Machine Words in Isabelle/HOL

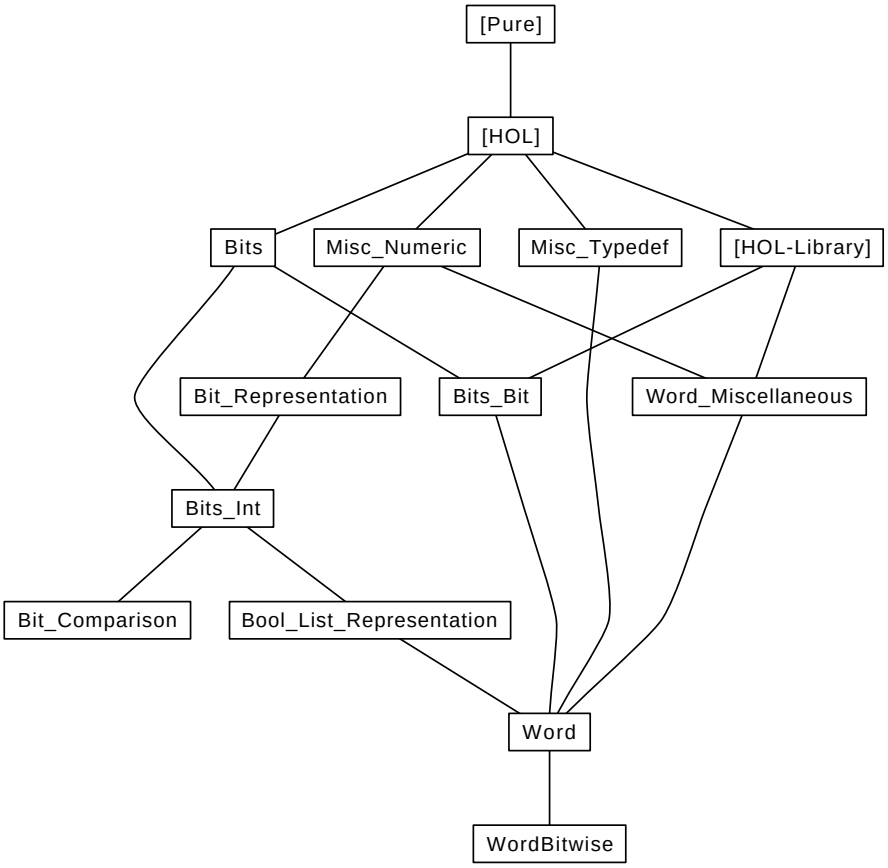
Jeremy Dawson, Paul Graunke, Brian Huffman, Gerwin Klein, and John Matthews

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

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 ⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bit-ops-same* [*simp*]:
x AND x = *x*
x OR x = *x*
x XOR x = *0*
for *x* :: *bit*
⟨*proof*⟩

lemma *bit-not-not* [*simp*]: *NOT* (*NOT x*) = *x*
for *x* :: *bit*
⟨*proof*⟩

lemma *bit-or-def*: *b OR c* = *NOT* (*NOT b AND NOT c*)
for *b c* :: *bit*
⟨*proof*⟩

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

for b c :: *bit*
⟨*proof*⟩

lemma *bit-NOT-eq-1-iff* [*simp*]: $NOT\ b = 1 \longleftrightarrow b = 0$
for b :: *bit*
⟨*proof*⟩

lemma *bit-AND-eq-1-iff* [*simp*]: $a\ AND\ b = 1 \longleftrightarrow a = 1 \wedge b = 1$
for a b :: *bit*
⟨*proof*⟩

end

3 Useful Numerical Lemmas

theory *Misc-Numeric*
imports *Main*
begin

lemma *one-mod-exp-eq-one* [*simp*]: $1\ mod\ (2 * 2\ \hat{}\ n) = (1::int)$
⟨*proof*⟩

lemma *mod-2-neq-1-eq-eq-0*: $k\ mod\ 2 \neq 1 \longleftrightarrow k\ mod\ 2 = 0$
for k :: *int*
⟨*proof*⟩

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

lemma *diff-le-eq'*: $a - b \leq c \longleftrightarrow a \leq b + c$
for a b c :: *int*
⟨*proof*⟩

lemma *emep1*: $even\ n \Longrightarrow even\ d \Longrightarrow 0 \leq d \Longrightarrow (n + 1)\ mod\ d = (n\ mod\ d) + 1$
for n d :: *int*
⟨*proof*⟩

lemma *int-mod-ge*: $a < n \Longrightarrow 0 < n \Longrightarrow a \leq a\ mod\ n$
for a n :: *int*
⟨*proof*⟩

lemma *int-mod-ge'*: $b < 0 \Longrightarrow 0 < n \Longrightarrow b + n \leq b\ mod\ n$
for b n :: *int*
⟨*proof*⟩

lemma *int-mod-le'*: $0 \leq b - n \Longrightarrow b\ mod\ n \leq b - n$
for b n :: *int*

⟨*proof*⟩

lemma *zless2*: $0 < (2 :: int)$
⟨*proof*⟩

lemma *zless2p*: $0 < (2 \hat{} n :: int)$
⟨*proof*⟩

lemma *zle2p*: $0 \leq (2 \hat{} n :: int)$
⟨*proof*⟩

lemma *m1mod2k*: $-1 \bmod 2 \hat{} n = (2 \hat{} n - 1 :: int)$
⟨*proof*⟩

lemma *p1mod22k'*: $(1 + 2 * b) \bmod (2 * 2 \hat{} n) = 1 + 2 * (b \bmod 2 \hat{} n)$
for $b :: int$
⟨*proof*⟩

lemma *p1mod22k*: $(2 * b + 1) \bmod (2 * 2 \hat{} n) = 2 * (b \bmod 2 \hat{} n) + 1$
for $b :: int$
⟨*proof*⟩

lemma *int-mod-lem*: $0 < n \Longrightarrow 0 \leq b \wedge b < n \longleftrightarrow b \bmod n = b$
for $b\ n :: int$
⟨*proof*⟩

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*: $k\ BIT\ False = k + k$
⟨*proof*⟩

lemma *Bit-B1*: $k\ BIT\ True = k + k + 1$
⟨*proof*⟩

lemma *Bit-B0-2t*: $k\ BIT\ False = 2 * k$
⟨*proof*⟩

lemma *Bit-B1-2t*: $k\ BIT\ True = 2 * k + 1$

⟨*proof*⟩

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

lemma *bin-last-odd*: *bin-last = odd*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bin-rest-BIT* [*simp*]: *bin-rest* (*x BIT b*) = *x*
⟨*proof*⟩

lemma *bin-last-BIT* [*simp*]: *bin-last* (*x BIT b*) = *b*
⟨*proof*⟩

lemma *BIT-eq-iff* [*iff*]: *u BIT b = v BIT c* ⟷ *u = v* ∧ *b = c*
⟨*proof*⟩

lemma *BIT-bin-simps* [*simp*]:
numeral k BIT False = numeral (*Num.Bit0 k*)
numeral k BIT True = numeral (*Num.Bit1 k*)
(− *numeral k*) *BIT False* = − *numeral* (*Num.Bit0 k*)
(− *numeral k*) *BIT True* = − *numeral* (*Num.BitM k*)
⟨*proof*⟩

lemma *BIT-special-simps* [*simp*]:
shows *0 BIT False = 0*
and *0 BIT True = 1*
and *1 BIT False = 2*
and *1 BIT True = 3*
and (− *1*) *BIT False* = − *2*
and (− *1*) *BIT True* = − *1*
⟨*proof*⟩

lemma *Bit-eq-0-iff*: *w BIT b = 0* ⟷ *w = 0* ∧ ¬ *b*
⟨*proof*⟩

lemma *Bit-eq-m1-iff*: *w BIT b* = −*1* ⟷ *w* = −*1* ∧ *b*
⟨*proof*⟩

lemma *BitM-inc*: *Num.BitM* (*Num.inc w*) = *Num.Bit1 w*
⟨*proof*⟩

lemma *expand-BIT*:

numeral (Num.Bit0 w) = numeral w BIT False
numeral (Num.Bit1 w) = numeral w BIT True
− numeral (Num.Bit0 w) = (− numeral w) BIT False
− numeral (Num.Bit1 w) = (− numeral (w + Num.One)) BIT True
⟨*proof*⟩

lemma *bin-last-numeral-simps* [*simp*]:
¬ bin-last 0
bin-last 1
bin-last (− 1)
bin-last Numeral1
¬ bin-last (numeral (Num.Bit0 w))
bin-last (numeral (Num.Bit1 w))
¬ bin-last (− numeral (Num.Bit0 w))
bin-last (− numeral (Num.Bit1 w))
⟨*proof*⟩

lemma *bin-rest-numeral-simps* [*simp*]:
bin-rest 0 = 0
bin-rest 1 = 0
bin-rest (− 1) = − 1
bin-rest Numeral1 = 0
bin-rest (numeral (Num.Bit0 w)) = numeral w
bin-rest (numeral (Num.Bit1 w)) = numeral w
bin-rest (− numeral (Num.Bit0 w)) = − numeral w
bin-rest (− numeral (Num.Bit1 w)) = − numeral (w + Num.One)
⟨*proof*⟩

lemma *less-Bits*: *v BIT b < w BIT c ⟷ v < w ∨ v ≤ w ∧ ¬ b ∧ c*
⟨*proof*⟩

lemma *le-Bits*: *v BIT b ≤ w BIT c ⟷ v < w ∨ v ≤ w ∧ (¬ b ∨ c)*
⟨*proof*⟩

lemma *pred-BIT-simps* [*simp*]:
x BIT False − 1 = (x − 1) BIT True
x BIT True − 1 = x BIT False
⟨*proof*⟩

lemma *succ-BIT-simps* [*simp*]:
x BIT False + 1 = x BIT True
x BIT True + 1 = (x + 1) BIT False
⟨*proof*⟩

lemma *add-BIT-simps* [*simp*]:
x BIT False + y BIT False = (x + y) BIT False
x BIT False + y BIT True = (x + y) BIT True
x BIT True + y BIT False = (x + y) BIT True
x BIT True + y BIT True = (x + y + 1) BIT False

⟨*proof*⟩

lemma *mult-BIT-simps* [*simp*]:
x BIT False ∗ *y* = (*x* ∗ *y*) *BIT False*
x ∗ *y BIT False* = (*x* ∗ *y*) *BIT False*
x BIT True ∗ *y* = (*x* ∗ *y*) *BIT False* + *y*
⟨*proof*⟩

lemma *B-mod-2'*: *X* = *2* ⟹ (*w BIT True*) *mod X* = *1* ∧ (*w BIT False*) *mod X* = *0*
⟨*proof*⟩

lemma *bin-ex-rl*: ∃ *w b*. *w BIT b* = *bin*
⟨*proof*⟩

lemma *bin-exhaust*:
assumes *that*: ⋀*x b*. *bin* = *x BIT b* ⟹ *Q*
shows *Q*
⟨*proof*⟩

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

lemma *bin-abs-lem*: *bin* = (*w BIT b*) ⟹ *bin* ≠ −*1* ⟶ *bin* ≠ *0* ⟶ *nat* |*w*| < *nat* |*bin*|
⟨*proof*⟩

lemma *bin-induct*:
assumes *PPls*: *P 0*
and *PMin*: *P* (− *1*)
and *PBit*: ⋀*bin bit*. *P bin* ⟹ *P* (*bin BIT bit*)
shows *P bin*
⟨*proof*⟩

lemma *Bit-div2* [*simp*]: (*w BIT b*) *div 2* = *w*
⟨*proof*⟩

lemma *bin-nth-eq-iff*: *bin-nth x* = *bin-nth y* ⟷ *x* = *y*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bin-nth-zero* [*simp*]: ¬ *bin-nth 0 n*
⟨*proof*⟩

lemma *bin-nth-1* [*simp*]: *bin-nth 1 n* $\longleftrightarrow$ *n = 0*
⟨*proof*⟩

lemma *bin-nth-minus1* [*simp*]: *bin-nth (− 1) n*
⟨*proof*⟩

lemma *bin-nth-0-BIT*: *bin-nth (w BIT b) 0* $\longleftrightarrow$ *b*
⟨*proof*⟩

lemma *bin-nth-Suc-BIT*: *bin-nth (w BIT b) (Suc n) = bin-nth w n*
⟨*proof*⟩

lemma *bin-nth-minus* [*simp*]: *0 < n* $\Longrightarrow$ *bin-nth (w BIT b) n = bin-nth w (n − 1)*
⟨*proof*⟩

lemma *bin-nth-numeral*: *bin-rest x = y* $\Longrightarrow$ *bin-nth x (numeral n) = bin-nth y (pred-numeral n)*
⟨*proof*⟩

lemmas *bin-nth-numeral-simps* [*simp*] =
bin-nth-numeral [*OF bin-rest-numeral-simps*(*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* $\Rightarrow$ *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
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma *bintrunc-mod2p*: *bintrunc n w* = *w mod 2* ^ *n*
⟨*proof*⟩

lemma *sbintrunc-mod2p*: *sbintrunc n w* = (*w* + *2* ^ *n*) *mod 2* ^ (*Suc n*) − *2* ^ *n*
⟨*proof*⟩

4.3 Simplifications for (s)bintrunc

lemma *bintrunc-n-0* [*simp*]: *bintrunc n 0* = *0*
⟨*proof*⟩

lemma *sbintrunc-n-0* [*simp*]: *sbintrunc n 0* = *0*
⟨*proof*⟩

lemma *sbintrunc-n-minus1* [*simp*]: *sbintrunc n* (− *1*) = −*1*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *sbintrunc-Suc-numeral*:
sbintrunc (Suc n) 1 = 1
sbintrunc (Suc n) (numeral (Num.Bit0 w)) = 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
⟨*proof*⟩

lemma *bin-sign-lem*: *(bin-sign (sbintrunc n bin) = −1) = bin-nth bin n*
⟨*proof*⟩

lemma *nth-bintr*: *bin-nth (bintrunc m w) n ⟷ n < m ∧ bin-nth w n*
⟨*proof*⟩

lemma *nth-sbintr*: *bin-nth (sbintrunc m w) n = (if n < m then bin-nth w n else bin-nth w m)*
⟨*proof*⟩

lemma *bin-nth-Bit*: *bin-nth (w BIT b) n ⟷ n = 0 ∧ b ∨ (∃ m. n = Suc m ∧ bin-nth w m)*
⟨*proof*⟩

lemma *bin-nth-Bit0*:
bin-nth (numeral (Num.Bit0 w)) n ⟷
(∃ m. n = Suc m ∧ bin-nth (numeral w) m)
⟨*proof*⟩

lemma *bin-nth-Bit1*:
bin-nth (numeral (Num.Bit1 w)) n ⟷
n = 0 ∨ (∃ m. n = Suc m ∧ bin-nth (numeral w) m)
⟨*proof*⟩

lemma *bintrunc-bintrunc-l*: *n ≤ m ⟹ bintrunc m (bintrunc n w) = bintrunc n w*
⟨*proof*⟩

lemma *sbintrunc-sbintrunc-l*: *n ≤ m ⟹ sbintrunc m (sbintrunc n w) = sbintrunc n w*
⟨*proof*⟩

lemma *bintrunc-bintrunc-ge*: *n ≤ m ⟹ bintrunc n (bintrunc m w) = bintrunc n w*
⟨*proof*⟩

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

lemma *sbintrunc-sbintrunc-min* [*simp*]: *sbintrunc m* (*sbintrunc n w*) = *sbintrunc* (*min m n*) *w*
⟨*proof*⟩

lemmas *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*: $0 < n \Longrightarrow bintrunc\ (Suc\ (n - 1))\ w = bintrunc\ n\ w$
⟨*proof*⟩

lemma *sbintrunc-minus*: $0 < n \Longrightarrow sbintrunc\ (Suc\ (n - 1))\ w = sbintrunc\ n\ w$
⟨*proof*⟩

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** $f = \lambda w.\ w\ BIT\ b$] **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*: $bintrunc\ (Suc\ n)\ x = y \Longrightarrow m = Suc\ n \Longrightarrow bintrunc\ m\ x = y$
⟨*proof*⟩

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*: $sbintrunc\ (Suc\ n)\ x = y \Longrightarrow m = Suc\ n \Longrightarrow sbintrunc\ m\ x = y$
⟨*proof*⟩

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

lemma *sbintrunc-bintrunc-lt*: $m > n \Longrightarrow sbintrunc\ n\ (bintrunc\ m\ w) = sbintrunc\ n\ w$
⟨*proof*⟩

lemma *bintrunc-sbintrunc-le*: $m \leq Suc\ n \Longrightarrow bintrunc\ m\ (sbintrunc\ n\ w) = bintrunc\ m\ w$
⟨*proof*⟩

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

lemma *bintrunc-sbintrunc'* [*simp*]: $0 < n \Longrightarrow bintrunc\ n\ (sbintrunc\ (n - 1)\ w) = bintrunc\ n\ w$
⟨*proof*⟩

lemma *sbintrunc-bintrunc'* [*simp*]: $0 < n \Longrightarrow sbintrunc\ (n - 1)\ (bintrunc\ n\ w) = sbintrunc\ (n - 1)\ w$
⟨*proof*⟩

lemma *bin-sbin-eq-iff*: $bintrunc\ (Suc\ n)\ x = bintrunc\ (Suc\ n)\ y \longleftrightarrow sbintrunc\ n\ x = sbintrunc\ n\ y$
⟨*proof*⟩

lemma *bin-sbin-eq-iff'*:
$0 < n \Longrightarrow bintrunc\ n\ x = bintrunc\ n\ y \longleftrightarrow sbintrunc\ (n - 1)\ x = sbintrunc\ (n - 1)\ y$
⟨*proof*⟩

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

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

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

lemma *bintrunc-numeral*:
$bintrunc\ (numeral\ k)\ x = bintrunc\ (pred\text{-}numeral\ k)\ (bin\text{-}rest\ x)\ BIT\ bin\text{-}last\ x$
⟨*proof*⟩

lemma *sbintrunc-numeral*:
$sbintrunc\ (numeral\ k)\ x = sbintrunc\ (pred\text{-}numeral\ k)\ (bin\text{-}rest\ x)\ BIT\ bin\text{-}last\ x$
⟨*proof*⟩

lemma *bintrunc-numeral-simps* [*simp*]:
$bintrunc\ (numeral\ k)\ (numeral\ (Num.Bit0\ w)) = bintrunc\ (pred\text{-}numeral\ k)\ (numeral\ w)\ BIT\ False$

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *no-bintr-alt1*: *bintrunc n = (λw. w mod 2 ^ n :: int)*
⟨*proof*⟩

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

lemma *no-sbintr-alt2*: *sbintrunc n = (λw. (w + 2 ^ n) mod 2 ^ Suc n − 2 ^ n :: int)*
⟨*proof*⟩

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

lemma *sb-inc-lem*: *a + 2^k < 0 ⟹ a + 2^k + 2^(Suc k) ≤ (a + 2^k) mod 2^(Suc k)*
for *a :: int*
⟨*proof*⟩

lemma *sb-inc-lem'*: *a < − (2^k) ⟹ a + 2^k + 2^(Suc k) ≤ (a + 2^k) mod 2^(Suc k)*
for *a :: int*
⟨*proof*⟩

lemma *sbintrunc-inc*: *x < − (2^n) ⟹ x + 2^(Suc n) ≤ sbintrunc n x*
⟨*proof*⟩

lemma *sb-dec-lem*: *0 ≤ − (2 ^ k) + a ⟹ (a + 2 ^ k) mod (2 * 2 ^ k) ≤ − (2 ^ k) + a*

for a :: *int*
⟨*proof*⟩

lemma *sb-dec-lem'*: $2 \hat{} k \leq a \Longrightarrow (a + 2 \hat{} k) \bmod (2 * 2 \hat{} k) \leq -(2 \hat{} k) + a$
for a :: *int*
⟨*proof*⟩

lemma *sbintrunc-dec*: $x \geq (2 \hat{} n) \Longrightarrow x - 2 \hat{} (Suc\ n) >= sbintrunc\ n\ x$
⟨*proof*⟩

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

lemma *bintr-ge0*: $0 \leq bintrunc\ n\ w$
⟨*proof*⟩

lemma *bintr-lt2p*: $bintrunc\ n\ w < 2 \hat{} n$
⟨*proof*⟩

lemma *bintr-Min*: $bintrunc\ n\ (-1) = 2 \hat{} n - 1$
⟨*proof*⟩

lemma *sbintr-ge*: $-(2 \hat{} n) \leq sbintrunc\ n\ w$
⟨*proof*⟩

lemma *sbintr-lt*: $sbintrunc\ n\ w < 2 \hat{} n$
⟨*proof*⟩

lemma *sign-Pls-ge-0*: $bin\text{-}sign\ bin = 0 \longleftrightarrow bin \geq 0$
for *bin* :: *int*
⟨*proof*⟩

lemma *sign-Min-lt-0*: $bin\text{-}sign\ bin = -1 \longleftrightarrow bin < 0$
for *bin* :: *int*
⟨*proof*⟩

lemma *bin-rest-trunc*: $bin\text{-}rest\ (bintrunc\ n\ bin) = bintrunc\ (n - 1)\ (bin\text{-}rest\ bin)$
⟨*proof*⟩

lemma *bin-rest-power-trunc*:
$(bin\text{-}rest\ \hat{}\hat{}\ k)\ (bintrunc\ n\ bin) = bintrunc\ (n - k)\ ((bin\text{-}rest\ \hat{}\hat{}\ k)\ bin)$
⟨*proof*⟩

lemma *bin-rest-trunc-i*: $bintrunc\ n\ (bin\text{-}rest\ bin) = bin\text{-}rest\ (bintrunc\ (Suc\ n)\ bin)$
⟨*proof*⟩

lemma *bin-rest-strunc*: $bin\text{-}rest\ (sbintrunc\ (Suc\ n)\ bin) = sbintrunc\ n\ (bin\text{-}rest\ bin)$
⟨*proof*⟩

lemma *bintrunc-rest* [*simp*]: *bintrunc n* (*bin-rest* (*bintrunc n bin*)) = *bin-rest* (*bintrunc n bin*)
⟨*proof*⟩

lemma *sbintrunc-rest* [*simp*]: *sbintrunc n* (*bin-rest* (*sbintrunc n bin*)) = *bin-rest* (*sbintrunc n bin*)
⟨*proof*⟩

lemma *bintrunc-rest'*: *bintrunc n* ∘ *bin-rest* ∘ *bintrunc n* = *bin-rest* ∘ *bintrunc n*
⟨*proof*⟩

lemma *sbintrunc-rest'*: *sbintrunc n* ∘ *bin-rest* ∘ *sbintrunc n* = *bin-rest* ∘ *sbintrunc n*
⟨*proof*⟩

lemma *rco-lem*: *f* ∘ *g* ∘ *f* = *g* ∘ *f* ⟹ *f* ∘ (*g* ∘ *f*) ^^ *n* = *g* ^^ *n* ∘ *f*
⟨*proof*⟩

lemmas *rco-bintr* = *bintrunc-rest'*
[*THEN rco-lem* [*THEN fun-cong*], *unfolded o-def*]
lemmas *rco-sbintr* = *sbintrunc-rest'*
[*THEN rco-lem* [*THEN fun-cong*], *unfolded o-def*]

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*)
⟨*proof*⟩

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*: $bitNOT = (\lambda x{::}int.\ -\ x\ -\ 1)$

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*))
⟨*proof*⟩

termination
⟨*proof*⟩

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

definition *int-or-def*:
$bitOR = (\lambda x\ y{::}int.\ NOT\ (NOT\ x\ AND\ NOT\ y))$

definition *int-xor-def*:
$bitXOR = (\lambda x\ y{::}int.\ (x\ AND\ NOT\ y)\ OR\ (NOT\ x\ AND\ y))$

instance ⟨*proof*⟩

end

5.1.1 Basic simplification rules

lemma *int-not-BIT* [*simp*]:
$NOT\ (w\ BIT\ b) = (NOT\ w)\ BIT\ (\neg\ b)$
⟨*proof*⟩

lemma *int-not-simps* [*simp*]:
$NOT\ (0{::}int) = -1$
$NOT\ (1{::}int) = -2$
$NOT\ (-\ 1{::}int) = 0$
$NOT\ (numeral\ w{::}int) = -\ numeral\ (w + Num.One)$
$NOT\ (-\ numeral\ (Num.Bit0\ w){::}int) = numeral\ (Num.BitM\ w)$
$NOT\ (-\ numeral\ (Num.Bit1\ w){::}int) = numeral\ (Num.Bit0\ w)$
⟨*proof*⟩

lemma *int-not-not* [*simp*]: $NOT\ (NOT\ (x{::}int)) = x$
⟨*proof*⟩

lemma *int-and-0* [*simp*]: $(0::int)\ AND\ x = 0$
⟨*proof*⟩

lemma *int-and-m1* [*simp*]: $(-1::int)\ AND\ x = x$
⟨*proof*⟩

lemma *int-and-Bits* [*simp*]:
$(x\ BIT\ b)\ AND\ (y\ BIT\ c) = (x\ AND\ y)\ BIT\ (b \wedge c)$
⟨*proof*⟩

lemma *int-or-zero* [*simp*]: $(0::int)\ OR\ x = x$
⟨*proof*⟩

lemma *int-or-minus1* [*simp*]: $(-1::int)\ OR\ x = -1$
⟨*proof*⟩

lemma *int-or-Bits* [*simp*]:
$(x\ BIT\ b)\ OR\ (y\ BIT\ c) = (x\ OR\ y)\ BIT\ (b \vee c)$
⟨*proof*⟩

lemma *int-xor-zero* [*simp*]: $(0::int)\ XOR\ x = x$
⟨*proof*⟩

lemma *int-xor-Bits* [*simp*]:
$(x\ BIT\ b)\ XOR\ (y\ BIT\ c) = (x\ XOR\ y)\ BIT\ ((b \vee c) \wedge \neg\ (b \wedge c))$
⟨*proof*⟩

5.1.2 Binary destructors

lemma *bin-rest-NOT* [*simp*]: $bin\text{-}rest\ (NOT\ x) = NOT\ (bin\text{-}rest\ x)$
⟨*proof*⟩

lemma *bin-last-NOT* [*simp*]: $bin\text{-}last\ (NOT\ x) \longleftrightarrow \neg\ bin\text{-}last\ x$
⟨*proof*⟩

lemma *bin-rest-AND* [*simp*]: $bin\text{-}rest\ (x\ AND\ y) = bin\text{-}rest\ x\ AND\ bin\text{-}rest\ y$
⟨*proof*⟩

lemma *bin-last-AND* [*simp*]: $bin\text{-}last\ (x\ AND\ y) \longleftrightarrow bin\text{-}last\ x \wedge bin\text{-}last\ y$
⟨*proof*⟩

lemma *bin-rest-OR* [*simp*]: $bin\text{-}rest\ (x\ OR\ y) = bin\text{-}rest\ x\ OR\ bin\text{-}rest\ y$
⟨*proof*⟩

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

lemma *bin-rest-XOR* [*simp*]: $bin\text{-}rest\ (x\ XOR\ y) = bin\text{-}rest\ x\ XOR\ bin\text{-}rest\ y$

⟨*proof*⟩

lemma *bin-last-XOR* [*simp*]: *bin-last* (*x XOR y*) ⟷ (*bin-last x* ∨ *bin-last y*) ∧ ¬ (*bin-last x* ∧ *bin-last y*)
⟨*proof*⟩

lemma *bin-nth-ops*:
!!*x y*. *bin-nth* (*x AND y*) *n* = (*bin-nth x n* & *bin-nth y n*)
!!*x y*. *bin-nth* (*x OR y*) *n* = (*bin-nth x n* | *bin-nth y n*)
!!*x y*. *bin-nth* (*x XOR y*) *n* = (*bin-nth x n* ~= *bin-nth y n*)
!!*x*. *bin-nth* (*NOT x*) *n* = (~ *bin-nth x n*)
⟨*proof*⟩

5.1.3 Derived properties

lemma *int-xor-minus1* [*simp*]: (−1::*int*) *XOR x* = *NOT x*
⟨*proof*⟩

lemma *int-xor-extra-simps* [*simp*]:
w XOR (0::*int*) = *w*
w XOR (−1::*int*) = *NOT w*
⟨*proof*⟩

lemma *int-or-extra-simps* [*simp*]:
w OR (0::*int*) = *w*
w OR (−1::*int*) = −1
⟨*proof*⟩

lemma *int-and-extra-simps* [*simp*]:
w AND (0::*int*) = 0
w AND (−1::*int*) = *w*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bin-ops-same* [*simp*]:
(*x*::*int*) *AND x* = *x*
(*x*::*int*) *OR x* = *x*
(*x*::*int*) *XOR x* = 0
⟨*proof*⟩

lemmas *bin-log-esimps* =
int-and-extra-simps int-or-extra-simps int-xor-extra-simps

int-and-0 int-and-m1 int-or-zero int-or-minus1 int-xor-zero int-xor-minus1

lemma *bbw-ao-absorb*:
!!y::*int*. x *AND* (y *OR* x) = x & x *OR* (y *AND* x) = x
⟨*proof*⟩

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*)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *int-and-assoc*:
(x *AND* y) *AND* (z::*int*) = x *AND* (y *AND* z)
⟨*proof*⟩

lemma *int-or-assoc*:
(x *OR* y) *OR* (z::*int*) = x *OR* (y *OR* z)
⟨*proof*⟩

lemma *int-xor-assoc*:
(x *XOR* y) *XOR* (z::*int*) = x *XOR* (y *XOR* z)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *bbw-oa-dist*:
!!y z::*int*. (x *AND* y) *OR* z =

$(x\ OR\ z)\ AND\ (y\ OR\ z)$
⟨*proof*⟩

lemma *bbw-ao-dist*:
!!$y\ z$::*int*. $(x\ OR\ y)\ AND\ z =$
$(x\ AND\ z)\ OR\ (y\ AND\ z)$
⟨*proof*⟩

5.1.4 Simplification with numerals

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

lemma *bin-rl-eqI*: $[\![bin\text{-}rest\ x = bin\text{-}rest\ y;\ bin\text{-}last\ x = bin\text{-}last\ y]\!] \Longrightarrow x = y$
⟨*proof*⟩

lemma *bin-rest-neg-numeral-BitM* [*simp*]:
bin-rest $(-\ numeral\ (Num.BitM\ w)) = -\ numeral\ w$
⟨*proof*⟩

lemma *bin-last-neg-numeral-BitM* [*simp*]:
bin-last $(-\ numeral\ (Num.BitM\ w))$
⟨*proof*⟩

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
⟨*proof*⟩

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*)
⟨*proof*⟩

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))
⟨*proof*⟩

5.1.5 Interactions with arithmetic

lemma *plus-and-or* [*rule-format*]:
ALL y::int. (x AND y) + (x OR y) = x + y
⟨*proof*⟩

lemma *le-int-or*:
bin-sign (y::int) = 0 ==> x <= x OR y
⟨*proof*⟩

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)*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *bin-trunc-xor*:
!!x y. bintrunc n (bintrunc n x XOR bintrunc n y) =
bintrunc n (x XOR y)
⟨*proof*⟩

lemma *bin-trunc-not*:
!!x. bintrunc n (NOT (bintrunc n x)) = bintrunc n (NOT x)
⟨*proof*⟩

lemma *bintr-bintr-i*:
x = bintrunc n y ==> bintrunc n x = bintrunc n y
⟨*proof*⟩

lemmas *bin-trunc-and = bin-trunc-ao(1) [THEN bintr-bintr-i]*
lemmas *bin-trunc-or = bin-trunc-ao(2) [THEN bintr-bintr-i]*

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*
⟨*proof*⟩

lemma *bin-sc-sc-same* [*simp*]:
bin-sc n c (*bin-sc n b w*) = *bin-sc n c w*
⟨*proof*⟩

lemma *bin-sc-sc-diff*:
m ~= *n* ==>
bin-sc m c (*bin-sc n b w*) = *bin-sc n b* (*bin-sc m c w*)
⟨*proof*⟩

lemma *bin-nth-sc-gen*:
bin-nth (*bin-sc n b w*) *m* = (*if m* = *n then b else bin-nth w m*)
⟨*proof*⟩

lemma *bin-sc-nth* [*simp*]:
(*bin-sc n* (*bin-nth w n*) *w*) = *w*
⟨*proof*⟩

lemma *bin-sign-sc* [*simp*]:
bin-sign (*bin-sc n b w*) = *bin-sign w*
⟨*proof*⟩

lemma *bin-sc-bintr* [*simp*]:
bintrunc m (*bin-sc n x* (*bintrunc m* (*w*))) = *bintrunc m* (*bin-sc n x w*)
⟨*proof*⟩

lemma *bin-clr-le*:
bin-sc n False w <= *w*
⟨*proof*⟩

lemma *bin-set-ge*:
bin-sc n True w >= *w*
⟨*proof*⟩

lemma *bintr-bin-clr-le*:
bintrunc n (*bin-sc m False w*) <= *bintrunc n w*
⟨*proof*⟩

lemma *bintr-bin-set-ge*:

bintrunc n (bin-sc m True w) >= bintrunc n w
⟨*proof*⟩

lemma *bin-sc-FP* [*simp*]: *bin-sc n False 0 = 0*
⟨*proof*⟩

lemma *bin-sc-TM* [*simp*]: *bin-sc n True (− 1) = − 1*
⟨*proof*⟩

lemmas *bin-sc-simps = bin-sc.Z bin-sc.Suc bin-sc-TM bin-sc-FP*

lemma *bin-sc-minus*:
0 < n ==> bin-sc (Suc (n − 1)) b w = bin-sc n b w
⟨*proof*⟩

lemmas *bin-sc-Suc-minus =*
trans [*OF bin-sc-minus* [*symmetric*] *bin-sc.Suc*]

lemma *bin-sc-numeral* [*simp*]:
bin-sc (numeral k) b w =
bin-sc (pred-numeral k) b (bin-rest w) BIT bin-last w
⟨*proof*⟩

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
⟨*proof*⟩

lemma *bin-cat-Suc-Bit*:
bin-cat w (Suc n) (v BIT b) = bin-cat w n v BIT b
⟨*proof*⟩

lemma *bin-nth-cat*:
bin-nth (bin-cat x k y) n =
(if n < k then bin-nth y n else bin-nth x (n − k))
⟨*proof*⟩

lemma *bin-nth-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))
⟨*proof*⟩

lemma *bin-cat-assoc*:
bin-cat (bin-cat x m y) n z = bin-cat x (m + n) (bin-cat y n z)
⟨*proof*⟩

lemma *bin-cat-assoc-sym*:
bin-cat x m (bin-cat y n z) = bin-cat (bin-cat x (m − n) y) (min m n) z
⟨*proof*⟩

lemma *bin-cat-zero* [*simp*]: *bin-cat 0 n w = bintrunc n w*
⟨*proof*⟩

lemma *bintr-cat1*:
bintrunc (k + n) (bin-cat a n b) = bin-cat (bintrunc k a) n b
⟨*proof*⟩

lemma *bintr-cat*: *bintrunc m (bin-cat a n b) =*
bin-cat (bintrunc (m − n) a) n (bintrunc (min m n) b)
⟨*proof*⟩

lemma *bintr-cat-same* [*simp*]:
bintrunc n (bin-cat a n b) = bintrunc n b
⟨*proof*⟩

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

⟨*proof*⟩

lemma *split-bintrunc*:
bin-split n c = (a, b) ==> b = bintrunc n c
⟨*proof*⟩

lemma *bin-cat-split*:
bin-split n w = (u, v) ==> w = bin-cat u n v
⟨*proof*⟩

lemma *bin-split-cat*:
bin-split n (bin-cat v n w) = (v, bintrunc n w)
⟨*proof*⟩

lemma *bin-split-zero* [*simp*]: *bin-split n 0 = (0, 0)*
⟨*proof*⟩

lemma *bin-split-minus1* [*simp*]:
bin-split n (- 1) = (- 1, bintrunc n (- 1))
⟨*proof*⟩

lemma *bin-split-trunc*:
bin-split (min m n) c = (a, b) ==>
bin-split n (bintrunc m c) = (bintrunc (m - n) a, b)
⟨*proof*⟩

lemma *bin-split-trunc1*:
bin-split n c = (a, b) ==>
bin-split n (bintrunc m c) = (bintrunc (m - n) a, bintrunc m b)
⟨*proof*⟩

lemma *bin-cat-num*:
*bin-cat a n b = a * 2 ^ n + bintrunc n b*
⟨*proof*⟩

lemma *bin-split-num*:
bin-split n b = (b div 2 ^ n, b mod 2 ^ n)
⟨*proof*⟩

5.4 Miscellaneous lemmas

lemma *nth-2p-bin*:
bin-nth (2 ^ n) m = (m = n)
⟨*proof*⟩

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

m))
⟨*proof*⟩

lemma *power-BIT*: $2 \hat{} (Suc\ n) - 1 = (2 \hat{} n - 1)\ BIT\ True$
⟨*proof*⟩

lemma *mod-BIT*:
$bin\ BIT\ bit\ mod\ 2 \hat{} Suc\ n = (bin\ mod\ 2 \hat{} n)\ BIT\ bit$
⟨*proof*⟩

lemma *AND-mod*:
fixes $x :: int$
shows $x\ AND\ 2 \hat{} n - 1 = x\ mod\ 2 \hat{} n$
⟨*proof*⟩

end

6 Bool lists and integers

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

definition *map2* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a\ list \Rightarrow 'b\ list \Rightarrow 'c\ list$
where $map2\ f\ as\ bs = map\ (case\text{-}prod\ f)\ (zip\ as\ bs)$

lemma *map2-Nil* [*simp*, *code*]: $map2\ f\ []\ ys = []$
⟨*proof*⟩

lemma *map2-Nil2* [*simp*, *code*]: $map2\ f\ xs\ [] = []$
⟨*proof*⟩

lemma *map2-Cons* [*simp*, *code*]: $map2\ f\ (x \# xs)\ (y \# ys) = f\ x\ y \# map2\ f\ xs\ ys$
⟨*proof*⟩

6.1 Operations on lists of booleans

primrec *bl-to-bin-aux* :: $bool\ list \Rightarrow int \Rightarrow int$
where
Nil: $bl\text{-}to\text{-}bin\text{-}aux\ []\ w = w$
| *Cons*: $bl\text{-}to\text{-}bin\text{-}aux\ (b \# bs)\ w = bl\text{-}to\text{-}bin\text{-}aux\ bs\ (w\ BIT\ b)$

definition *bl-to-bin* :: $bool\ list \Rightarrow int$
where $bl\text{-}to\text{-}bin\ bs = bl\text{-}to\text{-}bin\text{-}aux\ bs\ 0$

primrec *bin-to-bl-aux* :: $nat \Rightarrow int \Rightarrow bool\ list \Rightarrow bool\ list$
where
Z: $bin\text{-}to\text{-}bl\text{-}aux\ 0\ w\ bl = bl$

| *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*
⟨*proof*⟩

lemma *bin-to-bl-aux-zero-minus-simp* [*simp*]:
0 < n ⟹ bin-to-bl-aux n 0 bl = bin-to-bl-aux (n − 1) 0 (False # bl)
⟨*proof*⟩

lemma *bin-to-bl-aux-minus1-minus-simp* [*simp*]:
0 < n ⟹ bin-to-bl-aux n (− 1) bl = bin-to-bl-aux (n − 1) (− 1) (True # bl)
⟨*proof*⟩

lemma *bin-to-bl-aux-one-minus-simp* [*simp*]:
0 < n ⟹ bin-to-bl-aux n 1 bl = bin-to-bl-aux (n − 1) 0 (True # bl)
⟨*proof*⟩

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

lemma *bin-to-bl-aux-Bit0-minus-simp* [*simp*]:
0 < n ⟹
bin-to-bl-aux n (numeral (Num.Bit0 w)) bl = bin-to-bl-aux (n − 1) (numeral w) (False # bl)
⟨*proof*⟩

lemma *bin-to-bl-aux-Bit1-minus-simp* [*simp*]:
0 < n ⟹
bin-to-bl-aux n (numeral (Num.Bit1 w)) bl = bin-to-bl-aux (n − 1) (numeral w) (True # bl)
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *bin-to-bl-aux-append*: *bin-to-bl-aux n w bs @ cs = bin-to-bl-aux n w (bs @ cs)*
⟨*proof*⟩

lemma *bl-to-bin-append*: *bl-to-bin (bs @ cs) = bl-to-bin-aux cs (bl-to-bin bs)*
⟨*proof*⟩

lemma *bin-to-bl-aux-alt*: *bin-to-bl-aux n w bs = bin-to-bl n w @ bs*
⟨*proof*⟩

lemma *bin-to-bl-0* [*simp*]: *bin-to-bl 0 bs = []*
⟨*proof*⟩

lemma *size-bin-to-bl-aux*: *size (bin-to-bl-aux n w bs) = n + length bs*
⟨*proof*⟩

lemma *size-bin-to-bl* [*simp*]: *size* (*bin-to-bl n w*) = *n*
⟨*proof*⟩

lemma *bin-bl-bin'*: *bl-to-bin* (*bin-to-bl-aux n w bs*) = *bl-to-bin-aux bs* (*bintrunc n w*)
⟨*proof*⟩

lemma *bin-bl-bin* [*simp*]: *bl-to-bin* (*bin-to-bl n w*) = *bintrunc n w*
⟨*proof*⟩

lemma *bl-bin-bl'*: *bin-to-bl* (*n* + *length bs*) (*bl-to-bin-aux bs w*) = *bin-to-bl-aux n w bs*
⟨*proof*⟩

lemma *bl-bin-bl* [*simp*]: *bin-to-bl* (*length bs*) (*bl-to-bin bs*) = *bs*
⟨*proof*⟩

lemma *bl-to-bin-inj*: *bl-to-bin bs* = *bl-to-bin cs* ⟹ *length bs* = *length cs* ⟹ *bs* = *cs*
⟨*proof*⟩

lemma *bl-to-bin-False* [*simp*]: *bl-to-bin* (*False* # *bl*) = *bl-to-bin bl*
⟨*proof*⟩

lemma *bl-to-bin-Nil* [*simp*]: *bl-to-bin* [] = *0*
⟨*proof*⟩

lemma *bin-to-bl-zero-aux*: *bin-to-bl-aux n 0 bl* = *replicate n False* @ *bl*
⟨*proof*⟩

lemma *bin-to-bl-zero*: *bin-to-bl n 0* = *replicate n False*
⟨*proof*⟩

lemma *bin-to-bl-minus1-aux*: *bin-to-bl-aux n* (− *1*) *bl* = *replicate n True* @ *bl*
⟨*proof*⟩

lemma *bin-to-bl-minus1*: *bin-to-bl n* (− *1*) = *replicate n True*
⟨*proof*⟩

lemma *bl-to-bin-rep-F*: *bl-to-bin* (*replicate n False* @ *bl*) = *bl-to-bin bl*
⟨*proof*⟩

lemma *bin-to-bl-trunc* [*simp*]: *n* ≤ *m* ⟹ *bin-to-bl n* (*bintrunc m w*) = *bin-to-bl n w*
⟨*proof*⟩

lemma *bin-to-bl-aux-bintr*:
bin-to-bl-aux n (*bintrunc m bin*) *bl* =

replicate $(n - m)$ *False* @ *bin-to-bl-aux* $(min\ n\ m)$ *bin bl*
⟨*proof*⟩

lemma *bin-to-bl-bintr*:
bin-to-bl n (*bintrunc m bin*) = *replicate* $(n - m)$ *False* @ *bin-to-bl* $(min\ n\ m)$ *bin*
⟨*proof*⟩

lemma *bl-to-bin-rep-False*: *bl-to-bin* (*replicate n False*) = *0*
⟨*proof*⟩

lemma *len-bin-to-bl-aux*: *length* (*bin-to-bl-aux n w bs*) = *n* + *length bs*
⟨*proof*⟩

lemma *len-bin-to-bl*: *length* (*bin-to-bl n w*) = *n*
⟨*proof*⟩

lemma *sign-bl-bin'*: *bin-sign* (*bl-to-bin-aux bs w*) = *bin-sign w*
⟨*proof*⟩

lemma *sign-bl-bin*: *bin-sign* (*bl-to-bin bs*) = *0*
⟨*proof*⟩

lemma *bl-sbin-sign-aux*: *hd* (*bin-to-bl-aux* (*Suc n*) *w bs*) = (*bin-sign* (*sbintrunc n w*) = *−1*)
⟨*proof*⟩

lemma *bl-sbin-sign*: *hd* (*bin-to-bl* (*Suc n*) *w*) = (*bin-sign* (*sbintrunc n w*) = *−1*)
⟨*proof*⟩

lemma *bin-nth-of-bl-aux*:
bin-nth (*bl-to-bin-aux bl w*) *n* =
(*n* < *size bl* ∧ *rev bl* ! *n* | *n* ≥ *length bl* ∧ *bin-nth w* (*n* − *size bl*))
⟨*proof*⟩

lemma *bin-nth-of-bl*: *bin-nth* (*bl-to-bin bl*) *n* = (*n* < *length bl* ∧ *rev bl* ! *n*)
⟨*proof*⟩

lemma *bin-nth-bl*: *n* < *m* ⟹ *bin-nth w n* = *nth* (*rev* (*bin-to-bl m w*)) *n*
⟨*proof*⟩

lemma *nth-rev*: *n* < *length xs* ⟹ *rev xs* ! *n* = *xs* ! (*length xs* − *1* − *n*)
⟨*proof*⟩

lemma *nth-rev-alt*: *n* < *length ys* ⟹ *ys* ! *n* = *rev ys* ! (*length ys* − *Suc n*)
⟨*proof*⟩

lemma *nth-bin-to-bl-aux*:
n < *m* + *length bl* ⟹ (*bin-to-bl-aux m w bl*) ! *n* =

(*if* $n < m$ *then bin-nth* w $(m - 1 - n)$ *else* bl ! $(n - m)$)
⟨*proof*⟩

lemma *nth-bin-to-bl*: $n < m \Longrightarrow$ (*bin-to-bl* m w) ! n = *bin-nth* w $(m -$ *Suc* $n)$
⟨*proof*⟩

lemma *bl-to-bin-lt2p-aux*: *bl-to-bin-aux bs* $w <$ $(w + 1) * (2$ ^ *length bs*)
⟨*proof*⟩

lemma *bl-to-bin-lt2p-drop*: *bl-to-bin bs* $<$ 2 ^ *length* (*dropWhile Not bs*)
⟨*proof*⟩

lemma *bl-to-bin-lt2p*: *bl-to-bin bs* $<$ 2 ^ *length bs*
⟨*proof*⟩

lemma *bl-to-bin-ge2p-aux*: *bl-to-bin-aux bs* $w \geq w * (2$ ^ *length bs*)
⟨*proof*⟩

lemma *bl-to-bin-ge0*: *bl-to-bin bs* ≥ 0
⟨*proof*⟩

lemma *butlast-rest-bin*: *butlast* (*bin-to-bl* n w) = *bin-to-bl* $(n - 1)$ (*bin-rest* w)
⟨*proof*⟩

lemma *butlast-bin-rest*: *butlast bl* = *bin-to-bl* (*length bl* − *Suc 0*) (*bin-rest* (*bl-to-bin bl*))
⟨*proof*⟩

lemma *butlast-rest-bl2bin-aux*:
bl $\neq$ [] $\Longrightarrow$ *bl-to-bin-aux* (*butlast bl*) w = *bin-rest* (*bl-to-bin-aux bl w*)
⟨*proof*⟩

lemma *butlast-rest-bl2bin*: *bl-to-bin* (*butlast bl*) = *bin-rest* (*bl-to-bin bl*)
⟨*proof*⟩

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

lemma *trunc-bl2bin*: *bintrunc* m (*bl-to-bin bl*) = *bl-to-bin* (*drop* (*length bl* − m) *bl*)
⟨*proof*⟩

lemma *trunc-bl2bin-len* [*simp*]: *bintrunc* (*length bl*) (*bl-to-bin bl*) = *bl-to-bin bl*
⟨*proof*⟩

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

lemma *nth-rest-power-bin*: *bin-nth* ((*bin-rest* ^^ *k*) *w*) *n* = *bin-nth w* (*n* + *k*)
⟨*proof*⟩

lemma *take-rest-power-bin*: *m* ≤ *n* ⟹ *take m* (*bin-to-bl n w*) = *bin-to-bl m* ((*bin-rest* ^^ (*n* − *m*)) *w*)
⟨*proof*⟩

lemma *hd-butlast*: *size xs* > *1* ⟹ *hd* (*butlast xs*) = *hd xs*
⟨*proof*⟩

lemma *last-bin-last'*: *size xs* > *0* ⟹ *last xs* ⟷ *bin-last* (*bl-to-bin-aux xs w*)
⟨*proof*⟩

lemma *last-bin-last*: *size xs* > *0* ⟹ *last xs* ⟷ *bin-last* (*bl-to-bin xs*)
⟨*proof*⟩

lemma *bin-last-last*: *bin-last w* ⟷ *last* (*bin-to-bl* (*Suc n*) *w*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bl-not-aux-bin*: *map Not* (*bin-to-bl-aux n w cs*) = *bin-to-bl-aux n* (*NOT w*) (*map Not cs*)
⟨*proof*⟩

lemma *bl-not-bin*: *map Not* (*bin-to-bl n w*) = *bin-to-bl n* (*NOT w*)
⟨*proof*⟩

lemma *bl-and-bin*: *map2* (*op* ∧) (*bin-to-bl n v*) (*bin-to-bl n w*) = *bin-to-bl n* (*v AND w*)
⟨*proof*⟩

lemma *bl-or-bin*: *map2* (*op* ∨) (*bin-to-bl n v*) (*bin-to-bl n w*) = *bin-to-bl n* (*v OR w*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *drop-bin2bl-aux*:
drop m (*bin-to-bl-aux n bin bs*) =
bin-to-bl-aux (*n* − *m*) *bin* (*drop* (*m* − *n*) *bs*)
⟨*proof*⟩

lemma *drop-bin2bl*: *drop m* (*bin-to-bl n bin*) = *bin-to-bl* (*n* − *m*) *bin*
⟨*proof*⟩

lemma *take-bin2bl-lem1*: *take m* (*bin-to-bl-aux m w bs*) = *bin-to-bl m w*
⟨*proof*⟩

lemma *take-bin2bl-lem*: *take m* (*bin-to-bl-aux* (*m* + *n*) *w bs*) = *take m* (*bin-to-bl* (*m* + *n*) *w*)
⟨*proof*⟩

lemma *bin-split-take*: *bin-split n c* = (*a*, *b*) ⟹ *bin-to-bl m a* = *take m* (*bin-to-bl* (*m* + *n*) *c*)
⟨*proof*⟩

lemma *bin-split-take1*:
k = *m* + *n* ⟹ *bin-split n c* = (*a*, *b*) ⟹ *bin-to-bl m a* = *take m* (*bin-to-bl k c*)
⟨*proof*⟩

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

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

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

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

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

lemma *take-takefill'*: ⋀*w n*. *n* = *k* + *m* ⟹ *take k* (*takefill fill n w*) = *takefill fill k w*

⟨*proof*⟩

lemma *drop-takefill*: ⋀*w*. *drop k* (*takefill fill* (*m* + *k*) *w*) = *takefill fill m* (*drop k w*)
⟨*proof*⟩

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

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

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

lemma *takefill-same'*: *l* = *length xs* ⟹ *takefill fill l xs* = *xs*
⟨*proof*⟩

lemmas *takefill-same* [*simp*] = *takefill-same'* [*OF refl*]

lemma *takefill-bintrunc*: *takefill False n bl* = *rev* (*bin-to-bl n* (*bl-to-bin* (*rev bl*)))
⟨*proof*⟩

lemma *bl-bin-bl-rtf*: *bin-to-bl n* (*bl-to-bin bl*) = *rev* (*takefill False n* (*rev bl*))
⟨*proof*⟩

lemma *bl-bin-bl-rep-drop*:
bin-to-bl n (*bl-to-bin bl*) =
replicate (*n* − *length bl*) *False* @ *drop* (*length bl* − *n*) *bl*
⟨*proof*⟩

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

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

lemmas *takefill-Suc-cases* =
list.cases [*THEN takefill.Suc* [*THEN trans*]]

lemmas *takefill-Suc-Nil* = *takefill-Suc-cases* (*1*)
lemmas *takefill-Suc-Cons* = *takefill-Suc-cases* (*2*)

lemmas *takefill-minus-simps* = *takefill-Suc-cases* [*THEN* [*2*]
takefill-minus [*symmetric*, *THEN trans*]]

lemma *takefill-numeral-Nil* [*simp*]:
takefill fill (*numeral k*) [] = *fill* # *takefill fill* (*pred-numeral k*) []
⟨*proof*⟩

lemma *takefill-numeral-Cons* [*simp*]:
takefill fill (*numeral k*) (*x* # *xs*) = *x* # *takefill fill* (*pred-numeral k*) *xs*
⟨*proof*⟩

6.4 Links with function *bl-to-bin*

lemma *bl-to-bin-aux-cat*:
⋀*nv v*. *bl-to-bin-aux bs* (*bin-cat w nv v*) =
bin-cat w (*nv* + *length bs*) (*bl-to-bin-aux bs v*)
⟨*proof*⟩

lemma *bin-to-bl-aux-cat*:
⋀*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*)
⟨*proof*⟩

lemma *bl-to-bin-aux-alt*: *bl-to-bin-aux bs w* = *bin-cat w* (*length bs*) (*bl-to-bin bs*)
⟨*proof*⟩

lemma *bin-to-bl-cat*:
bin-to-bl (*nv* + *nw*) (*bin-cat v nw w*) =
bin-to-bl-aux nv v (*bin-to-bl nw w*)
⟨*proof*⟩

lemmas *bl-to-bin-aux-app-cat* =
trans [*OF bl-to-bin-aux-append bl-to-bin-aux-alt*]

lemmas *bin-to-bl-aux-cat-app* =
trans [*OF bin-to-bl-aux-cat bin-to-bl-aux-alt*]

lemma *bl-to-bin-app-cat*:
bl-to-bin (*bsa* @ *bs*) = *bin-cat* (*bl-to-bin bsa*) (*length bs*) (*bl-to-bin bs*)
⟨*proof*⟩

lemma *bin-to-bl-cat-app*:
bin-to-bl (*n* + *nw*) (*bin-cat w nw wa*) = *bin-to-bl n w* @ *bin-to-bl nw wa*
⟨*proof*⟩

bl-to-bin-app-cat-alt and *bl-to-bin-app-cat* are easily interderivable.

lemma *bl-to-bin-app-cat-alt*: *bin-cat* (*bl-to-bin cs*) *n w* = *bl-to-bin* (*cs* @ *bin-to-bl n w*)
⟨*proof*⟩

lemma *mask-lem*: (*bl-to-bin* (*True* # *replicate n False*)) = *bl-to-bin* (*replicate n True*) + *1*

⟨*proof*⟩

6.5 Function *bl-of-nth*

lemma *length-bl-of-nth* [*simp*]: *length* (*bl-of-nth n f*) = *n*
⟨*proof*⟩

lemma *nth-bl-of-nth* [*simp*]: *m* < *n* ⟹ *rev* (*bl-of-nth n f*) ! *m* = *f m*
⟨*proof*⟩

lemma *bl-of-nth-inj*: (⋀*k*. *k* < *n* ⟹ *f k* = *g k*) ⟹ *bl-of-nth n f* = *bl-of-nth n g*
⟨*proof*⟩

lemma *bl-of-nth-nth-le*: *n* ≤ *length xs* ⟹ *bl-of-nth n* (*nth* (*rev xs*)) = *drop* (*length xs* − *n*) *xs*
⟨*proof*⟩

lemma *bl-of-nth-nth* [*simp*]: *bl-of-nth* (*length xs*) (*op* ! (*rev xs*)) = *xs*
⟨*proof*⟩

lemma *size-rbl-pred*: *length* (*rbl-pred bl*) = *length bl*
⟨*proof*⟩

lemma *size-rbl-succ*: *length* (*rbl-succ bl*) = *length bl*
⟨*proof*⟩

lemma *size-rbl-add*: *length* (*rbl-add bl cl*) = *length bl*
⟨*proof*⟩

lemma *size-rbl-mult*: *length* (*rbl-mult bl cl*) = *length bl*
⟨*proof*⟩

lemmas *rbl-sizes* [*simp*] =
size-rbl-pred size-rbl-succ size-rbl-add size-rbl-mult

lemmas *rbl-Nils* =
rbl-pred.Nil rbl-succ.Nil rbl-add.Nil rbl-mult.Nil

lemma *rbl-pred*: *rbl-pred* (*rev* (*bin-to-bl n bin*)) = *rev* (*bin-to-bl n* (*bin* − *1*))
⟨*proof*⟩

lemma *rbl-succ*: *rbl-succ* (*rev* (*bin-to-bl n bin*)) = *rev* (*bin-to-bl n* (*bin* + *1*))
⟨*proof*⟩

lemma *rbl-add*:
⋀*bina binb*. *rbl-add* (*rev* (*bin-to-bl n bina*)) (*rev* (*bin-to-bl n binb*)) =
rev (*bin-to-bl n* (*bina* + *binb*))
⟨*proof*⟩

lemma *rbl-add-app2*: *length blb* $\geq$ *length bla* $\Longrightarrow$ *rbl-add bla* (*blb* @ *blc*) = *rbl-add bla blb*
⟨*proof*⟩

lemma *rbl-add-take2*:
length blb >= *length bla* ==> *rbl-add bla* (*take* (*length bla*) *blb*) = *rbl-add bla blb*
⟨*proof*⟩

lemma *rbl-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*))
⟨*proof*⟩

lemma *rbl-mult-app2*: *length blb* $\geq$ *length bla* $\Longrightarrow$ *rbl-mult bla* (*blb* @ *blc*) = *rbl-mult bla blb*
⟨*proof*⟩

lemma *rbl-mult-take2*:
length blb $\geq$ *length bla* $\Longrightarrow$ *rbl-mult bla* (*take* (*length bla*) *blb*) = *rbl-mult bla blb*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemmas *rbl-mult-Suc* = *lessI* [*THEN rbl-mult-gt*]

lemma *rbbl-Cons*: *b* # *rev* (*bin-to-bl n x*) = *rev* (*bin-to-bl* (*Suc n*) (*x BIT b*))
⟨*proof*⟩

lemma *rbl-mult*:
rbl-mult (*rev* (*bin-to-bl n bina*)) (*rev* (*bin-to-bl n binb*)) =
rev (*bin-to-bl n* (*bina* * *binb*))
⟨*proof*⟩

lemma *rbl-add-split*:
P (*rbl-add* (*y* # *ys*) (*x* # *xs*)) =
(∀ *ws*. *length ws* = *length ys* ⟶ *ws* = *rbl-add ys xs* ⟶
(*y* ⟶ ((*x* ⟶ *P* (*False* # *rbl-succ ws*)) ∧ (¬ *x* ⟶ *P* (*True* # *ws*)))) ∧
(¬ *y* ⟶ *P* (*x* # *ws*)))

⟨*proof*⟩

lemma *rbl-mult-split*:
P (*rbl-mult* (*y* # *ys*) *xs*) =
(∀ *ws*. *length ws* = *Suc* (*length ys*) ⟶ *ws* = *False* # *rbl-mult ys xs* ⟶
(*y* ⟶ *P* (*rbl-add ws xs*)) ∧ (¬ *y* ⟶ *P ws*))
⟨*proof*⟩

6.6 Repeated splitting or concatenation

lemma *sclem*: *size* (*concat* (*map* (*bin-to-bl n*) *xs*)) = *length xs* ∗ *n*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bin-rcat-bl*: *bin-rcat n wl* = *bl-to-bin* (*concat* (*map* (*bin-to-bl n*) *wl*))
⟨*proof*⟩

lemmas *bin-rsplit-aux-simps* = *bin-rsplit-aux.simps bin-rsplitl-aux.simps*
lemmas *rsplit-aux-simps* = *bin-rsplit-aux-simps*

lemmas *th-if-simp1* = *if-split* [**where** *P* = *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*
⟨*proof*⟩

lemma *bin-rsplitl-aux-append*: *bin-rsplitl-aux n m c* (*bs* @ *cs*) = *bin-rsplitl-aux n m c bs* @ *cs*
⟨*proof*⟩

lemmas *rsplit-aux-apps* [**where** *bs* = []] =
bin-rsplit-aux-append bin-rsplitl-aux-append

lemmas *rsplit-def-auxs* = *bin-rsplit-def bin-rsplitl-def*

lemmas *rsplit-aux-alts* = *rsplit-aux-apps*
[*unfolded append-Nil rsplit-def-auxs* [*symmetric*]]

lemma *bin-split-minus*: $0 < n \Longrightarrow$ *bin-split* (*Suc* ($n - 1$)) w = *bin-split* n w
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *bin-rsplit-aux-simp-alt*:
bin-rsplit-aux n m c bs =
(*if m* = *0* ∨ *n* = *0 then bs*
else let (*a*, *b*) = *bin-split n c in bin-rsplit n* (*m* − *n*, *a*) @ *b* # *bs*)
⟨*proof*⟩

lemmas *bin-rsplit-simp-alt* =
trans [*OF bin-rsplit-def bin-rsplit-aux-simp-alt*]

lemmas *bthrs* = *bin-rsplit-simp-alt* [*THEN* [*2*] *trans*]

lemma *bin-rsplit-size-sign'* [*rule-format*]:
$n > 0 \Longrightarrow$ *rev sw* = *bin-rsplit n* (*nw*, *w*) ⟹ ∀ *v*∈*set sw*. *bintrunc n v* = *v*
⟨*proof*⟩

lemmas *bin-rsplit-size-sign* = *bin-rsplit-size-sign'* [*OF asm-rl*
rev-rev-ident [*THEN trans*] *set-rev* [*THEN equalityD2* [*THEN subsetD*]]]

lemma *bin-nth-rsplit* [*rule-format*] :
$n > 0 \Longrightarrow m < n \Longrightarrow$
∀ *w k nw*.
rev sw = *bin-rsplit n* (*nw*, *w*) ⟶
$k <$ *size sw* ⟶ *bin-nth* (*sw* ! *k*) m = *bin-nth w* ($k * n + m$)
⟨*proof*⟩

lemma *bin-rsplit-all*: $0 < nw \Longrightarrow nw \leq n \Longrightarrow$ *bin-rsplit n* (*nw*, *w*) = [*bintrunc n w*]
⟨*proof*⟩

lemma *bin-rsplit-l* [*rule-format*]:

$\forall$ *bin. bin-rsplitl n* (*m*, *bin*) = *bin-rsplit n* (*m*, *bintrunc m bin*)
⟨*proof*⟩

lemma *bin-rsplit-rcat* [*rule-format*]:
n > *0* ⟶ *bin-rsplit n* (*n* ∗ *size ws*, *bin-rcat n ws*) = *map* (*bintrunc n*) *ws*
⟨*proof*⟩

lemma *bin-rsplit-aux-len-le* [*rule-format*] :
$\forall$ *ws m. n* ≠ *0* ⟶ *ws* = *bin-rsplit-aux n nw w bs* ⟶
length ws ≤ *m* ⟷ *nw* + *length bs* ∗ *n* ≤ *m* ∗ *n*
⟨*proof*⟩

lemma *bin-rsplit-len-le*: *n* ≠ *0* ⟶ *ws* = *bin-rsplit n* (*nw*, *w*) ⟶ *length ws* ≤ *m* ⟷ *nw* ≤ *m* ∗ *n*
⟨*proof*⟩

lemma *bin-rsplit-aux-len*:
n ≠ *0* ⟹ *length* (*bin-rsplit-aux n nw w cs*) = (*nw* + *n* − *1*) *div n* + *length cs*
⟨*proof*⟩

lemma *bin-rsplit-len*: *n* ≠ *0* ⟹ *length* (*bin-rsplit n* (*nw*, *w*)) = (*nw* + *n* − *1*) *div n*
⟨*proof*⟩

lemma *bin-rsplit-aux-len-indep*:
n ≠ *0* ⟹ *length bs* = *length cs* ⟹
length (*bin-rsplit-aux n nw v bs*) =
length (*bin-rsplit-aux n nw w cs*)
⟨*proof*⟩

lemma *bin-rsplit-len-indep*:
n ≠ *0* ⟹ *length* (*bin-rsplit n* (*nw*, *v*)) = *length* (*bin-rsplit n* (*nw*, *w*))
⟨*proof*⟩

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 $\exists\, n.\ \forall\, n' \geq n.\ f\ n'$ *then*
let $n = LEAST\ n.\ \forall\, n' \geq n.\ f\ n'$
in sbintrunc n (*bl-to-bin* (*True* # *rev* (*map f* [*0*..<*n*])))
else 0 :: *int*)

definition *shiftl x n* = $x * 2\ \hat{}\ n$ **for** x :: *int*

definition *shiftr x n* = *x div* $2\ \hat{}\ n$ **for** x :: *int*

definition *msb x* $\longleftrightarrow$ $x < 0$ **for** x :: *int*

instance ⟨*proof*⟩

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) $= y$
⟨*proof*⟩

lemma *td-nat-int*:
type-definition int nat (*Collect* (*op* <= *0*))
⟨*proof*⟩

context *type-definition*
begin

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

lemma *Abs-eqD*: *Abs x* = *Abs y* ==> $x \in A$ ==> $y \in A$ ==> $x = y$
⟨*proof*⟩

lemma *Abs-inverse'*:
$r : A$ ==> *Abs r* = a ==> *Rep* $a = r$
⟨*proof*⟩

lemma *Rep-comp-inverse*:

Rep ∘ *f* = *g* ==> *Abs* ∘ *g* = *f*
⟨*proof*⟩

lemma *Rep-eqD* [*elim*!]: *Rep x* = *Rep y* ==> *x* = *y*
⟨*proof*⟩

lemma *Rep-inverse'*: *Rep a* = *r* ==> *Abs r* = *a*
⟨*proof*⟩

lemma *comp-Abs-inverse*:
f ∘ *Abs* = *g* ==> *g* ∘ *Rep* = *f*
⟨*proof*⟩

lemma *set-Rep*:
A = *range Rep*
⟨*proof*⟩

lemma *set-Rep-Abs*: *A* = *range* (*Rep* ∘ *Abs*)
⟨*proof*⟩

lemma *Abs-inj-on*: *inj-on Abs A*
⟨*proof*⟩

lemma *image*: *Abs* ' *A* = *UNIV*
⟨*proof*⟩

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

lemma *fns1*:
Rep ∘ *fa* = *fr* ∘ *Rep* | *fa* ∘ *Abs* = *Abs* ∘ *fr* ==> *Abs* ∘ *fr* ∘ *Rep* = *fa*
⟨*proof*⟩

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

lemma *fns4*:
Rep ∘ *fa* ∘ *Abs* = *fr* ==>
Rep ∘ *fa* = *fr* ∘ *Rep* & *fa* ∘ *Abs* = *Abs* ∘ *fr*
⟨*proof*⟩

end

interpretation *nat-int*: *type-definition int nat Collect* (*op* <= *0*)
⟨*proof*⟩

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

nat-int.Abs-induct [*induct del*]

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*)
⟨*proof*⟩

lemma *fn-comm-power*:
fa ∘ *tr* = *tr* ∘ *fr* ==> *fa* ^^ *n* ∘ *tr* = *tr* ∘ *fr* ^^ *n*
⟨*proof*⟩

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

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

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

lemma *td-th*:
g ∘ *Abs* = *f* ==> *f* (*Rep x*) = *g x*
⟨*proof*⟩

lemma *eq-norm'*: *Rep* ∘ *Abs* = *norm*
⟨*proof*⟩

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

lemmas *td* = *td-thm*

lemma *set-iff-norm*: *w* : *A* ⟷ *w* = *norm w*
⟨*proof*⟩

lemma *inverse-norm*:
(*Abs n* = *w*) = (*Rep w* = *norm n*)
⟨*proof*⟩

lemma *norm-eq-iff*:
(*norm x* = *norm y*) = (*Abs x* = *Abs y*)
⟨*proof*⟩

lemma *norm-comps*:
Abs ∘ *norm* = *Abs*
norm ∘ *Rep* = *Rep*
norm ∘ *norm* = *norm*
⟨*proof*⟩

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

lemma *fns5*:
Rep ∘ *fa* ∘ *Abs* = *fr* ==>
fr ∘ *norm* = *fr* & *norm* ∘ *fr* = *fr*
⟨*proof*⟩

lemma *fns2*:
Abs ∘ *fr* ∘ *Rep* = *fa* ==>
(*norm* ∘ *fr* ∘ *norm* = *fr* ∘ *norm*) = (*Rep* ∘ *fa* = *fr* ∘ *Rep*)
⟨*proof*⟩

lemma *fns3*:
Abs ∘ *fr* ∘ *Rep* = *fa* ==>
(*norm* ∘ *fr* ∘ *norm* = *norm* ∘ *fr*) = (*fa* ∘ *Abs* = *Abs* ∘ *fr*)
⟨*proof*⟩

lemma *fns*:
fr ∘ *norm* = *norm* ∘ *fr* ==>
(*fa* ∘ *Abs* = *Abs* ∘ *fr*) = (*Rep* ∘ *fa* = *fr* ∘ *Rep*)
⟨*proof*⟩

lemma *range-norm*:
range (*Rep* ∘ *Abs*) = *A*
⟨*proof*⟩

end

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

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*)
⟨*proof*⟩

lemma *funpow-minus-simp*: $0 < n \Longrightarrow f \,\hat{}\hat{}\, n = f \circ f \,\hat{}\hat{}\, (n - 1)$
⟨*proof*⟩

lemma *power-numeral*: $a \,\hat{}\, numeral\ k = a * a \,\hat{}\, (pred\text{-}numeral\ k)$
⟨*proof*⟩

lemma *funpow-numeral* [*simp*]: $f \,\hat{}\hat{}\, numeral\ k = f \circ f \,\hat{}\hat{}\, (pred\text{-}numeral\ k)$
⟨*proof*⟩

lemma *replicate-numeral* [*simp*]: $replicate\ (numeral\ k)\ x = x \# replicate\ (pred\text{-}numeral\ k)\ x$
⟨*proof*⟩

lemma *rco-alt*: $(f \circ g) \,\hat{}\hat{}\, n \circ f = f \circ (g \circ f) \,\hat{}\hat{}\, n$
⟨*proof*⟩

lemma *list-exhaust-size-gt0*:
assumes *y*: $\bigwedge a\ list.\ y = a \# list \Longrightarrow P$
shows $0 < length\ y \Longrightarrow P$
⟨*proof*⟩

lemma *list-exhaust-size-eq0*:
assumes *y*: $y = [] \Longrightarrow P$
shows $length\ y = 0 \Longrightarrow P$
⟨*proof*⟩

lemma *size-Cons-lem-eq*: $y = xa \# list \Longrightarrow size\ y = Suc\ k \Longrightarrow size\ list = k$
⟨*proof*⟩

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

lemma *not-B1-is-B0*: $y \neq 1 \Longrightarrow y = 0$
for $y :: bit$
⟨*proof*⟩

lemma *B1-ass-B0*:
fixes $y :: bit$
assumes *y*: $y = 0 \Longrightarrow y = 1$
shows $y = 1$
⟨*proof*⟩
lemmas *n2s-ths* [*THEN eq-reflection*] = *add-2-eq-Suc add-2-eq-Suc'*

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

lemma *and-len*: $xs = ys \Longrightarrow xs = ys \wedge length\ xs = length\ ys$
⟨*proof*⟩

lemma *size-if*: $size\ (if\ p\ then\ xs\ else\ ys) = (if\ p\ then\ size\ xs\ else\ size\ ys)$
⟨*proof*⟩

lemma *tl-if*: *tl* (*if p then xs else ys*) = (*if p then tl xs else tl ys*)
⟨*proof*⟩

lemma *hd-if*: *hd* (*if p then xs else ys*) = (*if p then hd xs else hd ys*)
⟨*proof*⟩

lemma *if-Not-x*: (*if p then* ¬ *x else x*) = (*p* = (¬ *x*))
⟨*proof*⟩

lemma *if-x-Not*: (*if p then x else* ¬ *x*) = (*p* = *x*)
⟨*proof*⟩

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

lemma *if-same-eq*: (*If p x y* = (*If p u v*)) = (*if p then x* = *u else y* = *v*)
⟨*proof*⟩

lemma *if-same-eq-not*: (*If p x y* = (¬ *If p u v*)) = (*if p then x* = (¬ *u*) *else y* = (¬ *v*))
⟨*proof*⟩

lemma *if-Cons*: (*if p then x # xs else y # ys*) = *If p x y # If p xs ys*
⟨*proof*⟩

lemma *if-single*: (*if xc then* [*xab*] *else* [*an*]) = [*if xc then xab else an*]
⟨*proof*⟩

lemma *if-bool-simps*:
If p True y = (*p* ∨ *y*) ∧ *If p False y* = (¬ *p* ∧ *y*) ∧
If p y True = (*p* ⟶ *y*) ∧ *If p y False* = (*p* ∧ *y*)
⟨*proof*⟩

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*} ⟹ *the-elem y* = *x*
⟨*proof*⟩

lemma *nonemptyE*: *S* ≠ {} ⟹ (⋀*x*. *x* ∈ *S* ⟹ *R*) ⟹ *R*
⟨*proof*⟩

lemma *gt-or-eq-0*: *0* < *y* ∨ *0* = *y*
for *y* :: *nat*
⟨*proof*⟩

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

lemmas *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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *le-diff-eq'*: $a \leq c - b \longleftrightarrow b + a \leq c$
for $a\ b\ c$ :: *int*
⟨*proof*⟩

lemma *less-diff-eq'*: $a < c - b \longleftrightarrow b + a < c$
for $a\ b\ c$ :: *int*
⟨*proof*⟩

lemma *diff-less-eq'*: $a - b < c \longleftrightarrow a < b + c$
for $a\ b\ c$ :: *int*
⟨*proof*⟩

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

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

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*
⟨*proof*⟩

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

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

lemmas *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 ⟷ a mod m = b mod m*
for *a b m x* :: *nat*
⟨*proof*⟩

lemma *nat-minus-mod*: *(n − n mod m) mod m = 0*
for *m n* :: *nat*
⟨*proof*⟩

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

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

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

lemma *nat-mod-eq*: *b < n ⟹ a mod n = b mod n ⟹ a mod n = b*
for *a b n* :: *nat*
⟨*proof*⟩

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

lemma *nat-mod-lem*: $0 < n \Longrightarrow b < n \longleftrightarrow b \bmod n = b$
for $b\ n :: nat$
⟨*proof*⟩

lemma *mod-nat-add*: $x < z \Longrightarrow y < z \Longrightarrow (x + y) \bmod z = (\textit{if } x + y < z \textit{ then } x + y \textit{ else } x + y - z)$
for $x\ y\ z :: nat$
⟨*proof*⟩

lemma *mod-nat-sub*: $x < z \Longrightarrow (x - y) \bmod z = x - y$
for $x\ y :: nat$
⟨*proof*⟩

lemma *int-mod-eq*: $0 \leq b \Longrightarrow b < n \Longrightarrow a \bmod n = b \bmod n \Longrightarrow a \bmod n = b$
for $a\ b\ n :: int$
⟨*proof*⟩

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

lemma *int-mod-le*: $0 \leq a \Longrightarrow a \bmod n \leq a$
for $a :: int$
⟨*proof*⟩

lemma *mod-add-if-z*:
$x < z \Longrightarrow y < z \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x \Longrightarrow 0 \leq z \Longrightarrow$
$(x + y) \bmod z = (\textit{if } x + y < z \textit{ then } x + y \textit{ else } x + y - z)$
for $x\ y\ z :: int$
⟨*proof*⟩

lemma *mod-sub-if-z*:
$x < z \Longrightarrow y < z \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x \Longrightarrow 0 \leq z \Longrightarrow$
$(x - y) \bmod z = (\textit{if } y \leq x \textit{ then } x - y \textit{ else } x - y + z)$
for $x\ y\ z :: int$
⟨*proof*⟩

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*: $m \neq 0 \Longrightarrow (a + m * n) \textit{ div } m = a \textit{ div } m + n$
for $a\ m\ n :: int$
⟨*proof*⟩

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

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *pl-pl-mm*: $a + b = c + d \Longrightarrow a - c = d - b$
for *a b c d* :: *nat*
⟨*proof*⟩

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*: $0 < c \Longrightarrow a \geq b * c \longleftrightarrow a \ div \ c \geq b$
for *a b c* :: *nat*
⟨*proof*⟩

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

lemma *div-mult-le*: $a \ div \ b * b \leq a$
for *a b* :: *nat*
⟨*proof*⟩

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

lemma *given-quot*: $f > 0 \Longrightarrow (f * l + (f - 1)) \ div \ f = l$
for *f l* :: *nat*
⟨*proof*⟩

lemma *given-quot-alt*: $f > 0 \Longrightarrow (l * f + f - Suc\ 0) \ div \ f = l$
for *f l* :: *nat*
⟨*proof*⟩

lemma *diff-mod-le*: $a < d \Longrightarrow b \ dvd \ d \Longrightarrow a - a \ mod \ b \leq d - b$
for *a b d* :: *nat*
⟨*proof*⟩

lemma *less-le-mult'*: $w * c < b * c \Longrightarrow 0 \leq c \Longrightarrow (w + 1) * c \leq b * c$
for *b c w* :: *int*
⟨*proof*⟩

lemma *less-le-mult*: $w * c < b * c \Longrightarrow 0 \leq c \Longrightarrow w * c + c \leq b * c$
for *b c w* :: *int*
⟨*proof*⟩

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

lemma *gen-minus*: $0 < n \Longrightarrow f\ n = f\ (Suc\ (n - 1))$
⟨*proof*⟩

lemma *mpl-lem*: $j \leq i \Longrightarrow k < j \Longrightarrow i - j + k < i$
for *i j k* :: *nat*
⟨*proof*⟩

lemma *nonneg-mod-div*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq (a\ mod\ b) \wedge 0 \leq a\ div\ b$
for *a b* :: *int*
⟨*proof*⟩

declare *iszero-0* [*intro*]

lemma *min-pm* [*simp*]: $min\ a\ b + (a - b) = a$
for *a b* :: *nat*
⟨*proof*⟩

lemma *min-pm1* [*simp*]: $a - b + min\ a\ b = a$
for *a b* :: *nat*
⟨*proof*⟩

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

lemma *rev-min-pm1* [*simp*]: $a - b + min\ b\ a = a$
for *a b* :: *nat*
⟨*proof*⟩

lemma *min-minus* [*simp*]: $min\ m\ (m - k) = m - k$
for *m k* :: *nat*
⟨*proof*⟩

lemma *min-minus'* [*simp*]: $min\ (m - k)\ m = m - k$
for *m k* :: *nat*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *uint-nonnegative*: $0 \leq uint\ w$
⟨*proof*⟩

lemma *uint-bounded*: $uint\ w < 2 \hat{} len\text{-}of\ TYPE('a)$
for *w* :: *'a::len0 word*
⟨*proof*⟩

lemma *uint-idem*: $uint\ w\ mod\ 2 \hat{} len\text{-}of\ TYPE('a) = uint\ w$
for *w* :: *'a::len0 word*
⟨*proof*⟩

lemma *word-uint-eq-iff*: $a = b \longleftrightarrow uint\ a = uint\ b$
⟨*proof*⟩

lemma *word-uint-eqI*: $uint\ a = uint\ b \Longrightarrow a = b$
⟨*proof*⟩

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 $word\text{-}of\text{-}int\ k = Abs\text{-}word\ (k\ mod\ 2 \hat{} len\text{-}of\ TYPE('a))$

lemma *uint-word-of-int*: $uint\ (word\text{-}of\text{-}int\ k :: 'a::len0\ word) = k\ mod\ 2 \hat{} len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemma *word-of-int-uint*: $word\text{-}of\text{-}int\ (uint\ w) = w$
⟨*proof*⟩

lemma *split-word-all*: $(\bigwedge x::'a::len0\ word.\ PROP\ P\ x) \equiv (\bigwedge x.\ PROP\ P\ (word\text{-}of\text{-}int\ x))$
⟨*proof*⟩

10.2 Type conversions and casting

definition *sint* :: *'a::len word* ⇒ *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* ⇒ *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*}
⟨*proof*⟩

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

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

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 ⟨*proof*⟩

end

lemma *word-size-gt-0* [*iff*]: *0* < *size w*
for *w* :: *'a::len word*
⟨*proof*⟩

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

lemma *lens-not-0* [*iff*]:
fixes $w :: 'a::len\ word$
shows $size\ w \neq 0$
and $len\text{-}of\ TYPE('a) \neq 0$
⟨*proof*⟩

definition *source-size* :: $('a::len0\ word \Rightarrow 'b) \Rightarrow nat$
— whether a cast (or other) function is to a longer or shorter length
where [*code del*]: $source\text{-}size\ c = (let\ arb = undefined;\ x = c\ arb\ in\ size\ arb)$

definition *target-size* :: $('a \Rightarrow 'b::len0\ word) \Rightarrow nat$
where [*code del*]: $target\text{-}size\ c = size\ (c\ undefined)$

definition *is-up* :: $('a::len0\ word \Rightarrow 'b::len0\ word) \Rightarrow bool$
where $is\text{-}up\ c \longleftrightarrow source\text{-}size\ c \leq target\text{-}size\ c$

definition *is-down* :: $('a::len0\ word \Rightarrow 'b::len0\ word) \Rightarrow bool$
where $is\text{-}down\ c \longleftrightarrow target\text{-}size\ c \leq source\text{-}size\ c$

definition *of-bl* :: $bool\ list \Rightarrow 'a::len0\ word$
where $of\text{-}bl\ bl = word\text{-}of\text{-}int\ (bl\text{-}to\text{-}bin\ bl)$

definition *to-bl* :: $'a::len0\ word \Rightarrow bool\ list$
where $to\text{-}bl\ w = bin\text{-}to\text{-}bl\ (len\text{-}of\ TYPE('a))\ (uint\ w)$

definition *word-reverse* :: $'a::len0\ word \Rightarrow 'a\ word$
where $word\text{-}reverse\ w = of\text{-}bl\ (rev\ (to\text{-}bl\ w))$

definition *word-int-case* :: $(int \Rightarrow 'b) \Rightarrow 'a::len0\ word \Rightarrow 'b$
where $word\text{-}int\text{-}case\ f\ w = f\ (uint\ w)$

translations
$case\ x\ of\ XCONST\ of\text{-}int\ y \Rightarrow b \rightleftharpoons CONST\ word\text{-}int\text{-}case\ (\lambda y.\ b)\ x$
$case\ x\ of\ (XCONST\ of\text{-}int :: 'a)\ y \Rightarrow b \rightharpoonup CONST\ word\text{-}int\text{-}case\ (\lambda y.\ b)\ x$

10.3 Correspondence relation for theorem transfer

definition *cr-word* :: $int \Rightarrow 'a::len0\ word \Rightarrow bool$
where $cr\text{-}word = (\lambda x\ y.\ word\text{-}of\text{-}int\ x = y)$

lemma *Quotient-word*:
$Quotient\ (\lambda x\ y.\ bintrunc\ (len\text{-}of\ TYPE('a))\ x = bintrunc\ (len\text{-}of\ TYPE('a))\ y)$
$word\text{-}of\text{-}int\ uint\ (cr\text{-}word :: - \Rightarrow 'a::len0\ word \Rightarrow bool)$
⟨*proof*⟩

lemma *reflp-word*:
$reflp\ (\lambda x\ y.\ bintrunc\ (len\text{-}of\ TYPE('a::len0))\ x = bintrunc\ (len\text{-}of\ TYPE('a))\ y)$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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
⟨*proof*⟩

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 ⟨*proof*⟩

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))
⟨*proof*⟩

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)
⟨*proof*⟩

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

lemma *td-ext-ubin*:
td-ext (uint :: 'a word ⇒ int) word-of-int (uints (len-of TYPE('a::len0)))
(bintrunc (len-of TYPE('a)))
⟨*proof*⟩

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))
⟨*proof*⟩

10.6 Arithmetic operations

lift-definition *word-succ :: 'a::len0 word ⇒ 'a word* **is** *λx. x + 1*
⟨*proof*⟩

lift-definition *word-pred :: 'a::len0 word ⇒ 'a word* **is** *λx. x − 1*
⟨*proof*⟩

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

lift-definition *zero-word :: 'a word* **is** *0* ⟨*proof*⟩

lift-definition *one-word :: 'a word* **is** *1* ⟨*proof*⟩

lift-definition *plus-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op +*
⟨*proof*⟩

lift-definition *minus-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op −*
⟨*proof*⟩

lift-definition *uminus-word* :: *'a word* ⇒ *'a word* **is** *uminus*
⟨*proof*⟩

lift-definition *times-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op ∗*
⟨*proof*⟩

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
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*⟩

lemma *word-of-nat*: *of-nat n* = *word-of-int* (*int n*)
⟨*proof*⟩

lemma *word-of-int*: *of-int* = *word-of-int*
⟨*proof*⟩

definition *udvd* :: *'a*::*len word* ⇒ *'a*::*len word* ⇒ *bool* (**infixl** *udvd 50*)
where *a udvd b* = (∃ *n*≥*0*. *uint b* = *n* ∗ *uint a*)

10.7 Ordering

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

definition *word-le-def*: *a* ≤ *b* ⟷ *uint a* ≤ *uint b*

definition *word-less-def*: *a* < *b* ⟷ *uint a* < *uint b*

instance
⟨*proof*⟩

end

definition *word-sle* :: *'a*::*len word* ⇒ *'a word* ⇒ *bool* ((-/ <=*s* -) [*50*, *51*] *50*)
where *a* <=*s b* ⟷ *sint a* ≤ *sint b*

definition *word-sless* :: *'a*::*len word* ⇒ *'a word* ⇒ *bool* ((-/ <*s* -) [*50*, *51*] *50*)
where *x* <*s y* ⟷ *x* <=*s y* ∧ *x* ≠ *y*

10.8 Bit-wise operations

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

lift-definition *bitNOT-word* :: *'a word* ⇒ *'a word* **is** *bitNOT*
⟨*proof*⟩

lift-definition *bitAND-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitAND*
⟨*proof*⟩

lift-definition *bitOR-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitOR*
⟨*proof*⟩

lift-definition *bitXOR-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitXOR*
⟨*proof*⟩

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 or as signed, 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 ⟨*proof*⟩

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)*
⟨*proof*⟩

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

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

instance ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *uint-sint*: *uint w* = *bintrunc* (*len-of TYPE('a)*) (*sint w*)
for *w* :: *'a::len word*
⟨*proof*⟩

lemma *bintr-uint*: *len-of TYPE('a)* ≤ *n* ⟹ *bintrunc n* (*uint w*) = *uint w*
for *w* :: *'a::len0 word*
⟨*proof*⟩

lemma *wi-bintr*:
len-of TYPE('a::len0) ≤ *n* ⟹
word-of-int (*bintrunc n w*) = (*word-of-int w* :: *'a word*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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))
⟨*proof*⟩

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)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *word-numeral-alt*: *numeral b* = *word-of-int* (*numeral b*)
⟨*proof*⟩

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

lemma *word-neg-numeral-alt*: − *numeral b* = *word-of-int* (− *numeral b*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *uint-bintrunc* [*simp*]:
uint (*numeral bin* :: '*a word*) =
bintrunc (*len-of TYPE*('*a*::*len0*)) (*numeral bin*)
⟨*proof*⟩

lemma *uint-bintrunc-neg* [*simp*]:
uint (− *numeral bin* :: '*a word*) = *bintrunc* (*len-of TYPE*('*a*::*len0*)) (− *numeral bin*)
⟨*proof*⟩

lemma *sint-sbintrunc* [*simp*]:

sint (*numeral bin* :: *'a word*) = *sbintrunc* (*len-of TYPE('a::len)* − *1*) (*numeral bin*)
⟨*proof*⟩

lemma *sint-sbintrunc-neg* [*simp*]:
sint (− *numeral bin* :: *'a word*) = *sbintrunc* (*len-of TYPE('a::len)* − *1*) (− *numeral bin*)
⟨*proof*⟩

lemma *unat-bintrunc* [*simp*]:
unat (*numeral bin* :: *'a::len0 word*) = *nat* (*bintrunc* (*len-of TYPE('a)*) (*numeral bin*))
⟨*proof*⟩

lemma *unat-bintrunc-neg* [*simp*]:
unat (− *numeral bin* :: *'a::len0 word*) = *nat* (*bintrunc* (*len-of TYPE('a)*) (− *numeral bin*))
⟨*proof*⟩

lemma *size-0-eq*: *size w* = *0* ⟹ *v* = *w*
for *v w* :: *'a::len0 word*
⟨*proof*⟩

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

lemma *uint-lt2p* [*iff*]: *uint x* < *2* ^ *len-of TYPE('a)*
for *x* :: *'a::len0 word*
⟨*proof*⟩

lemma *sint-ge*: − (*2* ^ (*len-of TYPE('a)* − *1*)) ≤ *sint x*
for *x* :: *'a::len word*
⟨*proof*⟩

lemma *sint-lt*: *sint x* < *2* ^ (*len-of TYPE('a)* − *1*)
for *x* :: *'a::len word*
⟨*proof*⟩

lemma *sign-uint-Pls* [*simp*]: *bin-sign* (*uint x*) = *0*
⟨*proof*⟩

lemma *uint-m2p-neg*: *uint x* − *2* ^ *len-of TYPE('a)* < *0*
for *x* :: *'a::len0 word*
⟨*proof*⟩

lemma *uint-m2p-not-non-neg*: ¬ *0* ≤ *uint x* − *2* ^ *len-of TYPE('a)*
for *x* :: *'a::len0 word*
⟨*proof*⟩

lemma *lt2p-lem*: *len-of TYPE('a) ≤ n ⟹ uint w < 2 ^ n*
for *w :: 'a::len0 word*
⟨*proof*⟩

lemma *uint-le-0-iff* [*simp*]: *uint x ≤ 0 ⟷ uint x = 0*
⟨*proof*⟩

lemma *uint-nat*: *uint w = int (unat w)*
⟨*proof*⟩

lemma *uint-numeral*: *uint (numeral b :: 'a::len0 word) = numeral b mod 2 ^ len-of TYPE('a)*
⟨*proof*⟩

lemma *uint-neg-numeral*: *uint (- numeral b :: 'a::len0 word) = - numeral b mod 2 ^ len-of TYPE('a)*
⟨*proof*⟩

lemma *unat-numeral*: *unat (numeral b :: 'a::len0 word) = numeral b mod 2 ^ len-of TYPE('a)*
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *word-of-int-0* [*simp*, *code-post*]: *word-of-int 0 = 0*
⟨*proof*⟩

lemma *word-of-int-1* [*simp*, *code-post*]: *word-of-int 1 = 1*
⟨*proof*⟩

lemma *word-of-int-neg-1* [*simp*]: *word-of-int (- 1) = - 1*
⟨*proof*⟩

lemma *word-of-int-numeral* [*simp*] : *(word-of-int (numeral bin) :: 'a::len0 word) = numeral bin*
⟨*proof*⟩

lemma *word-of-int-neg-numeral* [*simp*]:
(word-of-int (- numeral bin) :: 'a::len0 word) = - numeral bin
⟨*proof*⟩

lemma *word-int-case-wi*:
word-int-case f (word-of-int i :: 'b word) = f (i mod 2 ^ len-of TYPE('b::len0))

⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *sint-range-size*: − (*2* ^ (*size w* − *Suc 0*)) ≤ *sint w* ∧ *sint w* < *2* ^ (*size*
w − *Suc 0*)
⟨*proof*⟩

lemma *sint-above-size*: *2* ^ (*size w* − *1*) ≤ *x* ⟹ *sint w* < *x*
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *sint-below-size*: *x* ≤ − (*2* ^ (*size w* − *1*)) ⟹ *x* ≤ *sint w*
for *w* :: *'a*::*len word*
⟨*proof*⟩

10.13 Testing bits

lemma *test-bit-eq-iff*: *test-bit u* = *test-bit v* ⟷ *u* = *v*
for *u v* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *test-bit-size* [*rule-format*] : *w* !! *n* ⟶ *n* < *size w*
for *w* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *word-eq-iff*: *x* = *y* ⟷ (∀ *n*<*len-of TYPE*(*'a*). *x* !! *n* = *y* !! *n*)
for *x y* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *word-eqI*: (⋀*n*. *n* < *size u* ⟶ *u* !! *n* = *v* !! *n*) ⟹ *u* = *v*
for *u* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *word-eqD*: $u = v \Longrightarrow u \mathbin{!!} x = v \mathbin{!!} x$
for $u\ v :: 'a{::}len0\ word$
⟨*proof*⟩

lemma *test-bit-bin'*: $w \mathbin{!!} n \longleftrightarrow n < size\ w \wedge bin\text{-}nth\ (uint\ w)\ n$
⟨*proof*⟩

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 $w :: 'a{::}len0\ word$
⟨*proof*⟩

lemma *bin-nth-sint*:
$len\text{-}of\ TYPE('a) \leq n \Longrightarrow$
$bin\text{-}nth\ (sint\ w)\ n = bin\text{-}nth\ (sint\ w)\ (len\text{-}of\ TYPE('a) - 1)$
for $w :: 'a{::}len\ word$
⟨*proof*⟩

lemma *td-bl*:
type-definition
$(to\text{-}bl :: 'a{::}len0\ word \Rightarrow bool\ list)$
of-bl
$\{bl.\ length\ bl = len\text{-}of\ TYPE('a)\}$
⟨*proof*⟩

interpretation *word-bl*:
type-definition
$to\text{-}bl :: 'a{::}len0\ word \Rightarrow bool\ list$
of-bl
$\{bl.\ length\ bl = len\text{-}of\ TYPE('a{::}len0)\}$
⟨*proof*⟩

lemmas *word-bl-Rep'* = *word-bl.Rep* [*unfolded mem-Collect-eq*, *iff*]

lemma *word-size-bl*: $size\ w = size\ (to\text{-}bl\ w)$
⟨*proof*⟩

lemma *to-bl-use-of-bl*: $to\text{-}bl\ w = bl \longleftrightarrow w = of\text{-}bl\ bl \wedge length\ bl = length\ (to\text{-}bl\ w)$
⟨*proof*⟩

lemma *to-bl-word-rev*: $to\text{-}bl\ (word\text{-}reverse\ w) = rev\ (to\text{-}bl\ w)$
⟨*proof*⟩

lemma *word-rev-rev* [*simp*] : $word\text{-}reverse\ (word\text{-}reverse\ w) = w$
⟨*proof*⟩

lemma *word-rev-gal*: *word-reverse w = u ⟹ word-reverse u = w*
⟨*proof*⟩

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

lemma *length-bl-gt-0* [*iff*]: *0 < length (to-bl x)*
for *x* :: *'a*::*len word*
⟨*proof*⟩

lemma *bl-not-Nil* [*iff*]: *to-bl x ≠ []*
for *x* :: *'a*::*len word*
⟨*proof*⟩

lemma *length-bl-neq-0* [*iff*]: *length (to-bl x) ≠ 0*
for *x* :: *'a*::*len word*
⟨*proof*⟩

lemma *hd-bl-sign-sint*: *hd (to-bl w) = (bin-sign (sint w) = −1)*
⟨*proof*⟩

lemma *of-bl-drop'*:
lend = length bl − len-of TYPE('a::*len0) ⟹*
of-bl (drop lend bl) = (of-bl bl :: 'a word)
⟨*proof*⟩

lemma *test-bit-of-bl*:
(of-bl bl::*'a*::*len0 word) !! n = (rev bl ! n ∧ n < len-of TYPE('a) ∧ n < length bl)*
⟨*proof*⟩

lemma *no-of-bl*: *(numeral bin ::'a*::*len0 word) = of-bl (bin-to-bl (len-of TYPE('a)) (numeral bin))*
⟨*proof*⟩

lemma *uint-bl*: *to-bl w = bin-to-bl (size w) (uint w)*
⟨*proof*⟩

lemma *to-bl-bin*: *bl-to-bin (to-bl w) = uint w*
⟨*proof*⟩

lemma *to-bl-of-bin*: *to-bl (word-of-int bin*::*'a*::*len0 word) = bin-to-bl (len-of TYPE('a)) bin*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *to-bl-neg-numeral* [*simp*]:
to-bl (− *numeral bin*::′*a*::*len0 word*) =
bin-to-bl (*len-of TYPE*(′*a*)) (− *numeral bin*)
⟨*proof*⟩

lemma *to-bl-to-bin* [*simp*] : *bl-to-bin* (*to-bl w*) = *uint w*
⟨*proof*⟩

lemma *uint-bl-bin*: *bl-to-bin* (*bin-to-bl* (*len-of TYPE*(′*a*)) (*uint x*)) = *uint x*
for *x* :: ′*a*::*len0 word*
⟨*proof*⟩

lemma *uints-unats*: *uints n* = *int* ‘ *unats n*
⟨*proof*⟩

lemma *unats-uints*: *unats n* = *nat* ‘ *uints n*
⟨*proof*⟩

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

lemma *num-of-bintr′*:
bintrunc (*len-of TYPE*(′*a*::*len0*)) (*numeral a*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: ′*a word*)
⟨*proof*⟩

lemma *num-of-sbintr′*:
sbintrunc (*len-of TYPE*(′*a*::*len*) − *1*) (*numeral a*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: ′*a word*)
⟨*proof*⟩

lemma *num-abs-bintr*:
(*numeral x* :: ′*a word*) =
word-of-int (*bintrunc* (*len-of TYPE*(′*a*::*len0*)) (*numeral x*))
⟨*proof*⟩

lemma *num-abs-sbintr*:
(*numeral x* :: ′*a word*) =
word-of-int (*sbintrunc* (*len-of TYPE*(′*a*::*len*) − *1*) (*numeral x*))
⟨*proof*⟩

lemma *ucast-id*: *ucast w = w*
⟨*proof*⟩

lemma *scast-id*: *scast w = w*
⟨*proof*⟩

lemma *ucast-bl*: *ucast w = of-bl (to-bl w)*
⟨*proof*⟩

lemma *nth-ucast*: *(ucast w::'a::len0 word) !! n = (w !! n ∧ n < len-of TYPE('a))*
⟨*proof*⟩

lemma *ucast-bintr* [*simp*]:
ucast (numeral w :: 'a::len0 word) =
word-of-int (bintrunc (len-of TYPE('a)) (numeral w))
⟨*proof*⟩

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

lemma *source-size*: *source-size (c::'a::len0 word ⇒ -) = len-of TYPE('a)*
⟨*proof*⟩

lemma *target-size*: *target-size (c::- ⇒ 'b::len0 word) = len-of TYPE('b)*
⟨*proof*⟩

lemma *is-down*: *is-down c ⟷ len-of TYPE('b) ≤ len-of TYPE('a)*
for *c :: 'a::len0 word ⇒ 'b::len0 word*
⟨*proof*⟩

lemma *is-up*: *is-up c ⟷ len-of TYPE('a) ≤ len-of TYPE('b)*
for *c :: 'a::len0 word ⇒ 'b::len0 word*
⟨*proof*⟩

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

lemma *down-cast-same* [*OF refl*]: *uc = ucast ⟹ is-down uc ⟹ uc = scast*
⟨*proof*⟩

lemma *word-rev-tf*:
to-bl (of-bl bl::'a::len0 word) =
rev (takefill False (len-of TYPE('a)) (rev bl))

⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ucast-down-drop* [*OF refl*]:
uc = *ucast* ⟹ *source-size uc* = *target-size uc* + *n* ⟹
to-bl (*uc w*) = *drop n* (*to-bl w*)
⟨*proof*⟩

lemma *scast-down-drop* [*OF refl*]:
sc = *scast* ⟹ *source-size sc* = *target-size sc* + *n* ⟹
to-bl (*sc w*) = *drop n* (*to-bl w*)
⟨*proof*⟩

lemma *sint-up-scast* [*OF refl*]: *sc* = *scast* ⟹ *is-up sc* ⟹ *sint* (*sc w*) = *sint w*
⟨*proof*⟩

lemma *uint-up-ucast* [*OF refl*]: *uc* = *ucast* ⟹ *is-up uc* ⟹ *uint* (*uc w*) = *uint w*
⟨*proof*⟩

lemma *ucast-up-ucast* [*OF refl*]: *uc* = *ucast* ⟹ *is-up uc* ⟹ *ucast* (*uc w*) = *ucast w*
⟨*proof*⟩

lemma *scast-up-scast* [*OF refl*]: *sc* = *scast* ⟹ *is-up sc* ⟹ *scast* (*sc w*) = *scast w*
⟨*proof*⟩

lemma *ucast-of-bl-up* [*OF refl*]: *w* = *of-bl bl* ⟹ *size bl* ≤ *size w* ⟹ *ucast w* = *of-bl bl*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *up-scast-surj*:
is-up (*scast* :: *'b*::*len word* ⇒ *'a*::*len word*) ⟹
surj (*scast* :: *'a word* ⇒ *'b word*)
⟨*proof*⟩

lemma *down-scast-inj*:
is-down (*scast* :: *'b*::*len word* ⇒ *'a*::*len word*) ⟹
inj-on (*ucast* :: *'a word* ⇒ *'b word*) *A*
⟨*proof*⟩

lemma *down-ucast-inj*:
is-down (*ucast* :: *'b*::*len0 word* ⇒ *'a*::*len0 word*) ⟹
inj-on (*ucast* :: *'a word* ⇒ *'b word*) *A*
⟨*proof*⟩

lemma *of-bl-append-same*: *of-bl* (*X* @ *to-bl w*) = *w*
⟨*proof*⟩

lemma *ucast-down-wi* [*OF refl*]: *uc* = *ucast* ⟹ *is-down uc* ⟹ *uc* (*word-of-int x*) = *word-of-int x*
⟨*proof*⟩

lemma *ucast-down-no* [*OF refl*]: *uc* = *ucast* ⟹ *is-down uc* ⟹ *uc* (*numeral bin*) = *numeral bin*
⟨*proof*⟩

lemma *ucast-down-bl* [*OF refl*]: *uc* = *ucast* ⟹ *is-down uc* ⟹ *uc* (*of-bl bl*) = *of-bl bl*
⟨*proof*⟩

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*: $a < b \longleftrightarrow uint\ a < uint\ b$
⟨*proof*⟩

lemma *signed-linorder*: *class.linorder word-sle word-sless*
⟨*proof*⟩

interpretation *signed*: *linorder word-sle word-sless*
⟨*proof*⟩

lemma *udvdI*: $0 \leq n \Longrightarrow uint\ b = n * uint\ a \Longrightarrow a\ udvd\ b$
⟨*proof*⟩

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*: $-\ 1 = word\text{-}of\text{-}int\ (-\ 1)$
⟨*proof*⟩

lemma *word-0-bl* [*simp*]: *of-bl* [] = *0*
⟨*proof*⟩

lemma *word-1-bl*: *of-bl* [*True*] = *1*
⟨*proof*⟩

lemma *uint-eq-0* [*simp*]: *uint 0* = *0*
⟨*proof*⟩

lemma *of-bl-0* [*simp*]: *of-bl* (*replicate n False*) = *0*
⟨*proof*⟩

lemma *to-bl-0* [*simp*]: *to-bl* (*0*::*'a*::*len0 word*) = *replicate* (*len-of TYPE*(*'a*)) *False*
⟨*proof*⟩

lemma *uint-0-iff*: *uint x* = *0* $\longleftrightarrow$ *x* = *0*
⟨*proof*⟩

lemma *unat-0-iff*: *unat x* = *0* $\longleftrightarrow$ *x* = *0*
⟨*proof*⟩

lemma *unat-0* [*simp*]: *unat 0* = *0*
⟨*proof*⟩

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

⟨*proof*⟩

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* ⟷ *x* ≠ *0*
⟨*proof*⟩

lemma *ucast-0* [*simp*]: *ucast 0* = *0*
⟨*proof*⟩

lemma *sint-0* [*simp*]: *sint 0* = *0*
⟨*proof*⟩

lemma *scast-0* [*simp*]: *scast 0* = *0*
⟨*proof*⟩

lemma *sint-n1* [*simp*] : *sint* (− *1*) = − *1*
⟨*proof*⟩

lemma *scast-n1* [*simp*]: *scast* (− *1*) = − *1*
⟨*proof*⟩

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

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

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

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*
⟨*proof*⟩

lemma *uint-cong*: *x* = *y* ⟹ *uint x* = *uint y*
⟨*proof*⟩

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* $(1 :: 'a\ word) = 1\ mod\ 2\ \hat{}\ len\text{-}of\ TYPE('a)$
⟨*proof*⟩

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* $(1 :: 'a\ word) = bintrunc\ (len\text{-}of\ TYPE('a))\ 1$
⟨*proof*⟩

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* $(1 :: 'a\ word) = sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ 1$
⟨*proof*⟩

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* $(- 1)$
⟨*proof*⟩

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 $(- numeral\ w) = - numeral\ w - 1$
⟨*proof*⟩

lemma *word-sp-01* [*simp*]:
word-succ $(- 1) = 0 \wedge word\text{-}succ\ 0 = 1 \wedge word\text{-}pred\ 0 = - 1 \wedge word\text{-}pred\ 1 =$

0
⟨*proof*⟩

lemma *word-of-int-Ex*: ∃ *y*. *x* = *word-of-int y*
⟨*proof*⟩

10.16 Order on fixed-length words

lemma *word-zero-le* [*simp*]: *0* ≤ *y*
for *y* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *word-m1-ge* [*simp*] : *word-pred 0* ≥ *y*
⟨*proof*⟩

lemma *word-n1-ge* [*simp*]: *y* ≤ −*1*
for *y* :: *'a*::*len0 word*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-le-nat-alt*: *a* ≤ *b* ⟷ *unat a* ≤ *unat b*
⟨*proof*⟩

lemma *word-less-nat-alt*: *a* < *b* ⟷ *unat a* < *unat b*
⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *udvd-nat-alt*: *a udvd b* ⟷ (∃ *n*≥*0*. *unat b* = *n* ∗ *unat a*)

⟨*proof*⟩

lemma *udvd-iff-dvd*: *x udvd y* ⟷ *unat x dvd unat y*
⟨*proof*⟩

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

lemma *unat-minus-one*:
assumes *w* ≠ *0*
shows *unat* (*w* − *1*) = *unat w* − *1*
⟨*proof*⟩

lemma *measure-unat*: *p* ≠ *0* ⟹ *unat* (*p* − *1*) < *unat p*
⟨*proof*⟩

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* ^ *len-of TYPE*('*a*)
for *x* :: '*a*::*len0 word* **and** *y* :: '*b*::*len0 word*
⟨*proof*⟩

10.17 Conditions for the addition (etc) of two words to overflow

lemma *uint-add-lem*:
(*uint x* + *uint y* < *2* ^ *len-of TYPE*('*a*)) =
(*uint* (*x* + *y*) = *uint x* + *uint y*)
for *x y* :: '*a*::*len0 word*
⟨*proof*⟩

lemma *uint-mult-lem*:
(*uint x* ∗ *uint y* < *2* ^ *len-of TYPE*('*a*)) =
(*uint* (*x* ∗ *y*) = *uint x* ∗ *uint y*)
for *x y* :: '*a*::*len0 word*
⟨*proof*⟩

lemma *uint-sub-lem*: *uint x* ≥ *uint y* ⟷ *uint* (*x* − *y*) = *uint x* − *uint y*
⟨*proof*⟩

lemma *uint-add-le*: *uint* (*x* + *y*) ≤ *uint x* + *uint y*
⟨*proof*⟩

lemma *uint-sub-ge*: *uint* (*x* − *y*) ≥ *uint x* − *uint y*
⟨*proof*⟩

lemma *mod-add-if-z*:
x < *z* ⟹ *y* < *z* ⟹ *0* ≤ *y* ⟹ *0* ≤ *x* ⟹ *0* ≤ *z* ⟹
(*x* + *y*) *mod z* = (*if x* + *y* < *z then x* + *y else x* + *y* − *z*)

for *x y z* :: *int*
⟨*proof*⟩

lemma *uint-plus-if'*:
uint (*a* + *b*) =
(*if uint a* + *uint b* < 2 ^ *len-of TYPE*('*a*) *then uint a* + *uint b*
else uint a + *uint b* − 2 ^ *len-of TYPE*('*a*))
for *a b* :: '*a*::*len0 word*
⟨*proof*⟩

lemma *mod-sub-if-z*:
x < *z* ⟹ *y* < *z* ⟹ 0 ≤ *y* ⟹ 0 ≤ *x* ⟹ 0 ≤ *z* ⟹
(*x* − *y*) *mod z* = (*if y* ≤ *x then x* − *y else x* − *y* + *z*)
for *x y z* :: *int*
⟨*proof*⟩

lemma *uint-sub-if'*:
uint (*a* − *b*) =
(*if uint b* ≤ *uint a then uint a* − *uint b*
else uint a − *uint b* + 2 ^ *len-of TYPE*('*a*))
for *a b* :: '*a*::*len0 word*
⟨*proof*⟩

10.18 Definition of *uint-arith*

lemma *word-of-int-inverse*:
word-of-int r = *a* ⟹ 0 ≤ *r* ⟹ *r* < 2 ^ *len-of TYPE*('*a*) ⟹ *uint a* = *r*
for *a* :: '*a*::*len0 word*
⟨*proof*⟩

lemma *uint-split*:
P (*uint x*) = (∀ *i*. *word-of-int i* = *x* ∧ 0 ≤ *i* ∧ *i* < 2^*len-of TYPE*('*a*) ⟶ *P i*)
for *x* :: '*a*::*len0 word*
⟨*proof*⟩

lemma *uint-split-asm*:
P (*uint x*) = (∄ *i*. *word-of-int i* = *x* ∧ 0 ≤ *i* ∧ *i* < 2^*len-of TYPE*('*a*) ∧ ¬ *P i*)
for *x* :: '*a*::*len0 word*
⟨*proof*⟩

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* ⟹ *a* ^ *b* = *c* ^ *d*

⟨*proof*⟩

⟨*ML*⟩

10.19 More on overflows and monotonicity

lemma *no-plus-overflow-uint-size*: $x \leq x + y \longleftrightarrow uint\ x + uint\ y < 2\ \hat{}\ size\ x$
for $x\ y :: 'a::len0\ word$
⟨*proof*⟩

lemmas *no-olen-add* = *no-plus-overflow-uint-size* [*unfolded word-size*]

lemma *no-ulen-sub*: $x \geq x - y \longleftrightarrow uint\ y \leq uint\ x$
for $x\ y :: 'a::len0\ word$
⟨*proof*⟩

lemma *no-olen-add'*: $x \leq y + x \longleftrightarrow uint\ y + uint\ x < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
for $x\ y :: 'a::len0\ word$
⟨*proof*⟩

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 $x :: 'a::len\ word$
⟨*proof*⟩

lemma *word-le-sub1*: $x \neq 0 \Longrightarrow 1 \leq x \longleftrightarrow 0 \leq x - 1$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *sub-wrap-lt*: $x < x - z \longleftrightarrow x < z$
for $x\ z :: 'a::len0\ word$
⟨*proof*⟩

lemma *sub-wrap*: $x \leq x - z \longleftrightarrow z = 0 \vee x < z$
for $x\ z :: 'a::len0\ word$
⟨*proof*⟩

lemma *plus-minus-not-NULL-ab*: $x \leq ab - c \Longrightarrow c \leq ab \Longrightarrow c \neq 0 \Longrightarrow x + c \neq 0$
for $x\ ab\ c :: 'a::len0\ word$

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *le-minus'*: $a + c \leq b \Longrightarrow a \leq a + c \Longrightarrow c \leq b - a$
for *a b c* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *le-plus'*: $a \leq b \Longrightarrow c \leq b - a \Longrightarrow a + c \leq b$
for *a b c* :: *'a*::*len0 word*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-less-minus-cancel*: $y - x < z - x \Longrightarrow x \leq z \Longrightarrow y < z$
for *x y z* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *word-less-minus-mono-left*: $y < z \Longrightarrow x \leq y \Longrightarrow y - x < z - x$
for *x y z* :: *'a*::*len0 word*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-less-add-right*: $x < y - z \Longrightarrow z \leq y \Longrightarrow x + z < y$
for $x\ y\ z$ :: $'a$::*len0 word*
⟨*proof*⟩

lemma *word-less-sub-right*: $x < y + z \Longrightarrow y \leq x \Longrightarrow x - y < z$
for $x\ y\ z$ :: $'a$::*len0 word*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *inc-le*: $i < m \Longrightarrow i + 1 \leq m$
for $i\ m$ :: $'a$::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *udvd-incr-lem*:
$up < uq \Longrightarrow up = ua + n * uint\ K \Longrightarrow$
$uq = ua + n' * uint\ K \Longrightarrow up + uint\ K \leq uq$
⟨*proof*⟩

lemma *udvd-incr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
$uint\ q = ua + n' * uint\ K \Longrightarrow p + K \leq q$
⟨*proof*⟩

lemma *udvd-decr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$

*uint q = ua + n' * uint K ⟹ p ≤ q − K*
⟨*proof*⟩

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 ⟹ z udvd xy ⟹ z udvd k ⟹ xy ≤ k − z*
⟨*proof*⟩

lemma *udvd-incr2-K*:
p < a + s ⟹ a ≤ a + s ⟹ K udvd s ⟹ K udvd p − a ⟹ a ≤ p ⟹
0 < K ⟹ p ≤ p + K ∧ p + K ≤ a + s
⟨*proof*⟩

lemma *word-succ-rbl*: *to-bl w = bl ⟹ to-bl (word-succ w) = rev (rbl-succ (rev bl))*
⟨*proof*⟩

lemma *word-pred-rbl*: *to-bl w = bl ⟹ to-bl (word-pred w) = rev (rbl-pred (rev bl))*
⟨*proof*⟩

lemma *word-add-rbl*:
to-bl v = vbl ⟹ to-bl w = wbl ⟹
to-bl (v + w) = rev (rbl-add (rev vbl) (rev wbl))
⟨*proof*⟩

lemma *word-mult-rbl*:
to-bl v = vbl ⟹ to-bl w = wbl ⟹
*to-bl (v * w) = rev (rbl-mult (rev vbl) (rev wbl))*
⟨*proof*⟩

lemma *rtb-rbl-ariths*:
rev (to-bl w) = ys ⟹ rev (to-bl (word-succ w)) = rbl-succ ys
rev (to-bl w) = ys ⟹ rev (to-bl (word-pred w)) = rbl-pred ys
*rev (to-bl v) = ys ⟹ rev (to-bl w) = xs ⟹ rev (to-bl (v * w)) = rbl-mult ys xs*
rev (to-bl v) = ys ⟹ rev (to-bl w) = xs ⟹ rev (to-bl (v + w)) = rbl-add ys xs
⟨*proof*⟩

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)*
⟨*proof*⟩

lemma *iszero-word-no* [*simp*]:
iszero (*numeral bin* :: *'a*::*len word*) =
iszero (*bintrunc* (*len-of TYPE*(*'a*)) (*numeral bin*))
⟨*proof*⟩

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*)
⟨*proof*⟩

lemmas *unat-of-nat* = *td-ext-unat* [*THEN td-ext.eq-norm*]

interpretation *word-unat*:
td-ext
unat::*'a*::*len word* ⇒ *nat*
of-nat
unats (*len-of TYPE*(*'a*::*len*))
λ*i*. *i mod 2* ^ *len-of TYPE*(*'a*::*len*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-nchotomy*: ∀ *w* :: *'a*::*len word*. ∃ *n*. *w* = *of-nat n* ∧ *n* < *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *of-nat-eq*: *of-nat n* = *w* ⟷ (∃ *q*. *n* = *unat w* + *q* ∗ *2* ^ *len-of TYPE*(*'a*))
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *of-nat-eq-size*: *of-nat n* = *w* ⟷ (∃ *q*. *n* = *unat w* + *q* ∗ *2* ^ *size w*)
⟨*proof*⟩

lemma *of-nat-0*: *of-nat m* = (*0*::*'a*::*len word*) ⟷ (∃ *q*. *m* = *q* ∗ *2* ^ *len-of TYPE*(*'a*))

⟨*proof*⟩

lemma *of-nat-2p* [*simp*]: *of-nat* (*2* ^ *len-of TYPE*('*a*)) = (*0*::'*a*::*len word*)
⟨*proof*⟩

lemma *of-nat-gt-0*: *of-nat k* ≠ *0* ⟹ *0* < *k*
⟨*proof*⟩

lemma *of-nat-neq-0*: *0* < *k* ⟹ *k* < *2* ^ *len-of TYPE*('*a*::*len*) ⟹ *of-nat k* ≠ (*0* :: '*a word*)
⟨*proof*⟩

lemma *Abs-fnat-hom-add*: *of-nat a* + *of-nat b* = *of-nat* (*a* + *b*)
⟨*proof*⟩

lemma *Abs-fnat-hom-mult*: *of-nat a* * *of-nat b* = (*of-nat* (*a* * *b*) :: '*a*::*len word*)
⟨*proof*⟩

lemma *Abs-fnat-hom-Suc*: *word-succ* (*of-nat a*) = *of-nat* (*Suc a*)
⟨*proof*⟩

lemma *Abs-fnat-hom-0*: (*0*::'*a*::*len word*) = *of-nat 0*
⟨*proof*⟩

lemma *Abs-fnat-hom-1*: (*1*::'*a*::*len word*) = *of-nat* (*Suc 0*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *word-arith-nat-mult*: *a* * *b* = *of-nat* (*unat a* * *unat b*)
⟨*proof*⟩

lemma *word-arith-nat-Suc*: *word-succ a* = *of-nat* (*Suc* (*unat a*))
⟨*proof*⟩

lemma *word-arith-nat-div*: *a div b* = *of-nat* (*unat a div unat b*)
⟨*proof*⟩

lemma *word-arith-nat-mod*: *a mod b* = *of-nat* (*unat a mod unat b*)
⟨*proof*⟩

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*: $x = y \Longrightarrow unat\ x = unat\ y$
⟨*proof*⟩

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*:
$unat\ x + unat\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a) \longleftrightarrow unat\ (x + y) = unat\ x + unat\ y$
for *x y* :: *'a*::*len word*
⟨*proof*⟩

lemma *unat-mult-lem*:
$unat\ x * unat\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a) \longleftrightarrow unat\ (x * y) = unat\ x * unat\ y$
for *x y* :: *'a*::*len word*
⟨*proof*⟩

lemmas *unat-plus-if'* =
trans [*OF unat-word-ariths*(*1*) *mod-nat-add*, *simplified*]

lemma *le-no-overflow*: $x \leq b \Longrightarrow a \leq a + b \Longrightarrow x \leq a + b$
for *a b x* :: *'a*::*len0 word*
⟨*proof*⟩

lemmas *un-ui-le* =
trans [*OF word-le-nat-alt* [*symmetric*] *word-le-def*]

lemma *unat-sub-if-size*:
$unat\ (x - y) =$
$(if\ unat\ y \leq unat\ x$
$\ then\ unat\ x - unat\ y$
$\ else\ unat\ x + 2\ \hat{}\ size\ x - unat\ y)$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *unat-mod*: $unat\ (x\ mod\ y) = unat\ x\ mod\ unat\ y$
for *x y* :: *'a*::*len word*
⟨*proof*⟩

lemma *uint-div*: *uint* (*x div y*) = *uint x div uint y*
for *x y* :: *'a*::*len word*
⟨*proof*⟩

lemma *uint-mod*: *uint* (*x mod y*) = *uint x mod uint y*
for *x y* :: *'a*::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *no-plus-overflow-unat-size*: *x* ≤ *x* + *y* ⟷ *unat x* + *unat y* < *2* ^ *size x*
for *x y* :: *'a*::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *div-lt'*: *i* ≤ *k div x* ⟹ *unat i* * *unat x* < *2* ^ *len-of TYPE*(*'a*)

for $i\ k\ x :: 'a$::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *div-le-mult*: $i \leq k\ div\ x \Longrightarrow 0 < x \Longrightarrow i * x \leq k$
for $i\ k\ x :: 'a$::*len word*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-div-mult-le*: $a\ div\ b * b \leq a$
for $a\ b :: 'a$::*len word*

⟨*proof*⟩

lemma *word-mod-less-divisor*: *0 < n ⟹ m mod n < n*
for *m n* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-of-int-power-hom*: *word-of-int a ^ n = (word-of-int (a ^ n)* :: *'a*::*len word)*
⟨*proof*⟩

lemma *word-arith-power-alt*: *a ^ n = (word-of-int (uint a ^ n)* :: *'a*::*len word)*
⟨*proof*⟩

lemma *of-bl-length-less*:
length x = k ⟹ k < len-of TYPE('a) ⟹ (of-bl x :: *'a*::*len word) < 2 ^ k*
⟨*proof*⟩

10.23 Cardinality, finiteness of set of words

instance *word* :: (*len0*) *finite*
⟨*proof*⟩

lemma *card-word*: *CARD('a*::*len0 word) = 2 ^ len-of TYPE('a)*
⟨*proof*⟩

lemma *card-word-size*: *card* (*UNIV* :: *'a word set*) = (*2 ^ size x*)
for *x* :: *'a*::*len0 word*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *uint-or*: *uint* (*x OR y*) = *uint x OR uint y*
⟨*proof*⟩

lemma *uint-and*: *uint* (*x AND y*) = *uint x AND uint y*
⟨*proof*⟩

lemma *test-bit-wi* [*simp*]:
(*word-of-int x* :: *'a*::*len0 word*) !! *n* ⟷ *n* < *len-of TYPE*(*'a*) ∧ *bin-nth x n*
⟨*proof*⟩

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* ⇒ -)
⟨*proof*⟩

lemma *word-ops-nth-size*:
n < *size x* ⟹
(*x OR y*) !! *n* = (*x* !! *n* | *y* !! *n*) ∧
(*x AND y*) !! *n* = (*x* !! *n* ∧ *y* !! *n*) ∧
(*x XOR y*) !! *n* = (*x* !! *n* ≠ *y* !! *n*) ∧
(*NOT x*) !! *n* = (¬ *x* !! *n*)
for *x* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *word-ao-nth*:
(*x OR y*) !! *n* = (*x* !! *n* | *y* !! *n*) ∧
(*x AND y*) !! *n* = (*x* !! *n* ∧ *y* !! *n*)
for *x* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *test-bit-numeral* [*simp*]:
(*numeral w* :: *'a*::*len0 word*) !! *n* ⟷
n < *len-of TYPE*(*'a*) ∧ *bin-nth* (*numeral w*) *n*
⟨*proof*⟩

lemma *test-bit-neg-numeral* [*simp*]:
(− *numeral w* :: *'a*::*len0 word*) !! *n* ⟷
n < *len-of TYPE*(*'a*) ∧ *bin-nth* (− *numeral w*) *n*
⟨*proof*⟩

lemma *test-bit-1* [*simp*]: (*1* :: *'a*::*len word*) !! *n* ⟷ *n* = *0*
⟨*proof*⟩

lemma *nth-0* [*simp*]: ¬ (*0* :: *'a*::*len0 word*) !! *n*
⟨*proof*⟩

lemma *nth-minus1* [*simp*]: (*−1* :: *'a*::*len0 word*) !! *n* ⟷ *n* < *len-of TYPE*(*'a*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-bw-same*:
$x\ AND\ x = x$
$x\ OR\ x = x$

x XOR x = 0
for *x* :: *'a::len0 word*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-not-not* [*simp*]: *NOT NOT x = x*
for *x* :: *'a::len0 word*
⟨*proof*⟩

lemma *word-ao-dist*: *(x OR y) AND z = x AND z OR y AND z*
for *x* :: *'a::len0 word*
⟨*proof*⟩

lemma *word-oa-dist*: *x AND y OR z = (x OR z) AND (y OR z)*
for *x* :: *'a::len0 word*
⟨*proof*⟩

lemma *word-add-not* [*simp*]: *x + NOT x = -1*
for *x* :: *'a::len0 word*
⟨*proof*⟩

lemma *word-plus-and-or* [*simp*]: *(x AND y) + (x OR y) = x + y*
for *x* :: *'a::len0 word*
⟨*proof*⟩

lemma *leoa*: *w = x OR y ⟹ y = w AND y*
for *x* :: *'a::len0 word*
⟨*proof*⟩

lemma *leao*: *w' = x' AND y' ⟹ x' = x' OR w'*
for *x'* :: *'a::len0 word*
⟨*proof*⟩

lemma *word-ao-equiv*: *w = w OR w' ⟷ w' = w AND w'*
for *w w'* :: *'a::len0 word*
⟨*proof*⟩

lemma *le-word-or2*: *x ≤ x OR y*

for *x y* :: *'a*::*len0 word*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *bl-word-xor*: *to-bl* (*v XOR w*) = *map2 op* ≠ (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *bl-word-or*: *to-bl* (*v OR w*) = *map2 op* ∨ (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *bl-word-and*: *to-bl* (*v AND w*) = *map2 op* ∧ (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *word-lsb-alt*: *lsb w* = *test-bit w 0*
for *w* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *word-lsb-1-0* [*simp*]: *lsb* (*1*::*'a*::*len word*) ∧ ¬ *lsb* (*0*::*'b*::*len0 word*)
⟨*proof*⟩

lemma *word-lsb-last*: *lsb w* = *last* (*to-bl w*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-lsb-int*: *lsb w* ⟷ *uint w mod 2* = *1*
⟨*proof*⟩

lemma *word-msb-sint*: *msb w* ⟷ *sint w* < *0*
⟨*proof*⟩

lemma *msb-word-of-int*: *msb* (*word-of-int x*::*'a*::*len word*) = *bin-nth x* (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *word-msb-numeral* [*simp*]:
msb (*numeral w*::*'a*::*len word*) = *bin-nth* (*numeral w*) (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *word-msb-neg-numeral* [*simp*]:
msb (− *numeral w*::*'a*::*len word*) = *bin-nth* (− *numeral w*) (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

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

lemma *word-msb-1* [*simp*]: *msb* (*1*::*'a*::*len word*) ⟷ *len-of TYPE*(*'a*) = *1*
⟨*proof*⟩

lemma *word-msb-nth*: *msb w* = *bin-nth* (*uint w*) (*len-of TYPE*(*'a*) − *1*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-msb-alt*: *msb w* = *hd* (*to-bl w*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-set-nth* [*simp*]: *set-bit w n* (*test-bit w n*) = *w*
for *w* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *bin-nth-uint'*: *bin-nth* (*uint w*) *n* ⟷ *rev* (*bin-to-bl* (*size w*) (*uint w*)) ! *n* ∧ *n* < *size w*
⟨*proof*⟩

lemmas *bin-nth-uint* = *bin-nth-uint'* [*unfolded word-size*]

lemma *test-bit-bl*: *w* !! *n* ⟷ *rev* (*to-bl w*) ! *n* ∧ *n* < *size w*
⟨*proof*⟩

lemma *to-bl-nth*: *n* < *size w* ⟹ *to-bl w* ! *n* = *w* !! (*size w* − *Suc n*)
⟨*proof*⟩

lemma *test-bit-set*: (*set-bit w n x*) !! *n* ⟷ *n* < *size w* ∧ *x*
for *w* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *test-bit-set-gen*:
test-bit (*set-bit w n x*) *m* = (*if m* = *n then n* < *size w* ∧ *x else test-bit w m*)
for *w* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *of-bl-rep-False*: *of-bl* (*replicate n False* @ *bs*) = *of-bl bs*
⟨*proof*⟩

lemma *msb-nth*: *msb w* = *w* !! (*len-of TYPE*(*'a*) − *1*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemmas *msb0* = *len-gt-0* [*THEN diff-Suc-less*, *THEN word-ops-nth-size* [*unfolded word-size*]]
lemmas *msb1* = *msb0* [**where** *i* = *0*]

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*]:
  $n = size\ w \Longrightarrow ofn = set\text{-}bits \Longrightarrow [w, ofn\ g] = l \Longrightarrow$
    td-ext test-bit ofn $\{f.\ \forall i.\ f\ i \longrightarrow i < n\}$ $(\lambda h\ i.\ h\ i \wedge i < n)$
  for $w :: 'a::len0\ word$
  ⟨*proof*⟩

interpretation *test-bit*:
  td-ext
    op !! :: $'a::len0\ word \Rightarrow nat \Rightarrow bool$
    set-bits
    $\{f.\ \forall i.\ f\ i \longrightarrow i < len\text{-}of\ TYPE('a::len0)\}$
    $(\lambda h\ i.\ h\ i \wedge i < len\text{-}of\ TYPE('a::len0))$
  ⟨*proof*⟩

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 $w :: 'a::len0\ word$
  ⟨*proof*⟩

lemma *word-set-set-diff*:
  fixes $w :: 'a::len0\ word$
  assumes $m \neq n$
  shows *set-bit* (*set-bit w m x*) *n y* = *set-bit* (*set-bit w n y*) *m x*
  ⟨*proof*⟩

lemma *nth-sint*:
  fixes $w :: 'a::len\ word$
  defines $l \equiv len\text{-}of\ TYPE('a)$
  shows *bin-nth* (*sint w*) *n* = (*if n* < *l* − *1 then w* !! *n else w* !! (*l* − *1*))
  ⟨*proof*⟩

lemma *word-lsb-numeral* [*simp*]:
  lsb (*numeral bin* :: $'a::len\ word$) ⟷ *bin-last* (*numeral bin*)
  ⟨*proof*⟩

lemma *word-lsb-neg-numeral* [*simp*]:
  lsb (− *numeral bin* :: $'a::len\ word$) ⟷ *bin-last* (− *numeral bin*)
  ⟨*proof*⟩

lemma *set-bit-word-of-int*: *set-bit* (*word-of-int x*) *n b* = *word-of-int* (*bin-sc n b x*)
  ⟨*proof*⟩

lemma *word-set-numeral* [*simp*]:

set-bit (*numeral bin*::*'a*::*len0 word*) *n b* =
word-of-int (*bin-sc n b* (*numeral bin*))
⟨*proof*⟩

lemma *word-set-neg-numeral* [*simp*]:
set-bit (− *numeral bin*::*'a*::*len0 word*) *n b* =
word-of-int (*bin-sc n b* (− *numeral bin*))
⟨*proof*⟩

lemma *word-set-bit-0* [*simp*]: *set-bit 0 n b* = *word-of-int* (*bin-sc n b 0*)
⟨*proof*⟩

lemma *word-set-bit-1* [*simp*]: *set-bit 1 n b* = *word-of-int* (*bin-sc n b 1*)
⟨*proof*⟩

lemma *setBit-no* [*simp*]: *setBit* (*numeral bin*) *n* = *word-of-int* (*bin-sc n True* (*numeral bin*))
⟨*proof*⟩

lemma *clearBit-no* [*simp*]:
clearBit (*numeral bin*) *n* = *word-of-int* (*bin-sc n False* (*numeral bin*))
⟨*proof*⟩

lemma *to-bl-n1*: *to-bl* (−*1*::*'a*::*len0 word*) = *replicate* (*len-of TYPE*(*'a*)) *True*
⟨*proof*⟩

lemma *word-msb-n1* [*simp*]: *msb* (−*1*::*'a*::*len word*)
⟨*proof*⟩

lemma *word-set-nth-iff*: *set-bit w n b* = *w* ⟷ *w* !! *n* = *b* ∨ *n* ≥ *size w*
for *w* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *test-bit-2p*: (*word-of-int* (*2* ^ *n*)::*'a*::*len word*) !! *m* ⟷ *m* = *n* ∧ *m* < *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *nth-w2p*: ((*2*::*'a*::*len word*) ^ *n*) !! *m* ⟷ *m* = *n* ∧ *m* < *len-of TYPE*(*'a*::*len*)
⟨*proof*⟩

lemma *uint-2p*: (*0*::*'a*::*len word*) < *2* ^ *n* ⟹ *uint* (*2* ^ *n*::*'a*::*len word*) = *2* ^ *n*
⟨*proof*⟩

lemma *word-of-int-2p*: (*word-of-int* (*2* ^ *n*) :: *'a*::*len word*) = *2* ^ *n*
⟨*proof*⟩

lemma *bang-is-le*: *x* !! *m* ⟹ *2* ^ *m* ≤ *x*
for *x* :: *'a*::*len word*
⟨*proof*⟩

lemma *word-clr-le*: $w \geq$ *set-bit w n False*
for w :: *'a::len0 word*
⟨*proof*⟩

lemma *word-set-ge*: $w \leq$ *set-bit w n True*
for w :: *'a::len word*
⟨*proof*⟩

10.25 Shifting, Rotating, and Splitting Words

lemma *shiftl1-wi* [*simp*]: *shiftl1* (*word-of-int w*) = *word-of-int* (*w BIT False*)
⟨*proof*⟩

lemma *shiftl1-numeral* [*simp*]: *shiftl1* (*numeral w*) = *numeral* (*Num.Bit0 w*)
⟨*proof*⟩

lemma *shiftl1-neg-numeral* [*simp*]: *shiftl1* (− *numeral w*) = − *numeral* (*Num.Bit0 w*)
⟨*proof*⟩

lemma *shiftl1-0* [*simp*] : *shiftl1 0* = *0*
⟨*proof*⟩

lemma *shiftl1-def-u*: *shiftl1 w* = *word-of-int* (*uint w BIT False*)
⟨*proof*⟩

lemma *shiftl1-def-s*: *shiftl1 w* = *word-of-int* (*sint w BIT False*)
⟨*proof*⟩

lemma *shiftr1-0* [*simp*]: *shiftr1 0* = *0*
⟨*proof*⟩

lemma *sshiftr1-0* [*simp*]: *sshiftr1 0* = *0*
⟨*proof*⟩

lemma *sshiftr1-n1* [*simp*]: *sshiftr1* (− *1*) = − *1*
⟨*proof*⟩

lemma *shiftl-0* [*simp*]: (*0*::*'a::len0 word*) << *n* = *0*
⟨*proof*⟩

lemma *shiftr-0* [*simp*]: (*0*::*'a::len0 word*) >> *n* = *0*
⟨*proof*⟩

lemma *sshiftr-0* [*simp*]: *0* >>> *n* = *0*
⟨*proof*⟩

lemma *sshiftr-n1* [*simp*]: *−1* >>> *n* = *−1*

⟨*proof*⟩

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

lemma *nth-shiftl'*: (*w* << *m*) !! *n* ⟷ *n* < *size w* ∧ *n* >= *m* ∧ *w* !! (*n* − *m*)
for *w* :: *'a*::*len0 word*
⟨*proof*⟩

lemmas *nth-shiftl* = *nth-shiftl'* [*unfolded word-size*]

lemma *nth-shiftr1*: *shiftr1 w* !! *n* = *w* !! *Suc n*
⟨*proof*⟩

lemma *nth-shiftr*: (*w* >> *m*) !! *n* = *w* !! (*n* + *m*)
for *w* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *uint-shiftr1*: *uint* (*shiftr1 w*) = *bin-rest* (*uint w*)
⟨*proof*⟩

lemma *nth-sshiftr1*: *sshiftr1 w* !! *n* = (*if n* = *size w* − *1 then w* !! *n else w* !! *Suc n*)
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *shiftr1-div-2*: *uint* (*shiftr1 w*) = *uint w div 2*
⟨*proof*⟩

lemma *sshiftr1-div-2*: *sint* (*sshiftr1 w*) = *sint w div 2*
⟨*proof*⟩

lemma *shiftr-div-2n*: *uint* (*shiftr w n*) = *uint w div 2* ^ *n*
⟨*proof*⟩

lemma *sshiftr-div-2n*: *sint* (*sshiftr w n*) = *sint w div 2* ^ *n*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *shiftl1-of-bl*: *shiftl1* (*of-bl bl*) = *of-bl* (*bl* @ [*False*])
⟨*proof*⟩

lemma *shiftl1-bl*: *shiftl1 w* = *of-bl* (*to-bl w* @ [*False*])
for *w* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *bl-shiftl1*: *to-bl* (*shiftl1 w*) = *tl* (*to-bl w*) @ [*False*]
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *bl-shiftl1'*: *to-bl* (*shiftl1 w*) = *tl* (*to-bl w* @ [*False*])
⟨*proof*⟩

lemma *shiftr1-bl*: *shiftr1 w* = *of-bl* (*butlast* (*to-bl w*))
⟨*proof*⟩

lemma *bl-shiftr1*: *to-bl* (*shiftr1 w*) = *False* # *butlast* (*to-bl w*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *bl-shiftr1'*: *to-bl* (*shiftr1 w*) = *butlast* (*False* # *to-bl w*)
⟨*proof*⟩

lemma *shiftl1-rev*: *shiftl1 w* = *word-reverse* (*shiftr1* (*word-reverse w*))
⟨*proof*⟩

lemma *shiftl-rev*: *shiftl w n* = *word-reverse* (*shiftr* (*word-reverse w*) *n*)
⟨*proof*⟩

lemma *rev-shiftl*: *word-reverse w* << *n* = *word-reverse* (*w* >> *n*)
⟨*proof*⟩

lemma *shiftr-rev*: *w* >> *n* = *word-reverse* (*word-reverse w* << *n*)
⟨*proof*⟩

lemma *rev-shiftr*: *word-reverse w* >> *n* = *word-reverse* (*w* << *n*)
⟨*proof*⟩

lemma *bl-sshiftr1*: *to-bl* (*sshiftr1 w*) = *hd* (*to-bl w*) # *butlast* (*to-bl w*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *drop-shiftr*: *drop n* (*to-bl* (*w* >> *n*)) = *take* (*size w* − *n*) (*to-bl w*)

for w :: $'a$::*len word*
⟨*proof*⟩

lemma *drop-sshiftr*: *drop n* (*to-bl* (*w* >>> *n*)) = *take* (*size w* − *n*) (*to-bl w*)
for w :: $'a$::*len word*
⟨*proof*⟩

lemma *take-shiftr*: *n* ≤ *size w* ⟹ *take n* (*to-bl* (*w* >> *n*)) = *replicate n False*
⟨*proof*⟩

lemma *take-sshiftr'* [*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*
⟨*proof*⟩

lemmas *hd-sshiftr* = *take-sshiftr'* [*THEN conjunct1*]
lemmas *take-sshiftr* = *take-sshiftr'* [*THEN conjunct2*]

lemma *atd-lem*: *take n xs* = *t* ⟹ *drop n xs* = *d* ⟹ *xs* = *t* @ *d*
⟨*proof*⟩

lemmas *bl-shiftr* = *atd-lem* [*OF take-shiftr drop-shiftr*]
lemmas *bl-sshiftr* = *atd-lem* [*OF take-sshiftr drop-sshiftr*]

lemma *shiftl-of-bl*: *of-bl bl* << *n* = *of-bl* (*bl* @ *replicate n False*)
⟨*proof*⟩

lemma *shiftl-bl*: *w* << *n* = *of-bl* (*to-bl w* @ *replicate n False*)
for w :: $'a$::*len0 word*
⟨*proof*⟩

lemmas *shiftl-numeral* [*simp*] = *shiftl-def* [**where** *w*=*numeral w*] **for** *w*

lemma *bl-shiftl*: *to-bl* (*w* << *n*) = *drop n* (*to-bl w*) @ *replicate* (*min* (*size w*) *n*) *False*
⟨*proof*⟩

lemma *shiftl-zero-size*: *size x* ≤ *n* ⟹ *x* << *n* = *0*
for x :: $'a$::*len0 word*
⟨*proof*⟩

lemma *shiftl1-2t*: *shiftl1 w* = *2* ∗ *w*
for w :: $'a$::*len word*
⟨*proof*⟩

lemma *shiftl1-p*: *shiftl1 w* = *w* + *w*

for $w :: 'a::len\ word$
⟨*proof*⟩

lemma *shiftl-t2n*: $shiftl\ w\ n = 2\ \hat{}\ n * w$
for $w :: 'a::len\ word$
⟨*proof*⟩

lemma *shiftr1-bintr* [*simp*]:
$(shiftr1\ (numeral\ w) :: 'a::len0\ word) =$
$word\text{-}of\text{-}int\ (bin\text{-}rest\ (bintrunc\ (len\text{-}of\ TYPE('a))\ (numeral\ w)))$
⟨*proof*⟩

lemma *sshiftr1-sbintr* [*simp*]:
$(sshiftr1\ (numeral\ w) :: 'a::len\ word) =$
$word\text{-}of\text{-}int\ (bin\text{-}rest\ (sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ (numeral\ w)))$
⟨*proof*⟩

lemma *shiftr-no* [*simp*]:

$(numeral\ w::'a::len0\ word) >>\ n = word\text{-}of\text{-}int$
$((bin\text{-}rest\ \hat{}\hat{}\ n)\ (bintrunc\ (len\text{-}of\ TYPE('a))\ (numeral\ w)))$
⟨*proof*⟩

lemma *sshiftr-no* [*simp*]:

$(numeral\ w::'a::len\ word) >>>\ n = word\text{-}of\text{-}int$
$((bin\text{-}rest\ \hat{}\hat{}\ n)\ (sbintrunc\ (len\text{-}of\ TYPE('a) - 1)\ (numeral\ w)))$
⟨*proof*⟩

lemma *shiftr1-bl-of*:
$length\ bl \leq len\text{-}of\ TYPE('a) \Longrightarrow$
$shiftr1\ (of\text{-}bl\ bl::'a::len0\ word) = of\text{-}bl\ (butlast\ bl)$
⟨*proof*⟩

lemma *shiftr-bl-of*:
$length\ bl \leq len\text{-}of\ TYPE('a) \Longrightarrow$
$(of\text{-}bl\ bl::'a::len0\ word) >>\ n = of\text{-}bl\ (take\ (length\ bl - n)\ bl)$
⟨*proof*⟩

lemma *shiftr-bl*: $x >>\ n \equiv of\text{-}bl\ (take\ (len\text{-}of\ TYPE('a) - n)\ (to\text{-}bl\ x))$
for $x :: 'a::len0\ word$
⟨*proof*⟩

lemma *msb-shift*: $msb\ w \longleftrightarrow w >>\ (len\text{-}of\ TYPE('a) - 1) \neq 0$
for $w :: 'a::len\ word$
⟨*proof*⟩

lemma *zip-replicate*: $n \geq length\ ys \Longrightarrow zip\ (replicate\ n\ x)\ ys = map\ (\lambda y.\ (x,\ y))\ ys$

⟨*proof*⟩

lemma *align-lem-or* [*rule-format*] :
∀ *x m*. *length x* = *n* + *m* ⟶ *length y* = *n* + *m* ⟶
drop m x = *replicate n False* ⟶ *take m y* = *replicate m False* ⟶
map2 op | *x y* = *take m x* @ *drop m y*
⟨*proof*⟩

lemma *align-lem-and* [*rule-format*] :
∀ *x m*. *length x* = *n* + *m* ⟶ *length y* = *n* + *m* ⟶
drop m x = *replicate n False* ⟶ *take m y* = *replicate m False* ⟶
map2 op ∧ *x y* = *replicate* (*n* + *m*) *False*
⟨*proof*⟩

lemma *aligned-bl-add-size* [*OF refl*]:
size x − *n* = *m* ⟹ *n* ≤ *size x* ⟹ *drop m* (*to-bl x*) = *replicate n False* ⟹
take m (*to-bl y*) = *replicate m False* ⟹
to-bl (*x* + *y*) = *take m* (*to-bl x*) @ *drop m* (*to-bl y*)
⟨*proof*⟩

10.25.2 Mask

lemma *nth-mask* [*OF refl*, *simp*]: *m* = *mask n* ⟹ *test-bit m i* ⟷ *i* < *n* ∧ *i* < *size m*
⟨*proof*⟩

lemma *mask-bl*: *mask n* = *of-bl* (*replicate n True*)
⟨*proof*⟩

lemma *mask-bin*: *mask n* = *word-of-int* (*bintrunc n* (− *1*))
⟨*proof*⟩

lemma *and-mask-bintr*: *w AND mask n* = *word-of-int* (*bintrunc n* (*uint w*))
⟨*proof*⟩

lemma *and-mask-wi*: *word-of-int i AND mask n* = *word-of-int* (*bintrunc n i*)
⟨*proof*⟩

lemma *and-mask-wi'*:
word-of-int i AND mask n = (*word-of-int* (*bintrunc* (*min LENGTH*('*a*) *n*) *i*) :: '*a*::*len word*)
⟨*proof*⟩

lemma *and-mask-no*: *numeral i AND mask n* = *word-of-int* (*bintrunc n* (*numeral i*))
⟨*proof*⟩

lemma *bl-and-mask'*:
to-bl (*w AND mask n* :: '*a*::*len word*) =

replicate (*len-of TYPE*('*a*) − *n*) *False* @
drop (*len-of TYPE*('*a*) − *n*) (*to-bl w*)
⟨*proof*⟩

lemma *and-mask-mod-2p*: *w AND mask n* = *word-of-int* (*uint w mod 2* ^ *n*)
⟨*proof*⟩

lemma *and-mask-lt-2p*: *uint* (*w AND mask n*) < *2* ^ *n*
⟨*proof*⟩

lemma *eq-mod-iff*: *0* < *n* ⟹ *b* = *b mod n* ⟷ *0* ≤ *b* ∧ *b* < *n*
for *b n* :: *int*
⟨*proof*⟩

lemma *mask-eq-iff*: *w AND mask n* = *w* ⟷ *uint w* < *2* ^ *n*
⟨*proof*⟩

lemma *and-mask-dvd*: *2* ^ *n dvd uint w* ⟷ *w AND mask n* = *0*
⟨*proof*⟩

lemma *and-mask-dvd-nat*: *2* ^ *n dvd unat w* ⟷ *w AND mask n* = *0*
⟨*proof*⟩

lemma *word-2p-lem*: *n* < *size w* ⟹ *w* < *2* ^ *n* = (*uint w* < *2* ^ *n*)
for *w* :: '*a*::*len word*
⟨*proof*⟩

lemma *less-mask-eq*: *x* < *2* ^ *n* ⟹ *x AND mask n* = *x*
for *x* :: '*a*::*len word*
⟨*proof*⟩

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* ⟹ *x AND mask n* < *2*^*n*
⟨*proof*⟩

lemma *word-mod-2p-is-mask* [*OF refl*]: *c* = *2* ^ *n* ⟹ *c* > *0* ⟹ *x mod c* = *x AND mask n*
for *c x* :: '*a*::*len word*
⟨*proof*⟩

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*

$(b\ AND\ mask\ n) * a\ AND\ mask\ n = b * a\ AND\ mask\ n$
$(a\ AND\ mask\ n) + (b\ AND\ mask\ n)\ AND\ mask\ n = a + b\ AND\ mask\ n$
$(a\ AND\ mask\ n) - (b\ AND\ mask\ n)\ AND\ mask\ n = a - b\ AND\ mask\ n$
$(a\ AND\ mask\ n) * (b\ AND\ mask\ n)\ AND\ mask\ n = a * b\ AND\ mask\ n$
$-(a\ AND\ mask\ n)\ AND\ mask\ n = -a\ AND\ mask\ n$
$word\text{-}succ\ (a\ AND\ mask\ n)\ AND\ mask\ n = word\text{-}succ\ a\ AND\ mask\ n$
$word\text{-}pred\ (a\ AND\ mask\ n)\ AND\ mask\ n = word\text{-}pred\ a\ AND\ mask\ n$
⟨*proof*⟩

lemma *mask-power-eq*: $(x\ AND\ mask\ n)\ \hat{}\ k\ AND\ mask\ n = x\ \hat{}\ k\ AND\ mask\ n$
⟨*proof*⟩

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*:
$to\text{-}bl\ (revcast\ w :: 'a{::}len0\ word) = takefill\ False\ (len\text{-}of\ TYPE('a))\ (to\text{-}bl\ w)$
⟨*proof*⟩

lemma *revcast-rev-ucast* [*OF refl refl refl*]:
$cs = [rc, uc] \Longrightarrow rc = revcast\ (word\text{-}reverse\ w) \Longrightarrow uc = ucast\ w \Longrightarrow$
$rc = word\text{-}reverse\ uc$
⟨*proof*⟩

lemma *revcast-ucast*: $revcast\ w = word\text{-}reverse\ (ucast\ (word\text{-}reverse\ w))$
⟨*proof*⟩

lemma *ucast-revcast*: $ucast\ w = word\text{-}reverse\ (revcast\ (word\text{-}reverse\ w))$
⟨*proof*⟩

lemma *ucast-rev-revcast*: $ucast\ (word\text{-}reverse\ w) = word\text{-}reverse\ (revcast\ w)$
⟨*proof*⟩

linking revcast and cast via shift

lemmas *wsst-TYs* = *source-size target-size word-size*

lemma *revcast-down-uu* [*OF refl*]:
$rc = revcast \Longrightarrow source\text{-}size\ rc = target\text{-}size\ rc + n \Longrightarrow rc\ w = ucast\ (w >> n)$
for $w :: 'a{::}len\ word$
⟨*proof*⟩

lemma *revcast-down-us* [*OF refl*]:
$rc = revcast \Longrightarrow source\text{-}size\ rc = target\text{-}size\ rc + n \Longrightarrow rc\ w = ucast\ (w >>> n)$
for $w :: 'a{::}len\ word$

⟨*proof*⟩

lemma *revcast-down-su* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹ *rc w* = *scast* (*w* >> *n*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *revcast-down-ss* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* = *target-size rc* + *n* ⟹ *rc w* = *scast* (*w* >>> *n*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *cast-down-rev*:
uc = *ucast* ⟹ *source-size uc* = *target-size uc* + *n* ⟹ *uc w* = *revcast* (*w* << *n*)
for *w* :: *'a*::*len word*
⟨*proof*⟩

lemma *revcast-up* [*OF refl*]:
rc = *revcast* ⟹ *source-size rc* + *n* = *target-size rc* ⟹
rc w = (*ucast w* :: *'a*::*len word*) << *n*
⟨*proof*⟩

lemmas *rc1* = *revcast-up* [*THEN*
revcast-rev-ucast [*symmetric*, *THEN trans*, *THEN word-rev-gal*, *symmetric*]]
lemmas *rc2* = *revcast-down-uu* [*THEN*
revcast-rev-ucast [*symmetric*, *THEN trans*, *THEN word-rev-gal*, *symmetric*]]

lemmas *ucast-up* =
rc1 [*simplified rev-shiftr* [*symmetric*] *revcast-ucast* [*symmetric*]]
lemmas *ucast-down* =
rc2 [*simplified rev-shiftr revcast-ucast* [*symmetric*]]

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *slice1-0* [*simp*] : *slice1 n 0* = *0*

⟨*proof*⟩

lemma *slice-0* [*simp*] : *slice n 0 = 0*
⟨*proof*⟩

lemma *slice-take'*: *slice n w = of-bl* (*take* (*size w − n*) (*to-bl w*))
⟨*proof*⟩

lemmas *slice-take = slice-take'* [*unfolded word-size*]

— shiftr to a word of the same size is just slice, slice is just shiftr then ucast
lemmas *shiftr-slice = trans* [*OF shiftr-bl* [*THEN meta-eq-to-obj-eq*] *slice-take* [*symmetric*]]

lemma *slice-shiftr*: *slice n w = ucast* (*w >> n*)
⟨*proof*⟩

lemma *nth-slice*: (*slice n w* :: *'a*::*len0 word*) !! *m* = (*w* !! (*m + n*) ∧ *m < len-of TYPE*(*'a*))
⟨*proof*⟩

lemma *slice1-down-alt'*:
sl = slice1 n w ⟹ *fs = size sl* ⟹ *fs + k = n* ⟹
to-bl sl = takefill False fs (*drop k* (*to-bl w*))
⟨*proof*⟩

lemma *slice1-up-alt'*:
sl = slice1 n w ⟹ *fs = size sl* ⟹ *fs = n + k* ⟹
to-bl sl = takefill False fs (*replicate k False* @ (*to-bl w*))
⟨*proof*⟩

lemmas *sd1 = slice1-down-alt'* [*OF refl refl*, *unfolded word-size*]
lemmas *su1 = slice1-up-alt'* [*OF refl refl*, *unfolded word-size*]
lemmas *slice1-down-alt = le-add-diff-inverse* [*THEN sd1*]
lemmas *slice1-up-alts =*
le-add-diff-inverse [*symmetric*, *THEN su1*]
le-add-diff-inverse2 [*symmetric*, *THEN su1*]

lemma *ucast-slice1*: *ucast w = slice1* (*size w*) *w*
⟨*proof*⟩

lemma *ucast-slice*: *ucast w = slice 0 w*
⟨*proof*⟩

lemma *slice-id*: *slice 0 t = t*
⟨*proof*⟩

lemma *revcast-slice1* [*OF refl*]: *rc = revcast w* ⟹ *slice1* (*size rc*) *w = rc*
⟨*proof*⟩

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*))))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*)))
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *word-cat-bl*: *word-cat a b = of-bl (to-bl a @ to-bl b)*
⟨*proof*⟩

lemma *of-bl-append*:
*(of-bl (xs @ ys) :: 'a::len word) = of-bl xs * 2^(length ys) + of-bl ys*
⟨*proof*⟩

lemma *of-bl-False* [*simp*]: *of-bl (False#xs) = of-bl xs*
⟨*proof*⟩

lemma *of-bl-True* [*simp*]: *(of-bl (True # xs) :: 'a::len word) = 2^length xs + of-bl xs*
⟨*proof*⟩

lemma *of-bl-Cons*: *of-bl (x#xs) = of-bool x * 2^length xs + of-bl xs*
⟨*proof*⟩

lemma *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*
⟨*proof*⟩

lemma *word-split-bl'*:
std = size c − size b ⟹ (word-split c = (a, b)) ⟹
(a = of-bl (take std (to-bl c)) ∧ b = of-bl (drop std (to-bl c)))
⟨*proof*⟩

lemma *word-split-bl*: *std = size c − size b ⟹*
(a = of-bl (take std (to-bl c)) ∧ b = of-bl (drop std (to-bl c))) ⟷
word-split c = (a, b)
⟨*proof*⟩

lemma *word-split-bl-eq*:
(word-split c :: ('c::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*
⟨*proof*⟩
lemma *test-bit-split'*:
word-split c = (a, b) ⟶
(∀ n m.
b !! n = (n < size b ∧ c !! n) ∧
a !! m = (m < size a ∧ c !! (m + size b)))
⟨*proof*⟩

lemma *test-bit-split*:
word-split c = (a, b) ⟹
(∀ n::nat. b !! n ⟷ n < size b ∧ c !! n) ∧
(∀ m::nat. a !! m ⟷ m < size a ∧ c !! (m + size b))

⟨*proof*⟩

lemma *test-bit-split-eq*:
word-split c = (a, b) ⟷
((∀ n::nat. b !! n = (n < size b ∧ c !! n)) ∧
(∀ m::nat. a !! m = (m < size a ∧ c !! (m + size b))))
⟨*proof*⟩

lemma *word-cat-id*: *word-cat a b = b*
⟨*proof*⟩
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))
⟨*proof*⟩

lemma *word-cat-split-alt*: *size w ≤ size u + size v ⟹ word-split w = (u, v) ⟹ word-cat u v = w*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *slice-cat1* [*OF refl*]:
wc = word-cat a b ⟹ size wc >= size a + size b ⟹ slice (size b) wc = a
⟨*proof*⟩

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
⟨*proof*⟩

lemma *word-split-cat-alt*:
w = word-cat u v ⟹ size u + size v ≤ size w ⟹ word-split w = (u, v)
⟨*proof*⟩

lemmas *word-cat-bl-no-bin* [*simp*] =
word-cat-bl [**where** *a=numeral a* **and** *b=numeral b*, *unfolded to-bl-numeral*]
for *a b*

lemmas *word-split-bl-no-bin* [*simp*] =
word-split-bl-eq [**where** *c=numeral c*, *unfolded to-bl-numeral*] **for** *c*

This odd result arises from the fact that the statement of the result implies

that the decoded words are of the same type, and therefore of the same length, as the original word.

lemma *word-rsplit-same*: *word-rsplit w = [w]*
⟨*proof*⟩

lemma *word-rsplit-empty-iff-size*: *word-rsplit w = [] ⟷ size w = 0*
⟨*proof*⟩

lemma *test-bit-rsplit*:
sw = word-rsplit w ⟹ m < size (hd sw) ⟹
*k < length sw ⟹ (rev sw ! k) !! m = w !! (k * size (hd sw) + m)*
for *sw :: 'a::len word list*
⟨*proof*⟩

lemma *word-rcat-bl*: *word-rcat wl = of-bl (concat (map to-bl wl))*
⟨*proof*⟩

lemma *size-rcat-lem'*: *size (concat (map to-bl wl)) = length wl * size (hd wl)*
⟨*proof*⟩

lemmas *size-rcat-lem = size-rcat-lem' [unfolded word-size]*

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))
⟨*proof*⟩

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

lemma *foldl-eq-foldr*: *foldl op + x xs = foldr op + (x # xs) 0*
for *x :: 'a::comm-monoid-add*
⟨*proof*⟩

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
⟨*proof*⟩

lemma *length-word-rsplit-size*:
n = len-of TYPE('a::len) ⟹
*length (word-rsplit w :: 'a word list) ≤ m ⟷ size w ≤ m * n*
⟨*proof*⟩

lemmas *length-word-rsplit-lt-size =*
length-word-rsplit-size [unfolded Not-eq-iff linorder-not-less [symmetric]]

lemma *length-word-rsplit-exp-size*:
n = len-of TYPE('a::len) ⟹
length (word-rsplit w :: 'a word list) = (size w + n − 1) div n
⟨*proof*⟩

lemma *length-word-rsplit-even-size*:
*n = len-of TYPE('a::len) ⟹ size w = m * n ⟹*
length (word-rsplit w :: 'a word list) = m
⟨*proof*⟩

lemmas *length-word-rsplit-exp-size' = refl [THEN length-word-rsplit-exp-size]*

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *msrevs*:
*0 < n ⟹ (k * n + m) div n = m div n + k*
*(k * n + m) mod n = m mod n*
for *n :: nat*
⟨*proof*⟩

lemma *word-rsplit-rcat-size [OF refl]*:
word-rcat ws = frcw ⟹
*size frcw = length ws * len-of TYPE('a) ⟹ word-rsplit frcw = ws*
for *ws :: 'a::len word list*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *rotate1-rl* [*simp*] : *rotater1* (*rotate1 l*) = *l*
⟨*proof*⟩

lemma *rotate1-lr* [*simp*] : *rotate1* (*rotater1 l*) = *l*
⟨*proof*⟩

lemma *rotater1-rev'*: *rotater1* (*rev xs*) = *rev* (*rotate1 xs*)
⟨*proof*⟩

lemma *rotater-rev'*: *rotater n* (*rev xs*) = *rev* (*rotate n xs*)
⟨*proof*⟩

lemma *rotater-rev*: *rotater n ys* = *rev* (*rotate n* (*rev ys*))
⟨*proof*⟩

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

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

lemma *rotate-inv-plus* [*rule-format*] :
∀ *k*. *k* = *m* + *n* ⟶ *rotater k* (*rotate n xs*) = *rotater m xs* ∧
rotate k (*rotater n xs*) = *rotate m xs* ∧
rotater n (*rotate k xs*) = *rotate m xs* ∧

rotate n (rotater k xs) = rotater m xs
⟨*proof*⟩

lemmas *rotate-inv-rel = le-add-diff-inverse2* [*symmetric*, *THEN rotate-inv-plus*]

lemmas *rotate-inv-eq = order-refl* [*THEN rotate-inv-rel*, *simplified*]

lemmas *rotate-lr* [*simp*] = *rotate-inv-eq* [*THEN conjunct1*]
lemmas *rotate-rl* [*simp*] = *rotate-inv-eq* [*THEN conjunct2*, *THEN conjunct1*]

lemma *rotate-gal*: *rotater n xs = ys ⟷ rotate n ys = xs*
⟨*proof*⟩

lemma *rotate-gal'*: *ys = rotater n xs ⟷ xs = rotate n ys*
⟨*proof*⟩

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

lemma *restrict-to-left*: *x = y ⟹ x = z ⟷ y = z*
⟨*proof*⟩

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 ≠ [] ⟹ butlast (map f xs) = map f (butlast xs)*
⟨*proof*⟩

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

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

lemma *but-last-zip* [*rule-format*] :
∀ *ys*. *length xs = length ys ⟶ xs ≠ [] ⟶*
last (zip xs ys) = (last xs, last ys) ∧
butlast (zip xs ys) = zip (butlast xs) (butlast ys)
⟨*proof*⟩

lemma *but-last-map2* [*rule-format*] :
∀ *ys*. *length xs = length ys ⟶ xs ≠ [] ⟶*
last (map2 f xs ys) = f (last xs) (last ys) ∧

butlast (*map2 f xs ys*) = *map2 f* (*butlast xs*) (*butlast ys*)
⟨*proof*⟩

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

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

lemmas *lrth* =
box-equals [*OF asm-rl length-rotater* [*symmetric*]
length-rotater [*symmetric*],
THEN rotater1-map2]

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

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

lemmas *lth* = *box-equals* [*OF asm-rl length-rotate* [*symmetric*]
length-rotate [*symmetric*], *THEN rotate1-map2*]

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

lemma *to-bl-rotl*: *to-bl* (*word-rotl n w*) = *rotate n* (*to-bl w*)
⟨*proof*⟩

lemmas *blrs0* = *rotate-eqs* [*THEN to-bl-rotl* [*THEN trans*]]

lemmas *word-rotl-eqs* =
blrs0 [*simplified word-bl-Rep′ word-bl.Rep-inject to-bl-rotl* [*symmetric*]]

lemma *to-bl-rotr*: *to-bl* (*word-rotr n w*) = *rotater n* (*to-bl w*)
⟨*proof*⟩

lemmas *brrs0* = *rotater-eqs* [*THEN to-bl-rotr* [*THEN trans*]]

lemmas *word-rotr-eqs* =
brrs0 [*simplified word-bl-Rep' word-bl.Rep-inject to-bl-rotr* [*symmetric*]]

declare *word-rotr-eqs* (*1*) [*simp*]
declare *word-rotl-eqs* (*1*) [*simp*]

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*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-rotr-rev*: *word-rotr n w* = *word-reverse* (*word-rotl n* (*word-reverse w*))
⟨*proof*⟩

lemma *word-roti-0* [*simp*]: *word-roti 0 w* = *w*
⟨*proof*⟩

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*⟩

lemma *word-roti-conv-mod'*: *word-roti n w* = *word-roti* (*n mod int* (*size w*)) *w*
⟨*proof*⟩

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
⟨*proof*⟩
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)
⟨*proof*⟩

lemmas *bl-word-roti-dt = bl-word-roti-dt'* [*unfolded word-size*]

lemmas *word-rotl-dt = bl-word-rotl-dt* [*THEN word-bl.Rep-inverse'* [*symmetric*]]
lemmas *word-rotr-dt = bl-word-rotr-dt* [*THEN word-bl.Rep-inverse'* [*symmetric*]]
lemmas *word-roti-dt = bl-word-roti-dt* [*THEN word-bl.Rep-inverse'* [*symmetric*]]

lemma *word-rotx-0* [*simp*] : *word-rotr i 0 = 0 ∧ word-rotl i 0 = 0*
⟨*proof*⟩

lemma *word-roti-0'* [*simp*] : *word-roti n 0 = 0*
⟨*proof*⟩

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** $x = word\text{-}of\text{-}int\ n$ **and** $0 \leq n$ **and** $n < 2\hat{}len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemma *word-nat-cases* [*cases type*: *word*]:
fixes $x :: 'a::len\ word$
obtains n **where** $x = of\text{-}nat\ n$ **and** $n < 2\hat{}len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemma *max-word-eq*: $(max\text{-}word::'a::len\ word) = 2\hat{}len\text{-}of\ TYPE('a) - 1$
⟨*proof*⟩

lemma *max-word-max* [*simp*,*intro*!]: $n \leq max\text{-}word$
⟨*proof*⟩

lemma *word-of-int-2p-len*: $word\text{-}of\text{-}int\ (2\ \hat{}\ len\text{-}of\ TYPE('a)) = (0::'a::len0\ word)$
⟨*proof*⟩

lemma *word-pow-0*: $(2::'a::len\ word)\ \hat{}\ len\text{-}of\ TYPE('a) = 0$
⟨*proof*⟩

lemma *max-word-wrap*: $x + 1 = 0 \Longrightarrow x = max\text{-}word$
⟨*proof*⟩

lemma *max-word-minus*: $max\text{-}word = (-1::'a::len\ word)$
⟨*proof*⟩

lemma *max-word-bl* [*simp*]: $to\text{-}bl\ (max\text{-}word::'a::len\ word) = replicate\ (len\text{-}of\ TYPE('a))\ True$
⟨*proof*⟩

lemma *max-test-bit* [*simp*]: $(max\text{-}word::'a::len\ word)\ !!\ n \longleftrightarrow n < len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemma *word-and-max* [*simp*]: $x\ AND\ max\text{-}word = x$
⟨*proof*⟩

lemma *word-or-max* [*simp*]: $x\ OR\ max\text{-}word = max\text{-}word$
⟨*proof*⟩

lemma *word-ao-dist2*: $x\ AND\ (y\ OR\ z) = x\ AND\ y\ OR\ x\ AND\ z$
for $x\ y\ z :: 'a::len0\ word$
⟨*proof*⟩

lemma *word-oa-dist2*: $x\ OR\ y\ AND\ z = (x\ OR\ y)\ AND\ (x\ OR\ z)$
for $x\ y\ z :: 'a::len0\ word$
⟨*proof*⟩

lemma *word-and-not* [*simp*]: $x\ AND\ NOT\ x = 0$

for $x :: 'a::len0\ word$
⟨*proof*⟩

lemma *word-or-not* [*simp*]: $x\ OR\ NOT\ x = max\text{-}word$
⟨*proof*⟩

lemma *word-boolean*: $boolean\ (op\ AND)\ (op\ OR)\ bitNOT\ 0\ max\text{-}word$
⟨*proof*⟩

interpretation *word-bool-alg*: *boolean op AND op OR bitNOT 0 max-word*
⟨*proof*⟩

lemma *word-xor-and-or*: $x\ XOR\ y = x\ AND\ NOT\ y\ OR\ NOT\ x\ AND\ y$
for $x\ y :: 'a::len0\ word$
⟨*proof*⟩

interpretation *word-bool-alg*: *boolean-xor op AND op OR bitNOT 0 max-word op XOR*
⟨*proof*⟩

lemma *shiftr-x-0* [*iff*]: $x >> 0 = x$
for $x :: 'a::len0\ word$
⟨*proof*⟩

lemma *shiftl-x-0* [*simp*]: $x << 0 = x$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *shiftl-1* [*simp*]: $(1::'a::len\ word) << n = 2\hat{}n$
⟨*proof*⟩

lemma *uint-lt-0* [*simp*]: $uint\ x < 0 = False$
⟨*proof*⟩

lemma *shiftr1-1* [*simp*]: $shiftr1\ (1::'a::len\ word) = 0$
⟨*proof*⟩

lemma *shiftr-1*[*simp*]: $(1::'a::len\ word) >> n = (if\ n = 0\ then\ 1\ else\ 0)$
⟨*proof*⟩

lemma *word-less-1* [*simp*]: $x < 1 \longleftrightarrow x = 0$
for $x :: 'a::len\ word$
⟨*proof*⟩

lemma *to-bl-mask*:
$to\text{-}bl\ (mask\ n :: 'a::len\ word) =$
$replicate\ (len\text{-}of\ TYPE('a) - n)\ False\ @$
$\quad replicate\ (min\ (len\text{-}of\ TYPE('a))\ n)\ True$
⟨*proof*⟩

lemma *map-replicate-True*:
n = length xs ⟹
map (λ(x,y). x ∧ y) (zip xs (replicate n True)) = xs
⟨*proof*⟩

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

lemma *bl-and-mask*:
fixes *w :: 'a::len word*
and *n :: nat*
defines *n' ≡ len-of TYPE('a) − n*
shows *to-bl (w AND mask n) = replicate n' False @ drop n' (to-bl w)*
⟨*proof*⟩

lemma *drop-rev-takefill*:
length xs ≤ n ⟹
drop (n − length xs) (rev (takefill False n (rev xs))) = xs
⟨*proof*⟩

lemma *map-nth-0* [*simp*]: *map (op !! (0::'a::len0 word)) xs = replicate (length xs) False*
⟨*proof*⟩

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)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-neq-0-conv*: *w ≠ 0 ⟷ 0 < w*
for *w :: 'a::len word*
⟨*proof*⟩

lemma *max-lt*: *unat (max a b div c) = unat (max a b) div unat c*
for *c :: 'a::len word*
⟨*proof*⟩

lemma *uint-sub-if-size*:
uint $(x - y) =$
$($*if uint* $y \leq$ *uint* x
then uint $x -$ *uint* y
else uint $x -$ *uint* $y + 2\hat{}$*size* $x)$
⟨*proof*⟩

lemma *unat-sub*: $b \leq a \Longrightarrow$ *unat* $(a - b) =$ *unat* $a -$ *unat* b
⟨*proof*⟩

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* $(2\hat{}$*len-of* $TYPE('a) - i) =$ (*word-of-int* $(-i)::'a::$*len word*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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'
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-induct*: $P\ 0 \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (1 + n)) \Longrightarrow P\ m$
for $P :: 'a::$*len word* $\Rightarrow$ *bool*

⟨*proof*⟩

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 $P :: 'b::len\ word \Rightarrow bool$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *word-rec-Suc*: $1 + n \neq 0 \Longrightarrow$ *word-rec z s* $(1 + n) =$ *s n* (*word-rec z s n*)
for $n :: 'a::len\ word$
⟨*proof*⟩

lemma *word-rec-Pred*: $n \neq 0 \Longrightarrow$ *word-rec z s n = s* $(n - 1)$ (*word-rec z s* $(n - 1)$)
⟨*proof*⟩

lemma *word-rec-in*: *f* (*word-rec z* (λ-. *f*) *n*) = *word-rec* (*f z*) (λ-. *f*) *n*
⟨*proof*⟩

lemma *word-rec-in2*: *f n* (*word-rec z f n*) = *word-rec* (*f 0 z*) (*f* ∘ *op* + *1*) *n*
⟨*proof*⟩

lemma *word-rec-twice*:
$m \leq n \Longrightarrow$ *word-rec z f n = word-rec* (*word-rec z f* $(n - m)$) (*f* ∘ *op* + $(n - m)$) *m*
⟨*proof*⟩

lemma *word-rec-id*: *word-rec z* (λ-. *id*) *n = z*
⟨*proof*⟩

lemma *word-rec-id-eq*: $\forall m < n.\ f\ m = id \Longrightarrow$ *word-rec z f n = z*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *unatSuc*: $1 + n \neq 0 \Longrightarrow$ *unat* $(1 + n) =$ *Suc* (*unat n*)
for $n :: 'a::len\ word$
⟨*proof*⟩

declare *bin-to-bl-def* [*simp*]

⟨*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*)
⟨*proof*⟩

lemma *xor3-simps*:
xor3 True a b = (*a* = *b*)
xor3 a True b = (*a* = *b*)
xor3 a b True = (*a* = *b*)
xor3 False a b = (*a* ≠ *b*)
xor3 a False b = (*a* ≠ *b*)
xor3 a b False = (*a* ≠ *b*)
⟨*proof*⟩

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

lemma *word-eq-rbl-eq*: *x* = *y* ⟷ *rev* (*to-bl x*) = *rev* (*to-bl y*)
⟨*proof*⟩

lemma *rbl-word-or*: *rev* (*to-bl* (*x OR y*)) = *map2 op* ∨ (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
⟨*proof*⟩

lemma *rbl-word-and*: *rev* (*to-bl* (*x AND y*)) = *map2 op* ∧ (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
⟨*proof*⟩

lemma *rbl-word-xor*: *rev* (*to-bl* (*x XOR y*)) = *map2 op* ≠ (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
⟨*proof*⟩

lemma *rbl-word-not*: *rev* (*to-bl* (*NOT x*)) = *map Not* (*rev* (*to-bl x*))
⟨*proof*⟩

lemma *bl-word-sub*: *to-bl* (*x* − *y*) = *to-bl* (*x* + (− *y*))
⟨*proof*⟩

lemma *rbl-word-1*: *rev* (*to-bl* (*1* :: *'a*::*len0 word*)) = *takefill False* (*len-of TYPE*(*'a*)) [*True*]
⟨*proof*⟩

lemma *rbl-word-if*: *rev* (*to-bl* (*if P then x else y*)) = *map2* (*If P*) (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
⟨*proof*⟩

lemma *rbl-add-carry-Cons*:
(*if car then rbl-succ else id*) (*rbl-add* (*x* # *xs*) (*y* # *ys*)) =
xor3 x y car # (*if carry x y car then rbl-succ else id*) (*rbl-add xs ys*)
⟨*proof*⟩

lemma *rbl-add-suc-carry-fold*:
length xs = *length ys* ⟹
∀ *car*. (*if car then rbl-succ else id*) (*rbl-add xs ys*) =
(*foldr* (λ(*x*, *y*) *res car*. *xor3 x y car* # *res* (*carry x y car*)) (*zip xs ys*) (λ-. [])) *car*
⟨*proof*⟩

lemma *to-bl-plus-carry*:
to-bl (*x* + *y*) =
rev (*foldr* (λ(*x*, *y*) *res car*. *xor3 x y car* # *res* (*carry x y car*))
(*rev* (*zip* (*to-bl x*) (*to-bl y*))) (λ-. []) *False*)
⟨*proof*⟩

definition *rbl-plus cin xs ys* =
foldr (λ(*x*, *y*) *res car*. *xor3 x y car* # *res* (*carry x y car*)) (*zip xs ys*) (λ-. []) *cin*

lemma *rbl-plus-simps*:
rbl-plus cin (*x* # *xs*) (*y* # *ys*) = *xor3 x y cin* # *rbl-plus* (*carry x y cin*) *xs ys*
rbl-plus cin [] *ys* = []
rbl-plus cin xs [] = []
⟨*proof*⟩

lemma *rbl-word-plus*: *rev* (*to-bl* (*x* + *y*)) = *rbl-plus False* (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
⟨*proof*⟩

definition *rbl-succ2 b xs* = (*if b then rbl-succ xs else xs*)

lemma *rbl-succ2-simps*:
rbl-succ2 b [] = []
rbl-succ2 b (*x* # *xs*) = (*b* ≠ *x*) # *rbl-succ2* (*x* ∧ *b*) *xs*
⟨*proof*⟩

lemma *twos-complement*: − *x* = *word-succ* (*NOT x*)
⟨*proof*⟩

lemma *rbl-word-neg*: *rev* (*to-bl* (− *x*)) = *rbl-succ2 True* (*map Not* (*rev* (*to-bl x*)))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)))
⟨*proof*⟩

lemma *rbl-word-ucast*:
rev (*to-bl* (*ucast x* :: *'a*::*len0 word*)) = *takefill False* (*len-of TYPE*(*'a*)) (*rev* (*to-bl x*))
⟨*proof*⟩

lemma *rbl-shiftl*:
rev (*to-bl* (*w* << *n*)) = *takefill False* (*size w*) (*replicate n False* @ *rev* (*to-bl w*))
⟨*proof*⟩

lemma *rbl-shiftr*:
rev (*to-bl* (*w* >> *n*)) = *takefill False* (*size w*) (*drop n* (*rev* (*to-bl w*)))
⟨*proof*⟩

definition *drop-nonempty v n xs* = (*if n* < *length xs then drop n xs else* [*last* (*v* # *xs*)])

lemma *drop-nonempty-simps*:
drop-nonempty v (*Suc n*) (*x* # *xs*) = *drop-nonempty x n xs*
drop-nonempty v 0 (*x* # *xs*) = (*x* # *xs*)
drop-nonempty v n [] = [*v*]
⟨*proof*⟩

definition *takefill-last x n xs* = *takefill* (*last* (*x* # *xs*)) *n xs*

lemma *takefill-last-simps*:

takefill-last z (Suc n) (x # xs) = x # takefill-last x n xs
takefill-last z 0 xs = []
takefill-last z n [] = replicate n z
⟨*proof*⟩

lemma *rbl-sshiftr*:
rev (to-bl (w >>> n)) = takefill-last False (size w) (drop-nonempty False n (rev (to-bl w)))
⟨*proof*⟩

lemma *nth-word-of-int*:
(word-of-int x :: 'a::len0 word) !! n = (n < len-of TYPE('a) ∧ bin-nth x n)
⟨*proof*⟩

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)))
⟨*proof*⟩

lemma *rbl-word-scast*:
rev (to-bl (scast x :: 'a::len word)) = takefill-last False (len-of TYPE('a)) (rev (to-bl x))
⟨*proof*⟩

definition *rbl-mul :: bool list ⇒ bool list ⇒ bool list*
where *rbl-mul xs ys = foldr (λx sm. rbl-plus False (map (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 = []
⟨*proof*⟩

lemma *takefill-le2*: *length xs ≤ n ⟹ takefill x m (takefill x n xs) = takefill x m xs*
⟨*proof*⟩

lemma *take-rbl-plus*: *∀ n b. take n (rbl-plus b xs ys) = rbl-plus b (take n xs) (take n ys)*
⟨*proof*⟩

lemma *word-rbl-mul-induct*:
length xs ≤ size y ⟹
rbl-mul xs (rev (to-bl y)) = take (length xs) (rev (to-bl (of-bl (rev xs) ∗ y)))
for *y :: 'a::len word*
⟨*proof*⟩

lemma *rbl-word-mul*: *rev* (*to-bl* (*x* ∗ *y*)) = *rbl-mul* (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
for *x* :: *'a*::*len word*
⟨*proof*⟩

Breaking up inequalities into bitlist properties.

definition
rev-bl-order F xs ys =
(*length xs* = *length ys* ∧
((*xs* = *ys* ∧ *F*)
∨ (∃ *n* < *length xs*. *drop* (*Suc n*) *xs* = *drop* (*Suc n*) *ys*
∧ ¬ *xs* ! *n* ∧ *ys* ! *n*)))

lemma *rev-bl-order-simps*:
rev-bl-order F [] [] = *F*
rev-bl-order F (*x* # *xs*) (*y* # *ys*) = *rev-bl-order* ((*y* ∧ ¬ *x*) ∨ ((*y* ∨ ¬ *x*) ∧ *F*)) *xs ys*
⟨*proof*⟩

lemma *rev-bl-order-rev-simp*:
length xs = *length ys* ⟹
rev-bl-order F (*xs* @ [*x*]) (*ys* @ [*y*]) = ((*y* ∧ ¬ *x*) ∨ ((*y* ∨ ¬ *x*) ∧ *rev-bl-order F xs ys*))
⟨*proof*⟩

lemma *rev-bl-order-bl-to-bin*:
length xs = *length ys* ⟹
rev-bl-order True xs ys = (*bl-to-bin* (*rev xs*) ≤ *bl-to-bin* (*rev ys*)) ∧
rev-bl-order False xs ys = (*bl-to-bin* (*rev xs*) < *bl-to-bin* (*rev ys*))
⟨*proof*⟩

lemma *word-le-rbl*: *x* ≤ *y* ⟷ *rev-bl-order True* (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
for *x y* :: *'a*::*len0 word*
⟨*proof*⟩

lemma *word-less-rbl*: *x* < *y* ⟷ *rev-bl-order False* (*rev* (*to-bl x*)) (*rev* (*to-bl y*))
for *x y* :: *'a*::*len0 word*
⟨*proof*⟩

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

lemma *word-sle-msb-le*: *x* <=*s y* ⟷ (*msb y* ⟶ *msb x*) ∧ ((*msb x* ∧ ¬ *msb y*) ∨ *x* ≤ *y*)
⟨*proof*⟩

lemma *word-sless-msb-less*: *x* <*s y* ⟷ (*msb y* ⟶ *msb x*) ∧ ((*msb x* ∧ ¬ *msb y*) ∨ *x* < *y*)
⟨*proof*⟩

definition *map-last f xs* = (*if xs* = [] *then* [] *else butlast xs* @ [*f* (*last xs*)])

lemma *map-last-simps*:
map-last f [] = []
map-last f [*x*] = [*f x*]
map-last f (*x* # *y* # *zs*) = *x* # *map-last f* (*y* # *zs*)
⟨*proof*⟩

lemma *word-sle-rbl*:
x <=*s y* ⟷ *rev-bl-order True* (*map-last Not* (*rev* (*to-bl x*))) (*map-last Not* (*rev* (*to-bl y*)))
⟨*proof*⟩

lemma *word-sless-rbl*:
x <*s y* ⟷ *rev-bl-order False* (*map-last Not* (*rev* (*to-bl x*))) (*map-last Not* (*rev* (*to-bl y*)))
⟨*proof*⟩

Lemmas for unpacking *rev* (*to-bl n*) for numerals n and also for irreducible values and expressions.

lemma *rev-bin-to-bl-simps*:
rev (*bin-to-bl 0 x*) = []
rev (*bin-to-bl* (*Suc n*) (*numeral* (*num.Bit0 nm*))) = *False* # *rev* (*bin-to-bl n* (*numeral nm*))
rev (*bin-to-bl* (*Suc n*) (*numeral* (*num.Bit1 nm*))) = *True* # *rev* (*bin-to-bl n* (*numeral nm*))
rev (*bin-to-bl* (*Suc n*) (*numeral* (*num.One*))) = *True* # *replicate n False*
rev (*bin-to-bl* (*Suc n*) (− *numeral* (*num.Bit0 nm*))) = *False* # *rev* (*bin-to-bl n* (− *numeral nm*))
rev (*bin-to-bl* (*Suc n*) (− *numeral* (*num.Bit1 nm*))) =
True # *rev* (*bin-to-bl n* (− *numeral* (*nm* + *num.One*)))
rev (*bin-to-bl* (*Suc n*) (− *numeral* (*num.One*))) = *True* # *replicate n True*
rev (*bin-to-bl* (*Suc n*) (− *numeral* (*num.Bit0 nm* + *num.One*))) =
True # *rev* (*bin-to-bl n* (− *numeral* (*nm* + *num.One*)))
rev (*bin-to-bl* (*Suc n*) (− *numeral* (*num.Bit1 nm* + *num.One*))) =
False # *rev* (*bin-to-bl n* (− *numeral* (*nm* + *num.One*)))
rev (*bin-to-bl* (*Suc n*) (− *numeral* (*num.One* + *num.One*))) =
False # *rev* (*bin-to-bl n* (− *numeral num.One*))
⟨*proof*⟩

lemma *to-bl-upt*: *to-bl x* = *rev* (*map* (*op* !! *x*) [*0* ..< *size x*])
⟨*proof*⟩

lemma *rev-to-bl-upt*: *rev* (*to-bl x*) = *map* (*op* !! *x*) [*0* ..< *size x*]
⟨*proof*⟩

lemma *upt-eq-list-intros*:
j ≤ *i* ⟹ [*i* ..< *j*] = []
i = *x* ⟹ *x* < *j* ⟹ [*x* + *1* ..< *j*] = *xs* ⟹ [*i* ..< *j*] = (*x* # *xs*)

⟨*proof*⟩

10.30 Tactic definition

⟨*ML*⟩

end

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

lemma *AND-lower* [*simp*]:
fixes x :: *int* **and** y :: *int*
assumes $0 \leq x$
shows $0 \leq x$ *AND* y
⟨*proof*⟩

lemma *OR-lower* [*simp*]:
fixes x :: *int* **and** y :: *int*
assumes $0 \leq x$ $0 \leq y$
shows $0 \leq x$ *OR* y
⟨*proof*⟩

lemma *XOR-lower* [*simp*]:
fixes x :: *int* **and** y :: *int*
assumes $0 \leq x$ $0 \leq y$
shows $0 \leq x$ *XOR* y
⟨*proof*⟩

lemma *AND-upper1* [*simp*]:
fixes x :: *int* **and** y :: *int*
assumes $0 \leq x$
shows x *AND* $y \leq x$
⟨*proof*⟩

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 $0 \leq y$
shows x *AND* $y \leq y$
⟨*proof*⟩

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 $0 \leq x\ x < 2\ \hat{}\ n\ y < 2\ \hat{}\ n$
shows $x\ OR\ y < 2\ \hat{}\ n$
⟨*proof*⟩

lemma *XOR-upper*:
fixes $x :: int$ **and** $y :: int$
assumes $0 \leq x\ x < 2\ \hat{}\ n\ y < 2\ \hat{}\ n$
shows $x\ XOR\ y < 2\ \hat{}\ n$
⟨*proof*⟩

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.