 \hat{} (size\ w - 1) \leq x \Longrightarrow sint\ w < x$
for $w :: 'a::len\ word$
unfolding *word-size* **by** (*rule less-le-trans* [*OF sint-lt*])

lemma *sint-below-size*: $x \leq - (2 \hat{} (size\ w - 1)) \Longrightarrow x \leq sint\ w$
for $w :: 'a::len\ word$
unfolding *word-size* **by** (*rule order-trans* [*OF - sint-ge*])

10.13 Testing bits

lemma *test-bit-eq-iff*: $test\text{-}bit\ u = test\text{-}bit\ v \longleftrightarrow u = v$
for $u\ v :: 'a::len0\ word$
unfolding *word-test-bit-def* **by** (*simp add*: *bin-nth-eq-iff*)

lemma *test-bit-size* [*rule-format*] : $w\ !!\ n \longrightarrow n < size\ w$
for $w :: 'a::len0\ word$
apply (*unfold word-test-bit-def*)
apply (*subst word-ubin.norm-Rep* [*symmetric*])
apply (*simp only*: *nth-bintr word-size*)
apply *fast*
done

lemma *word-eq-iff*: $x = y \longleftrightarrow (\forall n{<}len\text{-}of\ TYPE('a).\ x\ !!\ n = y\ !!\ n)$
for $x\ y :: 'a::len0\ word$
unfolding *uint-inject* [*symmetric*] *bin-eq-iff word-test-bit-def* [*symmetric*]
by (*metis test-bit-size* [*unfolded word-size*])

lemma *word-eqI*: $(\bigwedge n.\ n < size\ u \longrightarrow u\ !!\ n = v\ !!\ n) \Longrightarrow u = v$
for $u :: 'a::len0\ word$
by (*simp add*: *word-size word-eq-iff*)

lemma *word-eqD*: $u = v \Longrightarrow u\ !!\ x = v\ !!\ x$
for $u\ v :: 'a::len0\ word$
by *simp*

lemma *test-bit-bin'*: $w\ !!\ n \longleftrightarrow n < size\ w \wedge bin\text{-}nth\ (uint\ w)\ n$
by (*simp add*: *word-test-bit-def word-size nth-bintr* [*symmetric*])

lemmas *test-bit-bin* = *test-bit-bin'* [*unfolded word-size*]

lemma *bin-nth-uint-imp*: $bin\text{-}nth\ (uint\ w)\ n \Longrightarrow n < len\text{-}of\ TYPE('a)$
for $w :: 'a::len0\ word$
apply (*rule nth-bintr* [*THEN iffD1*, *THEN conjunct1*])
apply (*subst word-ubin.norm-Rep*)
apply *assumption*

done

lemma *bin-nth-sint*:
len-of TYPE('a) ≤ n ⟹
bin-nth (sint w) n = bin-nth (sint w) (len-of TYPE('a) − 1)
for *w :: 'a::len word*
apply (*subst word-sbin.norm-Rep* [*symmetric*])
apply (*auto simp add*: *nth-sbintr*)
done

lemma *td-bl*:
type-definition
(*to-bl :: 'a::len0 word ⇒ bool list*)
of-bl
{*bl. length bl = len-of TYPE('a)*}
apply (*unfold type-definition-def of-bl-def to-bl-def*)
apply (*simp add*: *word-ubin.eq-norm*)
apply *safe*
apply (*drule sym*)
apply *simp*
done

interpretation *word-bl*:
type-definition
to-bl :: 'a::len0 word ⇒ bool list
of-bl
{*bl. length bl = len-of TYPE('a::len0)*}
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 *to-bl-word-rev*: *to-bl* (*word-reverse w*) = *rev* (*to-bl w*)
by (*simp add*: *word-reverse-def word-bl.Abs-inverse*)

lemma *word-rev-rev* [*simp*] : *word-reverse* (*word-reverse w*) = *w*
by (*simp add*: *word-reverse-def word-bl.Abs-inverse*)

lemma *word-rev-gal*: *word-reverse w = u ⟹ word-reverse u = w*
by (*metis word-rev-rev*)

lemma *word-rev-gal'*: *u = word-reverse w ⟹ w = word-reverse u*

by *simp*

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-bl-sign-sint*: *hd* (*to-bl w*) = (*bin-sign* (*sint w*) = *−1*)
apply (*unfold to-bl-def sint-uint*)
apply (*rule trans* [*OF - bl-sbin-sign*])
apply *simp*
done

lemma *of-bl-drop'*:
lend = *length bl − len-of TYPE*('*a*::*len0*) ⟹
of-bl (*drop lend bl*) = (*of-bl bl* :: *'a word*)
by (*auto simp*: *of-bl-def trunc-bl2bin* [*symmetric*])

lemma *test-bit-of-bl*:
(*of-bl bl*::*'a*::*len0 word*) !! *n* = (*rev bl* ! *n* ∧ *n < len-of TYPE*('*a*) ∧ *n < length bl*)
by (*auto simp add*: *of-bl-def word-test-bit-def word-size word-ubin.eq-norm nth-bintr bin-nth-of-bl*)

lemma *no-of-bl*: (*numeral bin* ::*'a*::*len0 word*) = *of-bl* (*bin-to-bl* (*len-of TYPE*('*a*)) (*numeral bin*))
by (*simp add*: *of-bl-def*)

lemma *uint-bl*: *to-bl w* = *bin-to-bl* (*size w*) (*uint w*)
by (*auto simp*: *word-size to-bl-def*)

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*::*len0 word*) = *bin-to-bl* (*len-of TYPE*('*a*)) *bin*
by (*auto simp*: *uint-bl word-ubin.eq-norm word-size*)

lemma *to-bl-numeral* [*simp*]:
to-bl (*numeral bin*::*'a*::*len0 word*) =
bin-to-bl (*len-of TYPE*('*a*)) (*numeral bin*)
unfolding *word-numeral-alt* **by** (*rule to-bl-of-bin*)

lemma *to-bl-neg-numeral* [*simp*]:
to-bl (− *numeral bin*::*'a*::*len0 word*) =
bin-to-bl (*len-of TYPE*(*'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* (*len-of TYPE*(*'a*)) (*uint x*)) = *uint x*
for *x* :: *'a*::*len0 word*
by (*rule trans* [*OF bin-bl-bin word-ubin.norm-Rep*])

lemma *uints-unats*: *uints n* = *int* ' *unats n*
apply (*unfold unats-def uints-num*)
apply *safe*
apply (*rule-tac image-eqI*)
apply (*erule-tac nat-0-le* [*symmetric*])
apply *auto*
apply (*erule-tac nat-less-iff* [*THEN iffD2*])
apply (*rule-tac* [*2*] *zless-nat-eq-int-zless* [*THEN iffD1*])
apply (*auto simp*: *nat-power-eq*)
done

lemma *unats-uints*: *unats n* = *nat* ' *uints n*
by (*auto simp*: *uints-unats image-iff*)

lemmas *bintr-num* =
word-ubin.norm-eq-iff [*of numeral a numeral b*, *symmetric*, *folded word-numeral-alt*]
for *a b*
lemmas *sbintr-num* =
word-sbin.norm-eq-iff [*of numeral a numeral b*, *symmetric*, *folded word-numeral-alt*]
for *a b*

lemma *num-of-bintr'*:
bintrunc (*len-of TYPE*(*'a*::*len0*)) (*numeral a*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: *'a word*)
unfolding *bintr-num* **by** (*erule subst*, *simp*)

lemma *num-of-sbintr'*:
sbintrunc (*len-of TYPE*(*'a*::*len*) − *1*) (*numeral a*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: *'a word*)
unfolding *sbintr-num* **by** (*erule subst*, *simp*)

lemma *num-abs-bintr*:
(*numeral x* :: *'a word*) =
word-of-int (*bintrunc* (*len-of TYPE*(*'a*::*len0*)) (*numeral x*))
by (*simp only*: *word-ubin.Abs-norm word-numeral-alt*)

lemma *num-abs-sbintr*:
(*numeral x* :: *'a word*) =
word-of-int (*sbintrunc* (*len-of TYPE('a::len)* − *1*) (*numeral x*))
by (*simp only*: *word-sbin.Abs-norm word-numeral-alt*)

lemma *ucast-id*: *ucast w* = *w*
by (*auto simp*: *ucast-def*)

lemma *scast-id*: *scast w* = *w*
by (*auto simp*: *scast-def*)

lemma *ucast-bl*: *ucast w* = *of-bl* (*to-bl w*)
by (*auto simp*: *ucast-def of-bl-def uint-bl word-size*)

lemma *nth-ucast*: (*ucast w::'a::len0 word*) !! *n* = (*w* !! *n* ∧ *n* < *len-of TYPE('a)*)
by (*simp add*: *ucast-def test-bit-bin word-ubin.eq-norm nth-bintr word-size*)
(*fast elim*!: *bin-nth-uint-imp*)

lemma *ucast-bintr* [*simp*]:
ucast (*numeral w* :: *'a::len0 word*) =
word-of-int (*bintrunc* (*len-of TYPE('a)*) (*numeral w*))
by (*simp add*: *ucast-def*)

lemma *scast-sbintr* [*simp*]:
scast (*numeral w* ::*'a::len word*) =
word-of-int (*sbintrunc* (*len-of TYPE('a)* − *Suc 0*) (*numeral w*))
by (*simp add*: *scast-def*)

lemma *source-size*: *source-size* (*c::'a::len0 word* ⇒ -) = *len-of TYPE('a)*
unfolding *source-size-def word-size Let-def* **..**

lemma *target-size*: *target-size* (*c::-* ⇒ *'b::len0 word*) = *len-of TYPE('b)*
unfolding *target-size-def word-size Let-def* **..**

lemma *is-down*: *is-down c* ⟷ *len-of TYPE('b)* ≤ *len-of TYPE('a)*
for *c* :: *'a::len0 word* ⇒ *'b::len0 word*
by (*simp only*: *is-down-def source-size target-size*)

lemma *is-up*: *is-up c* ⟷ *len-of TYPE('a)* ≤ *len-of TYPE('b)*
for *c* :: *'a::len0 word* ⇒ *'b::len0 word*
by (*simp only*: *is-up-def source-size target-size*)

lemmas *is-up-down = trans [OF is-up is-down [symmetric]]*

lemma *down-cast-same [OF refl]*: *uc = ucast ⟹ is-down uc ⟹ uc = scast*
apply (*unfold is-down*)
apply *safe*
apply (*rule ext*)
apply (*unfold ucast-def scast-def uint-sint*)
apply (*rule word-ubin.norm-eq-iff [THEN iffD1]*)
apply *simp*
done

lemma *word-rev-tf*:
to-bl (of-bl bl::'a::len0 word) =
rev (takefill False (len-of TYPE('a)) (rev bl))
by (*auto simp*: *of-bl-def uint-bl bl-bin-bl-rtf word-ubin.eq-norm word-size*)

lemma *word-rep-drop*:
to-bl (of-bl bl::'a::len0 word) =
replicate (len-of TYPE('a) − length bl) False @
drop (length bl − len-of TYPE('a)) bl
by (*simp add*: *word-rev-tf takefill-alt rev-take*)

lemma *to-bl-ucast*:
to-bl (ucast (w::'b::len0 word) ::'a::len0 word) =
replicate (len-of TYPE('a) − len-of TYPE('b)) False @
drop (len-of TYPE('b) − len-of TYPE('a)) (to-bl w)
apply (*unfold ucast-bl*)
apply (*rule trans*)
apply (*rule word-rep-drop*)
apply *simp*
done

lemma *ucast-up-app [OF refl]*:
uc = ucast ⟹ source-size uc + n = target-size uc ⟹
to-bl (uc w) = replicate n False @ (to-bl w)
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 *sint-up-scast* [*OF refl*]: *sc* = *scast* ⟹ *is-up sc* ⟹ *sint* (*sc w*) = *sint w*
apply (*unfold is-up*)
apply *safe*
apply (*simp add*: *scast-def word-sbin.eq-norm*)
apply (*rule box-equals*)
prefer *3*
apply (*rule word-sbin.norm-Rep*)
apply (*rule sbintrunc-sbintrunc-l*)
defer
apply (*subst word-sbin.norm-Rep*)
apply (*rule refl*)
apply *simp*
done

lemma *uint-up-ucast* [*OF refl*]: *uc* = *ucast* ⟹ *is-up uc* ⟹ *uint* (*uc w*) = *uint w*
apply (*unfold is-up*)
apply *safe*
apply (*rule bin-eqI*)
apply (*fold word-test-bit-def*)
apply (*auto simp add*: *nth-ucast*)
apply (*auto simp add*: *test-bit-bin*)
done

lemma *ucast-up-ucast* [*OF refl*]: *uc* = *ucast* ⟹ *is-up uc* ⟹ *ucast* (*uc w*) = *ucast w*
apply (*simp* (*no-asm*) *add*: *ucast-def*)
apply (*clarsimp simp add*: *uint-up-ucast*)
done

lemma *scast-up-scast* [*OF refl*]: *sc* = *scast* ⟹ *is-up sc* ⟹ *scast* (*sc w*) = *scast w*
apply (*simp* (*no-asm*) *add*: *scast-def*)
apply (*clarsimp simp add*: *sint-up-scast*)
done

lemma *ucast-of-bl-up* [*OF refl*]: *w* = *of-bl bl* ⟹ *size bl* ≤ *size w* ⟹ *ucast w* = *of-bl bl*
by (*auto simp add* : *nth-ucast word-size test-bit-of-bl intro*!: *word-eqI*)

lemmas *ucast-up-ucast-id* = *trans* [*OF ucast-up-ucast ucast-id*]
lemmas *scast-up-scast-id* = *trans* [*OF scast-up-scast scast-id*]

lemmas *isduu* = *is-up-down* [**where** *c* = *ucast*, *THEN iffD2*]
lemmas *isdus* = *is-up-down* [**where** *c* = *scast*, *THEN iffD2*]

lemmas *ucast-down-ucast-id* = *isduu* [*THEN ucast-up-ucast-id*]
lemmas *scast-down-scast-id* = *isdus* [*THEN ucast-up-ucast-id*]

lemma *up-ucast-surj*:
is-up (*ucast* :: *'b*::*len0 word* ⇒ *'a*::*len0 word*) ⟹
surj (*ucast* :: *'a word* ⇒ *'b word*)
by (*rule surjI*) (*erule ucast-up-ucast-id*)

lemma *up-scast-surj*:
is-up (*scast* :: *'b*::*len word* ⇒ *'a*::*len word*) ⟹
surj (*scast* :: *'a word* ⇒ *'b word*)
by (*rule surjI*) (*erule scast-up-scast-id*)

lemma *down-scast-inj*:
is-down (*scast* :: *'b*::*len word* ⇒ *'a*::*len word*) ⟹
inj-on (*ucast* :: *'a word* ⇒ *'b word*) *A*
by (*rule inj-on-inverseI*, *erule scast-down-scast-id*)

lemma *down-ucast-inj*:
is-down (*ucast* :: *'b*::*len0 word* ⇒ *'a*::*len0 word*) ⟹
inj-on (*ucast* :: *'a word* ⇒ *'b word*) *A*
by (*rule inj-on-inverseI*) (*erule ucast-down-ucast-id*)

lemma *of-bl-append-same*: *of-bl* (*X* @ *to-bl w*) = *w*
by (*rule word-bl.Rep-eqD*) (*simp add*: *word-rep-drop*)

lemma *ucast-down-wi* [*OF refl*]: *uc* = *ucast* ⟹ *is-down uc* ⟹ *uc* (*word-of-int x*) = *word-of-int x*
apply (*unfold is-down*)
apply (*clarsimp simp add*: *ucast-def word-ubin.eq-norm*)
apply (*rule word-ubin.norm-eq-iff* [*THEN iffD1*])
apply (*erule bintrunc-bintrunc-ge*)
done

lemma *ucast-down-no* [*OF refl*]: *uc* = *ucast* ⟹ *is-down uc* ⟹ *uc* (*numeral bin*) = *numeral bin*
unfolding *word-numeral-alt* **by** *clarify* (*rule ucast-down-wi*)

lemma *ucast-down-bl* [*OF refl*]: *uc* = *ucast* ⟹ *is-down uc* ⟹ *uc* (*of-bl bl*) = *of-bl bl*
unfolding *of-bl-def* **by** *clarify* (*erule ucast-down-wi*)

lemmas *slice-def'* = *slice-def* [*unfolded word-size*]
lemmas *test-bit-def'* = *word-test-bit-def* [*THEN fun-cong*]

lemmas *word-log-defs* = *word-and-def word-or-def word-xor-def word-not-def*

10.14 Word Arithmetic

lemma *word-less-alt*: $a < b \longleftrightarrow uint\ a < uint\ b$
by (*fact word-less-def*)

lemma *signed-linorder*: *class.linorder word-sle word-sless*
by *standard* (*auto simp*: *word-sle-def word-sless-def*)

interpretation *signed*: *linorder word-sle word-sless*
by (*rule signed-linorder*)

lemma *udvdI*: $0 \leq n \Longrightarrow uint\ b = n * uint\ a \Longrightarrow a\ udvd\ b$
by (*auto simp*: *udvd-def*)

lemmas *word-div-no* [*simp*] = *word-div-def* [*of numeral a numeral b*] **for** *a b*
lemmas *word-mod-no* [*simp*] = *word-mod-def* [*of numeral a numeral b*] **for** *a b*
lemmas *word-less-no* [*simp*] = *word-less-def* [*of numeral a numeral b*] **for** *a b*
lemmas *word-le-no* [*simp*] = *word-le-def* [*of numeral a numeral b*] **for** *a b*
lemmas *word-sless-no* [*simp*] = *word-sless-def* [*of numeral a numeral b*] **for** *a b*
lemmas *word-sle-no* [*simp*] = *word-sle-def* [*of numeral a numeral b*] **for** *a b*

lemma *word-m1-wi*: $-1 = word\text{-}of\text{-}int\ (-1)$
by (*simp add*: *word-neg-numeral-alt* [*of Num.One*])

lemma *word-0-bl* [*simp*]: *of-bl* [] = *0*
by (*simp add*: *of-bl-def*)

lemma *word-1-bl*: *of-bl* [*True*] = *1*
by (*simp add*: *of-bl-def bl-to-bin-def*)

lemma *uint-eq-0* [*simp*]: *uint 0* = *0*
unfolding *word-0-wi word-ubin.eq-norm* **by** *simp*

lemma *of-bl-0* [*simp*]: *of-bl* (*replicate n False*) = *0*
by (*simp add*: *of-bl-def bl-to-bin-rep-False*)

lemma *to-bl-0* [*simp*]: *to-bl* (*0*::*'a*::*len0 word*) = *replicate* (*len-of TYPE*(*'a*)) *False*
by (*simp add*: *uint-bl word-size bin-to-bl-zero*)

lemma *uint-0-iff*: $uint\ x = 0 \longleftrightarrow x = 0$
by (*simp add*: *word-uint-eq-iff*)

lemma *unat-0-iff*: $unat\ x = 0 \longleftrightarrow x = 0$
by (*auto simp*: *unat-def nat-eq-iff uint-0-iff*)

lemma *unat-0* [*simp*]: *unat 0* = *0*
by (*auto simp*: *unat-def*)

lemma *size-0-same'*: *size w* = *0* $\Longrightarrow w = v$
for *v w* :: *'a*::*len0 word*

apply (*unfold word-size*)
apply (*rule box-equals*)
defer
apply (*rule word-uint.Rep-inverse*)+
apply (*rule word-ubin.norm-eq-iff* [*THEN iffD1*])
apply *simp*
done

lemmas *size-0-same* = *size-0-same'* [*unfolded word-size*]

lemmas *unat-eq-0* = *unat-0-iff*
lemmas *unat-eq-zero* = *unat-0-iff*

lemma *unat-gt-0*: $0 < unat\ x \longleftrightarrow x \neq 0$
by (*auto simp*: *unat-0-iff* [*symmetric*])

lemma *ucast-0* [*simp*]: *ucast 0* = *0*
by (*simp add*: *ucast-def*)

lemma *sint-0* [*simp*]: *sint 0* = *0*
by (*simp add*: *sint-uint*)

lemma *scast-0* [*simp*]: *scast 0* = *0*
by (*simp add*: *scast-def*)

lemma *sint-n1* [*simp*] : *sint* (− *1*) = − *1*
by (*simp only*: *word-m1-wi word-sbin.eq-norm*) *simp*

lemma *scast-n1* [*simp*]: *scast* (− *1*) = − *1*
by (*simp add*: *scast-def*)

lemma *uint-1* [*simp*]: *uint* (*1*::*'a*::*len word*) = *1*
by (*simp only*: *word-1-wi word-ubin.eq-norm*) (*simp add*: *bintrunc-minus-simps*(*4*))

lemma *unat-1* [*simp*]: *unat* (*1*::*'a*::*len word*) = *1*
by (*simp add*: *unat-def*)

lemma *ucast-1* [*simp*]: *ucast* (*1*::*'a*::*len word*) = *1*
by (*simp add*: *ucast-def*)

10.15 Transferring goals from words to ints

lemma *word-ths*:
shows *word-succ-p1*: *word-succ a* = *a* + *1*
and *word-pred-m1*: *word-pred a* = *a* − *1*
and *word-pred-succ*: *word-pred* (*word-succ a*) = *a*
and *word-succ-pred*: *word-succ* (*word-pred a*) = *a*
and *word-mult-succ*: *word-succ a* ∗ *b* = *b* + *a* ∗ *b*
by (*transfer*, *simp add*: *algebra-simps*)+

lemma *uint-cong*: $x = y \Longrightarrow uint\ x = uint\ y$
by *simp*

lemma *uint-word-ariths*:
fixes $a\ b :: 'a::len0\ word$
shows $uint\ (a + b) = (uint\ a + uint\ b)\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a::len0)$
and $uint\ (a - b) = (uint\ a - uint\ b)\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a)$
and $uint\ (a * b) = uint\ a * uint\ b\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a)$
and $uint\ (-\ a) = -\ uint\ a\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a)$
and $uint\ (word\text{-}succ\ a) = (uint\ a + 1)\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a)$
and $uint\ (word\text{-}pred\ a) = (uint\ a - 1)\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a)$
and $uint\ (0 :: 'a\ word) = 0\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a)$
and $uint\ (1 :: 'a\ word) = 1\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a)$
by (*simp-all add*: *word-arith-wis* [*THEN trans* [*OF uint-cong int-word-uint*]])

lemma *uint-word-arith-bintrs*:
fixes $a\ b :: 'a::len0\ word$
shows $uint\ (a + b) = bintrunc\ (len\text{-}of\ TYPE('a))\ (uint\ a + uint\ b)$
and $uint\ (a - b) = bintrunc\ (len\text{-}of\ TYPE('a))\ (uint\ a - uint\ b)$
and $uint\ (a * b) = bintrunc\ (len\text{-}of\ TYPE('a))\ (uint\ a * uint\ b)$
and $uint\ (-\ a) = bintrunc\ (len\text{-}of\ TYPE('a))\ (-\ uint\ a)$
and $uint\ (word\text{-}succ\ a) = bintrunc\ (len\text{-}of\ TYPE('a))\ (uint\ a + 1)$
and $uint\ (word\text{-}pred\ a) = bintrunc\ (len\text{-}of\ TYPE('a))\ (uint\ a - 1)$
and $uint\ (0 :: 'a\ word) = bintrunc\ (len\text{-}of\ TYPE('a))\ 0$
and $uint\ (1 :: 'a\ word) = bintrunc\ (len\text{-}of\ TYPE('a))\ 1$
by (*simp-all add*: *uint-word-ariths bintrunc-mod2p*)

lemma *sint-word-ariths*:
fixes $a\ b :: 'a::len\ word$
shows $sint\ (a + b) = sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ (sint\ a + sint\ b)$
and $sint\ (a - b) = sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ (sint\ a - sint\ b)$
and $sint\ (a * b) = sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ (sint\ a * sint\ b)$
and $sint\ (-\ a) = sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ (-\ sint\ a)$
and $sint\ (word\text{-}succ\ a) = sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ (sint\ a + 1)$
and $sint\ (word\text{-}pred\ a) = sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ (sint\ a - 1)$
and $sint\ (0 :: 'a\ word) = sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ 0$
and $sint\ (1 :: 'a\ word) = sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ 1$
apply (*simp-all only*: *word-sbin.inverse-norm* [*symmetric*])
apply (*simp-all add*: *wi-hom-syms*)
apply *transfer* **apply** *simp*
apply *transfer* **apply** *simp*
done

lemmas *uint-div-alt* = *word-div-def* [*THEN trans* [*OF uint-cong int-word-uint*]]
lemmas *uint-mod-alt* = *word-mod-def* [*THEN trans* [*OF uint-cong int-word-uint*]]

lemma *word-pred-0-n1*: *word-pred 0* = *word-of-int* (− *1*)
unfolding *word-pred-m1* **by** *simp*

lemma *succ-pred-no* [*simp*]:
word-succ (*numeral w*) = *numeral w* + *1*
word-pred (*numeral w*) = *numeral w* − *1*
word-succ (− *numeral w*) = − *numeral w* + *1*
word-pred (− *numeral w*) = − *numeral w* − *1*
by (*simp-all add*: *word-succ-p1 word-pred-m1*)

lemma *word-sp-01* [*simp*]:
word-succ (− *1*) = *0* ∧ *word-succ 0* = *1* ∧ *word-pred 0* = − *1* ∧ *word-pred 1* = *0*
by (*simp-all add*: *word-succ-p1 word-pred-m1*)

lemma *word-of-int-Ex*: ∃ *y*. *x* = *word-of-int y*
by (*rule-tac x*=*uint x* **in** *exI*) *simp*

10.16 Order on fixed-length words

lemma *word-zero-le* [*simp*]: *0* ≤ *y*
for *y* :: *'a*::*len0 word*
unfolding *word-le-def* **by** *auto*

lemma *word-m1-ge* [*simp*] : *word-pred 0* ≥ *y*
by (*simp only*: *word-le-def word-pred-0-n1 word-uint.eq-norm m1mod2k*) *auto*

lemma *word-n1-ge* [*simp*]: *y* ≤ −*1*
for *y* :: *'a*::*len0 word*
by (*simp only*: *word-le-def word-m1-wi word-uint.eq-norm m1mod2k*) *auto*

lemmas *word-not-simps* [*simp*] =
word-zero-le [*THEN leD*] *word-m1-ge* [*THEN leD*] *word-n1-ge* [*THEN leD*]

lemma *word-gt-0*: *0* < *y* ⟷ *0* ≠ *y*
for *y* :: *'a*::*len0 word*
by (*simp add*: *less-le*)

lemmas *word-gt-0-no* [*simp*] = *word-gt-0* [*of numeral y*] **for** *y*

lemma *word-sless-alt*: *a* <*s b* ⟷ *sint a* < *sint b*
by (*auto simp add*: *word-sle-def word-sless-def less-le*)

lemma *word-le-nat-alt*: *a* ≤ *b* ⟷ *unat a* ≤ *unat b*
unfolding *unat-def word-le-def*
by (*rule nat-le-eq-zle* [*symmetric*]) *simp*

lemma *word-less-nat-alt*: *a* < *b* ⟷ *unat a* < *unat b*
unfolding *unat-def word-less-alt*
by (*rule nat-less-eq-zless* [*symmetric*]) *simp*

lemma *wi-less*:
(*word-of-int n* < (*word-of-int m* :: *'a*::*len0 word*)) =
(*n mod 2* ^ *len-of TYPE*(*'a*) < *m mod 2* ^ *len-of TYPE*(*'a*))
unfolding *word-less-alt* **by** (*simp add*: *word-uint.eq-norm*)

lemma *wi-le*:
(*word-of-int n* ≤ (*word-of-int m* :: *'a*::*len0 word*)) =
(*n mod 2* ^ *len-of TYPE*(*'a*) ≤ *m mod 2* ^ *len-of TYPE*(*'a*))
unfolding *word-le-def* **by** (*simp add*: *word-uint.eq-norm*)

lemma *udvd-nat-alt*: *a udvd b* ⟷ (∃ *n*≥*0*. *unat b* = *n* ∗ *unat a*)
apply (*unfold udvd-def*)
apply *safe*
apply (*simp add*: *unat-def nat-mult-distrib*)
apply (*simp add*: *uint-nat*)
apply (*rule exI*)
apply *safe*
prefer *2*
apply (*erule notE*)
apply (*rule refl*)
apply *force*
done

lemma *udvd-iff-dvd*: *x udvd y* ⟷ *unat x dvd unat y*
unfolding *dvd-def udvd-nat-alt* **by** *force*

lemmas *unat-mono* = *word-less-nat-alt* [*THEN iffD1*]

lemma *unat-minus-one*:
assumes *w* ≠ *0*
shows *unat* (*w* − *1*) = *unat w* − *1*
proof −
have *0* ≤ *uint w* **by** (*fact uint-nonnegative*)
moreover from *assms* **have** *0* ≠ *uint w*
by (*simp add*: *uint-0-iff*)
ultimately have *1* ≤ *uint w*
by *arith*
from *uint-lt2p* [*of w*] **have** *uint w* − *1* < *2* ^ *len-of TYPE*(*'a*)
by *arith*
with ‹*1* ≤ *uint w*› **have** (*uint w* − *1*) *mod 2* ^ *len-of TYPE*(*'a*) = *uint w* − *1*
by (*auto intro*: *mod-pos-pos-trivial*)
with ‹*1* ≤ *uint w*› **have** *nat* ((*uint w* − *1*) *mod 2* ^ *len-of TYPE*(*'a*)) = *nat* (*uint w*) − *1*
by *auto*
then show *?thesis*
by (*simp only*: *unat-def int-word-uint word-arith-wis mod-diff-right-eq*)
qed

lemma *measure-unat*: $p \neq 0 \Longrightarrow unat\ (p - 1) < unat\ p$
by (*simp add*: *unat-minus-one*) (*simp add*: *unat-0-iff* [*symmetric*])

lemmas *uint-add-ge0* [*simp*] = *add-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]
lemmas *uint-mult-ge0* [*simp*] = *mult-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]

lemma *uint-sub-lt2p* [*simp*]: $uint\ x - uint\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
for $x :: 'a$::*len0 word* **and** $y :: 'b$::*len0 word*
using *uint-ge-0* [*of y*] *uint-lt2p* [*of x*] **by** *arith*

10.17 Conditions for the addition (etc) of two words to overflow

lemma *uint-add-lem*:
$(uint\ x + uint\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a)) =$
$(uint\ (x + y) = uint\ x + uint\ y)$
for $x\ y :: 'a$::*len0 word*
by (*unfold uint-word-ariths*) (*auto intro*!: *trans* [*OF* - *int-mod-lem*])

lemma *uint-mult-lem*:
$(uint\ x * uint\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a)) =$
$(uint\ (x * y) = uint\ x * uint\ y)$
for $x\ y :: 'a$::*len0 word*
by (*unfold uint-word-ariths*) (*auto intro*!: *trans* [*OF* - *int-mod-lem*])

lemma *uint-sub-lem*: $uint\ x \geq uint\ y \longleftrightarrow uint\ (x - y) = uint\ x - uint\ y$
by (*auto simp*: *uint-word-ariths intro*!: *trans* [*OF* - *int-mod-lem*])

lemma *uint-add-le*: $uint\ (x + y) \leq uint\ x + uint\ y$
unfolding *uint-word-ariths* **by** (*metis uint-add-ge0 zmod-le-nonneg-dividend*)

lemma *uint-sub-ge*: $uint\ (x - y) \geq uint\ x - uint\ y$
unfolding *uint-word-ariths* **by** (*metis int-mod-ge uint-sub-lt2p zless2p*)

lemma *mod-add-if-z*:
$x < z \Longrightarrow y < z \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x \Longrightarrow 0 \leq z \Longrightarrow$
$(x + y)\ mod\ z = (if\ x + y < z\ then\ x + y\ else\ x + y - z)$
for $x\ y\ z :: int$
by (*auto intro*: *int-mod-eq*)

lemma *uint-plus-if'*:
$uint\ (a + b) =$
$(if\ uint\ a + uint\ b < 2\ \hat{}\ len\text{-}of\ TYPE('a)\ then\ uint\ a + uint\ b$
$else\ uint\ a + uint\ b - 2\ \hat{}\ len\text{-}of\ TYPE('a))$
for $a\ b :: 'a$::*len0 word*
using *mod-add-if-z* [*of uint a* - *uint b*] **by** (*simp add*: *uint-word-ariths*)

lemma *mod-sub-if-z*:
$x < z \Longrightarrow y < z \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x \Longrightarrow 0 \leq z \Longrightarrow$

$(x - y)\ mod\ z = (if\ y \leq x\ then\ x - y\ else\ x - y + z)$
for $x\ y\ z :: int$
by (*auto intro*: *int-mod-eq*)

lemma *uint-sub-if'*:
$uint\ (a - b) =$
$(if\ uint\ b \leq uint\ a\ then\ uint\ a - uint\ b$
$else\ uint\ a - uint\ b + 2\ \hat{}\ len\text{-}of\ TYPE('a))$
for $a\ b :: 'a::len0\ word$
using *mod-sub-if-z* [*of uint a - uint b*] **by** (*simp add*: *uint-word-ariths*)

10.18 Definition of *uint-arith*

lemma *word-of-int-inverse*:
$word\text{-}of\text{-}int\ r = a \Longrightarrow 0 \leq r \Longrightarrow r < 2\ \hat{}\ len\text{-}of\ TYPE('a) \Longrightarrow uint\ a = r$
for $a :: 'a::len0\ word$
apply (*erule word-uint.Abs-inverse'* [*rotated*])
apply (*simp add*: *uints-num*)
done

lemma *uint-split*:
$P\ (uint\ x) = (\forall i.\ word\text{-}of\text{-}int\ i = x \wedge 0 \leq i \wedge i < 2\hat{}len\text{-}of\ TYPE('a) \longrightarrow P\ i)$
for $x :: 'a::len0\ word$
apply (*fold word-int-case-def*)
apply (*auto dest*!: *word-of-int-inverse simp*: *int-word-uint mod-pos-pos-trivial*
split: *word-int-split*)
done

lemma *uint-split-asm*:
$P\ (uint\ x) = (\nexists i.\ word\text{-}of\text{-}int\ i = x \wedge 0 \leq i \wedge i < 2\hat{}len\text{-}of\ TYPE('a) \wedge \neg\ P\ i)$
for $x :: 'a::len0\ word$
by (*auto dest*!: *word-of-int-inverse*
simp: *int-word-uint mod-pos-pos-trivial*
split: *uint-split*)

lemmas *uint-splits* = *uint-split uint-split-asm*

lemmas *uint-arith-simps* =
word-le-def word-less-alt
word-uint.Rep-inject [*symmetric*]
uint-sub-if' uint-plus-if'

lemma *power-False-cong*: $False \Longrightarrow a\ \hat{}\ b = c\ \hat{}\ d$
by *auto*

ML ‹
fun uint-arith-simpset ctxt =

```
  ctxt addsimps @{thms uint-arith-simps}
    delsimps @{thms word-uint.Rep-inject}
    |> fold Splitter.add-split @{thms if-split-asm}
    |> fold Simplifier.add-cong @{thms power-False-cong}

fun uint-arith-tacs ctxt =
  let
    fun arith-tac' n t =
      Arith-Data.arith-tac ctxt n t
        handle Cooper.COOPER - => Seq.empty;
  in
    [ clarify-tac ctxt 1,
      full-simp-tac (uint-arith-simpset ctxt) 1,
      ALLGOALS (full-simp-tac
        (put-simpset HOL-ss ctxt
          |> fold Splitter.add-split @{thms uint-splits}
          |> fold Simplifier.add-cong @{thms power-False-cong})),
      rewrite-goals-tac ctxt @{thms word-size},
      ALLGOALS (fn n => REPEAT (resolve-tac ctxt [allI, impI] n) THEN
                        REPEAT (eresolve-tac ctxt [conjE] n) THEN
                        REPEAT (dresolve-tac ctxt @{thms word-of-int-inverse} n
                              THEN assume-tac ctxt n
                              THEN assume-tac ctxt n)),
      TRYALL arith-tac' ]
  end

fun uint-arith-tac ctxt = SELECT-GOAL (EVERY (uint-arith-tacs ctxt))
›

method-setup uint-arith =
  ‹Scan.succeed (SIMPLE-METHOD' o uint-arith-tac)›
  solving word arithmetic via integers and arith
```

10.19 More on overflows and monotonicity

```
lemma no-plus-overflow-uint-size: x ≤ x + y ⟷ uint x + uint y < 2 ^ size x
  for x y :: 'a::len0 word
  unfolding word-size by uint-arith

lemmas no-olen-add = no-plus-overflow-uint-size [unfolded word-size]

lemma no-ulen-sub: x ≥ x - y ⟷ uint y ≤ uint x
  for x y :: 'a::len0 word
  by uint-arith

lemma no-olen-add': x ≤ y + x ⟷ uint y + uint x < 2 ^ len-of TYPE('a)
  for x y :: 'a::len0 word
  by (simp add: ac-simps no-olen-add)
```

lemmas *olen-add-eqv* = *trans* [*OF no-olen-add no-olen-add'* [*symmetric*]]

lemmas *uint-plus-simple-iff* = *trans* [*OF no-olen-add uint-add-lem*]
lemmas *uint-plus-simple* = *uint-plus-simple-iff* [*THEN iffD1*]
lemmas *uint-minus-simple-iff* = *trans* [*OF no-ulen-sub uint-sub-lem*]
lemmas *uint-minus-simple-alt* = *uint-sub-lem* [*folded word-le-def*]
lemmas *word-sub-le-iff* = *no-ulen-sub* [*folded word-le-def*]
lemmas *word-sub-le* = *word-sub-le-iff* [*THEN iffD2*]

lemma *word-less-sub1*: $x \neq 0 \Longrightarrow 1 < x \longleftrightarrow 0 < x - 1$
for $x :: 'a::len\ word$
by *uint-arith*

lemma *word-le-sub1*: $x \neq 0 \Longrightarrow 1 \leq x \longleftrightarrow 0 \leq x - 1$
for $x :: 'a::len\ word$
by *uint-arith*

lemma *sub-wrap-lt*: $x < x - z \longleftrightarrow x < z$
for $x\ z :: 'a::len0\ word$
by *uint-arith*

lemma *sub-wrap*: $x \leq x - z \longleftrightarrow z = 0 \vee x < z$
for $x\ z :: 'a::len0\ word$
by *uint-arith*

lemma *plus-minus-not-NULL-ab*: $x \leq ab - c \Longrightarrow c \leq ab \Longrightarrow c \neq 0 \Longrightarrow x + c \neq 0$
for $x\ ab\ c :: 'a::len0\ word$
by *uint-arith*

lemma *plus-minus-no-overflow-ab*: $x \leq ab - c \Longrightarrow c \leq ab \Longrightarrow x \leq x + c$
for $x\ ab\ c :: 'a::len0\ word$
by *uint-arith*

lemma *le-minus'*: $a + c \leq b \Longrightarrow a \leq a + c \Longrightarrow c \leq b - a$
for $a\ b\ c :: 'a::len0\ word$
by *uint-arith*

lemma *le-plus'*: $a \leq b \Longrightarrow c \leq b - a \Longrightarrow a + c \leq b$
for $a\ b\ c :: 'a::len0\ word$
by *uint-arith*

lemmas *le-plus* = *le-plus'* [*rotated*]

lemmas *le-minus* = *leD* [*THEN thin-rl*, *THEN le-minus'*]

lemma *word-plus-mono-right*: $y \leq z \Longrightarrow x \leq x + z \Longrightarrow x + y \leq x + z$
for $x\ y\ z :: 'a::len0\ word$
by *uint-arith*

lemma *word-less-minus-cancel*: $y - x < z - x \Longrightarrow x \leq z \Longrightarrow y < z$
for *x y z* :: *'a*::*len0 word*
by *uint-arith*

lemma *word-less-minus-mono-left*: $y < z \Longrightarrow x \leq y \Longrightarrow y - x < z - x$
for *x y z* :: *'a*::*len0 word*
by *uint-arith*

lemma *word-less-minus-mono*: $a < c \Longrightarrow d < b \Longrightarrow a - b < a \Longrightarrow c - d < c \Longrightarrow a - b < c - d$
for *a b c d* :: *'a*::*len word*
by *uint-arith*

lemma *word-le-minus-cancel*: $y - x \leq z - x \Longrightarrow x \leq z \Longrightarrow y \leq z$
for *x y z* :: *'a*::*len0 word*
by *uint-arith*

lemma *word-le-minus-mono-left*: $y \leq z \Longrightarrow x \leq y \Longrightarrow y - x \leq z - x$
for *x y z* :: *'a*::*len0 word*
by *uint-arith*

lemma *word-le-minus-mono*:
$a \leq c \Longrightarrow d \leq b \Longrightarrow a - b \leq a \Longrightarrow c - d \leq c \Longrightarrow a - b \leq c - d$
for *a b c d* :: *'a*::*len word*
by *uint-arith*

lemma *plus-le-left-cancel-wrap*: $x + y' < x \Longrightarrow x + y < x \Longrightarrow x + y' < x + y \longleftrightarrow y' < y$
for *x y y'* :: *'a*::*len0 word*
by *uint-arith*

lemma *plus-le-left-cancel-nowrap*: $x \leq x + y' \Longrightarrow x \leq x + y \Longrightarrow x + y' < x + y \longleftrightarrow y' < y$
for *x y y'* :: *'a*::*len0 word*
by *uint-arith*

lemma *word-plus-mono-right2*: $a \leq a + b \Longrightarrow c \leq b \Longrightarrow a \leq a + c$
for *a b c* :: *'a*::*len0 word*
by *uint-arith*

lemma *word-less-add-right*: $x < y - z \Longrightarrow z \leq y \Longrightarrow x + z < y$
for *x y z* :: *'a*::*len0 word*
by *uint-arith*

lemma *word-less-sub-right*: $x < y + z \Longrightarrow y \leq x \Longrightarrow x - y < z$
for *x y z* :: *'a*::*len0 word*
by *uint-arith*

lemma *word-le-plus-either*: $x \leq y \vee x \leq z \Longrightarrow y \leq y + z \Longrightarrow x \leq y + z$
for *x y z* :: *'a::len0 word*
by *uint-arith*

lemma *word-less-nowrapI*: $x < z - k \Longrightarrow k \leq z \Longrightarrow 0 < k \Longrightarrow x < x + k$
for *x z k* :: *'a::len0 word*
by *uint-arith*

lemma *inc-le*: $i < m \Longrightarrow i + 1 \leq m$
for *i m* :: *'a::len word*
by *uint-arith*

lemma *inc-i*: $1 \leq i \Longrightarrow i < m \Longrightarrow 1 \leq i + 1 \wedge i + 1 \leq m$
for *i m* :: *'a::len word*
by *uint-arith*

lemma *udvd-incr-lem*:
$up < uq \Longrightarrow up = ua + n * uint\ K \Longrightarrow$
$uq = ua + n' * uint\ K \Longrightarrow up + uint\ K \leq uq$
apply *clarsimp*
apply (*drule less-le-mult*)
apply *safe*
done

lemma *udvd-incr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
$uint\ q = ua + n' * uint\ K \Longrightarrow p + K \leq q$
apply (*unfold word-less-alt word-le-def*)
apply (*drule* (*2*) *udvd-incr-lem*)
apply (*erule uint-add-le* [*THEN order-trans*])
done

lemma *udvd-decr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
$uint\ q = ua + n' * uint\ K \Longrightarrow p \leq q - K$
apply (*unfold word-less-alt word-le-def*)
apply (*drule* (*2*) *udvd-incr-lem*)
apply (*drule le-diff-eq* [*THEN iffD2*])
apply (*erule order-trans*)
apply (*rule uint-sub-ge*)
done

lemmas *udvd-incr-lem0* = *udvd-incr-lem* [**where** *ua=0*, *unfolded add-0-left*]
lemmas *udvd-incr0* = *udvd-incr'* [**where** *ua=0*, *unfolded add-0-left*]
lemmas *udvd-decr0* = *udvd-decr'* [**where** *ua=0*, *unfolded add-0-left*]

lemma *udvd-minus-le'*: $xy < k \Longrightarrow z\ udvd\ xy \Longrightarrow z\ udvd\ k \Longrightarrow xy \leq k - z$
apply (*unfold udvd-def*)
apply *clarify*

apply (*erule* (*2*) *udvd-decr0*)
done

lemma *udvd-incr2-K*:
$p < a + s \Longrightarrow a \leq a + s \Longrightarrow K\ udvd\ s \Longrightarrow K\ udvd\ p - a \Longrightarrow a \leq p \Longrightarrow$
$0 < K \Longrightarrow p \leq p + K \wedge p + K \leq a + s$
supply [[*simproc del*: *linordered-ring-less-cancel-factor*]]
apply (*unfold udvd-def*)
apply *clarify*
apply (*simp add*: *uint-arith-simps split*: *if-split-asm*)
prefer *2*
apply (*insert uint-range'* [*of s*])[*1*]
apply *arith*
apply (*drule add.commute* [*THEN xtr1*])
apply (*simp add*: *diff-less-eq* [*symmetric*])
apply (*drule less-le-mult*)
apply *arith*
apply *simp*
done

lemma *word-succ-rbl*: *to-bl w* = *bl* $\Longrightarrow$ *to-bl* (*word-succ w*) = *rev* (*rbl-succ* (*rev bl*))
apply (*unfold word-succ-def*)
apply *clarify*
apply (*simp add*: *to-bl-of-bin*)
apply (*simp add*: *to-bl-def rbl-succ*)
done

lemma *word-pred-rbl*: *to-bl w* = *bl* $\Longrightarrow$ *to-bl* (*word-pred w*) = *rev* (*rbl-pred* (*rev bl*))
apply (*unfold word-pred-def*)
apply *clarify*
apply (*simp add*: *to-bl-of-bin*)
apply (*simp add*: *to-bl-def rbl-pred*)
done

lemma *word-add-rbl*:
to-bl v = *vbl* $\Longrightarrow$ *to-bl w* = *wbl* $\Longrightarrow$
to-bl (*v* + *w*) = *rev* (*rbl-add* (*rev vbl*) (*rev wbl*))
apply (*unfold word-add-def*)
apply *clarify*
apply (*simp add*: *to-bl-of-bin*)
apply (*simp add*: *to-bl-def rbl-add*)
done

lemma *word-mult-rbl*:
to-bl v = *vbl* $\Longrightarrow$ *to-bl w* = *wbl* $\Longrightarrow$
to-bl (*v* * *w*) = *rev* (*rbl-mult* (*rev vbl*) (*rev wbl*))

apply (*unfold word-mult-def*)
apply *clarify*
apply (*simp add*: *to-bl-of-bin*)
apply (*simp add*: *to-bl-def 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*)

10.20 Arithmetic type class instantiations

lemmas *word-le-0-iff* [*simp*] =
word-zero-le [*THEN leD*, *THEN linorder-antisym-conv1*]

lemma *word-of-int-nat*: *0* ≤ *x* ⟹ *word-of-int x* = *of-nat* (*nat x*)
by (*simp add*: *word-of-int*)

lemma *iszero-word-no* [*simp*]:
iszero (*numeral bin* :: *'a*::*len word*) =
iszero (*bintrunc* (*len-of TYPE*(*'a*)) (*numeral bin*))
using *word-ubin.norm-eq-iff* [**where** *'a*=*'a*, *of numeral bin 0*]
by (*simp add*: *iszero-def* [*symmetric*])

Use *iszero* to simplify equalities between word numerals.

lemmas *word-eq-numeral-iff-iszero* [*simp*] =
eq-numeral-iff-iszero [**where** *'a*=*'a*::*len word*]

10.21 Word and nat

lemma *td-ext-unat* [*OF refl*]:
n = *len-of TYPE*(*'a*::*len*) ⟹
td-ext (*unat* :: *'a word* ⇒ *nat*) *of-nat* (*unats n*) (λ*i*. *i mod 2* ^ *n*)
apply (*unfold td-ext-def' unat-def word-of-nat unats-uints*)
apply (*auto intro*!: *imageI simp add* : *word-of-int-hom-syms*)
apply (*erule word-uint.Abs-inverse* [*THEN arg-cong*])
apply (*simp add*: *int-word-uint nat-mod-distrib nat-power-eq*)
done

lemmas *unat-of-nat* = *td-ext-unat* [*THEN td-ext.eq-norm*]

interpretation *word-unat*:
td-ext

unat::*'a*::*len word* ⇒ *nat*
of-nat
unats (*len-of TYPE*(*'a*::*len*))
λ*i*. *i mod* 2 ^ *len-of TYPE*(*'a*::*len*)
by (*rule td-ext-unat*)

lemmas *td-unat* = *word-unat.td-thm*

lemmas *unat-lt2p* [*iff*] = *word-unat.Rep* [*unfolded unats-def mem-Collect-eq*]

lemma *unat-le*: *y* ≤ *unat z* ⟹ *y* ∈ *unats* (*len-of TYPE*(*'a*))
for *z* :: *'a*::*len word*
apply (*unfold unats-def*)
apply *clarsimp*
apply (*rule xtrans*, *rule unat-lt2p*, *assumption*)
done

lemma *word-nchotomy*: ∀ *w* :: *'a*::*len word*. ∃ *n*. *w* = *of-nat n* ∧ *n* < 2 ^ *len-of TYPE*(*'a*)
apply (*rule allI*)
apply (*rule word-unat.Abs-cases*)
apply (*unfold unats-def*)
apply *auto*
done

lemma *of-nat-eq*: *of-nat n* = *w* ⟷ (∃ *q*. *n* = *unat w* + *q* ∗ 2 ^ *len-of TYPE*(*'a*))
for *w* :: *'a*::*len word*
apply (*rule trans*)
apply (*rule word-unat.inverse-norm*)
apply (*rule iffI*)
apply (*rule mod-eqD*)
apply *simp*
apply *clarsimp*
done

lemma *of-nat-eq-size*: *of-nat n* = *w* ⟷ (∃ *q*. *n* = *unat w* + *q* ∗ 2 ^ *size w*)
unfolding *word-size* **by** (*rule of-nat-eq*)

lemma *of-nat-0*: *of-nat m* = (0::*'a*::*len word*) ⟷ (∃ *q*. *m* = *q* ∗ 2 ^ *len-of TYPE*(*'a*))
by (*simp add*: *of-nat-eq*)

lemma *of-nat-2p* [*simp*]: *of-nat* (2 ^ *len-of TYPE*(*'a*)) = (0::*'a*::*len word*)
by (*fact mult-1* [*symmetric*, *THEN iffD2* [*OF of-nat-0 exI*]])

lemma *of-nat-gt-0*: *of-nat k* ≠ 0 ⟹ 0 < *k*
by (*cases k*) *auto*

lemma *of-nat-neq-0*: 0 < *k* ⟹ *k* < 2 ^ *len-of TYPE*(*'a*::*len*) ⟹ *of-nat k* ≠ (0

:: *'a word*)
by (*auto simp add* : *of-nat-0*)

lemma *Abs-fnat-hom-add*: *of-nat a* + *of-nat b* = *of-nat* (*a* + *b*)
by *simp*

lemma *Abs-fnat-hom-mult*: *of-nat a* ∗ *of-nat b* = (*of-nat* (*a* ∗ *b*) :: *'a*::*len word*)
by (*simp add*: *word-of-nat wi-hom-mult*)

lemma *Abs-fnat-hom-Suc*: *word-succ* (*of-nat a*) = *of-nat* (*Suc a*)
by (*simp add*: *word-of-nat wi-hom-succ ac-simps*)

lemma *Abs-fnat-hom-0*: (*0*::*'a*::*len word*) = *of-nat 0*
by *simp*

lemma *Abs-fnat-hom-1*: (*1*::*'a*::*len word*) = *of-nat* (*Suc 0*)
by *simp*

lemmas *Abs-fnat-homs* =
Abs-fnat-hom-add Abs-fnat-hom-mult Abs-fnat-hom-Suc
Abs-fnat-hom-0 Abs-fnat-hom-1

lemma *word-arith-nat-add*: *a* + *b* = *of-nat* (*unat a* + *unat b*)
by *simp*

lemma *word-arith-nat-mult*: *a* ∗ *b* = *of-nat* (*unat a* ∗ *unat b*)
by *simp*

lemma *word-arith-nat-Suc*: *word-succ a* = *of-nat* (*Suc* (*unat a*))
by (*subst Abs-fnat-hom-Suc* [*symmetric*]) *simp*

lemma *word-arith-nat-div*: *a div b* = *of-nat* (*unat a div unat b*)
by (*simp add*: *word-div-def word-of-nat zdiv-int uint-nat*)

lemma *word-arith-nat-mod*: *a mod b* = *of-nat* (*unat a mod unat b*)
by (*simp add*: *word-mod-def word-of-nat zmod-int uint-nat*)

lemmas *word-arith-nat-defs* =
word-arith-nat-add word-arith-nat-mult
word-arith-nat-Suc Abs-fnat-hom-0
Abs-fnat-hom-1 word-arith-nat-div
word-arith-nat-mod

lemma *unat-cong*: *x* = *y* ⟹ *unat x* = *unat y*
by *simp*

lemmas *unat-word-ariths* = *word-arith-nat-defs*
[*THEN trans* [*OF unat-cong unat-of-nat*]]

lemmas *word-sub-less-iff* = *word-sub-le-iff*
[*unfolded linorder-not-less* [*symmetric*] *Not-eq-iff*]

lemma *unat-add-lem*:
unat x + *unat y* < 2 ^ *len-of TYPE*('*a*) ⟷ *unat* (*x* + *y*) = *unat x* + *unat y*
for *x y* :: '*a*::*len word*
by (*auto simp*: *unat-word-ariths intro*!: *trans* [*OF* - *nat-mod-lem*])

lemma *unat-mult-lem*:
unat x ∗ *unat y* < 2 ^ *len-of TYPE*('*a*) ⟷ *unat* (*x* ∗ *y*) = *unat x* ∗ *unat y*
for *x y* :: '*a*::*len word*
by (*auto simp*: *unat-word-ariths intro*!: *trans* [*OF* - *nat-mod-lem*])

lemmas *unat-plus-if'* =
trans [*OF unat-word-ariths*(*1*) *mod-nat-add*, *simplified*]

lemma *le-no-overflow*: *x* ≤ *b* ⟹ *a* ≤ *a* + *b* ⟹ *x* ≤ *a* + *b*
for *a b x* :: '*a*::*len0 word*
apply (*erule order-trans*)
apply (*erule olen-add-eqv* [*THEN iffD1*])
done

lemmas *un-ui-le* =
trans [*OF word-le-nat-alt* [*symmetric*] *word-le-def*]

lemma *unat-sub-if-size*:
unat (*x* − *y*) =
(*if unat y* ≤ *unat x*
then unat x − *unat y*
else unat x + 2 ^ *size x* − *unat y*)
apply (*unfold word-size*)
apply (*simp add*: *un-ui-le*)
apply (*auto simp add*: *unat-def uint-sub-if'*)
apply (*rule nat-diff-distrib*)
prefer *3*
apply (*simp add*: *algebra-simps*)
apply (*rule nat-diff-distrib* [*THEN trans*])
prefer *3*
apply (*subst nat-add-distrib*)
prefer *3*
apply (*simp add*: *nat-power-eq*)
apply *auto*
apply *uint-arith*
done

lemmas *unat-sub-if'* = *unat-sub-if-size* [*unfolded word-size*]

lemma *unat-div*: *unat* (*x div y*) = *unat x div unat y*
for *x y* :: '*a*::*len word*

apply (*simp add* : *unat-word-ariths*)
apply (*rule unat-lt2p* [*THEN xtr7*, *THEN nat-mod-eq'*])
apply (*rule div-le-dividend*)
done

lemma *unat-mod*: *unat* (*x mod y*) = *unat x mod unat y*
for *x y* :: *'a*::*len word*
apply (*clarsimp simp add* : *unat-word-ariths*)
apply (*cases unat y*)
prefer *2*
apply (*rule unat-lt2p* [*THEN xtr7*, *THEN nat-mod-eq'*])
apply (*rule mod-le-divisor*)
apply *auto*
done

lemma *uint-div*: *uint* (*x div y*) = *uint x div uint y*
for *x y* :: *'a*::*len word*
by (*simp add*: *uint-nat unat-div zdiv-int*)

lemma *uint-mod*: *uint* (*x mod y*) = *uint x mod uint y*
for *x y* :: *'a*::*len word*
by (*simp add*: *uint-nat unat-mod zmod-int*)

10.22 Definition of *unat-arith* tactic

lemma *unat-split*: *P* (*unat x*) ⟷ (∀ *n*. *of-nat n* = *x* ∧ *n* < *2^len-of TYPE*('*a*) ⟶ *P n*)
for *x* :: *'a*::*len word*
by (*auto simp*: *unat-of-nat*)

lemma *unat-split-asm*: *P* (*unat x*) ⟷ (∄ *n*. *of-nat n* = *x* ∧ *n* < *2^len-of TYPE*('*a*) ∧ ¬ *P n*)
for *x* :: *'a*::*len word*
by (*auto simp*: *unat-of-nat*)

lemmas *of-nat-inverse* =
word-unat.Abs-inverse' [*rotated*, *unfolded unats-def*, *simplified*]

lemmas *unat-splits* = *unat-split unat-split-asm*

lemmas *unat-arith-simps* =
word-le-nat-alt word-less-nat-alt
word-unat.Rep-inject [*symmetric*]
unat-sub-if' unat-plus-if' unat-div unat-mod

ML ‹
fun unat-arith-simpset ctxt =
ctxt addsimps @{*thms unat-arith-simps*}

```
    delsimps @{thms word-unat.Rep-inject}
    |> fold Splitter.add-split @{thms if-split-asm}
    |> fold Simplifier.add-cong @{thms power-False-cong}

fun unat-arith-tacs ctxt =
  let
    fun arith-tac' n t =
      Arith-Data.arith-tac ctxt n t
        handle Cooper.COOPER - => Seq.empty;
  in
    [ clarify-tac ctxt 1,
      full-simp-tac (unat-arith-simpset ctxt) 1,
      ALLGOALS (full-simp-tac
        (put-simpset HOL-ss ctxt
          |> fold Splitter.add-split @{thms unat-splits}
          |> fold Simplifier.add-cong @{thms power-False-cong})),
      rewrite-goals-tac ctxt @{thms word-size},
      ALLGOALS (fn n => REPEAT (resolve-tac ctxt [allI, impI] n) THEN
                        REPEAT (eresolve-tac ctxt [conjE] n) THEN
                        REPEAT (dresolve-tac ctxt @{thms of-nat-inverse} n THEN
assume-tac ctxt n)),
      TRYALL arith-tac' ]
  end

fun unat-arith-tac ctxt = SELECT-GOAL (EVERY (unat-arith-tacs ctxt))
›

method-setup unat-arith =
  ‹Scan.succeed (SIMPLE-METHOD' o unat-arith-tac)›
  solving word arithmetic via natural numbers and arith

lemma no-plus-overflow-unat-size: x ≤ x + y ⟷ unat x + unat y < 2 ^ size x
  for x y :: 'a::len word
  unfolding word-size by unat-arith

lemmas no-olen-add-nat =
  no-plus-overflow-unat-size [unfolded word-size]

lemmas unat-plus-simple =
  trans [OF no-olen-add-nat unat-add-lem]

lemma word-div-mult: 0 < y ⟹ unat x * unat y < 2 ^ len-of TYPE('a) ⟹ x
* y div y = x
  for x y :: 'a::len word
  apply unat-arith
  apply clarsimp
  apply (subst unat-mult-lem [THEN iffD1])
   apply auto
  done
```

lemma *div-lt'*: $i \leq k\ div\ x \Longrightarrow unat\ i * unat\ x < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
for *i k x* :: *'a*::*len word*
apply *unat-arith*
apply *clarsimp*
apply (*drule mult-le-mono1*)
apply (*erule order-le-less-trans*)
apply (*rule xtr7* [*OF unat-lt2p div-mult-le*])
done

lemmas *div-lt''* = *order-less-imp-le* [*THEN div-lt'*]

lemma *div-lt-mult*: $i < k\ div\ x \Longrightarrow 0 < x \Longrightarrow i * x < k$
for *i k x* :: *'a*::*len word*
apply (*frule div-lt''* [*THEN unat-mult-lem* [*THEN iffD1*]])
apply (*simp add*: *unat-arith-simps*)
apply (*drule* (*1*) *mult-less-mono1*)
apply (*erule order-less-le-trans*)
apply (*rule div-mult-le*)
done

lemma *div-le-mult*: $i \leq k\ div\ x \Longrightarrow 0 < x \Longrightarrow i * x \leq k$
for *i k x* :: *'a*::*len word*
apply (*frule div-lt'* [*THEN unat-mult-lem* [*THEN iffD1*]])
apply (*simp add*: *unat-arith-simps*)
apply (*drule mult-le-mono1*)
apply (*erule order-trans*)
apply (*rule div-mult-le*)
done

lemma *div-lt-uint'*: $i \leq k\ div\ x \Longrightarrow uint\ i * uint\ x < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
for *i k x* :: *'a*::*len word*
apply (*unfold uint-nat*)
apply (*drule div-lt'*)
apply (*metis of-nat-less-iff of-nat-mult of-nat-numeral of-nat-power*)
done

lemmas *div-lt-uint''* = *order-less-imp-le* [*THEN div-lt-uint'*]

lemma *word-le-exists'*: $x \leq y \Longrightarrow \exists z.\ y = x + z \wedge uint\ x + uint\ z < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
for *x y z* :: *'a*::*len0 word*
apply (*rule exI*)
apply (*rule conjI*)
apply (*rule zadd-diff-inverse*)
apply *uint-arith*
done

lemmas *plus-minus-not-NULL* = *order-less-imp-le* [*THEN plus-minus-not-NULL-ab*]

lemmas *plus-minus-no-overflow* =
order-less-imp-le [*THEN plus-minus-no-overflow-ab*]

lemmas *mcs* = *word-less-minus-cancel word-less-minus-mono-left*
word-le-minus-cancel word-le-minus-mono-left

lemmas *word-l-diffs* = *mcs* [**where** *y* = *w* + *x*, *unfolded add-diff-cancel*] **for** *w x*
lemmas *word-diff-ls* = *mcs* [**where** *z* = *w* + *x*, *unfolded add-diff-cancel*] **for** *w x*
lemmas *word-plus-mcs* = *word-diff-ls* [**where** *y* = *v* + *x*, *unfolded add-diff-cancel*]
for *v x*

lemmas *le-unat-uoi* = *unat-le* [*THEN word-unat.Abs-inverse*]

lemmas *thd* = *refl* [*THEN* [*2*] *split-div-lemma* [*THEN iffD2*], *THEN conjunct1*]

lemmas *uno-simps* [*THEN le-unat-uoi*] = *mod-le-divisor div-le-dividend dtle*

lemma *word-mod-div-equality*: (*n div b*) * *b* + (*n mod b*) = *n*
for *n b* :: *'a*::*len word*
apply (*unfold word-less-nat-alt word-arith-nat-defs*)
apply (*cut-tac y*=*unat b* **in** *gt-or-eq-0*)
apply (*erule disjE*)
apply (*simp only*: *div-mult-mod-eq uno-simps Word.word-unat.Rep-inverse*)
apply *simp*
done

lemma *word-div-mult-le*: *a div b* * *b* ≤ *a*
for *a b* :: *'a*::*len word*
apply (*unfold word-le-nat-alt word-arith-nat-defs*)
apply (*cut-tac y*=*unat b* **in** *gt-or-eq-0*)
apply (*erule disjE*)
apply (*simp only*: *div-mult-le uno-simps Word.word-unat.Rep-inverse*)
apply *simp*
done

lemma *word-mod-less-divisor*: *0* < *n* ⟹ *m mod n* < *n*
for *m n* :: *'a*::*len word*
apply (*simp only*: *word-less-nat-alt word-arith-nat-defs*)
apply (*auto simp*: *uno-simps*)
done

lemma *word-of-int-power-hom*: *word-of-int a* ^ *n* = (*word-of-int* (*a* ^ *n*) :: *'a*::*len word*)
by (*induct n*) (*simp-all add*: *wi-hom-mult* [*symmetric*])

lemma *word-arith-power-alt*: *a* ^ *n* = (*word-of-int* (*uint a* ^ *n*) :: *'a*::*len word*)
by (*simp add* : *word-of-int-power-hom* [*symmetric*])

lemma *of-bl-length-less*:
length x = k ⟹ k < len-of TYPE('a) ⟹ (of-bl x :: 'a::len word) < 2 ^ k
apply (*unfold of-bl-def word-less-alt word-numeral-alt*)
apply *safe*
apply (*simp* (*no-asm*) *add*: *word-of-int-power-hom word-uint.eq-norm*
del: *word-of-int-numeral*)
apply (*simp add*: *mod-pos-pos-trivial*)
apply (*subst mod-pos-pos-trivial*)
apply (*rule bl-to-bin-ge0*)
apply (*rule order-less-trans*)
apply (*rule bl-to-bin-lt2p*)
apply *simp*
apply (*rule bl-to-bin-lt2p*)
done

10.23 Cardinality, finiteness of set of words

instance *word* :: (*len0*) *finite*
by *standard* (*simp add*: *type-definition.univ* [*OF type-definition-word*])

lemma *card-word*: *CARD('a::len0 word) = 2 ^ len-of TYPE('a)*
by (*simp add*: *type-definition.card* [*OF type-definition-word*] *nat-power-eq*)

lemma *card-word-size*: *card* (*UNIV* :: *'a word set*) = (*2 ^ size x*)
for *x* :: *'a::len0 word*
unfolding *word-size* **by** (*rule card-word*)

10.24 Bitwise Operations on Words

lemmas *bin-log-bintrs = bin-trunc-not bin-trunc-xor bin-trunc-and bin-trunc-or*

lemmas *wils1 = bin-log-bintrs* [*THEN word-ubin.norm-eq-iff* [*THEN iffD1*],
folded word-ubin.eq-norm, *THEN eq-reflection*]

lemmas *word-log-binary-defs =*
word-and-def word-or-def word-xor-def

lemma *word-wi-log-defs*:
NOT word-of-int a = word-of-int (*NOT a*)
word-of-int a AND word-of-int b = word-of-int (*a AND b*)
word-of-int a OR word-of-int b = word-of-int (*a OR b*)
word-of-int a XOR word-of-int b = word-of-int (*a XOR b*)
by (*transfer*, *rule refl*)+

lemma *word-no-log-defs* [*simp*]:

NOT (numeral a) = word-of-int (NOT (numeral a))
NOT (− numeral a) = word-of-int (NOT (− numeral a))
numeral a AND numeral b = word-of-int (numeral a AND numeral b)
numeral a AND − numeral b = word-of-int (numeral a AND − numeral b)
− numeral a AND numeral b = word-of-int (− numeral a AND numeral b)
− numeral a AND − numeral b = word-of-int (− numeral a AND − numeral b)
numeral a OR numeral b = word-of-int (numeral a OR numeral b)
numeral a OR − numeral b = word-of-int (numeral a OR − numeral b)
− numeral a OR numeral b = word-of-int (− numeral a OR numeral b)
− numeral a OR − numeral b = word-of-int (− numeral a OR − numeral b)
numeral a XOR numeral b = word-of-int (numeral a XOR numeral b)
numeral a XOR − numeral b = word-of-int (numeral a XOR − numeral b)
− numeral a XOR numeral b = word-of-int (− numeral a XOR numeral b)
− numeral a XOR − numeral b = word-of-int (− numeral a XOR − numeral b)
by (*transfer*, *rule refl*)+

Special cases for when one of the arguments equals 1.

lemma *word-bitwise-1-simps* [*simp*]:
NOT (1::'a::len0 word) = −2
1 AND numeral b = word-of-int (1 AND numeral b)
1 AND − numeral b = word-of-int (1 AND − numeral b)
numeral a AND 1 = word-of-int (numeral a AND 1)
− numeral a AND 1 = word-of-int (− numeral a AND 1)
1 OR numeral b = word-of-int (1 OR numeral b)
1 OR − numeral b = word-of-int (1 OR − numeral b)
numeral a OR 1 = word-of-int (numeral a OR 1)
− numeral a OR 1 = word-of-int (− numeral a OR 1)
1 XOR numeral b = word-of-int (1 XOR numeral b)
1 XOR − numeral b = word-of-int (1 XOR − numeral b)
numeral a XOR 1 = word-of-int (numeral a XOR 1)
− numeral a XOR 1 = word-of-int (− numeral a XOR 1)
by (*transfer*, *simp*)+

Special cases for when one of the arguments equals -1.

lemma *word-bitwise-m1-simps* [*simp*]:
NOT (−1::'a::len0 word) = 0
(−1::'a::len0 word) AND x = x
x AND (−1::'a::len0 word) = x
(−1::'a::len0 word) OR x = −1
x OR (−1::'a::len0 word) = −1
(−1::'a::len0 word) XOR x = NOT x
x XOR (−1::'a::len0 word) = NOT x
by (*transfer*, *simp*)+

lemma *uint-or*: *uint (x OR y) = uint x OR uint y*
by (*transfer*, *simp add*: *bin-trunc-ao*)

lemma *uint-and*: *uint (x AND y) = uint x AND uint y*
by (*transfer*, *simp add*: *bin-trunc-ao*)

lemma *test-bit-wi* [*simp*]:
(*word-of-int x* :: *'a*::*len0 word*) !! *n* ⟷ *n* < *len-of TYPE*(*'a*) ∧ *bin-nth x n*
by (*simp add*: *word-test-bit-def word-ubin.eq-norm nth-bintr*)

lemma *word-test-bit-transfer* [*transfer-rule*]:
(*rel-fun pcr-word* (*rel-fun op* = *op* =))
(λ*x n*. *n* < *len-of TYPE*(*'a*) ∧ *bin-nth x n*) (*test-bit* :: *'a*::*len0 word* ⇒ -)
unfolding *rel-fun-def word.pcr-cr-eq cr-word-def* **by** *simp*

lemma *word-ops-nth-size*:
n < *size x* ⟹
(*x OR y*) !! *n* = (*x* !! *n* | *y* !! *n*) ∧
(*x AND y*) !! *n* = (*x* !! *n* ∧ *y* !! *n*) ∧
(*x XOR y*) !! *n* = (*x* !! *n* ≠ *y* !! *n*) ∧
(*NOT x*) !! *n* = (¬ *x* !! *n*)
for *x* :: *'a*::*len0 word*
unfolding *word-size* **by** *transfer* (*simp add*: *bin-nth-ops*)

lemma *word-ao-nth*:
(*x OR y*) !! *n* = (*x* !! *n* | *y* !! *n*) ∧
(*x AND y*) !! *n* = (*x* !! *n* ∧ *y* !! *n*)
for *x* :: *'a*::*len0 word*
by *transfer* (*auto simp add*: *bin-nth-ops*)

lemma *test-bit-numeral* [*simp*]:
(*numeral w* :: *'a*::*len0 word*) !! *n* ⟷
n < *len-of TYPE*(*'a*) ∧ *bin-nth* (*numeral w*) *n*
by *transfer* (*rule refl*)

lemma *test-bit-neg-numeral* [*simp*]:
(− *numeral w* :: *'a*::*len0 word*) !! *n* ⟷
n < *len-of TYPE*(*'a*) ∧ *bin-nth* (− *numeral w*) *n*
by *transfer* (*rule refl*)

lemma *test-bit-1* [*simp*]: (*1* :: *'a*::*len word*) !! *n* ⟷ *n* = *0*
by *transfer auto*

lemma *nth-0* [*simp*]: ¬ (*0* :: *'a*::*len0 word*) !! *n*
by *transfer simp*

lemma *nth-minus1* [*simp*]: (*−1* :: *'a*::*len0 word*) !! *n* ⟷ *n* < *len-of TYPE*(*'a*)
by *transfer simp*

lemmas *bwsimps* =
wi-hom-add
word-wi-log-defs

lemma *word-bw-assocs*:
(*x AND y*) *AND z* = *x AND y AND z*
(*x OR y*) *OR z* = *x OR y OR z*
(*x XOR y*) *XOR z* = *x XOR y XOR z*
for *x* :: *'a*::*len0 word*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-bw-comms*:
x AND y = *y AND x*
x OR y = *y OR x*
x XOR y = *y XOR x*
for *x* :: *'a*::*len0 word*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-bw-lcs*:
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* :: *'a*::*len0 word*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-log-esimps* [*simp*]:
x AND 0 = *0*
x AND −1 = *x*
x OR 0 = *x*
x OR −1 = *−1*
x XOR 0 = *x*
x XOR −1 = *NOT x*
0 AND x = *0*
−1 AND x = *x*
0 OR x = *x*
−1 OR x = *−1*
0 XOR x = *x*
−1 XOR x = *NOT x*
for *x* :: *'a*::*len0 word*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-not-dist*:
NOT (*x OR y*) = *NOT x AND NOT y*
NOT (*x AND y*) = *NOT x OR NOT y*
for *x* :: *'a*::*len0 word*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-bw-same*:
x AND x = *x*
x OR x = *x*
x XOR x = *0*
for *x* :: *'a*::*len0 word*

by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-ao-absorbs* [*simp*]:
x AND (y OR x) = x
x OR y AND x = x
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* :: *'a*::*len0 word*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-not-not* [*simp*]: *NOT NOT x = x*
for *x* :: *'a*::*len0 word*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-ao-dist*: *(x OR y) AND z = x AND z OR y AND z*
for *x* :: *'a*::*len0 word*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-oa-dist*: *x AND y OR z = (x OR z) AND (y OR z)*
for *x* :: *'a*::*len0 word*
by (*auto simp*: *word-eq-iff word-ops-nth-size* [*unfolded word-size*])

lemma *word-add-not* [*simp*]: *x + NOT x = −1*
for *x* :: *'a*::*len0 word*
by *transfer* (*simp add*: *bin-add-not*)

lemma *word-plus-and-or* [*simp*]: *(x AND y) + (x OR y) = x + y*
for *x* :: *'a*::*len0 word*
by *transfer* (*simp add*: *plus-and-or*)

lemma *leoa*: *w = x OR y ⟹ y = w AND y*
for *x* :: *'a*::*len0 word*
by *auto*

lemma *leao*: *w' = x' AND y' ⟹ x' = x' OR w'*
for *x'* :: *'a*::*len0 word*
by *auto*

lemma *word-ao-equiv*: *w = w OR w' ⟷ w' = w AND w'*
for *w w'* :: *'a*::*len0 word*
by (*auto intro*: *leoa leao*)

lemma *le-word-or2*: *x ≤ x OR y*
for *x y* :: *'a*::*len0 word*
by (*auto simp*: *word-le-def uint-or intro*: *le-int-or*)

lemmas *le-word-or1* = *xtr3* [*OF word-bw-comms* (*2*) *le-word-or2*]
lemmas *word-and-le1* = *xtr3* [*OF word-ao-absorbs* (*4*) [*symmetric*] *le-word-or2*]
lemmas *word-and-le2* = *xtr3* [*OF word-ao-absorbs* (*8*) [*symmetric*] *le-word-or2*]

lemma *bl-word-not*: *to-bl* (*NOT w*) = *map Not* (*to-bl w*)
unfolding *to-bl-def word-log-defs bl-not-bin*
by (*simp add*: *word-ubin.eq-norm*)

lemma *bl-word-xor*: *to-bl* (*v XOR w*) = *map2 op* ≠ (*to-bl v*) (*to-bl w*)
unfolding *to-bl-def word-log-defs bl-xor-bin*
by (*simp add*: *word-ubin.eq-norm*)

lemma *bl-word-or*: *to-bl* (*v OR w*) = *map2 op* ∨ (*to-bl v*) (*to-bl w*)
unfolding *to-bl-def word-log-defs bl-or-bin*
by (*simp add*: *word-ubin.eq-norm*)

lemma *bl-word-and*: *to-bl* (*v AND w*) = *map2 op* ∧ (*to-bl v*) (*to-bl w*)
unfolding *to-bl-def word-log-defs bl-and-bin*
by (*simp add*: *word-ubin.eq-norm*)

lemma *word-lsb-alt*: *lsb w* = *test-bit w 0*
for *w* :: *'a*::*len0 word*
by (*auto simp*: *word-test-bit-def word-lsb-def*)

lemma *word-lsb-1-0* [*simp*]: *lsb* (*1*::*'a*::*len word*) ∧ ¬ *lsb* (*0*::*'b*::*len0 word*)
unfolding *word-lsb-def uint-eq-0 uint-1* **by** *simp*

lemma *word-lsb-last*: *lsb w* = *last* (*to-bl w*)
for *w* :: *'a*::*len word*
apply (*unfold word-lsb-def uint-bl bin-to-bl-def*)
apply (*rule-tac bin*=*uint w* **in** *bin-exhaust*)
apply (*cases size w*)
apply *auto*
apply (*auto simp add*: *bin-to-bl-aux-alt*)
done

lemma *word-lsb-int*: *lsb w* ⟷ *uint w mod 2* = *1*
by (*auto simp*: *word-lsb-def bin-last-def*)

lemma *word-msb-sint*: *msb w* ⟷ *sint w* < *0*
by (*simp only*: *word-msb-def sign-Min-lt-0*)

lemma *msb-word-of-int*: *msb* (*word-of-int x*::*'a*::*len word*) = *bin-nth x* (*len-of TYPE*(*'a*) − *1*)
by (*simp add*: *word-msb-def word-sbin.eq-norm bin-sign-lem*)

lemma *word-msb-numeral* [*simp*]:
msb (*numeral w*::*'a*::*len word*) = *bin-nth* (*numeral w*) (*len-of TYPE*(*'a*) − *1*)

unfolding *word-numeral-alt* **by** (*rule msb-word-of-int*)

lemma *word-msb-neg-numeral* [*simp*]:
msb (− *numeral w*::*'a*::*len word*) = *bin-nth* (− *numeral w*) (*len-of TYPE*(*'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*: *word-msb-def*)

lemma *word-msb-1* [*simp*]: *msb* (*1*::*'a*::*len word*) ⟷ *len-of TYPE*(*'a*) = *1*
unfolding *word-1-wi msb-word-of-int eq-iff* [**where** *'a*=*nat*]
by (*simp add*: *Suc-le-eq*)

lemma *word-msb-nth*: *msb w* = *bin-nth* (*uint w*) (*len-of TYPE*(*'a*) − *1*)
for *w* :: *'a*::*len word*
by (*simp add*: *word-msb-def sint-uint bin-sign-lem*)

lemma *word-msb-alt*: *msb w* = *hd* (*to-bl w*)
for *w* :: *'a*::*len word*
apply (*unfold word-msb-nth uint-bl*)
apply (*subst hd-conv-nth*)
apply (*rule length-greater-0-conv* [*THEN iffD1*])
apply *simp*
apply (*simp add* : *nth-bin-to-bl word-size*)
done

lemma *word-set-nth* [*simp*]: *set-bit w n* (*test-bit w n*) = *w*
for *w* :: *'a*::*len0 word*
by (*auto simp*: *word-test-bit-def word-set-bit-def*)

lemma *bin-nth-uint'*: *bin-nth* (*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*: *w* !! *n* ⟷ *rev* (*to-bl w*) ! *n* ∧ *n* < *size w*
unfolding *to-bl-def word-test-bit-def word-size* **by** (*rule bin-nth-uint*)

lemma *to-bl-nth*: *n* < *size w* ⟹ *to-bl w* ! *n* = *w* !! (*size w* − *Suc n*)
apply (*unfold test-bit-bl*)
apply *clarsimp*
apply (*rule trans*)
apply (*rule nth-rev-alt*)

apply (*auto simp add*: *word-size*)
done

lemma *test-bit-set*: (*set-bit w n x*) !! *n* ⟷ *n* < *size w* ∧ *x*
for *w* :: *'a*::*len0 word*
by (*auto simp*: *word-size word-test-bit-def word-set-bit-def word-ubin.eq-norm nth-bintr*)

lemma *test-bit-set-gen*:
test-bit (*set-bit w n x*) *m* = (*if m* = *n then n* < *size w* ∧ *x else test-bit w m*)
for *w* :: *'a*::*len0 word*
apply (*unfold word-size word-test-bit-def word-set-bit-def*)
apply (*clarsimp simp add*: *word-ubin.eq-norm nth-bintr bin-nth-sc-gen*)
apply (*auto elim*!: *test-bit-size* [*unfolded word-size*]
simp add: *word-test-bit-def* [*symmetric*])
done

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 *msb-nth*: *msb w* = *w* !! (*len-of TYPE*(*'a*) − *1*)
for *w* :: *'a*::*len word*
by (*simp add*: *word-msb-nth word-test-bit-def*)

lemmas *msb0* = *len-gt-0* [*THEN diff-Suc-less*, *THEN word-ops-nth-size* [*unfolded word-size*]]
lemmas *msb1* = *msb0* [**where** *i* = *0*]
lemmas *word-ops-msb* = *msb1* [*unfolded msb-nth* [*symmetric*, *unfolded One-nat-def*]]

lemmas *lsb0* = *len-gt-0* [*THEN word-ops-nth-size* [*unfolded word-size*]]
lemmas *word-ops-lsb* = *lsb0* [*unfolded word-lsb-alt*]

lemma *td-ext-nth* [*OF refl refl refl*, *unfolded word-size*]:
n = *size w* ⟹ *ofn* = *set-bits* ⟹ [*w*, *ofn g*] = *l* ⟹
td-ext test-bit ofn {*f*. ∀ *i*. *f i* ⟶ *i* < *n*} (λ*h i*. *h i* ∧ *i* < *n*)
for *w* :: *'a*::*len0 word*
apply (*unfold word-size td-ext-def'*)
apply *safe*
apply (*rule-tac* [*3*] *ext*)
apply (*rule-tac* [*4*] *ext*)
apply (*unfold word-size of-nth-def test-bit-bl*)
apply *safe*
defer
apply (*clarsimp simp*: *word-bl.Abs-inverse*)+
apply (*rule word-bl.Rep-inverse'*)
apply (*rule sym* [*THEN trans*])
apply (*rule bl-of-nth-nth*)
apply *simp*
apply (*rule bl-of-nth-inj*)

apply (*clarsimp simp add* : *test-bit-bl word-size*)
done

interpretation *test-bit*:
td-ext
op !! :: *'a::len0 word* ⇒ *nat* ⇒ *bool*
set-bits
{*f*. ∀ *i*. *f i* ⟶ *i* < *len-of TYPE('a::len0)*}
(λ*h i*. *h i* ∧ *i* < *len-of TYPE('a::len0)*)
by (*rule td-ext-nth*)

lemmas *td-nth* = *test-bit.td-thm*

lemma *word-set-set-same* [*simp*]: *set-bit* (*set-bit w n x*) *n y* = *set-bit w n y*
for *w* :: *'a::len0 word*
by (*rule word-eqI*) (*simp add* : *test-bit-set-gen word-size*)

lemma *word-set-set-diff*:
fixes *w* :: *'a::len0 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 *nth-sint*:
fixes *w* :: *'a::len word*
defines *l* ≡ *len-of TYPE('a)*
shows *bin-nth* (*sint w*) *n* = (*if n* < *l* − *1 then w* !! *n else w* !! (*l* − *1*))
unfolding *sint-uint l-def*
by (*auto simp*: *nth-sbintr word-test-bit-def* [*symmetric*])

lemma *word-lsb-numeral* [*simp*]:
lsb (*numeral bin* :: *'a::len word*) ⟷ *bin-last* (*numeral bin*)
unfolding *word-lsb-alt test-bit-numeral* **by** *simp*

lemma *word-lsb-neg-numeral* [*simp*]:
lsb (− *numeral bin* :: *'a::len word*) ⟷ *bin-last* (− *numeral bin*)
by (*simp add*: *word-lsb-alt*)

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 word-ubin.eq-norm nth-bintr*)

lemma *word-set-numeral* [*simp*]:
set-bit (*numeral bin*::*'a::len0 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::len0 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*)

lemma *setBit-no* [*simp*]: *setBit* (*numeral bin*) *n* = *word-of-int* (*bin-sc n True* (*numeral bin*))
by (*simp add*: *setBit-def*)

lemma *clearBit-no* [*simp*]:
clearBit (*numeral bin*) *n* = *word-of-int* (*bin-sc n False* (*numeral bin*))
by (*simp add*: *clearBit-def*)

lemma *to-bl-n1*: *to-bl* (−*1*::*'a*::*len0 word*) = *replicate* (*len-of TYPE*(*'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 *word-msb-n1* [*simp*]: *msb* (−*1*::*'a*::*len word*)
unfolding *word-msb-alt to-bl-n1* **by** *simp*

lemma *word-set-nth-iff*: *set-bit w n b* = *w* ⟷ *w* !! *n* = *b* ∨ *n* ≥ *size w*
for *w* :: *'a*::*len0 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 (*drule test-bit-size*)
apply *force*
done

lemma *test-bit-2p*: (*word-of-int* (*2* ^ *n*)::*'a*::*len word*) !! *m* ⟷ *m* = *n* ∧ *m* < *len-of TYPE*(*'a*)
by (*auto simp*: *word-test-bit-def word-ubin.eq-norm nth-bintr nth-2p-bin*)

lemma *nth-w2p*: ((*2*::*'a*::*len word*) ^ *n*) !! *m* ⟷ *m* = *n* ∧ *m* < *len-of TYPE*(*'a*::*len*)
by (*simp add*: *test-bit-2p* [*symmetric*] *word-of-int* [*symmetric*])

lemma *uint-2p*: $(0::'a::len\ word) < 2\ \hat{}\ n \Longrightarrow uint\ (2\ \hat{}\ n::'a::len\ word) = 2\ \hat{}\ n$
apply (*unfold word-arith-power-alt*)
apply (*case-tac len-of TYPE('a)*)
apply *clarsimp*
apply (*case-tac nat*)
apply *clarsimp*
apply (*case-tac n*)
apply *clarsimp*
apply *clarsimp*
apply (*drule word-gt-0* [*THEN iffD1*])
apply (*safe intro*!: *word-eqI*)
apply (*auto simp add*: *nth-2p-bin*)
apply (*erule notE*)
apply (*simp* (*no-asm-use*) *add*: *uint-word-of-int word-size*)
apply (*subst mod-pos-pos-trivial*)
apply *simp*
apply (*rule power-strict-increasing*)
apply *simp-all*
done

lemma *word-of-int-2p*: $(word\text{-}of\text{-}int\ (2\ \hat{}\ n) :: 'a::len\ word) = 2\ \hat{}\ n$
by (*induct n*) (*simp-all add*: *wi-hom-syms*)

lemma *bang-is-le*: $x\ !!\ m \Longrightarrow 2\ \hat{}\ m \leq x$
for $x :: 'a::len\ word$
apply (*rule xtr3*)
apply (*rule-tac* [*2*] $y = x$ **in** *le-word-or2*)
apply (*rule word-eqI*)
apply (*auto simp add*: *word-ao-nth nth-w2p word-size*)
done

lemma *word-clr-le*: $w \geq set\text{-}bit\ w\ n\ False$
for $w :: 'a::len0\ word$
apply (*unfold word-set-bit-def word-le-def word-ubin.eq-norm*)
apply (*rule order-trans*)
apply (*rule bintr-bin-clr-le*)
apply *simp*
done

lemma *word-set-ge*: $w \leq set\text{-}bit\ w\ n\ True$
for $w :: 'a::len\ word$
apply (*unfold word-set-bit-def word-le-def word-ubin.eq-norm*)
apply (*rule order-trans* [*OF - bintr-bin-set-ge*])
apply *simp*
done

10.25 Shifting, Rotating, and Splitting Words

lemma *shiftl1-wi* [*simp*]: *shiftl1* (*word-of-int w*) = *word-of-int* (*w BIT False*)
unfolding *shiftl1-def*
apply (*simp add*: *word-ubin.norm-eq-iff* [*symmetric*] *word-ubin.eq-norm*)
apply (*subst refl* [*THEN bintrunc-BIT-I*, *symmetric*])
apply (*subst bintrunc-bintrunc-min*)
apply *simp*
done

lemma *shiftl1-numeral* [*simp*]: *shiftl1* (*numeral w*) = *numeral* (*Num.Bit0 w*)
unfolding *word-numeral-alt shiftl1-wi* **by** *simp*

lemma *shiftl1-neg-numeral* [*simp*]: *shiftl1* (− *numeral w*) = − *numeral* (*Num.Bit0 w*)
unfolding *word-neg-numeral-alt shiftl1-wi* **by** *simp*

lemma *shiftl1-0* [*simp*] : *shiftl1 0* = *0*
by (*simp add*: *shiftl1-def*)

lemma *shiftl1-def-u*: *shiftl1 w* = *word-of-int* (*uint w BIT False*)
by (*simp only*: *shiftl1-def*)

lemma *shiftl1-def-s*: *shiftl1 w* = *word-of-int* (*sint w BIT False*)
by (*simp add*: *shiftl1-def Bit-B0 wi-hom-syms*)

lemma *shiftr1-0* [*simp*]: *shiftr1 0* = *0*
by (*simp add*: *shiftr1-def*)

lemma *sshiftr1-0* [*simp*]: *sshiftr1 0* = *0*
by (*simp add*: *sshiftr1-def*)

lemma *sshiftr1-n1* [*simp*]: *sshiftr1* (− *1*) = − *1*
by (*simp add*: *sshiftr1-def*)

lemma *shiftl-0* [*simp*]: (*0*::*'a*::*len0 word*) << *n* = *0*
by (*induct n*) (*auto simp*: *shiftl-def*)

lemma *shiftr-0* [*simp*]: (*0*::*'a*::*len0 word*) >> *n* = *0*
by (*induct n*) (*auto simp*: *shiftr-def*)

lemma *sshiftr-0* [*simp*]: *0* >>> *n* = *0*
by (*induct n*) (*auto simp*: *sshiftr-def*)

lemma *sshiftr-n1* [*simp*]: *−1* >>> *n* = *−1*
by (*induct n*) (*auto simp*: *sshiftr-def*)

lemma *nth-shiftl1*: *shiftl1 w* !! *n* ⟷ *n* < *size w* ∧ *n* > *0* ∧ *w* !! (*n* − *1*)
apply (*unfold shiftl1-def word-test-bit-def*)
apply (*simp add*: *nth-bintr word-ubin.eq-norm word-size*)

apply (*cases n*)
apply *auto*
done

lemma *nth-shiftl'*: (*w* << *m*) !! *n* ⟷ *n* < *size w* ∧ *n* >= *m* ∧ *w* !! (*n* − *m*)
for *w* :: *'a*::*len0 word*
apply (*unfold shiftl-def*)
apply (*induct m arbitrary*: *n*)
apply (*force elim*!: *test-bit-size*)
apply (*clarsimp simp add* : *nth-shiftl1 word-size*)
apply *arith*
done

lemmas *nth-shiftl* = *nth-shiftl'* [*unfolded word-size*]

lemma *nth-shiftr1*: *shiftr1 w* !! *n* = *w* !! *Suc n*
apply (*unfold shiftr1-def word-test-bit-def*)
apply (*simp add*: *nth-bintr word-ubin.eq-norm*)
apply *safe*
apply (*drule bin-nth.Suc* [*THEN iffD2*, *THEN bin-nth-uint-imp*])
apply *simp*
done

lemma *nth-shiftr*: (*w* >> *m*) !! *n* = *w* !! (*n* + *m*)
for *w* :: *'a*::*len0 word*
apply (*unfold shiftr-def*)
apply (*induct m arbitrary*: *n*)
apply (*auto simp add*: *nth-shiftr1*)
done

lemma *uint-shiftr1*: *uint* (*shiftr1 w*) = *bin-rest* (*uint w*)
apply (*unfold shiftr1-def word-ubin.eq-norm bin-rest-trunc-i*)
apply (*subst bintr-uint* [*symmetric*, *OF order-refl*])
apply (*simp only* : *bintrunc-bintrunc-l*)
apply *simp*
done

lemma *nth-sshiftr1*: *sshiftr1 w* !! *n* = (*if n* = *size w* − *1 then w* !! *n else w* !! *Suc n*)
apply (*unfold sshiftr1-def word-test-bit-def*)
apply (*simp add*: *nth-bintr word-ubin.eq-norm bin-nth.Suc* [*symmetric*] *word-size del*: *bin-nth.simps*)
apply (*simp add*: *nth-bintr uint-sint del* : *bin-nth.simps*)
apply (*auto simp add*: *bin-nth-sint*)
done

lemma *nth-sshiftr* [*rule-format*] :

```
  ∀ n. sshiftr w m !! n =
    (n < size w ∧ (if n + m ≥ size w then w !! (size w − 1) else w !! (n + m)))
  apply (unfold sshiftr-def)
  apply (induct-tac m)
   apply (simp add: test-bit-bl)
  apply (clarsimp simp add: nth-sshiftr1 word-size)
  apply safe
       apply arith
      apply arith
     apply (erule thin-rl)
     apply (case-tac n)
      apply safe
      apply simp
     apply simp
    apply (erule thin-rl)
    apply (case-tac n)
     apply safe
     apply simp
    apply simp
   apply arith+
  done

lemma shiftr1-div-2: uint (shiftr1 w) = uint w div 2
  apply (unfold shiftr1-def bin-rest-def)
  apply (rule word-uint.Abs-inverse)
  apply (simp add: uints-num pos-imp-zdiv-nonneg-iff)
  apply (rule xtr7)
   prefer 2
   apply (rule zdiv-le-dividend)
    apply auto
  done

lemma sshiftr1-div-2: sint (sshiftr1 w) = sint w div 2
  apply (unfold sshiftr1-def bin-rest-def [symmetric])
  apply (simp add: word-sbin.eq-norm)
  apply (rule trans)
   defer
   apply (subst word-sbin.norm-Rep [symmetric])
   apply (rule refl)
  apply (subst word-sbin.norm-Rep [symmetric])
  apply (unfold One-nat-def)
  apply (rule sbintrunc-rest)
  done

lemma shiftr-div-2n: uint (shiftr w n) = uint w div 2 ^ n
  apply (unfold shiftr-def)
  apply (induct n)
   apply simp
  apply (simp add: shiftr1-div-2 mult.commute zdiv-zmult2-eq [symmetric])
```

done

lemma *sshiftr-div-2n*: *sint* (*sshiftr w n*) = *sint w div* 2 ^ *n*
apply (*unfold sshiftr-def*)
apply (*induct n*)
apply *simp*
apply (*simp add*: *sshiftr1-div-2 mult.commute zdiv-zmult2-eq* [*symmetric*])
done

10.25.1 shift functions in terms of lists of bools

lemmas *bshiftr1-numeral* [*simp*] =
bshiftr1-def [**where** *w*=*numeral w*, *unfolded to-bl-numeral*] **for** *w*

lemma *bshiftr1-bl*: *to-bl* (*bshiftr1 b w*) = *b* # *butlast* (*to-bl w*)
unfolding *bshiftr1-def* **by** (*rule word-bl.Abs-inverse*) *simp*

lemma *shiftl1-of-bl*: *shiftl1* (*of-bl bl*) = *of-bl* (*bl* @ [*False*])
by (*simp add*: *of-bl-def bl-to-bin-append*)

lemma *shiftl1-bl*: *shiftl1 w* = *of-bl* (*to-bl w* @ [*False*])
for *w* :: *'a*::*len0 word*
proof −
have *shiftl1 w* = *shiftl1* (*of-bl* (*to-bl w*))
by *simp*
also have ... = *of-bl* (*to-bl w* @ [*False*])
by (*rule shiftl1-of-bl*)
finally show *?thesis* **.**
qed

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 *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*))
apply (*unfold shiftr1-def uint-bl of-bl-def*)
apply (*simp add*: *butlast-rest-bin word-size*)
apply (*simp add*: *bin-rest-trunc* [*symmetric*, *unfolded One-nat-def*])
done

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*])

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 *shiftl1-rev*: *shiftl1 w* = *word-reverse* (*shiftr1* (*word-reverse w*))
apply (*unfold word-reverse-def*)
apply (*rule word-bl.Rep-inverse'* [*symmetric*])
apply (*simp add*: *bl-shiftl1' bl-shiftr1' word-bl.Abs-inverse*)
apply (*cases to-bl w*)
apply *auto*
done

lemma *shiftl-rev*: *shiftl w n* = *word-reverse* (*shiftr* (*word-reverse w*) *n*)
by (*induct n*) (*auto simp add*: *shiftl-def shiftr-def shiftl1-rev*)

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*)

lemma *bl-sshiftr1*: *to-bl* (*sshiftr1 w*) = *hd* (*to-bl w*) # *butlast* (*to-bl w*)
for *w* :: *'a*::*len word*
apply (*unfold sshiftr1-def uint-bl word-size*)
apply (*simp add*: *butlast-rest-bin word-ubin.eq-norm*)
apply (*simp add*: *sint-uint*)
apply (*rule nth-equalityI*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac i*)
apply (*simp-all add*: *hd-conv-nth length-0-conv* [*symmetric*]
nth-bin-to-bl bin-nth.Suc [*symmetric*] *nth-sbintr*
del: *bin-nth.Suc*)
apply *force*
apply (*rule impI*)
apply (*rule-tac f* = *bin-nth* (*uint w*) **in** *arg-cong*)
apply *simp*
done

lemma *drop-shiftr*: *drop n* (*to-bl* (*w* >> *n*)) = *take* (*size w* − *n*) (*to-bl w*)
for *w* :: *'a*::*len word*
apply (*unfold shiftr-def*)
apply (*induct n*)
prefer *2*

```
  apply (simp add: drop-Suc bl-shiftr1 butlast-drop [symmetric])
  apply (rule butlast-take [THEN trans])
   apply (auto simp: word-size)
 done

lemma drop-sshiftr: drop n (to-bl (w >>> n)) = take (size w - n) (to-bl w)
 for w :: 'a::len word
 apply (unfold sshiftr-def)
 apply (induct n)
  prefer 2
  apply (simp add: drop-Suc bl-sshiftr1 butlast-drop [symmetric])
  apply (rule butlast-take [THEN trans])
   apply (auto simp: word-size)
 done

lemma take-shiftr: n ≤ size w ⟹ take n (to-bl (w >> n)) = replicate n False
 apply (unfold shiftr-def)
 apply (induct n)
  prefer 2
  apply (simp add: bl-shiftr1' length-0-conv [symmetric] word-size)
  apply (rule take-butlast [THEN trans])
   apply (auto simp: word-size)
 done

lemma take-sshiftr' [rule-format] :
 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 (unfold sshiftr-def)
 apply (induct n)
  prefer 2
  apply (simp add: bl-sshiftr1)
  apply (rule impI)
  apply (rule take-butlast [THEN trans])
   apply (auto simp: word-size)
 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)
 by (induct n) (auto simp: shiftl-def shiftl1-of-bl replicate-app-Cons-same)
```

lemma *shiftl-bl*: $w << n = of\text{-}bl\ (to\text{-}bl\ w\ @\ replicate\ n\ False)$
for $w :: 'a::len0\ word$
proof −
have $w << n = of\text{-}bl\ (to\text{-}bl\ w) << n$
by *simp*
also have $\ldots = of\text{-}bl\ (to\text{-}bl\ w\ @\ replicate\ n\ False)$
by (*rule shiftl-of-bl*)
finally show *?thesis* **.**
qed

lemmas *shiftl-numeral* [*simp*] = *shiftl-def* [**where** *w=numeral w*] **for** *w*

lemma *bl-shiftl*: $to\text{-}bl\ (w << n) = drop\ n\ (to\text{-}bl\ w)\ @\ replicate\ (min\ (size\ w)\ n)\ False$
by (*simp add*: *shiftl-bl word-rep-drop word-size*)

lemma *shiftl-zero-size*: $size\ x \leq n \Longrightarrow x << n = 0$
for $x :: 'a::len0\ word$
apply (*unfold word-size*)
apply (*rule word-eqI*)
apply (*clarsimp simp add*: *shiftl-bl word-size test-bit-of-bl nth-append*)
done

lemma *shiftl1-2t*: $shiftl1\ w = 2 * w$
for $w :: 'a::len\ word$
by (*simp add*: *shiftl1-def Bit-def wi-hom-mult* [*symmetric*])

lemma *shiftl1-p*: $shiftl1\ w = w + w$
for $w :: 'a::len\ word$
by (*simp add*: *shiftl1-2t*)

lemma *shiftl-t2n*: $shiftl\ w\ n = 2\ \hat{}\ n * w$
for $w :: 'a::len\ word$
by (*induct n*) (*auto simp*: *shiftl-def shiftl1-2t*)

lemma *shiftr1-bintr* [*simp*]:
$(shiftr1\ (numeral\ w) :: 'a::len0\ word) =$
$word\text{-}of\text{-}int\ (bin\text{-}rest\ (bintrunc\ (len\text{-}of\ TYPE('a))\ (numeral\ w)))$
unfolding *shiftr1-def word-numeral-alt* **by** (*simp add*: *word-ubin.eq-norm*)

lemma *sshiftr1-sbintr* [*simp*]:
$(sshiftr1\ (numeral\ w) :: 'a::len\ word) =$
$word\text{-}of\text{-}int\ (bin\text{-}rest\ (sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ (numeral\ w)))$
unfolding *sshiftr1-def word-numeral-alt* **by** (*simp add*: *word-sbin.eq-norm*)

lemma *shiftr-no* [*simp*]:

(*numeral w*::*'a*::*len0 word*) >> *n* = *word-of-int*
((*bin-rest* ^^ *n*) (*bintrunc* (*len-of TYPE*(*'a*)) (*numeral w*)))
by (*rule word-eqI*) (*auto simp*: *nth-shiftr nth-rest-power-bin nth-bintr word-size*)

lemma *sshiftr-no* [*simp*]:

(*numeral w*::*'a*::*len word*) >>> *n* = *word-of-int*
((*bin-rest* ^^ *n*) (*sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*numeral w*)))
apply (*rule word-eqI*)
apply (*auto simp*: *nth-sshiftr nth-rest-power-bin nth-sbintr word-size*)
apply (*subgoal-tac na* + *n* = *len-of TYPE*(*'a*) − *Suc 0*, *simp*, *simp*)+
done

lemma *shiftr1-bl-of*:
length bl ≤ *len-of TYPE*(*'a*) ⟹
shiftr1 (*of-bl bl*::*'a*::*len0 word*) = *of-bl* (*butlast bl*)
by (*clarsimp simp*: *shiftr1-def of-bl-def butlast-rest-bl2bin word-ubin.eq-norm trunc-bl2bin*)

lemma *shiftr-bl-of*:
length bl ≤ *len-of TYPE*(*'a*) ⟹
(*of-bl bl*::*'a*::*len0 word*) >> *n* = *of-bl* (*take* (*length bl* − *n*) *bl*)
apply (*unfold shiftr-def*)
apply (*induct n*)
apply *clarsimp*
apply *clarsimp*
apply (*subst shiftr1-bl-of*)
apply *simp*
apply (*simp add*: *butlast-take*)
done

lemma *shiftr-bl*: *x* >> *n* ≡ *of-bl* (*take* (*len-of TYPE*(*'a*) − *n*) (*to-bl x*))
for *x* :: *'a*::*len0 word*
using *shiftr-bl-of* [**where** *'a*=*'a*, *of to-bl x*] **by** *simp*

lemma *msb-shift*: *msb w* ⟷ *w* >> (*len-of TYPE*(*'a*) − *1*) ≠ *0*
for *w* :: *'a*::*len word*
apply (*unfold shiftr-bl word-msb-alt*)
apply (*simp add*: *word-size Suc-le-eq take-Suc*)
apply (*cases hd* (*to-bl w*))
apply (*auto simp*: *word-1-bl of-bl-rep-False* [**where** *n*=*1* **and** *bs*=[], *simplified*])
done

lemma *zip-replicate*: *n* ≥ *length ys* ⟹ *zip* (*replicate n x*) *ys* = *map* (λ*y*. (*x*, *y*))
ys
apply (*induct ys arbitrary*: *n*)
apply *simp-all*
apply (*case-tac n*)
apply *simp-all*
done

lemma *align-lem-or* [*rule-format*] :
 $\forall x\ m.\ length\ x = n + m \longrightarrow length\ y = n + m \longrightarrow$
  drop m x = replicate n False $\longrightarrow$ *take m y = replicate m False* $\longrightarrow$
  map2 op | x y = take m x @ drop m y
 apply (*induct y*)
 apply *force*
 apply *clarsimp*
 apply (*case-tac x*)
 apply *force*
 apply (*case-tac m*)
 apply *auto*
 apply (*drule-tac t=length xs* **for** *xs* **in** *sym*)
 apply (*auto simp*: *map2-def zip-replicate o-def*)
 done

lemma *align-lem-and* [*rule-format*] :
 $\forall x\ m.\ length\ x = n + m \longrightarrow length\ y = n + m \longrightarrow$
  drop m x = replicate n False $\longrightarrow$ *take m y = replicate m False* $\longrightarrow$
  map2 op $\wedge$ *x y = replicate (n + m) False*
 apply (*induct y*)
 apply *force*
 apply *clarsimp*
 apply (*case-tac x*)
 apply *force*
 apply (*case-tac m*)
 apply *auto*
 apply (*drule-tac t=length xs* **for** *xs* **in** *sym*)
 apply (*auto simp*: *map2-def zip-replicate o-def map-replicate-const*)
 done

lemma *aligned-bl-add-size* [*OF refl*]:
 size x − n = m $\Longrightarrow$ *n* $\leq$ *size x* $\Longrightarrow$ *drop m (to-bl x) = replicate n False* $\Longrightarrow$
  take m (to-bl y) = replicate m False $\Longrightarrow$
  to-bl (x + y) = take m (to-bl x) @ drop m (to-bl y)
 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

10.25.2 Mask

lemma *nth-mask* [*OF refl*, *simp*]: $m = mask\ n \Longrightarrow test\text{-}bit\ m\ i \longleftrightarrow i < n \wedge i < size\ m$
apply (*unfold mask-def test-bit-bl*)
apply (*simp only*: *word-1-bl* [*symmetric*] *shiftl-of-bl*)
apply (*clarsimp simp add*: *word-size*)
apply (*simp only*: *of-bl-def mask-lem word-of-int-hom-syms add-diff-cancel2*)
apply (*fold of-bl-def*)
apply (*simp add*: *word-1-bl*)
apply (*rule test-bit-of-bl* [*THEN trans*, *unfolded test-bit-bl word-size*])
apply *auto*
done

lemma *mask-bl*: $mask\ n = of\text{-}bl\ (replicate\ n\ True)$
by (*auto simp add* : *test-bit-of-bl word-size intro*: *word-eqI*)

lemma *mask-bin*: $mask\ n = word\text{-}of\text{-}int\ (bintrunc\ n\ (-\ 1))$
by (*auto simp add*: *nth-bintr word-size intro*: *word-eqI*)

lemma *and-mask-bintr*: $w\ AND\ mask\ n = word\text{-}of\text{-}int\ (bintrunc\ n\ (uint\ w))$
apply (*rule word-eqI*)
apply (*simp add*: *nth-bintr word-size word-ops-nth-size*)
apply (*auto simp add*: *test-bit-bin*)
done

lemma *and-mask-wi*: $word\text{-}of\text{-}int\ i\ AND\ mask\ n = word\text{-}of\text{-}int\ (bintrunc\ n\ i)$
by (*auto simp add*: *nth-bintr word-size word-ops-nth-size word-eq-iff*)

lemma *and-mask-wi'*:
$word\text{-}of\text{-}int\ i\ AND\ mask\ n = (word\text{-}of\text{-}int\ (bintrunc\ (min\ LENGTH('a)\ n)\ i) :: 'a::len\ word)$
by (*auto simp add*: *nth-bintr word-size word-ops-nth-size word-eq-iff*)

lemma *and-mask-no*: $numeral\ i\ AND\ mask\ n = word\text{-}of\text{-}int\ (bintrunc\ n\ (numeral\ i))$
unfolding *word-numeral-alt* **by** (*rule and-mask-wi*)

lemma *bl-and-mask'*:
$to\text{-}bl\ (w\ AND\ mask\ n :: 'a::len\ word) =$
$replicate\ (len\text{-}of\ TYPE('a) - n)\ False\ @$
$drop\ (len\text{-}of\ TYPE('a) - n)\ (to\text{-}bl\ w)$
apply (*rule nth-equalityI*)
apply *simp*
apply (*clarsimp simp add*: *to-bl-nth word-size*)
apply (*simp add*: *word-size word-ops-nth-size*)
apply (*auto simp add*: *word-size test-bit-bl nth-append nth-rev*)
done

lemma *and-mask-mod-2p*: $w\ AND\ mask\ n = word\text{-}of\text{-}int\ (uint\ w\ mod\ 2\ \hat{}\ n)$

by (*simp only*: *and-mask-bintr bintrunc-mod2p*)

lemma *and-mask-lt-2p*: *uint* (*w AND mask n*) < *2* ^ *n*
apply (*simp add*: *and-mask-bintr word-ubin.eq-norm*)
apply (*simp add*: *bintrunc-mod2p*)
apply (*rule xtr8*)
prefer *2*
apply (*rule pos-mod-bound*)
apply *auto*
done

lemma *eq-mod-iff*: *0* < *n* ⟹ *b* = *b mod n* ⟷ *0* ≤ *b* ∧ *b* < *n*
for *b n* :: *int*
by (*simp add*: *int-mod-lem eq-sym-conv*)

lemma *mask-eq-iff*: *w AND mask n* = *w* ⟷ *uint w* < *2* ^ *n*
apply (*simp add*: *and-mask-bintr*)
apply (*simp add*: *word-ubin.inverse-norm*)
apply (*simp add*: *eq-mod-iff bintrunc-mod2p min-def*)
apply (*fast intro*!: *lt2p-lem*)
done

lemma *and-mask-dvd*: *2* ^ *n dvd uint w* ⟷ *w AND mask n* = *0*
apply (*simp add*: *dvd-eq-mod-eq-0 and-mask-mod-2p*)
apply (*simp add*: *word-uint.norm-eq-iff* [*symmetric*] *word-of-int-homs del*: *word-of-int-0*)
apply (*subst word-uint.norm-Rep* [*symmetric*])
apply (*simp only*: *bintrunc-bintrunc-min bintrunc-mod2p* [*symmetric*] *min-def*)
apply *auto*
done

lemma *and-mask-dvd-nat*: *2* ^ *n dvd unat w* ⟷ *w AND mask n* = *0*
apply (*unfold unat-def*)
apply (*rule trans* [*OF - and-mask-dvd*])
apply (*unfold dvd-def*)
apply *auto*
apply (*drule uint-ge-0* [*THEN nat-int.Abs-inverse'* [*simplified*], *symmetric*])
apply *simp*
apply (*simp add*: *nat-mult-distrib nat-power-eq*)
done

lemma *word-2p-lem*: *n* < *size w* ⟹ *w* < *2* ^ *n* = (*uint w* < *2* ^ *n*)
for *w* :: *'a*::*len word*
apply (*unfold word-size word-less-alt word-numeral-alt*)
apply (*auto simp add*: *word-of-int-power-hom word-uint.eq-norm mod-pos-pos-trivial*
simp del: *word-of-int-numeral*)
done

lemma *less-mask-eq*: *x* < *2* ^ *n* ⟹ *x AND mask n* = *x*
for *x* :: *'a*::*len word*

apply (*unfold word-less-alt word-numeral-alt*)
apply (*clarsimp simp add*: *and-mask-mod-2p word-of-int-power-hom word-uint.eq-norm simp del*: *word-of-int-numeral*)
apply (*drule xtr8* [*rotated*])
apply (*rule int-mod-le*)
apply (*auto simp add* : *mod-pos-pos-trivial*)
done

lemmas *mask-eq-iff-w2p* = *trans* [*OF mask-eq-iff word-2p-lem* [*symmetric*]]

lemmas *and-mask-less'* = *iffD2* [*OF word-2p-lem and-mask-lt-2p*, *simplified word-size*]

lemma *and-mask-less-size*: $n < size\ x \Longrightarrow x\ AND\ mask\ n < 2\hat{}n$
unfolding *word-size* **by** (*erule and-mask-less'*)

lemma *word-mod-2p-is-mask* [*OF refl*]: $c = 2\ \hat{}\ n \Longrightarrow c > 0 \Longrightarrow x\ mod\ c = x\ AND\ mask\ n$
for *c x* :: *'a*::*len word*
by (*auto simp*: *word-mod-def uint-2p and-mask-mod-2p*)

lemma *mask-eqs*:
$(a\ AND\ mask\ n) + b\ AND\ mask\ n = a + b\ AND\ mask\ n$
$a + (b\ AND\ mask\ n)\ AND\ mask\ n = a + b\ AND\ mask\ n$
$(a\ AND\ mask\ n) - b\ AND\ mask\ n = a - b\ AND\ mask\ n$
$a - (b\ AND\ mask\ n)\ AND\ mask\ n = a - b\ AND\ mask\ n$
$a * (b\ AND\ mask\ n)\ AND\ mask\ n = a * b\ AND\ mask\ n$
$(b\ AND\ mask\ n) * a\ AND\ mask\ n = b * a\ AND\ mask\ n$
$(a\ AND\ mask\ n) + (b\ AND\ mask\ n)\ AND\ mask\ n = a + b\ AND\ mask\ n$
$(a\ AND\ mask\ n) - (b\ AND\ mask\ n)\ AND\ mask\ n = a - b\ AND\ mask\ n$
$(a\ AND\ mask\ n) * (b\ AND\ mask\ n)\ AND\ mask\ n = a * b\ AND\ mask\ n$
$-(a\ AND\ mask\ n)\ AND\ mask\ n = -\ a\ AND\ mask\ n$
$word\text{-}succ\ (a\ AND\ mask\ n)\ AND\ mask\ n = word\text{-}succ\ a\ AND\ mask\ n$
$word\text{-}pred\ (a\ AND\ mask\ n)\ AND\ mask\ n = word\text{-}pred\ a\ AND\ mask\ n$
using *word-of-int-Ex* [**where** *x*=*a*] *word-of-int-Ex* [**where** *x*=*b*]
by (*auto simp*: *and-mask-wi' word-of-int-homs word.abs-eq-iff bintrunc-mod2p mod-simps*)

lemma *mask-power-eq*: $(x\ AND\ mask\ n)\ \hat{}\ k\ AND\ mask\ n = x\ \hat{}\ k\ AND\ mask\ n$
using *word-of-int-Ex* [**where** *x*=*x*]
by (*auto simp*: *and-mask-wi' word-of-int-power-hom word.abs-eq-iff bintrunc-mod2p mod-simps*)

10.25.3 Revcast

lemmas *revcast-def'* = *revcast-def* [*simplified*]
lemmas *revcast-def''* = *revcast-def'* [*simplified word-size*]
lemmas *revcast-no-def* [*simp*] = *revcast-def'* [**where** *w*=*numeral w*, *unfolded word-size*]
for *w*

lemma *to-bl-revcast*:
to-bl (*revcast w* :: *'a*::*len0 word*) = *takefill False* (*len-of TYPE*(*'a*)) (*to-bl w*)
apply (*unfold revcast-def' word-size*)
apply (*rule word-bl.Abs-inverse*)
apply *simp*
done

lemma *revcast-rev-ucast* [*OF refl refl refl*]:
cs = [*rc*, *uc*] ⟹ *rc* = *revcast* (*word-reverse w*) ⟹ *uc* = *ucast w* ⟹
rc = *word-reverse uc*
apply (*unfold ucast-def revcast-def' Let-def word-reverse-def*)
apply (*auto simp*: *to-bl-of-bin takefill-bintrunc*)
apply (*simp add*: *word-bl.Abs-inverse word-size*)
done

lemma *revcast-ucast*: *revcast w* = *word-reverse* (*ucast* (*word-reverse w*))
using *revcast-rev-ucast* [*of word-reverse w*] **by** *simp*

lemma *ucast-revcast*: *ucast w* = *word-reverse* (*revcast* (*word-reverse w*))
by (*fact revcast-rev-ucast* [*THEN word-rev-gal'*])

lemma *ucast-rev-revcast*: *ucast* (*word-reverse w*) = *word-reverse* (*revcast w*)
by (*fact revcast-ucast* [*THEN word-rev-gal'*])

linking revcast and cast via shift

lemmas *wsst-TYs* = *source-size target-size word-size*

lemma *revcast-down-uu* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹ *rc w* = *ucast* (*w* >> *n*)
for *w* :: *'a*::*len word*
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans*, *rule ucast-down-drop*)
prefer *2*
apply (*rule trans*, *rule drop-shiftr*)
apply (*auto simp*: *takefill-alt wsst-TYs*)
done

lemma *revcast-down-us* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹ *rc w* = *ucast* (*w* >>> *n*)
for *w* :: *'a*::*len word*
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans*, *rule ucast-down-drop*)
prefer *2*
apply (*rule trans*, *rule drop-sshiftr*)
apply (*auto simp*: *takefill-alt wsst-TYs*)
done

lemma *revcast-down-su* [*OF refl*]:
rc = revcast ⟹ source-size rc = target-size rc + n ⟹ rc w = scast (w >> n)
for *w :: 'a::len word*
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans*, *rule scast-down-drop*)
prefer *2*
apply (*rule trans*, *rule drop-shiftr*)
apply (*auto simp*: *takefill-alt wsst-TYs*)
done

lemma *revcast-down-ss* [*OF refl*]:
rc = revcast ⟹ source-size rc = target-size rc + n ⟹ rc w = scast (w >>> n)
for *w :: 'a::len word*
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans*, *rule scast-down-drop*)
prefer *2*
apply (*rule trans*, *rule drop-sshiftr*)
apply (*auto simp*: *takefill-alt wsst-TYs*)
done

lemma *cast-down-rev*:
uc = ucast ⟹ source-size uc = target-size uc + n ⟹ uc w = revcast (w << n)
for *w :: 'a::len word*
apply (*unfold shiftl-rev*)
apply *clarify*
apply (*simp add*: *revcast-rev-ucast*)
apply (*rule word-rev-gal'*)
apply (*rule trans* [*OF - revcast-rev-ucast*])
apply (*rule revcast-down-uu* [*symmetric*])
apply (*auto simp add*: *wsst-TYs*)
done

lemma *revcast-up* [*OF refl*]:
rc = revcast ⟹ source-size rc + n = target-size rc ⟹
rc w = (ucast w :: 'a::len word) << n
apply (*simp add*: *revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*simp add*: *takefill-alt*)
apply (*rule bl-shiftl* [*THEN trans*])
apply (*subst ucast-up-app*)
apply (*auto simp add*: *wsst-TYs*)
done

lemmas *rc1* = *revcast-up* [*THEN*
revcast-rev-ucast [*symmetric*, *THEN trans*, *THEN word-rev-gal*, *symmetric*]]
lemmas *rc2* = *revcast-down-uu* [*THEN*
revcast-rev-ucast [*symmetric*, *THEN trans*, *THEN word-rev-gal*, *symmetric*]]

lemmas *ucast-up* =
rc1 [*simplified rev-shiftr* [*symmetric*] *revcast-ucast* [*symmetric*]]
lemmas *ucast-down* =
rc2 [*simplified rev-shiftr revcast-ucast* [*symmetric*]]

10.25.4 Slices

lemma *slice1-no-bin* [*simp*]:
slice1 n (*numeral w* :: *'b word*) = *of-bl* (*takefill False n* (*bin-to-bl* (*len-of TYPE*(*'b*::*len0*)) (*numeral w*)))
by (*simp add*: *slice1-def*)

lemma *slice-no-bin* [*simp*]:
slice n (*numeral w* :: *'b word*) = *of-bl* (*takefill False* (*len-of TYPE*(*'b*::*len0*) − *n*)
(*bin-to-bl* (*len-of TYPE*(*'b*::*len0*)) (*numeral w*)))
by (*simp add*: *slice-def word-size*)

lemma *slice1-0* [*simp*] : *slice1 n 0* = *0*
unfolding *slice1-def* **by** *simp*

lemma *slice-0* [*simp*] : *slice n 0* = *0*
unfolding *slice-def* **by** *auto*

lemma *slice-take'*: *slice n w* = *of-bl* (*take* (*size w* − *n*) (*to-bl w*))
unfolding *slice-def' slice1-def*
by (*simp add* : *takefill-alt word-size*)

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 *slice-shiftr*: *slice n w* = *ucast* (*w* >> *n*)
apply (*unfold slice-take shiftr-bl*)
apply (*rule ucast-of-bl-up* [*symmetric*])
apply (*simp add*: *word-size*)
done

lemma *nth-slice*: (*slice n w* :: *'a*::*len0 word*) !! *m* = (*w* !! (*m* + *n*) ∧ *m* < *len-of TYPE*(*'a*))
by (*simp add*: *slice-shiftr nth-ucast nth-shiftr*)

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))
by (*auto simp*: *slice1-def word-size of-bl-def uint-bl*
word-ubin.eq-norm bl-bin-bl-rep-drop drop-takefill)

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 (*unfold slice1-def word-size of-bl-def uint-bl*)
apply (*clarsimp simp*: *word-ubin.eq-norm bl-bin-bl-rep-drop takefill-append* [*symmetric*])
apply (*rule-tac f = λk. takefill False* (*len-of TYPE('a)*)
(*replicate k False @ bin-to-bl* (*len-of TYPE('b)*) (*uint w*)) **in** *arg-cong*)
apply *arith*
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 *ucast-slice1*: *ucast w = slice1* (*size w*) *w*
by (*simp add*: *slice1-def ucast-bl takefill-same' word-size*)

lemma *ucast-slice*: *ucast w = slice 0 w*
by (*simp add*: *slice-def ucast-slice1*)

lemma *slice-id*: *slice 0 t = t*
by (*simp only*: *ucast-slice* [*symmetric*] *ucast-id*)

lemma *revcast-slice1* [*OF refl*]: *rc = revcast w ⟹ slice1* (*size rc*) *w = rc*
by (*simp add*: *slice1-def revcast-def' word-size*)

lemma *slice1-tf-tf'*:
to-bl (*slice1 n w :: 'a::len0 word*) =
rev (*takefill False* (*len-of TYPE('a)*) (*rev* (*takefill False n* (*to-bl w*))))
unfolding *slice1-def* **by** (*rule word-rev-tf*)

lemmas *slice1-tf-tf = slice1-tf-tf'* [*THEN word-bl.Rep-inverse'*, *symmetric*]

lemma *rev-slice1*:
n + k = len-of TYPE('a) + len-of TYPE('b) ⟹
slice1 n (*word-reverse w :: 'b::len0 word*) =
word-reverse (*slice1 k w :: 'a::len0 word*)
apply (*unfold word-reverse-def slice1-tf-tf*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule rev-swap* [*THEN iffD1*])
apply (*rule trans* [*symmetric*])

```
  apply (rule tf-rev)
  apply (simp add: word-bl.Abs-inverse)
 apply (simp add: word-bl.Abs-inverse)
 done

lemma rev-slice:
 n + k + len-of TYPE('a::len0) = len-of TYPE('b::len0) ⟹
   slice n (word-reverse (w::'b word)) = word-reverse (slice k w :: 'a word)
 apply (unfold slice-def word-size)
 apply (rule rev-slice1)
 apply arith
 done

lemmas sym-notr =
 not-iff [THEN iffD2, THEN not-sym, THEN not-iff [THEN iffD1]]
```

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

```
lemma sofl-test:
 (sint x + sint y = sint (x + y)) =
   ((((x + y) XOR x) AND ((x + y) XOR y)) >> (size x - 1) = 0)
 for x y :: 'a::len word
 apply (unfold word-size)
 apply (cases len-of TYPE('a), simp)
 apply (subst msb-shift [THEN sym-notr])
 apply (simp add: word-ops-msb)
 apply (simp add: word-msb-sint)
 apply safe
     apply simp-all
   apply (unfold sint-word-ariths)
   apply (unfold word-sbin.set-iff-norm [symmetric] sints-num)
   apply safe
      apply (insert sint-range' [where x=x])
      apply (insert sint-range' [where x=y])
      defer
      apply (simp (no-asm), arith)
     apply (simp (no-asm), arith)
    defer
    defer
    apply (simp (no-asm), arith)
   apply (simp (no-asm), arith)
  apply (rule notI [THEN notnotD],
    drule leI not-le-imp-less,
    drule sbintrunc-inc sbintrunc-dec,
    simp)+
 done
```

10.26 Split and cat

lemmas *word-split-bin' = word-split-def*
lemmas *word-cat-bin' = word-cat-def*

lemma *word-rsplit-no*:
(*word-rsplit* (*numeral bin* :: *'b*::*len0 word*) :: *'a word list*) =
map word-of-int (*bin-rsplit* (*len-of TYPE*(*'a*::*len*))
(*len-of TYPE*(*'b*), *bintrunc* (*len-of TYPE*(*'b*)) (*numeral bin*)))
by (*simp add*: *word-rsplit-def word-ubin.eq-norm*)

lemmas *word-rsplit-no-cl* [*simp*] = *word-rsplit-no*
[*unfolded bin-rsplitl-def bin-rsplit-l* [*symmetric*]]

lemma *test-bit-cat*:
wc = *word-cat a b* ⟹ *wc* !! *n* = (*n* < *size wc* ∧
(*if n* < *size b then b* !! *n else a* !! (*n* − *size b*)))
apply (*auto simp*: *word-cat-bin' test-bit-bin word-ubin.eq-norm nth-bintr bin-nth-cat word-size*)
apply (*erule bin-nth-uint-imp*)
done

lemma *word-cat-bl*: *word-cat a b* = *of-bl* (*to-bl a* @ *to-bl b*)
by (*simp add*: *of-bl-def to-bl-def word-cat-bin' bl-to-bin-app-cat*)

lemma *of-bl-append*:
(*of-bl* (*xs* @ *ys*) :: *'a*::*len word*) = *of-bl xs* ∗ *2*^(*length ys*) + *of-bl ys*
apply (*simp add*: *of-bl-def bl-to-bin-app-cat bin-cat-num*)
apply (*simp add*: *word-of-int-power-hom* [*symmetric*] *word-of-int-hom-syms*)
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 *split-uint-lem*: *bin-split n* (*uint w*) = (*a*, *b*) ⟹
a = *bintrunc* (*len-of TYPE*(*'a*) − *n*) *a* ∧ *b* = *bintrunc* (*len-of TYPE*(*'a*)) *b*
for *w* :: *'a*::*len0 word*
apply (*frule word-ubin.norm-Rep* [*THEN ssubst*])
apply (*drule bin-split-trunc1*)
apply (*drule sym* [*THEN trans*])
apply *assumption*
apply *safe*
done

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 (*unfold word-split-bin'*)
apply *safe*
defer
apply (*clarsimp split*: *prod.splits*)
apply *hypsubst-thin*
apply (*drule word-ubin.norm-Rep* [*THEN ssubst*])
apply (*drule split-bintrunc*)
apply (*simp add*: *of-bl-def bl2bin-drop word-size*
word-ubin.norm-eq-iff [*symmetric*] *min-def del*: *word-ubin.norm-Rep*)
apply (*clarsimp split*: *prod.splits*)
apply (*frule split-uint-lem* [*THEN conjunct1*])
apply (*unfold word-size*)
apply (*cases len-of TYPE('a) ≥ len-of TYPE('b)*)
defer
apply *simp*
apply (*simp add* : *of-bl-def to-bl-def*)
apply (*subst bin-split-take1* [*symmetric*])
prefer *2*
apply *assumption*
apply *simp*
apply (*erule thin-rl*)
apply (*erule arg-cong* [*THEN trans*])
apply (*simp add* : *word-ubin.norm-eq-iff* [*symmetric*])
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::len0 word × 'd::len0 word)) =
(of-bl (take (len-of TYPE('a::len) − len-of TYPE('d::len0)) (to-bl c)),
of-bl (drop (len-of TYPE('a) − len-of TYPE('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

— keep quantifiers for use in simplification
lemma *test-bit-split'*:
word-split c = (a, b) ⟶
(∀ n m.
b !! n = (n < size b ∧ c !! n) ∧
a !! m = (m < size a ∧ c !! (m + size b)))
apply (*unfold word-split-bin' test-bit-bin*)
apply (*clarify*)
apply (*clarsimp simp*: *word-ubin.eq-norm nth-bintr word-size split*: *prod.splits*)
apply (*drule bin-nth-split*)
apply *safe*
apply (*simp-all add*: *add.commute*)
apply (*erule bin-nth-uint-imp*)+
done

lemma *test-bit-split*:
word-split c = (a, b) ⟹
(∀ n::nat. b !! n ⟷ n < size b ∧ c !! n) ∧
(∀ m::nat. a !! m ⟷ m < size a ∧ c !! (m + size b))
by (*simp add*: *test-bit-split'*)

lemma *test-bit-split-eq*:
word-split c = (a, b) ⟷
((∀ n::nat. b !! n = (n < size b ∧ c !! n)) ∧
(∀ m::nat. a !! m = (m < size a ∧ c !! (m + size b))))
apply (*rule-tac iffI*)
apply (*rule-tac conjI*)
apply (*erule test-bit-split* [*THEN conjunct1*])
apply (*erule test-bit-split* [*THEN conjunct2*])
apply (*case-tac word-split c*)
apply (*frule test-bit-split*)
apply (*erule trans*)
apply (*fastforce intro*!: *word-eqI simp add*: *word-size*)
done

— this odd result is analogous to *ucast-id*, result to the length given by the result type

lemma *word-cat-id*: *word-cat a b = b*
by (*simp add*: *word-cat-bin' word-ubin.inverse-norm*)

— limited hom result
lemma *word-cat-hom*:
len-of TYPE('a::len0) ≤ len-of TYPE('b::len0) + len-of TYPE('c::len0) ⟹
(word-cat (word-of-int w :: 'b word) (b :: 'c word) :: 'a word) =
word-of-int (bin-cat w (size b) (uint b))
by (*auto simp*: *word-cat-def word-size word-ubin.norm-eq-iff* [*symmetric*]

word-ubin.eq-norm bintr-cat min.absorb1)

lemma *word-cat-split-alt*: *size w* ≤ *size u* + *size v* ⟹ *word-split w* = (*u*, *v*) ⟹ *word-cat u v* = *w*
apply (*rule word-eqI*)
apply (*drule test-bit-split*)
apply (*clarsimp simp add* : *test-bit-cat word-size*)
apply *safe*
apply *arith*
done

lemmas *word-cat-split-size* = *sym* [*THEN* [*2*] *word-cat-split-alt* [*symmetric*]]

10.26.1 Split and slice

lemma *split-slices*: *word-split w* = (*u*, *v*) ⟹ *u* = *slice* (*size v*) *w* ∧ *v* = *slice 0 w*
apply (*drule test-bit-split*)
apply (*rule conjI*)
apply (*rule word-eqI*, *clarsimp simp*: *nth-slice word-size*)+
done

lemma *slice-cat1* [*OF refl*]:
wc = *word-cat a b* ⟹ *size wc* >= *size a* + *size b* ⟹ *slice* (*size b*) *wc* = *a*
apply *safe*
apply (*rule word-eqI*)
apply (*simp add*: *nth-slice test-bit-cat word-size*)
done

lemmas *slice-cat2* = *trans* [*OF slice-id word-cat-id*]

lemma *cat-slices*:
a = *slice n c* ⟹ *b* = *slice 0 c* ⟹ *n* = *size b* ⟹
size a + *size b* >= *size c* ⟹ *word-cat a b* = *c*
apply *safe*
apply (*rule word-eqI*)
apply (*simp add*: *nth-slice test-bit-cat word-size*)
apply *safe*
apply *arith*
done

lemma *word-split-cat-alt*:
w = *word-cat u v* ⟹ *size u* + *size v* ≤ *size w* ⟹ *word-split w* = (*u*, *v*)
apply (*case-tac word-split w*)
apply (*rule trans*, *assumption*)
apply (*drule test-bit-split*)
apply *safe*
apply (*rule word-eqI*, *clarsimp simp*: *test-bit-cat word-size*)+
done

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*

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*]
by (*simp add*: *word-rsplit-def bin-rsplit-all*)

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* ⟹ (*rev sw* ! *k*) !! *m* = *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* = λ*x*. *bin-nth 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* : *word-ubin.eq-norm*)
apply (*erule bin-rsplit-size-sign* [*OF len-gt-0 refl*])
done

lemma *word-rcat-bl*: *word-rcat wl* = *of-bl* (*concat* (*map to-bl wl*))
by (*auto simp*: *word-rcat-def to-bl-def' of-bl-def bin-rcat-bl*)

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*]

lemmas *td-gal-lt-len* = *len-gt-0* [*THEN td-gal-lt*]

lemma *nth-rcat-lem*:
n < *length* (*wl*::*'a word list*) ∗ *len-of TYPE*(*'a*::*len*) ⟹

```
    rev (concat (map to-bl wl)) ! n =
    rev (to-bl (rev wl ! (n div len-of TYPE('a)))) ! (n mod len-of TYPE('a))
  apply (induct wl)
   apply clarsimp
  apply (clarsimp simp add : nth-append size-rcat-lem)
  apply (simp (no-asm-use) only:  mult-Suc [symmetric]
        td-gal-lt-len less-Suc-eq-le minus-div-mult-eq-mod [symmetric])
  apply clarsimp
  done

lemma test-bit-rcat:
  sw = size (hd wl) ⟹ rc = word-rcat wl ⟹ rc !! n =
    (n < size rc ∧ n div sw < size wl ∧ (rev wl) ! (n div sw) !! (n mod sw))
  for wl :: 'a::len word list
  apply (unfold word-rcat-bl word-size)
  apply (clarsimp simp add: test-bit-of-bl size-rcat-lem word-size td-gal-lt-len)
  apply safe
  apply (auto simp: test-bit-bl word-size td-gal-lt-len [THEN iffD2, THEN nth-rcat-lem])
  done

lemma foldl-eq-foldr: foldl op + x xs = foldr op + (x # xs) 0
  for x :: 'a::comm-monoid-add
  by (induct xs arbitrary: x) (auto simp: add.assoc)

lemmas test-bit-cong = arg-cong [where f = test-bit, THEN fun-cong]

lemmas test-bit-rsplit-alt =
  trans [OF nth-rev-alt [THEN test-bit-cong]
  test-bit-rsplit [OF refl asm-rl diff-Suc-less]]

— lazy way of expressing that u and v, and su and sv, have same types
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 = len-of TYPE('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 = len-of TYPE('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 = len-of TYPE('a::len) ⟹ size w = m * n ⟹*
length (word-rsplit w :: 'a word list) = m
by (*auto simp*: *length-word-rsplit-exp-size given-quot-alt*)

lemmas *length-word-rsplit-exp-size' = refl* [*THEN length-word-rsplit-exp-size*]

lemmas *tdle = iffD2* [*OF split-div-lemma refl*, *THEN conjunct1*]
lemmas *dtle = xtr4* [*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 xtr7*, *rule len-gt-0* [*THEN dtle*])+
done

lemma *size-word-rsplit-rcat-size*:
*word-rcat ws = frcw ⟹ size frcw = length ws * len-of TYPE('a)*
⟹ length (word-rsplit frcw::'a word list) = length ws
for *ws* :: *'a::len word list* **and** *frcw* :: *'b::len0 word*
apply (*clarsimp simp*: *word-size length-word-rsplit-exp-size'*)
apply (*fast intro*: *given-quot-alt*)
done

lemma *msrevs*:
*0 < n ⟹ (k * n + m) div n = m div n + k*
*(k * n + m) mod n = m mod n*
for *n* :: *nat*
by (*auto simp*: *add.commute*)

lemma *word-rsplit-rcat-size* [*OF refl*]:
word-rcat ws = frcw ⟹
*size frcw = length ws * len-of TYPE('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 nth-rev*)
apply *arith*
apply (*simp add*: *le0* [*THEN* [*2*] *xtr7*, *THEN diff-Suc-less*])
apply *safe*
apply (*simp add*: *diff-mult-distrib*)
apply (*rule mpl-lem*)
apply (*cases size ws*)
apply *simp-all*
done

10.27 Rotation

lemmas *rotater-0'* [*simp*] = *rotater-def* [**where** *n* = *0*, *simplified*]

lemmas *word-rot-defs* = *word-roti-def word-rotr-def word-rotl-def*

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

lemmas *rotate-eqs* =
trans [*OF rotate0* [*THEN fun-cong*] *id-apply*]
rotate-rotate [*symmetric*]
rotate-id
rotate-conv-mod
rotate-eq-mod

10.27.1 Rotation of list to right

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 *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 *restrict-to-left*: *x = y ⟹ x = z ⟷ y = z*
by *simp*

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*)

10.27.2 map, map2, commuting with rotate(r)

lemma *butlast-map*: $xs \neq [] \Longrightarrow butlast\ (map\ f\ xs) = map\ f\ (butlast\ xs)$
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*] :
$\forall\, ys.\ length\ xs = length\ ys \longrightarrow xs \neq [] \longrightarrow$
$last\ (zip\ xs\ ys) = (last\ xs,\ last\ ys) \wedge$
$butlast\ (zip\ xs\ ys) = zip\ (butlast\ xs)\ (butlast\ ys)$
apply (*induct xs*)
apply *auto*
apply ((*case-tac ys*, *auto simp*: *neq-Nil-conv*)[*1*])+
done

lemma *but-last-map2* [*rule-format*] :
$\forall\, ys.\ length\ xs = length\ ys \longrightarrow xs \neq [] \longrightarrow$
$last\ (map2\ f\ xs\ ys) = f\ (last\ xs)\ (last\ ys) \wedge$
$butlast\ (map2\ f\ xs\ ys) = map2\ f\ (butlast\ xs)\ (butlast\ ys)$
apply (*induct xs*)
apply *auto*
apply (*unfold map2-def*)
apply ((*case-tac ys*, *auto simp*: *neq-Nil-conv*)[*1*])+
done

lemma *rotater1-zip*:
$length\ xs = length\ ys \Longrightarrow$
$rotater1\ (zip\ xs\ ys) = zip\ (rotater1\ xs)\ (rotater1\ ys)$
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*:
$length\ xs = length\ ys \Longrightarrow$
$rotater1\ (map2\ f\ xs\ ys) = map2\ f\ (rotater1\ xs)\ (rotater1\ ys)$
by (*simp add*: *map2-def 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 simp*: *map2-def*)

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*)

— corresponding equalities for word rotation

lemma *to-bl-rotl*: *to-bl (word-rotl n w) = rotate n (to-bl w)*
by (*simp add*: *word-bl.Abs-inverse' word-rotl-def*)

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-bl.Abs-inverse' word-rotr-def*)

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*]

lemma *word-rot-rl* [*simp*]: *word-rotl k (word-rotr k v) = v*
and *word-rot-lr* [*simp*]: *word-rotr k (word-rotl k v) = v*
by (*auto simp add*: *to-bl-rotr to-bl-rotl word-bl.Rep-inject* [*symmetric*])

lemma *word-rot-gal*: *word-rotr n v = w ⟷ word-rotl n w = v*
and *word-rot-gal'*: *w = word-rotr n v ⟷ v = word-rotl n w*
by (*auto simp*: *to-bl-rotr to-bl-rotl word-bl.Rep-inject* [*symmetric*] *dest*: *sym*)

lemma *word-rotr-rev*: *word-rotr n w = word-reverse (word-rotl n (word-reverse w))*

by (*simp only*: *word-bl.Rep-inject* [*symmetric*] *to-bl-word-rev to-bl-rotr to-bl-rotl rotater-rev*)

lemma *word-roti-0* [*simp*]: *word-roti 0 w = w*
by (*auto simp*: *word-rot-defs*)

lemmas *abl-cong = arg-cong* [**where** *f = of-bl*]

lemma *word-roti-add*: *word-roti (m + n) w = word-roti m (word-roti n w)*
proof −
have *rotater-eq-lem*: ⋀*m n xs*. *m = n* ⟹ *rotater m xs = rotater n xs*
by *auto*

have *rotate-eq-lem*: ⋀*m n xs*. *m = n* ⟹ *rotate m xs = rotate n xs*
by *auto*

note *rpts* [*symmetric*] =
rotate-inv-plus [*THEN conjunct1*]
rotate-inv-plus [*THEN conjunct2*, *THEN conjunct1*]
rotate-inv-plus [*THEN conjunct2*, *THEN conjunct2*, *THEN conjunct1*]
rotate-inv-plus [*THEN conjunct2*, *THEN conjunct2*, *THEN conjunct2*]

note *rrp = trans* [*symmetric*, *OF rotate-rotate rotate-eq-lem*]
note *rrrp = trans* [*symmetric*, *OF rotater-add* [*symmetric*] *rotater-eq-lem*]

show *?thesis*
apply (*unfold word-rot-defs*)
apply (*simp only*: *split*: *if-split*)
apply (*safe intro*!: *abl-cong*)
apply (*simp-all only*: *to-bl-rotl* [*THEN word-bl.Rep-inverse'*]
to-bl-rotl
to-bl-rotr [*THEN word-bl.Rep-inverse'*]
to-bl-rotr)
apply (*rule rrp rrrp rpts*,
simp add: *nat-add-distrib* [*symmetric*]
nat-diff-distrib [*symmetric*])+
done
qed

lemma *word-roti-conv-mod'*: *word-roti n w = word-roti (n mod int (size w)) w*
apply (*unfold word-rot-defs*)
apply (*cut-tac y=size w* **in** *gt-or-eq-0*)
apply (*erule disjE*)
apply *simp-all*
apply (*safe intro*!: *abl-cong*)
apply (*rule rotater-eqs*)
apply (*simp add*: *word-size nat-mod-distrib*)
apply (*simp add*: *rotater-add* [*symmetric*] *rotate-gal* [*symmetric*])
apply (*rule rotater-eqs*)

apply (*simp add*: *word-size nat-mod-distrib*)
apply (*rule of-nat-eq-0-iff* [*THEN iffD1*])
apply (*auto simp add*: *not-le mod-eq-0-iff-dvd zdvd-int nat-add-distrib* [*symmetric*])
using *mod-mod-trivial mod-eq-dvd-iff*
apply *blast*
done

lemmas *word-roti-conv-mod* = *word-roti-conv-mod'* [*unfolded word-size*]

10.27.3 "Word rotation commutes with bit-wise operations

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*

lemma *word-rot-logs*:
word-rotl n (*NOT v*) = *NOT word-rotl n v*
word-rotr n (*NOT v*) = *NOT word-rotr n v*
word-rotl n (*x AND y*) = *word-rotl n x AND word-rotl n y*
word-rotr n (*x AND y*) = *word-rotr n x AND word-rotr n y*
word-rotl n (*x OR y*) = *word-rotl n x OR word-rotl n y*
word-rotr n (*x OR y*) = *word-rotr n x OR word-rotr n y*
word-rotl n (*x XOR y*) = *word-rotl n x XOR word-rotl n y*
word-rotr n (*x XOR y*) = *word-rotr n x XOR word-rotr n y*
by (*rule word-bl.Rep-eqD*,
rule word-rot-defs' [*THEN trans*],
simp only: *blwl-syms* [*symmetric*],
rule th1s [*THEN trans*],
rule refl)+
end

lemmas *word-rot-logs* = *word-rotate.word-rot-logs*

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-def*)
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 (*simp add*: *nat-mult-distrib*)
apply (*simp add*: *nat-diff-distrib* [*OF pos-mod-sign pos-mod-conj*
[*THEN conjunct2*, *THEN order-less-imp-le*]]
nat-mod-distrib)
apply (*simp add*: *nat-mod-distrib*)
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*]]

lemma *word-rotx-0* [*simp*] : *word-rotr i 0 = 0 ∧ word-rotl i 0 = 0*
by (*simp add*: *word-rotr-dt word-rotl-dt replicate-add* [*symmetric*])

lemma *word-roti-0'* [*simp*] : *word-roti n 0 = 0*
by (*auto simp*: *word-roti-def*)

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*

declare *word-roti-def* [*simp*]

10.28 Maximum machine word

lemma *word-int-cases*:
fixes *x* :: *'a::len0 word*
obtains *n* **where** *x = word-of-int n* **and** *0 ≤ n* **and** *n < 2^len-of TYPE('a)*
by (*cases x rule*: *word-uint.Abs-cases*) (*simp add*: *uints-num*)

lemma *word-nat-cases* [*cases type*: *word*]:
fixes *x* :: *'a::len word*
obtains *n* **where** *x = of-nat n* **and** *n < 2^len-of TYPE('a)*
by (*cases x rule*: *word-unat.Abs-cases*) (*simp add*: *unats-def*)

lemma *max-word-eq*: (*max-word*::'*a*::*len word*) = *2^len-of TYPE*('*a*) − *1*
by (*simp add*: *max-word-def word-of-int-hom-syms word-of-int-2p*)

lemma *max-word-max* [*simp*,*intro*!]: *n* ≤ *max-word*
by (*cases n rule*: *word-int-cases*)
(*simp add*: *max-word-def word-le-def int-word-uint mod-pos-pos-trivial del*: *minus-mod-self1*)

lemma *word-of-int-2p-len*: *word-of-int* (*2* ^ *len-of TYPE*('*a*)) = (*0*::'*a*::*len0 word*)
by (*subst word-uint.Abs-norm* [*symmetric*]) *simp*

lemma *word-pow-0*: (*2*::'*a*::*len word*) ^ *len-of TYPE*('*a*) = *0*
proof −
have *word-of-int* (*2* ^ *len-of TYPE*('*a*)) = (*0*::'*a word*)
by (*rule word-of-int-2p-len*)
then show *?thesis* **by** (*simp add*: *word-of-int-2p*)
qed

lemma *max-word-wrap*: *x* + *1* = *0* ⟹ *x* = *max-word*
apply (*simp add*: *max-word-eq*)
apply *uint-arith*
apply (*auto simp*: *word-pow-0*)
done

lemma *max-word-minus*: *max-word* = (−*1*::'*a*::*len word*)
proof −
have −*1* + *1* = (*0*::'*a word*)
by *simp*
then show *?thesis*
by (*rule max-word-wrap* [*symmetric*])
qed

lemma *max-word-bl* [*simp*]: *to-bl* (*max-word*::'*a*::*len word*) = *replicate* (*len-of TYPE*('*a*)) *True*
by (*subst max-word-minus to-bl-n1*)+ *simp*

lemma *max-test-bit* [*simp*]: (*max-word*::'*a*::*len word*) !! *n* ⟷ *n* < *len-of TYPE*('*a*)
by (*auto simp*: *test-bit-bl word-size*)

lemma *word-and-max* [*simp*]: *x AND max-word* = *x*
by (*rule word-eqI*) (*simp add*: *word-ops-nth-size word-size*)

lemma *word-or-max* [*simp*]: *x OR max-word* = *max-word*
by (*rule word-eqI*) (*simp add*: *word-ops-nth-size word-size*)

lemma *word-ao-dist2*: *x AND* (*y OR z*) = *x AND y OR x AND z*
for *x y z* :: '*a*::*len0 word*
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

lemma *word-oa-dist2*: *x OR y AND z = (x OR y) AND (x OR z)*
for *x y z* :: *'a::len0 word*
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

lemma *word-and-not* [*simp*]: *x AND NOT x = 0*
for *x* :: *'a::len0 word*
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

lemma *word-or-not* [*simp*]: *x OR NOT x = max-word*
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

lemma *word-boolean*: *boolean (op AND) (op OR) bitNOT 0 max-word*
apply (*rule boolean.intro*)
apply (*rule word-bw-assocs*)
apply (*rule word-bw-assocs*)
apply (*rule word-bw-comms*)
apply (*rule word-bw-comms*)
apply (*rule word-ao-dist2*)
apply (*rule word-oa-dist2*)
apply (*rule word-and-max*)
apply (*rule word-log-esimps*)
apply (*rule word-and-not*)
apply (*rule word-or-not*)
done

interpretation *word-bool-alg*: *boolean op AND op OR bitNOT 0 max-word*
by (*rule word-boolean*)

lemma *word-xor-and-or*: *x XOR y = x AND NOT y OR NOT x AND y*
for *x y* :: *'a::len0 word*
by (*rule word-eqI*) (*auto simp add*: *word-ops-nth-size word-size*)

interpretation *word-bool-alg*: *boolean-xor op AND op OR bitNOT 0 max-word op XOR*
apply (*rule boolean-xor.intro*)
apply (*rule word-boolean*)
apply (*rule boolean-xor-axioms.intro*)
apply (*rule word-xor-and-or*)
done

lemma *shiftr-x-0* [*iff*]: *x >> 0 = x*
for *x* :: *'a::len0 word*
by (*simp add*: *shiftr-bl*)

lemma *shiftl-x-0* [*simp*]: *x << 0 = x*
for *x* :: *'a::len word*
by (*simp add*: *shiftl-t2n*)

lemma *shiftl-1* [*simp*]: (*1*::*'a*::*len word*) << *n* = *2*^*n*
by (*simp add*: *shiftl-t2n*)

lemma *uint-lt-0* [*simp*]: *uint x* < *0* = *False*
by (*simp add*: *linorder-not-less*)

lemma *shiftr1-1* [*simp*]: *shiftr1* (*1*::*'a*::*len word*) = *0*
unfolding *shiftr1-def* **by** *simp*

lemma *shiftr-1*[*simp*]: (*1*::*'a*::*len word*) >> *n* = (*if n* = *0 then 1 else 0*)
by (*induct n*) (*auto simp*: *shiftr-def*)

lemma *word-less-1* [*simp*]: *x* < *1* ⟷ *x* = *0*
for *x* :: *'a*::*len word*
by (*simp add*: *word-less-nat-alt unat-0-iff*)

lemma *to-bl-mask*:
to-bl (*mask n* :: *'a*::*len word*) =
replicate (*len-of TYPE*(*'a*) − *n*) *False* @
replicate (*min* (*len-of TYPE*(*'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*

lemma *bl-and-mask*:
fixes *w* :: *'a*::*len word*
and *n* :: *nat*
defines *n'* ≡ *len-of TYPE*(*'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 op* ∧ (*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 op* ∧ ... (*to-bl* (*mask n*::*'a*::*len word*)) =
replicate n' False @ *drop n'* (*to-bl w*)
unfolding *to-bl-mask n'-def map2-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*)

lemma *map-nth-0* [*simp*]: *map* (*op* !! (*0*::'*a*::*len0 word*)) *xs* = *replicate* (*length xs*) *False*
by (*induct xs*) *auto*

lemma *uint-plus-if-size*:
uint (*x* + *y*) =
(*if uint x* + *uint y* < *2*^*size x*
then uint x + *uint y*
else uint x + *uint y* − *2*^*size x*)
by (*simp add*: *word-arith-wis int-word-uint mod-add-if-z word-size*)

lemma *unat-plus-if-size*:
unat (*x* + *y*) =
(*if unat x* + *unat y* < *2*^*size x*
then unat x + *unat y*
else unat x + *unat y* − *2*^*size x*)
for *x y* :: '*a*::*len word*
apply (*subst word-arith-nat-defs*)
apply (*subst unat-of-nat*)
apply (*simp add*: *mod-nat-add word-size*)
done

lemma *word-neq-0-conv*: *w* ≠ *0* ⟷ *0* < *w*
for *w* :: '*a*::*len word*
by (*simp add*: *word-gt-0*)

lemma *max-lt*: *unat* (*max a b div c*) = *unat* (*max a b*) *div unat c*
for *c* :: '*a*::*len word*
by (*fact unat-div*)

lemma *uint-sub-if-size*:
uint (*x* − *y*) =
(*if uint y* ≤ *uint x*
then uint x − *uint y*
else uint x − *uint y* + *2*^*size x*)
by (*simp add*: *word-arith-wis int-word-uint mod-sub-if-z word-size*)

lemma *unat-sub*: *b* ≤ *a* ⟹ *unat* (*a* − *b*) = *unat a* − *unat b*
by (*simp add*: *unat-def uint-sub-if-size word-le-def nat-diff-distrib*)

lemmas *word-less-sub1-numberof* [*simp*] = *word-less-sub1* [*of numeral w*] **for** *w*
lemmas *word-le-sub1-numberof* [*simp*] = *word-le-sub1* [*of numeral w*] **for** *w*

lemma *word-of-int-minus*: *word-of-int* (*2*^*len-of TYPE*('*a*) − *i*) = (*word-of-int*

$(-i)::'a::len\ word)$
proof −
have ∗: *2^len-of TYPE('a) − i = −i + 2^len-of TYPE('a)*
by *simp*
show *?thesis*
apply (*subst* ∗)
apply (*subst word-uint.Abs-norm* [*symmetric*], *subst mod-add-self2*)
apply *simp*
done
qed

lemmas *word-of-int-inj* =
word-uint.Abs-inject [*unfolded uints-num*, *simplified*]

lemma *word-le-less-eq*: $x \leq y \longleftrightarrow x = y \vee x < y$
for $x\ y :: 'z::len\ word$
by (*auto simp add*: *order-class.le-less*)

lemma *mod-plus-cong*:
fixes $b\ b' :: int$
assumes *1*: $b = b'$
and *2*: $x\ mod\ b' = x'\ mod\ b'$
and *3*: $y\ mod\ b' = y'\ mod\ b'$
and *4*: $x' + y' = z'$
shows $(x + y)\ mod\ b = z'\ mod\ b'$
proof −
from *1 2*[*symmetric*] *3*[*symmetric*] **have** $(x + y)\ mod\ b = (x'\ mod\ b' + y'\ mod\ b')\ mod\ b'$
by (*simp add*: *mod-add-eq*)
also have $\ldots = (x' + y')\ mod\ b'$
by (*simp add*: *mod-add-eq*)
finally show *?thesis*
by (*simp add*: *4*)
qed

lemma *mod-minus-cong*:
fixes $b\ b' :: int$
assumes $b = b'$
and $x\ mod\ b' = x'\ mod\ b'$
and $y\ mod\ b' = y'\ mod\ b'$
and $x' - y' = z'$
shows $(x - y)\ mod\ b = z'\ mod\ b'$
using *assms* [*symmetric*] **by** (*auto intro*: *mod-diff-cong*)

lemma *word-induct-less*: $P\ 0 \Longrightarrow (\bigwedge n.\ n < m \Longrightarrow P\ n \Longrightarrow P\ (1 + n)) \Longrightarrow P\ m$
for $P :: 'a::len\ word \Rightarrow bool$
apply (*cases m*)
apply *atomize*
apply (*erule rev-mp*)+

apply (*rule-tac x=m* **in** *spec*)
apply (*induct-tac n*)
apply *simp*
apply *clarsimp*
apply (*erule impE*)
apply *clarsimp*
apply (*erule-tac x=n* **in** *allE*)
apply (*erule impE*)
apply (*simp add*: *unat-arith-simps*)
apply (*clarsimp simp*: *unat-of-nat*)
apply *simp*
apply (*erule-tac x=of-nat na* **in** *allE*)
apply (*erule impE*)
apply (*simp add*: *unat-arith-simps*)
apply (*clarsimp simp*: *unat-of-nat*)
apply *simp*
done

lemma *word-induct*: $P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (1 + n)) \Longrightarrow P\ m$
for $P :: 'a::len\ word \Rightarrow bool$
by (*erule word-induct-less*) *simp*

lemma *word-induct2* [*induct type*]: $P\ 0 \Longrightarrow (\bigwedge n.\ 1 + n \neq 0 \Longrightarrow P\ n \Longrightarrow P\ (1 + n)) \Longrightarrow P\ n$
for $P :: 'b::len\ word \Rightarrow bool$
apply (*rule word-induct*)
apply *simp*
apply (*case-tac 1 + n = 0*)
apply *auto*
done

10.29 Recursion combinator for words

definition *word-rec* :: $'a \Rightarrow ('b::len\ word \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b\ word \Rightarrow 'a$
where *word-rec forZero forSuc n = rec-nat forZero* (*forSuc* ∘ *of-nat*) (*unat n*)

lemma *word-rec-0*: *word-rec z s 0 = z*
by (*simp add*: *word-rec-def*)

lemma *word-rec-Suc*: $1 + n \neq 0 \Longrightarrow$ *word-rec z s* $(1 + n) =$ *s n* (*word-rec z s n*)
for $n :: 'a::len\ word$
apply (*simp add*: *word-rec-def unat-word-ariths*)
apply (*subst nat-mod-eq'*)
apply (*metis Suc-eq-plus1-left Suc-lessI of-nat-2p unat-1 unat-lt2p word-arith-nat-add*)
apply *simp*
done

lemma *word-rec-Pred*: $n \neq 0 \Longrightarrow$ *word-rec z s n = s* $(n - 1)$ (*word-rec z s* $(n - 1)$)

apply (*rule subst*[**where** *t=n* **and** *s=1 + (n − 1)*])
apply *simp*
apply (*subst word-rec-Suc*)
apply *simp*
apply *simp*
done

lemma *word-rec-in*: *f* (*word-rec z* (λ-. *f*) *n*) = *word-rec* (*f z*) (λ-. *f*) *n*
by (*induct n*) (*simp-all add*: *word-rec-0 word-rec-Suc*)

lemma *word-rec-in2*: *f n* (*word-rec z f n*) = *word-rec* (*f 0 z*) (*f* ∘ *op + 1*) *n*
by (*induct n*) (*simp-all add*: *word-rec-0 word-rec-Suc*)

lemma *word-rec-twice*:
m ≤ *n* ⟹ *word-rec z f n* = *word-rec* (*word-rec z f* (*n − m*)) (*f* ∘ *op +* (*n − m*)) *m*
apply (*erule rev-mp*)
apply (*rule-tac x=z* **in** *spec*)
apply (*rule-tac x=f* **in** *spec*)
apply (*induct n*)
apply (*simp add*: *word-rec-0*)
apply *clarsimp*
apply (*rule-tac t=1 + n − m* **and** *s=1 + (n − m)* **in** *subst*)
apply *simp*
apply (*case-tac 1 + (n − m) = 0*)
apply (*simp add*: *word-rec-0*)
apply (*rule-tac f = word-rec a b* **for** *a b* **in** *arg-cong*)
apply (*rule-tac t=m* **and** *s=m + (1 + (n − m))* **in** *subst*)
apply *simp*
apply (*simp* (*no-asm-use*))
apply (*simp add*: *word-rec-Suc word-rec-in2*)
apply (*erule impE*)
apply *uint-arith*
apply (*drule-tac x=x* ∘ *op + 1* **in** *spec*)
apply (*drule-tac x=x 0 xa* **in** *spec*)
apply *simp*
apply (*rule-tac t=*λ*a. x* (*1 + (n − m + a)*) **and** *s=*λ*a. x* (*1 + (n − m) + a*) **in** *subst*)
apply (*clarsimp simp add*: *fun-eq-iff*)
apply (*rule-tac t=*(*1 + (n − m + xb*)) **and** *s=1 + (n − m) + xb* **in** *subst*)
apply *simp*
apply (*rule refl*)
apply (*rule refl*)
done

lemma *word-rec-id*: *word-rec z* (λ-. *id*) *n* = *z*
by (*induct n*) (*auto simp add*: *word-rec-0 word-rec-Suc*)

lemma *word-rec-id-eq*: ∀ *m* < *n*. *f m* = *id* ⟹ *word-rec z f n* = *z*

apply (*erule rev-mp*)
apply (*induct n*)
apply (*auto simp add*: *word-rec-0 word-rec-Suc*)
apply (*drule spec*, *erule mp*)
apply *uint-arith*
apply (*drule-tac x=n* **in** *spec*, *erule impE*)
apply *uint-arith*
apply *simp*
done

lemma *word-rec-max*:
$\forall\, m{\geq}n.\ m \neq -1 \longrightarrow f\ m = id \Longrightarrow$ *word-rec z f* $(-1) =$ *word-rec z f n*
apply (*subst word-rec-twice*[**where** *n=−1* **and** *m=−1 − n*])
apply *simp*
apply *simp*
apply (*rule word-rec-id-eq*)
apply *clarsimp*
apply (*drule spec*, *rule mp*, *erule mp*)
apply (*rule word-plus-mono-right2*[*OF - order-less-imp-le*])
prefer *2*
apply *assumption*
apply *simp*
apply (*erule contrapos-pn*)
apply *simp*
apply (*drule arg-cong*[**where** *f=λx. x − n*])
apply *simp*
done

lemma *unatSuc*: $1 + n \neq 0 \Longrightarrow$ *unat* $(1 + n) =$ *Suc* (*unat n*)
for *n* :: *'a*::*len word*
by *unat-arith*

declare *bin-to-bl-def* [*simp*]

ML-file *Tools/word-lib.ML*
ML-file *Tools/smt-word.ML*

hide-const (**open**) *Word*

end

theory *WordBitwise*
imports *Word*
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 *word-eq-rbl-eq*: *x* = *y* ⟷ *rev* (*to-bl x*) = *rev* (*to-bl y*)
by *simp*

lemma *rbl-word-or*: *rev* (*to-bl* (*x OR y*)) = *map2 op* ∨ (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
by (*simp add*: *map2-def zip-rev bl-word-or rev-map*)

lemma *rbl-word-and*: *rev* (*to-bl* (*x AND y*)) = *map2 op* ∧ (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
by (*simp add*: *map2-def zip-rev bl-word-and rev-map*)

lemma *rbl-word-xor*: *rev* (*to-bl* (*x XOR y*)) = *map2 op* ≠ (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
by (*simp add*: *map2-def 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 *bl-word-sub*: *to-bl* (*x* − *y*) = *to-bl* (*x* + (− *y*))
by *simp*

lemma *rbl-word-1*: *rev* (*to-bl* (*1* :: *'a*::*len0 word*)) = *takefill False* (*len-of TYPE*(*'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 len-of TYPE*('*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*: *map2-def 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*=λ*a*. *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*)))
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*::*len0 word*)) =
takefill False (*len-of TYPE*(*'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*::*len0 word*)) =
takefill False (*len-of TYPE*(*'a*)) (*drop n* (*rev* (*to-bl w*)))
apply (*simp add*: *slice-take word-rev-tf rev-take*)
apply (*cases n* < *len-of TYPE*(*'b*), *simp-all*)
done

lemma *rbl-word-ucast*:
rev (*to-bl* (*ucast x* :: *'a*::*len0 word*)) = *takefill False* (*len-of TYPE*(*'a*)) (*rev* (*to-bl x*))
apply (*simp add*: *to-bl-ucast takefill-alt*)
apply (*simp add*: *rev-drop*)
apply (*cases len-of TYPE*(*'a*) < *len-of TYPE*(*'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 len-of TYPE('a)*, *simp-all*)
apply (*rule word-eqI*)
apply (*simp add*: *nth-sshiftr word-size test-bit-of-bl*
msb-nth)
done

lemma *nth-word-of-int*:
(word-of-int x :: 'a::len0 word) !! n = (n < len-of TYPE('a) ∧ bin-nth 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*:
(scast (x :: 'a::len word) :: 'b::len word) !! n =
(n < len-of TYPE('b) ∧
(if n < len-of TYPE('a) − 1 then x !! n
else x !! (len-of TYPE('a) − 1)))
by (*simp add*: *scast-def nth-sint*)

lemma *rbl-word-scast*:
rev (to-bl (scast x :: 'a::len word)) = takefill-last False (len-of TYPE('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 nth-rev to-bl-nth)
apply (*cases len-of TYPE*('*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* (λ*x sm*. *rbl-plus False* (*map* (*op* ∧ *x*) *ys*) (*False* # *sm*)) *xs* []

lemma *rbl-mul-simps*:
rbl-mul (*x* # *xs*) *ys* = *rbl-plus False* (*map* (*op* ∧ *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* (*op* ∧ *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*: *shiftl-t2n*)

have *zip-take-triv*: $\bigwedge$*xs ys n*. *n* = *length ys* $\Longrightarrow$ *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 zip-take-triv*)
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* $\Longrightarrow$
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*)
apply (*auto simp add*: *bl-to-bin-def Bit-B0 Bit-B1 add1-zle-eq Bit-def*)
done

lemma *word-le-rbl*: *x ≤ y ⟷ rev-bl-order True (rev (to-bl x)) (rev (to-bl y))*
for *x y* :: *'a*::*len0 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*::*len0 word*
by (*simp add*: *word-less-alt rev-bl-order-bl-to-bin*)

lemma *word-sint-msb-eq*: *sint x = uint x − (if msb x then 2 ^ size x else 0)*
apply (*cases msb x*)
apply (*rule word-sint.Abs-eqD*[**where** *'a='a*], *simp-all*)
apply (*simp add*: *word-size wi-hom-syms word-of-int-2p-len*)
apply (*simp add*: *sints-num word-size*)
apply (*rule conjI*)
apply (*simp add*: *le-diff-eq'*)
apply (*rule order-trans*[**where** *y=2 ^ (len-of TYPE('a) − 1)*])
apply (*simp add*: *power-Suc*[*symmetric*])
apply (*simp add*: *linorder-not-less*[*symmetric*] *mask-eq-iff*[*symmetric*])
apply (*rule notI*, *drule word-eqD*[**where** *x=size x − 1*])
apply (*simp add*: *msb-nth word-ops-nth-size word-size*)
apply (*simp add*: *order-less-le-trans*[**where** *y=0*])
apply (*rule word-uint.Abs-eqD*[**where** *'a='a*], *simp-all*)
apply (*simp add*: *linorder-not-less uints-num word-msb-sint*)
apply (*rule order-less-le-trans*[*OF sint-lt*])
apply *simp*
done

lemma *word-sle-msb-le*: *x <=s y ⟷ (msb y ⟶ msb x) ∧ ((msb x ∧ ¬ msb y) ∨ x ≤ y)*
apply (*simp add*: *word-sle-def 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-def word-sle-msb-le*)

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))
  apply simp-all
  apply (simp-all only: bin-to-bl-aux-alt)
  apply (simp-all)
  apply (simp-all add: bin-to-bl-zero-aux bin-to-bl-minus1-aux)
  done

lemma to-bl-upt: to-bl x = rev (map (op !! x) [0 ..< size x])
  apply (rule nth-equalityI)
   apply (simp add: word-size)
  apply (auto simp: to-bl-nth word-size nth-rev)
  done

lemma rev-to-bl-upt: rev (to-bl x) = map (op !! 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)
```

10.30 Tactic definition

ML ‹

```
structure Word-Bitwise-Tac =
struct

val word-ss = simpset-of @{theory-context Word};

fun mk-nat-clist ns = List.foldr
  (uncurry (Thm.mk-binop @{cterm Cons :: nat => -}))
  @{cterm [] :: nat list} ns;

fun upt-conv ctxt ct =
  case Thm.term-of ct of
    (@{const upt} $ 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 op = :: 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 =
  Simplifier.make-simproc @{context} expand-upt
   {lhss = [@{term upt x y}], proc = K upt-conv};

fun word-len-simproc-fn ctxt ct =
  case Thm.term-of ct of
    Const (@{const-name len-of}, -) $ 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 op = :: 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 =
  Simplifier.make-simproc @{context} word-len
   {lhss = [@{term len-of x}], proc = 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) = 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' = List.foldr (op $) (HOLogic.mk-number @{typ nat} j)
       (replicate i @{term Suc});
    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} nat-get-Suc
   {lhss = map (fn t => t $ @{term n :: nat}) ts,
    proc = K (nat-get-Suc-simproc-fn n-sucs)};

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 map2-Cons map2-Nil 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

end

theory *Bit-Comparison*
imports *Bits-Int*
begin

lemma *AND-lower* [*simp*]:
  fixes *x* :: *int* **and** *y* :: *int*
  assumes *0* ≤ *x*
  shows *0* ≤ *x AND y*
  using *assms*
proof (*induct x arbitrary*: *y rule*: *bin-induct*)
  case (*3 bin bit*)
  show *?case*
  proof (*cases y rule*: *bin-exhaust*)
    case (*1 bin′ bit′*)
    from *3* **have** *0* ≤ *bin*
      by (*cases bit*) (*simp-all add*: *Bit-def*)
    then have *0* ≤ *bin AND bin′* **by** (*rule 3*)
    with *1* **show** *?thesis*
      by *simp* (*simp add*: *Bit-def*)
  qed
next
  case *2*
  then show *?case* **by** (*simp only*: *Min-def*)
qed *simp*

lemma *OR-lower* [*simp*]:
  fixes *x* :: *int* **and** *y* :: *int*
  assumes *0* ≤ *x 0* ≤ *y*
  shows *0* ≤ *x OR y*
  using *assms*
proof (*induct x arbitrary*: *y rule*: *bin-induct*)
  case (*3 bin bit*)
  show *?case*
  proof (*cases y rule*: *bin-exhaust*)
    case (*1 bin′ bit′*)
    from *3* **have** *0* ≤ *bin*
      by (*cases bit*) (*simp-all add*: *Bit-def*)

```
    moreover from 1 3 have 0 ≤ bin'
      by (cases bit') (simp-all add: Bit-def)
    ultimately have 0 ≤ bin OR bin' by (rule 3)
    with 1 show ?thesis
      by simp (simp add: Bit-def)
  qed
qed simp-all

lemma XOR-lower [simp]:
  fixes x :: int and y :: int
  assumes 0 ≤ x 0 ≤ y
  shows 0 ≤ x XOR y
  using assms
proof (induct x arbitrary: y rule: bin-induct)
  case (3 bin bit)
  show ?case
  proof (cases y rule: bin-exhaust)
    case (1 bin' bit')
    from 3 have 0 ≤ bin
      by (cases bit) (simp-all add: Bit-def)
    moreover from 1 3 have 0 ≤ bin'
      by (cases bit') (simp-all add: Bit-def)
    ultimately have 0 ≤ bin XOR bin' by (rule 3)
    with 1 show ?thesis
      by simp (simp add: Bit-def)
  qed
next
  case 2
  then show ?case by (simp only: Min-def)
qed simp

lemma AND-upper1 [simp]:
  fixes x :: int and y :: int
  assumes 0 ≤ x
  shows x AND y ≤ x
  using assms
proof (induct x arbitrary: y rule: bin-induct)
  case (3 bin bit)
  show ?case
  proof (cases y rule: bin-exhaust)
    case (1 bin' bit')
    from 3 have 0 ≤ bin
      by (cases bit) (simp-all add: Bit-def)
    then have bin AND bin' ≤ bin by (rule 3)
    with 1 show ?thesis
      by simp (simp add: Bit-def)
  qed
next
  case 2
```

```
  then show ?case by (simp only: Min-def)
qed simp

lemmas AND-upper1' [simp] = order-trans [OF AND-upper1]
lemmas AND-upper1'' [simp] = order-le-less-trans [OF AND-upper1]

lemma AND-upper2 [simp]:
  fixes x :: int and y :: int
  assumes 0 ≤ y
  shows x AND y ≤ y
  using assms
proof (induct y arbitrary: x rule: bin-induct)
  case (3 bin bit)
  show ?case
  proof (cases x rule: bin-exhaust)
    case (1 bin' bit')
    from 3 have 0 ≤ bin
      by (cases bit) (simp-all add: Bit-def)
    then have bin' AND bin ≤ bin by (rule 3)
    with 1 show ?thesis
      by simp (simp add: Bit-def)
  qed
next
  case 2
  then show ?case by (simp only: Min-def)
qed simp

lemmas AND-upper2' [simp] = order-trans [OF AND-upper2]
lemmas AND-upper2'' [simp] = order-le-less-trans [OF AND-upper2]

lemma OR-upper:
  fixes x :: int and y :: int
  assumes 0 ≤ x x < 2 ^ n y < 2 ^ n
  shows x OR y < 2 ^ n
  using assms
proof (induct x arbitrary: y n rule: bin-induct)
  case (3 bin bit)
  show ?case
  proof (cases y rule: bin-exhaust)
    case (1 bin' bit')
    show ?thesis
    proof (cases n)
      case 0
      with 3 have bin BIT bit = 0 by simp
      then have bin = 0 and ¬ bit
        by (auto simp add: Bit-def split: if-splits) arith
      then show ?thesis using 0 1 ‹y < 2 ^ n›
        by simp
    next
```

```
      case (Suc m)
      from 3 have 0 ≤ bin
        by (cases bit) (simp-all add: Bit-def)
      moreover from 3 Suc have bin < 2 ^ m
        by (cases bit) (simp-all add: Bit-def)
      moreover from 1 3 Suc have bin' < 2 ^ m
        by (cases bit') (simp-all add: Bit-def)
      ultimately have bin OR bin' < 2 ^ m by (rule 3)
      with 1 Suc show ?thesis
        by simp (simp add: Bit-def)
    qed
  qed
qed simp-all

lemma XOR-upper:
  fixes x :: int and y :: int
  assumes 0 ≤ x x < 2 ^ n y < 2 ^ n
  shows x XOR y < 2 ^ n
  using assms
proof (induct x arbitrary: y n rule: bin-induct)
  case (3 bin bit)
  show ?case
  proof (cases y rule: bin-exhaust)
    case (1 bin' bit')
    show ?thesis
    proof (cases n)
      case 0
      with 3 have bin BIT bit = 0 by simp
      then have bin = 0 and ¬ bit
        by (auto simp add: Bit-def split: if-splits) arith
      then show ?thesis using 0 1 ‹y < 2 ^ n›
        by simp
    next
      case (Suc m)
      from 3 have 0 ≤ bin
        by (cases bit) (simp-all add: Bit-def)
      moreover from 3 Suc have bin < 2 ^ m
        by (cases bit) (simp-all add: Bit-def)
      moreover from 1 3 Suc have bin' < 2 ^ m
        by (cases bit') (simp-all add: Bit-def)
      ultimately have bin XOR bin' < 2 ^ m by (rule 3)
      with 1 Suc show ?thesis
        by simp (simp add: Bit-def)
    qed
  qed
next
  case 2
  then show ?case by (simp only: Min-def)
qed simp
```

end

References

[1] Jeremy Dawson. Isabelle theories for machine words. In Michael Goldsmith and Bill Roscoe, editors, *Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07)*, Electronic Notes in Theoretical Computer Science, page 15, Oxford, September 2007. Elsevier. to appear.