Machine Words in Isabelle/HOL

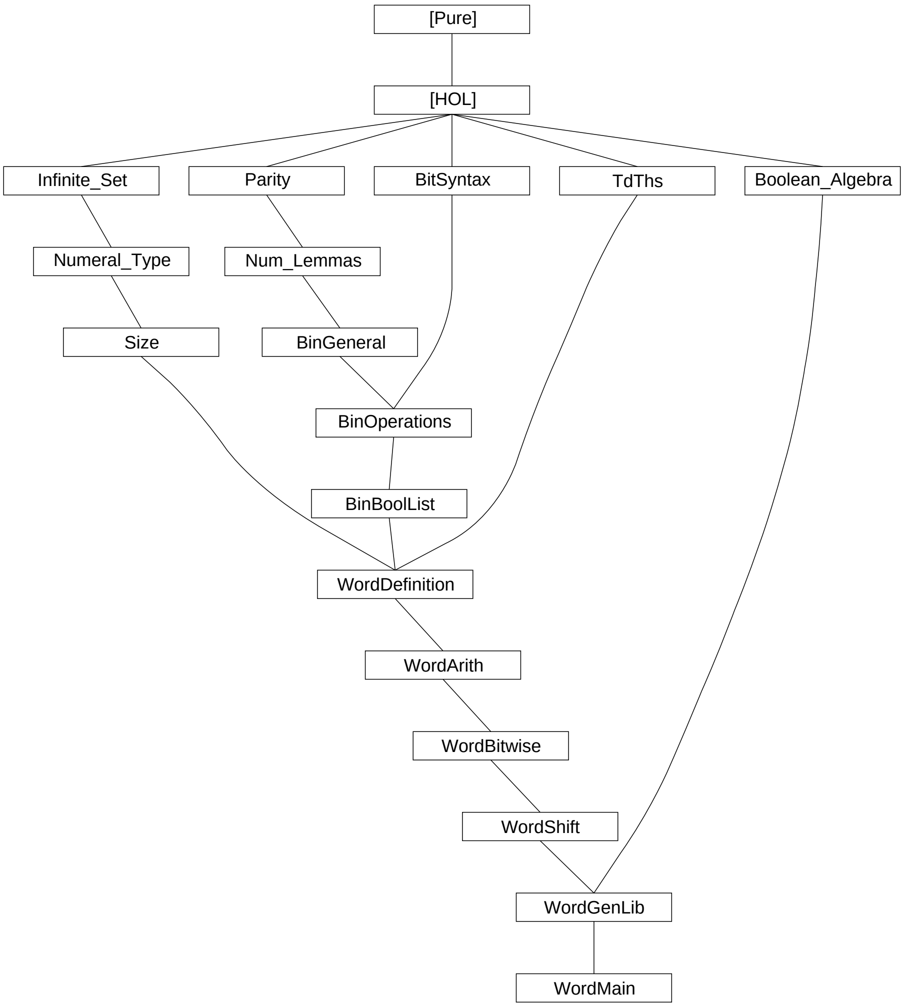
Jeremy Dawson, Paul Graunke, Brian Huffman, Gerwin Klein, and John Matthews

November 22, 2007

Abstract

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

Contents


[Pure]
[HOL]
Infinite_Set
Parity
BitSyntax
TdThs
Boolean_Algebra
Numeral_Type
Num_Lemmas
Size
BinGeneral
BinOperations
BinBoolList
WordDefinition
WordArith
WordBitwise
WordShift
WordGenLib
WordMain

1 Numeral-Type: Numeral Syntax for Types

theory *Numeral-Type*
imports *Infinite-Set*
begin

1.1 Preliminary lemmas

lemma *inj-Inl* [*simp*]: *inj-on Inl A*
⟨*proof*⟩

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
⟨*proof*⟩

lemma *inj-Some* [*simp*]: *inj-on Some A*
⟨*proof*⟩

lemma *card-Plus*:
[| *finite A*; *finite B* |] ==> *card* (*A* <+> *B*) = *card A* + *card B*
⟨*proof*⟩

lemma (**in** *type-definition*) *univ*:
UNIV = *Abs* ' *A*
⟨*proof*⟩

lemma (**in** *type-definition*) *card*: *card* (*UNIV* :: '*b set*) = *card A*
⟨*proof*⟩

1.2 Cardinalities of types

syntax *-type-card* :: *type* => *nat* ((*1CARD*/(*1*'(-'))))

translations *CARD*(*t*) => *card* (*UNIV*::*t set*)

⟨*ML*⟩

lemma *card-unit*: *CARD*(*unit*) = *1*
⟨*proof*⟩

lemma *card-bool*: *CARD*(*bool*) = *2*
⟨*proof*⟩

lemma *card-prod*: *CARD*('*a*::*finite* × '*b*::*finite*) = *CARD*('*a*) ∗ *CARD*('*b*)
⟨*proof*⟩

lemma *card-sum*: *CARD*('*a*::*finite* + '*b*::*finite*) = *CARD*('*a*) + *CARD*('*b*)
⟨*proof*⟩

lemma *card-option*: *CARD*('*a*::*finite option*) = *Suc CARD*('*a*)
⟨*proof*⟩

lemma *card-set*: $CARD('a::finite\ set) = 2\ \hat{}\ CARD('a)$
⟨*proof*⟩

1.3 Numeral Types

typedef (**open**) *num0* = *UNIV* :: *nat set* ⟨*proof*⟩
typedef (**open**) *num1* = *UNIV* :: *unit set* ⟨*proof*⟩
typedef (**open**) *'a bit0* = *UNIV* :: (*bool* ∗ *'a*) *set* ⟨*proof*⟩
typedef (**open**) *'a bit1* = *UNIV* :: (*bool* ∗ *'a*) *option set* ⟨*proof*⟩

instance *num1* :: *finite*
⟨*proof*⟩

instance *bit0* :: (*finite*) *finite*
⟨*proof*⟩

instance *bit1* :: (*finite*) *finite*
⟨*proof*⟩

lemma *card-num1*: $CARD(num1) = 1$
⟨*proof*⟩

lemma *card-bit0*: $CARD('a::finite\ bit0) = 2 * CARD('a)$
⟨*proof*⟩

lemma *card-bit1*: $CARD('a::finite\ bit1) = Suc\ (2 * CARD('a))$
⟨*proof*⟩

lemma *card-num0*: $CARD\ (num0) = 0$
⟨*proof*⟩

lemmas *card-univ-simps* [*simp*] =
card-unit
card-bool
card-prod
card-sum
card-option
card-set
card-num1
card-bit0
card-bit1
card-num0

1.4 Syntax

syntax
-NumeralType :: *num-const* => *type* (-)
-NumeralType0 :: *type* (*0*)
-NumeralType1 :: *type* (*1*)

translations
-NumeralType1 == (*type*) *num1*
-NumeralType0 == (*type*) *num0*

⟨*ML*⟩

1.5 Classes with at least 1 and 2

Class finite already captures "at least 1"

lemma *zero-less-card-finite* [*simp*]:
0 < *CARD*(*'a*::*finite*)
⟨*proof*⟩

lemma *one-le-card-finite* [*simp*]:
Suc 0 <= *CARD*(*'a*::*finite*)
⟨*proof*⟩

Class for cardinality "at least 2"

class *card2* = *finite* +
assumes *two-le-card*: *2* <= *CARD*(*'a*)

lemma *one-less-card*: *Suc 0* < *CARD*(*'a*::*card2*)
⟨*proof*⟩

instance *bit0* :: (*finite*) *card2*
⟨*proof*⟩

instance *bit1* :: (*finite*) *card2*
⟨*proof*⟩

1.6 Examples

term *TYPE*(*10*)

lemma *CARD*(*0*) = *0* ⟨*proof*⟩
lemma *CARD*(*17*) = *17* ⟨*proof*⟩

end

2 Size: The size class

theory *Size*
imports *Numeral-Type*
begin

The aim of this is to allow any type as index type, but to provide a default instantiation for numeral types. This independence requires some duplication with the definitions in Numeral_Type.

axclass *len0* < *type*

consts
len-of :: (*'a* :: *len0 itself*) => *nat*

Some theorems are only true on words with length greater 0.

axclass *len* < *len0*
len-gt-0 [*iff*]: *0* < *len-of TYPE* (*'a* :: *len0*)

instance *num0* :: *len0* ⟨*proof*⟩
instance *num1* :: *len0* ⟨*proof*⟩
instance *bit0* :: (*len0*) *len0* ⟨*proof*⟩
instance *bit1* :: (*len0*) *len0* ⟨*proof*⟩

defs (**overloaded**)
len-num0: *len-of* (*x*::*num0 itself*) == *0*
len-num1: *len-of* (*x*::*num1 itself*) == *1*
len-bit0: *len-of* (*x*::*'a*::*len0 bit0 itself*) == *2* ∗ *len-of TYPE* (*'a*)
len-bit1: *len-of* (*x*::*'a*::*len0 bit1 itself*) == *2* ∗ *len-of TYPE* (*'a*) + *1*

lemmas *len-of-numeral-defs* [*simp*] = *len-num0 len-num1 len-bit0 len-bit1*

instance *num1* :: *len* ⟨*proof*⟩
instance *bit0* :: (*len*) *len* ⟨*proof*⟩
instance *bit1* :: (*len0*) *len* ⟨*proof*⟩
lemma *len-of TYPE*(*17*) = *17* ⟨*proof*⟩
lemma *len-of TYPE*(*0*) = *0* ⟨*proof*⟩
lemma *len-of TYPE*(*'a*::*len0*) = *x*
⟨*proof*⟩

end

3 Num-Lemmas: Useful Numerical Lemmas

theory *Num-Lemmas* **imports** *Parity* **begin**

lemma *contentsI*: *y* = {*x*} ==> *contents y* = *x*
⟨*proof*⟩

lemma *prod-case-split*: *prod-case* = *split*
⟨*proof*⟩

lemmas *split-split* = *prod.split* [*unfolded prod-case-split*]

lemmas *split-split-asm* = *prod.split-asm* [*unfolded prod-case-split*]
lemmas *split.splits* = *split-split split-split-asm*

lemmas *funpow-0* = *funpow.simps*(*1*)
lemmas *funpow-Suc* = *funpow.simps*(*2*)

lemma *nonemptyE*: *S* ~= {} ==> (!!*x*. *x* : *S* ==> *R*) ==> *R*
⟨*proof*⟩

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

constdefs
mod-alt :: *'a* => *'a* => *'a* :: *Divides.div*
mod-alt n m == *n mod m*

— alternative way of defining *bin-last*, *bin-rest*
bin-rl :: *int* => *int* ∗ *bit*
bin-rl w == *SOME* (*r*, *l*). *w* = *r BIT l*

declare *iszero-0* [*iff*]

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*)

lemma *Min-ne-Pls* [*iff*]:
Numeral.Min ~= *Numeral.Pls*
⟨*proof*⟩

lemmas *Pls-ne-Min* [*iff*] = *Min-ne-Pls* [*symmetric*]

lemmas *PlsMin-defs* [*intro*!] =
Pls-def Min-def Pls-def [*symmetric*] *Min-def* [*symmetric*]

lemmas *PlsMin-simps* [*simp*] = *PlsMin-defs* [*THEN Eq-TrueI*]

lemma *number-of-False-cong*:
False ==> *number-of x* = *number-of y*
⟨*proof*⟩

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

lemma *sum-imp-diff*: *j* = *k* + *i* ==> *j* − *i* = (*k* :: *nat*)

⟨*proof*⟩

lemma *nobm1*:
$0 < (number\text{-}of\ w :: nat) ==>$
$number\text{-}of\ w - (1 :: nat) = number\text{-}of\ (Numeral.pred\ w)$
⟨*proof*⟩

lemma *of-int-power*:
$of\text{-}int\ (a\ \hat{}\ n) = (of\text{-}int\ a\ \hat{}\ n :: 'a :: \{recpower,\ comm\text{-}ring\text{-}1\})$
⟨*proof*⟩

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

lemmas *zless2p* [*simp*] = *zless2* [*THEN zero-less-power*]
lemmas *zle2p* [*simp*] = *zless2p* [*THEN order-less-imp-le*]

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

— the inverse(s) of *number-of*
lemma *nmod2*: $n\ mod\ (2::int) = 0\ |\ n\ mod\ 2 = 1$
⟨*proof*⟩

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

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

lemma *le-diff-eq'*: $(a \leq c - b) = (b + a \leq (c::int))$
⟨*proof*⟩

lemma *less-diff-eq'*: $(a < c - b) = (b + a < (c::int))$
⟨*proof*⟩

lemma *diff-le-eq'*: $(a - b \leq c) = (a \leq b + (c::int))$
⟨*proof*⟩

lemma *diff-less-eq'*: $(a - b < c) = (a < b + (c::int))$
⟨*proof*⟩

lemmas *m1mod2k* = *zless2p* [*THEN zmod-minus1*]
lemmas *m1mod22k* = *mult-pos-pos* [*OF zless2 zless2p*, *THEN zmod-minus1*]
lemmas *p1mod22k'* = *zless2p* [*THEN order-less-imp-le*, *THEN pos-zmod-mult-2*]
lemmas *z1pmod2'* = *zero-le-one* [*THEN pos-zmod-mult-2*, *simplified*]
lemmas *z1pdiv2'* = *zero-le-one* [*THEN pos-zdiv-mult-2*, *simplified*]

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

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

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

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

lemma *BIT-eq*: $u\ BIT\ b = v\ BIT\ c$ ==> $u = v\ \&\ b = c$
⟨*proof*⟩

lemmas *BIT-eqE* [*elim*!] = *BIT-eq* [*THEN conjE*, *standard*]

lemma *BIT-eq-iff* [*simp*]:
$(u\ BIT\ b = v\ BIT\ c) = (u = v \wedge b = c)$
⟨*proof*⟩

lemmas *BIT-eqI* [*intro*!] = *conjI* [*THEN BIT-eq-iff* [*THEN iffD2*]]

lemma *less-Bits*:
$(v\ BIT\ b < w\ BIT\ c) = (v < w \mid v <= w\ \&\ b = bit.B0\ \&\ c = bit.B1)$
⟨*proof*⟩

lemma *le-Bits*:
$(v\ BIT\ b <= w\ BIT\ c) = (v < w \mid v <= w\ \&\ (b\ \tilde{}{=}\ bit.B1 \mid c\ \tilde{}{=}\ bit.B0))$
⟨*proof*⟩

lemma *neB1E* [*elim*!]:
assumes *ne*: $y \neq bit.B1$
assumes *y*: $y = bit.B0 \Longrightarrow P$
shows P
⟨*proof*⟩

lemma *bin-ex-rl*: $EX\ w\ b.\ w\ BIT\ b = bin$
⟨*proof*⟩

lemma *bin-exhaust*:
assumes *Q*: $\bigwedge x\ b.\ bin = x\ BIT\ b \Longrightarrow Q$
shows Q

⟨*proof*⟩

lemma *bin-rl-char*: $(bin\text{-}rl\ w = (r, l)) = (r\ BIT\ l = w)$
⟨*proof*⟩

lemmas *bin-rl-simps* [*THEN bin-rl-char* [*THEN iffD2*], *standard*, *simp*] =
Pls-0-eq Min-1-eq refl

lemma *bin-abs-lem*:
$bin = (w\ BIT\ b) ==> {}^\sim\ bin = Numeral.Min\ --> {}^\sim\ bin = Numeral.Pls\ -->$
$nat\ (abs\ w) < nat\ (abs\ bin)$
⟨*proof*⟩

lemma *bin-induct*:
assumes *PPls*: $P\ Numeral.Pls$
and *PMin*: $P\ Numeral.Min$
and *PBit*: $!!bin\ bit.\ P\ bin ==> P\ (bin\ BIT\ bit)$
shows $P\ bin$
⟨*proof*⟩

lemma *no-no* [*simp*]: $number\text{-}of\ (number\text{-}of\ i) = i$
⟨*proof*⟩

lemma *Bit-B0*:
$k\ BIT\ bit.B0 = k + k$
⟨*proof*⟩

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

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

lemma *Bit-B1-2t*: $k\ BIT\ bit.B1 = 2 * k + 1$
⟨*proof*⟩

lemma *B-mod-2'*:
$X = 2 ==> (w\ BIT\ bit.B1)\ mod\ X = 1\ \&\ (w\ BIT\ bit.B0)\ mod\ X = 0$
⟨*proof*⟩

lemmas *B1-mod-2* [*simp*] = *B-mod-2'* [*OF refl*, *THEN conjunct1*, *standard*]
lemmas *B0-mod-2* [*simp*] = *B-mod-2'* [*OF refl*, *THEN conjunct2*, *standard*]

lemma *axxbyy*:
$a + m + m = b + n + n ==> (a = 0 \mid a = 1) ==> (b = 0 \mid b = 1) ==>$
$a = b\ \&\ m = (n :: int)$
⟨*proof*⟩

lemma *axxmod2*:
$(1 + x + x) \ mod \ 2 = (1 :: int) \ \& \ (0 + x + x) \ mod \ 2 = (0 :: int)$
⟨*proof*⟩

lemma *axxdiv2*:
$(1 + x + x) \ div \ 2 = (x :: int) \ \& \ (0 + x + x) \ div \ 2 = (x :: int)$
⟨*proof*⟩

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

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

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

lemma *zmod-uminus*: $- ((a :: int) \ mod \ b) \ mod \ b = -a \ mod \ b$
⟨*proof*⟩

lemma *zmod-zsub-distrib*: $((a::int) - b) \ mod \ c = (a \ mod \ c - b \ mod \ c) \ mod \ c$
⟨*proof*⟩

lemma *zmod-zsub-right-eq*: $((a::int) - b) \ mod \ c = (a - b \ mod \ c) \ mod \ c$
⟨*proof*⟩

lemma *zmod-zsub-left-eq*: $((a::int) - b) \ mod \ c = (a \ mod \ c - b) \ mod \ c$
⟨*proof*⟩

lemma *zmod-zsub-self* [*simp*]:
$((b :: int) - a) \ mod \ a = b \ mod \ a$
⟨*proof*⟩

lemma *zmod-zmult1-eq-rev*:
$b * a \ mod \ c = b \ mod \ c * a \ mod \ (c::int)$
⟨*proof*⟩

lemmas *rdmods* [*symmetric*] = *zmod-uminus* [*symmetric*]
zmod-zsub-left-eq zmod-zsub-right-eq zmod-zadd-left-eq
zmod-zadd-right-eq zmod-zmult1-eq zmod-zmult1-eq-rev

lemma *mod-plus-right*:
$((a + x) \ mod \ m = (b + x) \ mod \ m) = (a \ mod \ m = b \ mod \ (m :: nat))$
⟨*proof*⟩

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

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

lemmas *push-mods'* = *zmod-zadd1-eq* [*standard*]
zmod-zmult-distrib [*standard*] *zmod-zsub-distrib* [*standard*]
zmod-uminus [*symmetric*, *standard*]

lemmas *push-mods* = *push-mods'* [*THEN eq-reflection*, *standard*]
lemmas *pull-mods* = *push-mods* [*symmetric*] *rdmods* [*THEN eq-reflection*, *standard*]
lemmas *mod-simps* =
zmod-zmult-self1 [*THEN eq-reflection*] *zmod-zmult-self2* [*THEN eq-reflection*]
mod-mod-trivial [*THEN eq-reflection*]

lemma *nat-mod-eq*:
$!!b.\ b < n ==> a\ mod\ n = b\ mod\ n ==> a\ mod\ n = (b :: nat)$
⟨*proof*⟩

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

lemma *nat-mod-lem*:
$(0 :: nat) < n ==> b < n = (b\ mod\ n = b)$
⟨*proof*⟩

lemma *mod-nat-add*:
$(x :: nat) < z ==> y < z ==>$
$(x + y)\ mod\ z = (if\ x + y < z\ then\ x + y\ else\ x + y - z)$
⟨*proof*⟩

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

lemma *int-mod-lem*:
$(0 :: int) < n ==> (0 <= b\ \&\ b < n) = (b\ mod\ n = b)$
⟨*proof*⟩

lemma *int-mod-eq*:
$(0 :: int) <= b ==> b < n ==> a\ mod\ n = b\ mod\ n ==> a\ mod\ n = b$
⟨*proof*⟩

lemmas *int-mod-eq'* = *refl* [*THEN* [*3*] *int-mod-eq*]

lemma *int-mod-le*: $0 <= a ==> 0 < (n :: int) ==> a\ mod\ n <= a$
⟨*proof*⟩

lemma *int-mod-le'*: $0 <= b - n ==> 0 < (n :: int) ==> b\ mod\ n <= b - n$
⟨*proof*⟩

lemma *int-mod-ge*: $a < n ==> 0 < (n :: int) ==> a <= a\ mod\ n$

⟨*proof*⟩

lemma *int-mod-ge'*: $b < 0 ==> 0 < (n :: int) ==> b + n <= b \ mod \ n$
⟨*proof*⟩

lemma *mod-add-if-z*:
$(x :: int) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>$
$(x + y) \ mod \ z = (if \ x + y < z \ then \ x + y \ else \ x + y - z)$
⟨*proof*⟩

lemma *mod-sub-if-z*:
$(x :: int) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>$
$(x - y) \ mod \ z = (if \ y <= x \ then \ x - y \ else \ x - y + z)$
⟨*proof*⟩

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

lemma *zdiv-mult-self*: $m \ \tilde{}= (0 :: int) ==> (a + m * n) \ div \ m = a \ div \ m + n$
⟨*proof*⟩

lemma *eqne*: $equiv \ A \ r ==> X : A \ // \ r ==> X \ \tilde{}= \{\}$
⟨*proof*⟩

lemmas *Rep-Integ-ne* = *Integ.Rep-Integ*
[*THEN equiv-intrel* [*THEN eqne*, *simplified Integ-def* [*symmetric*]], *standard*]

lemmas *riq* = *Integ.Rep-Integ* [*simplified Integ-def*]
lemmas *intrel-refl* = *refl* [*THEN equiv-intrel-iff* [*THEN iffD1*], *standard*]
lemmas *Rep-Integ-equiv* = *quotient-eq-iff*
[*OF equiv-intrel riq riq*, *simplified Integ.Rep-Integ-inject*, *standard*]
lemmas *Rep-Integ-same* =
Rep-Integ-equiv [*THEN intrel-refl* [*THEN rev-iffD2*], *standard*]

lemma *RI-int*: $(a, 0) : Rep\text{-}Integ \ (int \ a)$
⟨*proof*⟩

lemmas *RI-intrel* [*simp*] = *UNIV-I* [*THEN quotientI*,
THEN Integ.Abs-Integ-inverse [*simplified Integ-def*], *standard*]

lemma *RI-minus*: $(a, b) : Rep\text{-}Integ \ x ==> (b, a) : Rep\text{-}Integ \ (- x)$
⟨*proof*⟩

lemma *RI-add*:
$(a, b) : Rep\text{-}Integ \ x ==> (c, d) : Rep\text{-}Integ \ y ==>$
$(a + c, b + d) : Rep\text{-}Integ \ (x + y)$
⟨*proof*⟩

lemma *mem-same*: $a : S$ ==> $a = b$ ==> $b : S$
⟨*proof*⟩

lemma *RI-eq-diff'*: (a, b) : *Rep-Integ* (*int* a − *int* b)
⟨*proof*⟩

lemma *RI-eq-diff*: $((a, b)$: *Rep-Integ* $x)$ = (*int* a − *int* $b = x$)
⟨*proof*⟩

lemma *mod-power-lem*:
$a > 1$ ==> a ˆ n *mod* a ˆ m = (*if* m <= n *then* 0 *else* (a :: *int*) ˆ n)
⟨*proof*⟩

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

lemmas *min-pm1* [*simp*] = *trans* [*OF add-commute min-pm*]

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

lemmas *rev-min-pm1* [*simp*] = *trans* [*OF add-commute rev-min-pm*]

lemma *pl-pl-rels*:
$a + b = c + d$ ==>
a >= c & b <= d | a <= c & b >= (d :: *nat*)
⟨*proof*⟩

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

lemma *minus-eq*: $(m - k = m)$ = ($k = 0$ | m = (0 :: *nat*))
⟨*proof*⟩

lemma *pl-pl-mm*: (a :: *nat*) + $b = c + d$ ==> $a - c = d - b$
⟨*proof*⟩

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

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

lemmas *min-minus'* [*simp*] = *trans* [*OF min-max.inf-commute min-minus*]

lemma *nat-no-eq-iff*:
(*number-of* b :: *int*) >= 0 ==> (*number-of* c :: *int*) >= 0 ==>
(*number-of* b = (*number-of* c :: *nat*)) = ($b = c$)
⟨*proof*⟩

lemmas *dme* = *box-equals* [*OF div-mod-equality add-0-right add-0-right*]
lemmas *dtle* = *xtr3* [*OF dme* [*symmetric*] *le-add1*]
lemmas *th2* = *order-trans* [*OF order-refl* [*THEN* [*2*] *mult-le-mono*] *dtle*]

lemma *td-gal*:
$0 < c ==> (a >= b * c) = (a \; div \; c >= (b :: nat))$
⟨*proof*⟩

lemmas *td-gal-lt* = *td-gal* [*simplified le-def*, *simplified*]

lemma *div-mult-le*: $(a :: nat) \; div \; b * b <= a$
⟨*proof*⟩

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

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

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

lemma *diff-mod-le*: $(a::nat) < d ==> b \; dvd \; d ==> a - a \; mod \; b <= d - b$
⟨*proof*⟩

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

lemmas *less-le-mult* = *less-le-mult'* [*simplified left-distrib*, *simplified*]

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

lemma *lrlem'*:
assumes *d*: $(i::nat) \leq j \vee m < j'$
assumes *R1*: $i * k \leq j * k \Longrightarrow R$
assumes *R2*: $Suc \; m * k' \leq j' * k' \Longrightarrow R$
shows R ⟨*proof*⟩

lemma *lrlem*: $(0::nat) < sc ==>$
$(sc - n + (n + lb * n) <= m * n) = (sc + lb * n <= m * n)$
⟨*proof*⟩

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

lemma *mpl-lem*: $j <= (i :: nat) ==> k < j ==> i - j + k < i$
⟨*proof*⟩

lemma *nonneg-mod-div*:
  0 <= a ==> 0 <= b ==> 0 <= (a mod b :: int) & 0 <= a div b
  ⟨*proof*⟩

end

4 BinGeneral: Basic Definitions for Binary Integers

theory *BinGeneral* **imports** *Num-Lemmas*

begin

4.1 Recursion combinator for binary integers

lemma *brlem*: (*bin* = *Numeral.Min*) = (− *bin* + *Numeral.pred 0* = *0*)
  ⟨*proof*⟩

function
  bin-rec′ :: *int* ∗ ′*a* ∗ ′*a* ∗ (*int* => *bit* => ′*a* => ′*a*) => ′*a*
  where
  bin-rec′ (*bin*, *f1*, *f2*, *f3*) = (*if bin* = *Numeral.Pls then f1*
    else if bin = *Numeral.Min then f2*
    else case bin-rl bin of (*w*, *b*) => *f3 w b* (*bin-rec′* (*w*, *f1*, *f2*, *f3*)))
  ⟨*proof*⟩

termination
  ⟨*proof*⟩

constdefs
  bin-rec :: ′*a* => ′*a* => (*int* => *bit* => ′*a* => ′*a*) => *int* => ′*a*
  bin-rec f1 f2 f3 bin == *bin-rec′* (*bin*, *f1*, *f2*, *f3*)

lemma *bin-rec-PM*:
  f = *bin-rec f1 f2 f3* ==> *f Numeral.Pls* = *f1* & *f Numeral.Min* = *f2*
  ⟨*proof*⟩

lemmas *bin-rec-Pls* = *refl* [*THEN bin-rec-PM*, *THEN conjunct1*, *standard*]
lemmas *bin-rec-Min* = *refl* [*THEN bin-rec-PM*, *THEN conjunct2*, *standard*]

lemma *bin-rec-Bit*:
  f = *bin-rec f1 f2 f3* ==> *f3 Numeral.Pls bit.B0 f1* = *f1* ==>
    f3 Numeral.Min bit.B1 f2 = *f2* ==> *f* (*w BIT b*) = *f3 w b* (*f w*)
  ⟨*proof*⟩

lemmas *bin-rec-simps* = *refl* [*THEN bin-rec-Bit*] *bin-rec-Pls bin-rec-Min*

4.2 Destructors for binary integers

consts
— corresponding operations analysing bins
bin-last :: *int* => *bit*
bin-rest :: *int* => *int*
bin-sign :: *int* => *int*
bin-nth :: *int* => *nat* => *bool*

primrec
Z : *bin-nth w 0* = (*bin-last w* = *bit.B1*)
Suc : *bin-nth w* (*Suc n*) = *bin-nth* (*bin-rest w*) *n*

defs
bin-rest-def : *bin-rest w* == *fst* (*bin-rl w*)
bin-last-def : *bin-last w* == *snd* (*bin-rl w*)
bin-sign-def : *bin-sign* == *bin-rec Numeral.Pls Numeral.Min* (%*w b s. s*)

lemma *bin-rl*: *bin-rl w* = (*bin-rest w*, *bin-last w*)
⟨*proof*⟩

lemmas *bin-rl-simp* [*simp*] = *iffD1* [*OF bin-rl-char bin-rl*]

lemma *bin-rest-simps* [*simp*]:
bin-rest Numeral.Pls = *Numeral.Pls*
bin-rest Numeral.Min = *Numeral.Min*
bin-rest (*w BIT b*) = *w*
⟨*proof*⟩

lemma *bin-last-simps* [*simp*]:
bin-last Numeral.Pls = *bit.B0*
bin-last Numeral.Min = *bit.B1*
bin-last (*w BIT b*) = *b*
⟨*proof*⟩

lemma *bin-sign-simps* [*simp*]:
bin-sign Numeral.Pls = *Numeral.Pls*
bin-sign Numeral.Min = *Numeral.Min*
bin-sign (*w BIT b*) = *bin-sign w*
⟨*proof*⟩

lemma *bin-r-l-extras* [*simp*]:
bin-last 0 = *bit.B0*
bin-last (− *1*) = *bit.B1*
bin-last −*1* = *bit.B1*
bin-last 1 = *bit.B1*
bin-rest 1 = *0*
bin-rest 0 = *0*
bin-rest (− *1*) = − *1*
bin-rest −*1* = −*1*

⟨*proof*⟩

lemma *bin-last-mod*:
bin-last w = (if w mod 2 = 0 then bit.B0 else bit.B1)
⟨*proof*⟩

lemma *bin-rest-div*:
bin-rest w = w div 2
⟨*proof*⟩

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

lemma *bin-nth-lem* [*rule-format*]:
ALL y. bin-nth x = bin-nth y --> x = y
⟨*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], standard]*

lemma *bin-nth-Pls* [*simp*]: *~ bin-nth Numeral.Pls n*
⟨*proof*⟩

lemma *bin-nth-Min* [*simp*]: *bin-nth Numeral.Min n*
⟨*proof*⟩

lemma *bin-nth-0-BIT*: *bin-nth (w BIT b) 0 = (b = bit.B1)*
⟨*proof*⟩

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

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

lemmas *bin-nth-0 = bin-nth.simps(1)*
lemmas *bin-nth-Suc = bin-nth.simps(2)*

lemmas *bin-nth-simps =*
bin-nth-0 bin-nth-Suc bin-nth-Pls bin-nth-Min bin-nth-minus

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

4.3 Truncating binary integers

consts
bintrunc :: *nat* => *int* => *int*
primrec
Z : *bintrunc 0 bin* = *Numeral.Pls*
Suc : *bintrunc* (*Suc n*) *bin* = *bintrunc n* (*bin-rest bin*) *BIT* (*bin-last bin*)

consts
sbintrunc :: *nat* => *int* => *int*
primrec
Z : *sbintrunc 0 bin* =
(*case bin-last bin of bit.B1* => *Numeral.Min* | *bit.B0* => *Numeral.Pls*)
Suc : *sbintrunc* (*Suc n*) *bin* = *sbintrunc n* (*bin-rest bin*) *BIT* (*bin-last bin*)

lemma *sign-bintr*:
!!*w*. *bin-sign* (*bintrunc n w*) = *Numeral.Pls*
⟨*proof*⟩

lemma *bintrunc-mod2p*:
!!*w*. *bintrunc n w* = (*w mod 2* ˆ *n* :: *int*)
⟨*proof*⟩

lemma *sbintrunc-mod2p*:
!!*w*. *sbintrunc n w* = ((*w* + *2* ˆ *n*) *mod 2* ˆ (*Suc n*) − *2* ˆ *n* :: *int*)
⟨*proof*⟩

4.4 Simplifications for (s)bintrunc

lemma *bit-bool*:
(*b* = (*b′* = *bit.B1*)) = (*b′* = (*if b then bit.B1 else bit.B0*))
⟨*proof*⟩

lemmas *bit-bool1* [*simp*] = *refl* [*THEN bit-bool* [*THEN iffD1*], *symmetric*]

lemma *bin-sign-lem*:
!!*bin*. (*bin-sign* (*sbintrunc n bin*) = *Numeral.Min*) = *bin-nth bin n*
⟨*proof*⟩

lemma *nth-bintr*:
!!*w m*. *bin-nth* (*bintrunc m w*) *n* = (*n* < *m* & *bin-nth w n*)
⟨*proof*⟩

lemma *nth-sbintr*:
!!*w m*. *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* = *bit.B1* | (*EX m*. *n* = *Suc m* & *bin-nth 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=Numeral.Pls*, *simplified bin-last-simps bin-rest-simps*, *standard*]

lemmas *bintrunc-Min* [*simp*] =
bintrunc.Suc [**where** *bin=Numeral.Min*, *simplified bin-last-simps bin-rest-simps*, *standard*]

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

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

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

lemmas *sbintrunc-Suc-Min* =
sbintrunc.Suc [**where** *bin=Numeral.Min*, *simplified bin-last-simps bin-rest-simps*, *standard*]

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

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

lemmas *sbintrunc-Pls* =
sbintrunc.Z [**where** *bin*=*Numeral.Pls*,
simplified bin-last-simps bin-rest-simps bit.simps, *standard*]

lemmas *sbintrunc-Min* =
sbintrunc.Z [**where** *bin*=*Numeral.Min*,
simplified bin-last-simps bin-rest-simps bit.simps, *standard*]

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

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

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

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

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

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

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

lemma *bintrunc-n-Pls* [*simp*]:
bintrunc n Numeral.Pls = Numeral.Pls
⟨*proof*⟩

lemma *sbintrunc-n-PM* [*simp*]:
sbintrunc n Numeral.Pls = Numeral.Pls
sbintrunc n Numeral.Min = Numeral.Min
⟨*proof*⟩

lemmas *thobini1* = *arg-cong* [**where** *f* = *%w. w BIT b*, *standard*]

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

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

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

lemma *bintrunc-0-Min*: *bintrunc 0 Numeral.Min* = *Numeral.Pls*
⟨*proof*⟩
lemma *bintrunc-0-BIT*: *bintrunc 0* (*w BIT b*) = *Numeral.Pls*
⟨*proof*⟩

lemma *bintrunc-Suc-lem*:
bintrunc (*Suc n*) *x* = *y* ==> *m* = *Suc n* ==> *bintrunc m x* = *y*
⟨*proof*⟩

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

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

lemmas *sbintrunc-Suc-Is* =
sbintrunc-Sucs [*THEN thobini1* [*THEN* [*2*] *trans*], *standard*]

lemmas *sbintrunc-Suc-minus-Is* =
sbintrunc-minus-simps [*THEN thobini1* [*THEN* [*2*] *trans*], *standard*]

lemma *sbintrunc-Suc-lem*:
sbintrunc (*Suc n*) *x* = *y* ==> *m* = *Suc n* ==> *sbintrunc m x* = *y*
⟨*proof*⟩

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

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

lemma *bintrunc-sbintrunc-le*:
m <= *Suc n* ==> *bintrunc m* (*sbintrunc n w*) = *bintrunc m w*
⟨*proof*⟩

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

lemma *bintrunc-sbintrunc'* [*simp*]:

$0 < n \Longrightarrow$ *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 <−>
sbintrunc n x = sbintrunc n y
⟨*proof*⟩

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

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

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

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

lemmas *bintrunc-pred-simps* [*simp*] =
bintrunc-minus-simps [*of number-of bin*, *simplified nobm1*, *standard*]

lemmas *sbintrunc-pred-simps* [*simp*] =
sbintrunc-minus-simps [*of number-of bin*, *simplified nobm1*, *standard*]

lemma *no-bintr-alt*:
number-of (bintrunc n w) = w mod 2 ˆ n
⟨*proof*⟩

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

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

lemma *no-bintr*:
number-of (bintrunc n w) = (number-of w mod 2 ˆ n :: int)
⟨*proof*⟩

lemma *no-sbintr-alt2*:

sbintrunc n = (%*w*. (*w* + *2* ^ *n*) *mod 2* ^ *Suc n* − *2* ^ *n* :: *int*)
⟨*proof*⟩

lemma *no-sbintr*:
number-of (*sbintrunc n w*) =
((*number-of 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*::*int*) + *2*^*k* < *0* ⟹ *a* + *2*^*k* + *2*^(*Suc k*) <= (*a* + *2*^*k*) *mod 2*^(*Suc k*)
⟨*proof*⟩

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

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

lemma *sb-dec-lem*:
(*0*::*int*) <= − (*2*^*k*) + *a* ==> (*a* + *2*^*k*) *mod* (*2* ∗ *2* ^ *k*) <= − (*2* ^ *k*) + *a*
⟨*proof*⟩

lemma *sb-dec-lem'*:
(*2*::*int*) ^ *k* <= *a* ==> (*a* + *2* ^ *k*) *mod* (*2* ∗ *2* ^ *k*) <= − (*2* ^ *k*) + *a*
⟨*proof*⟩

lemma *sbintrunc-dec*:
x >= (*2* ^ *n*) ==> *x* − *2* ^ (*Suc n*) >= *sbintrunc n x*
⟨*proof*⟩

lemmas *zmod-uminus'* = *zmod-uminus* [**where** *b*=*c*, *standard*]
lemmas *zpower-zmod'* = *zpower-zmod* [**where** *m*=*c* **and** *y*=*k*, *standard*]

lemmas *brdmod1s'* [*symmetric*] =
zmod-zadd-left-eq zmod-zadd-right-eq
zmod-zsub-left-eq zmod-zsub-right-eq
zmod-zmult1-eq zmod-zmult1-eq-rev

lemmas *brdmods'* [*symmetric*] =
zpower-zmod' [*symmetric*]
trans [*OF zmod-zadd-left-eq zmod-zadd-right-eq*]
trans [*OF zmod-zsub-left-eq zmod-zsub-right-eq*]
trans [*OF zmod-zmult1-eq zmod-zmult1-eq-rev*]

zmod-uminus′ [*symmetric*]
zmod-zadd-left-eq [**where** $b = 1$]
zmod-zsub-left-eq [**where** $b = 1$]

lemmas *bintr-arith1s* =
brdmod1s′ [**where** $c=2\hat{}n$, *folded pred-def succ-def bintrunc-mod2p*, *standard*]
lemmas *bintr-ariths* =
brdmods′ [**where** $c=2\hat{}n$, *folded pred-def succ-def bintrunc-mod2p*, *standard*]

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

lemma *bintr-ge0*: $(0 :: int) <= number\text{-}of\ (bintrunc\ n\ w)$
⟨*proof*⟩

lemma *bintr-lt2p*: $number\text{-}of\ (bintrunc\ n\ w) < (2\ \hat{}\ n :: int)$
⟨*proof*⟩

lemma *bintr-Min*:
$number\text{-}of\ (bintrunc\ n\ Numeral.Min) = (2\ \hat{}\ n :: int) - 1$
⟨*proof*⟩

lemma *sbintr-ge*: $(-\ (2\ \hat{}\ n) :: int) <= number\text{-}of\ (sbintrunc\ n\ w)$
⟨*proof*⟩

lemma *sbintr-lt*: $number\text{-}of\ (sbintrunc\ n\ w) < (2\ \hat{}\ n :: int)$
⟨*proof*⟩

lemma *bintrunc-Suc*:
bintrunc (*Suc n*) *bin* = *bintrunc n* (*bin-rest bin*) *BIT bin-last bin*
⟨*proof*⟩

lemma *sign-Pls-ge-0*:
(*bin-sign bin* = *Numeral.Pls*) = (*number-of bin* >= (*0* :: *int*))
⟨*proof*⟩

lemma *sign-Min-lt-0*:
(*bin-sign bin* = *Numeral.Min*) = (*number-of bin* < (*0* :: *int*))
⟨*proof*⟩

lemmas *sign-Min-neg* = *trans* [*OF sign-Min-lt-0 neg-def* [*symmetric*]]

lemma *bin-rest-trunc*:
!!*bin*. (*bin-rest* (*bintrunc n bin*)) = *bintrunc* (*n* − *1*) (*bin-rest bin*)
⟨*proof*⟩

lemma *bin-rest-power-trunc* [*rule-format*] :
(*bin-rest* ^ *k*) (*bintrunc n bin*) =
bintrunc (*n* − *k*) ((*bin-rest* ^ *k*) *bin*)
⟨*proof*⟩

lemma *bin-rest-trunc-i*:
bintrunc n (bin-rest bin) = bin-rest (bintrunc (Suc n) bin)
⟨*proof*⟩

lemma *bin-rest-strunc*:
!!*bin. bin-rest (sbintrunc (Suc n) bin) = sbintrunc n (bin-rest bin)*
⟨*proof*⟩

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

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

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

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

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

lemma *rco-alt*: *(f o g) ^ n o f = f o (g o f) ^ n*
⟨*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*]

4.5 Splitting and concatenation

consts
*bin-split :: nat => int => int * int*
primrec
Z : bin-split 0 w = (w, Numeral.Pls)
Suc : bin-split (Suc n) w = (let (w1, w2) = bin-split n (bin-rest w)
in (w1, w2 BIT bin-last w))

consts
bin-cat :: int => nat => int => int
primrec

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*

4.6 Miscellaneous lemmas

lemmas *funpow-minus-simp* =
trans [*OF gen-minus* [**where** *f* = *power f*] *funpow-Suc*, *standard*]

lemmas *funpow-pred-simp* [*simp*] =
funpow-minus-simp [*of number-of bin*, *simplified nobm1*, *standard*]

lemmas *replicate-minus-simp* =
trans [*OF gen-minus* [**where** *f* = %*n*. *replicate n x*] *replicate.replicate-Suc*,
standard]

lemmas *replicate-pred-simp* [*simp*] =
replicate-minus-simp [*of number-of bin*, *simplified nobm1*, *standard*]

lemmas *power-Suc-no* [*simp*] = *power-Suc* [*of number-of a*, *standard*]

lemmas *power-minus-simp* =
trans [*OF gen-minus* [**where** *f* = *power f*] *power-Suc*, *standard*]

lemmas *power-pred-simp* =
power-minus-simp [*of number-of bin*, *simplified nobm1*, *standard*]
lemmas *power-pred-simp-no* [*simp*] = *power-pred-simp* [**where** *f*= *number-of f*,
standard]

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* ==> *size y* = *Suc k* ==> *size list* = *k*
⟨*proof*⟩

lemma *size-Cons-lem-eq-bin*:
y = *xa* # *list* ==> *size y* = *number-of* (*Numeral.succ k*) ==>
size list = *number-of k*
⟨*proof*⟩

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

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

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

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

end

5 BitSyntax: Syntactic class for bitwise operations

theory *BitSyntax*
imports *Main*
begin

class *bit* = *type* +
fixes *bitNOT* :: $'a \Rightarrow 'a$
and *bitAND* :: $'a \Rightarrow 'a \Rightarrow 'a$
and *bitOR* :: $'a \Rightarrow 'a \Rightarrow 'a$
and *bitXOR* :: $'a \Rightarrow 'a \Rightarrow 'a$

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

notation
bitNOT (*NOT* - [*70*] *71*) **and**
bitAND (**infixr** *AND 64*) **and**
bitOR (**infixr** *OR 59*) **and**
bitXOR (**infixr** *XOR 59*)

Testing and shifting operations.

consts
test-bit :: $'a::bit \Rightarrow nat \Rightarrow bool$ (**infixl** !! *100*)
lsb :: $'a::bit \Rightarrow bool$
msb :: $'a::bit \Rightarrow bool$
set-bit :: $'a::bit \Rightarrow nat \Rightarrow bool \Rightarrow 'a$
set-bits :: $(nat \Rightarrow bool) \Rightarrow 'a::bit$ (**binder** *BITS 10*)
shiftl :: $'a::bit \Rightarrow nat \Rightarrow 'a$ (**infixl** << *55*)
shiftr :: $'a::bit \Rightarrow nat \Rightarrow 'a$ (**infixl** >> *55*)

5.1 Bitwise operations on type *bit*

instance *bit* :: *bit*
NOT-bit-def: $NOT\ x \equiv case\ x\ of\ bit.B0 \Rightarrow bit.B1 \mid bit.B1 \Rightarrow bit.B0$
AND-bit-def: $x\ AND\ y \equiv case\ x\ of\ bit.B0 \Rightarrow bit.B0 \mid bit.B1 \Rightarrow y$
OR-bit-def: $x\ OR\ y :: bit \equiv NOT\ (NOT\ x\ AND\ NOT\ y)$
XOR-bit-def: $x\ XOR\ y :: bit \equiv (x\ AND\ NOT\ y)\ OR\ (NOT\ x\ AND\ y)$
⟨*proof*⟩

lemma *bit-simps* [*simp*]:
$NOT\ bit.B0 = bit.B1$
$NOT\ bit.B1 = bit.B0$
$bit.B0\ AND\ y = bit.B0$
$bit.B1\ AND\ y = y$
$bit.B0\ OR\ y = y$
$bit.B1\ OR\ y = bit.B1$
$bit.B0\ XOR\ y = y$
$bit.B1\ XOR\ y = NOT\ y$
⟨*proof*⟩

lemma *bit-extra-simps* [*simp*]:
$x\ AND\ bit.B0 = bit.B0$
$x\ AND\ bit.B1 = x$
$x\ OR\ bit.B1 = bit.B1$
$x\ OR\ bit.B0 = x$
$x\ XOR\ bit.B1 = NOT\ x$
$x\ XOR\ bit.B0 = x$
⟨*proof*⟩

lemma *bit-ops-comm*:
$(x::bit)\ AND\ y = y\ AND\ x$
$(x::bit)\ OR\ y = y\ OR\ x$
$(x::bit)\ XOR\ y = y\ XOR\ x$
⟨*proof*⟩

lemma *bit-ops-same* [*simp*]:
$(x::bit)\ AND\ x = x$
$(x::bit)\ OR\ x = x$
$(x::bit)\ XOR\ x = bit.B0$
⟨*proof*⟩

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

end

6 BinOperations: Bitwise Operations on Binary Integers

theory *BinOperations* **imports** *BinGeneral BitSyntax*

begin

6.1 Logical operations

bit-wise logical operations on the int type

instance *int* :: *bit*
  int-not-def: $bitNOT \equiv bin\text{-}rec\ Numeral.Min\ Numeral.Pls$
    $(\lambda w\ b\ s.\ s\ BIT\ (NOT\ b))$
  int-and-def: $bitAND \equiv bin\text{-}rec\ (\lambda x.\ Numeral.Pls)\ (\lambda y.\ y)$
    $(\lambda w\ b\ s\ y.\ s\ (bin\text{-}rest\ y)\ BIT\ (b\ AND\ bin\text{-}last\ y))$
  int-or-def: $bitOR \equiv bin\text{-}rec\ (\lambda x.\ x)\ (\lambda y.\ Numeral.Min)$
    $(\lambda w\ b\ s\ y.\ s\ (bin\text{-}rest\ y)\ BIT\ (b\ OR\ bin\text{-}last\ y))$
  int-xor-def: $bitXOR \equiv bin\text{-}rec\ (\lambda x.\ x)\ bitNOT$
    $(\lambda w\ b\ s\ y.\ s\ (bin\text{-}rest\ y)\ BIT\ (b\ XOR\ bin\text{-}last\ y))$
  ⟨*proof*⟩

lemma *int-not-simps* [*simp*]:
  $NOT\ Numeral.Pls = Numeral.Min$
  $NOT\ Numeral.Min = Numeral.Pls$
  $NOT\ (w\ BIT\ b) = (NOT\ w)\ BIT\ (NOT\ b)$
  ⟨*proof*⟩

lemma *int-xor-Pls* [*simp*]:
  $Numeral.Pls\ XOR\ x = x$
  ⟨*proof*⟩

lemma *int-xor-Min* [*simp*]:
  $Numeral.Min\ XOR\ x = NOT\ x$
  ⟨*proof*⟩

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

lemma *int-xor-x-simps'*:
  $w\ XOR\ (Numeral.Pls\ BIT\ bit.B0) = w$
  $w\ XOR\ (Numeral.Min\ BIT\ bit.B1) = NOT\ w$
  ⟨*proof*⟩

lemmas *int-xor-extra-simps* [*simp*] = *int-xor-x-simps'* [*simplified arith-simps*]

lemma *int-or-Pls* [*simp*]:
  $Numeral.Pls\ OR\ x = x$

⟨*proof*⟩

lemma *int-or-Min* [*simp*]:
Numeral.Min OR x = Numeral.Min
⟨*proof*⟩

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

lemma *int-or-x-simps*′:
w OR (*Numeral.Pls BIT bit.B0*) = *w*
w OR (*Numeral.Min BIT bit.B1*) = *Numeral.Min*
⟨*proof*⟩

lemmas *int-or-extra-simps* [*simp*] = *int-or-x-simps*′ [*simplified arith-simps*]

lemma *int-and-Pls* [*simp*]:
Numeral.Pls AND x = Numeral.Pls
⟨*proof*⟩

lemma *int-and-Min* [*simp*]:
Numeral.Min AND x = x
⟨*proof*⟩

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

lemma *int-and-x-simps*′:
w AND (*Numeral.Pls BIT bit.B0*) = *Numeral.Pls*
w AND (*Numeral.Min BIT bit.B1*) = *w*
⟨*proof*⟩

lemmas *int-and-extra-simps* [*simp*] = *int-and-x-simps*′ [*simplified arith-simps*]

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 = Numeral.Pls*

⟨*proof*⟩

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

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

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 *bbw-assocs'*:
!!*y z*::*int*. (*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*)
⟨*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*⟩

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

lemma *le-int-or*:
!!*x*. *bin-sign y* = *Numeral.Pls* ==> *x* <= *x OR y*
⟨*proof*⟩

lemmas *int-and-le* =
xtr3 [*OF bbw-ao-absorbs* (*2*) [*THEN conjunct2*, *symmetric*] *le-int-or*]

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*⟩

lemma *bin-add-not*: *x* + *NOT x* = *Numeral.Min*
⟨*proof*⟩

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*]

6.2 Setting and clearing bits

consts
bin-sc :: *nat* => *bit* => *int* => *int*

primrec
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*]:
!!*w*. *bin-nth* (*bin-sc n b w*) *n* = (*b* = *bit.B1*)
⟨*proof*⟩

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

lemma *bin-sc-sc-diff*:
!!*w m*. *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*:
!!*w m*. *bin-nth* (*bin-sc n b w*) *m* = (*if m* = *n then b* = *bit.B1 else bin-nth w m*)
⟨*proof*⟩

lemma *bin-sc-nth* [*simp*]:

!!w. (bin-sc n (If (bin-nth w n) bit.B1 bit.B0) w) = w
⟨proof⟩

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

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

lemma bin-clr-le:
!!w. bin-sc n bit.B0 w <= w
⟨proof⟩

lemma bin-set-ge:
!!w. bin-sc n bit.B1 w >= w
⟨proof⟩

lemma bintr-bin-clr-le:
!!w m. bintrunc n (bin-sc m bit.B0 w) <= bintrunc n w
⟨proof⟩

lemma bintr-bin-set-ge:
!!w m. bintrunc n (bin-sc m bit.B1 w) >= bintrunc n w
⟨proof⟩

lemma bin-sc-FP [simp]: bin-sc n bit.B0 Numeral.Pls = Numeral.Pls
⟨proof⟩

lemma bin-sc-TM [simp]: bin-sc n bit.B1 Numeral.Min = Numeral.Min
⟨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, standard]

lemmas bin-sc-Suc-pred [simp] =
bin-sc-Suc-minus [of number-of bin, simplified nobm1, standard]

6.3 Operations on lists of booleans

consts
bin-to-bl :: nat => int => bool list

bin-to-bl-aux :: *nat* => *int* => *bool list* => *bool list*
bl-to-bin :: *bool list* => *int*
bl-to-bin-aux :: *int* => *bool list* => *int*

bl-of-nth :: *nat* => (*nat* => *bool*) => *bool list*

primrec
Nil : *bl-to-bin-aux w* [] = *w*
Cons : *bl-to-bin-aux w* (*b* # *bs*) =
bl-to-bin-aux (*w BIT* (*if b then bit.B1 else bit.B0*)) *bs*

primrec
Z : *bin-to-bl-aux 0 w bl* = *bl*
Suc : *bin-to-bl-aux* (*Suc n*) *w bl* =
bin-to-bl-aux n (*bin-rest w*) ((*bin-last w* = *bit.B1*) # *bl*)

defs
bin-to-bl-def : *bin-to-bl n w* == *bin-to-bl-aux n w* []
bl-to-bin-def : *bl-to-bin bs* == *bl-to-bin-aux Numeral.Pls bs*

primrec
Suc : *bl-of-nth* (*Suc n*) *f* = *f n* # *bl-of-nth n f*
Z : *bl-of-nth 0 f* = []

consts
takefill :: *'a* => *nat* => *'a list* => *'a list*
app2 :: (*'a* => *'b* => *'c*) => *'a list* => *'b list* => *'c list*

— takefill - like take but if argument list too short,
— extends result to get requested length
primrec
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*)

defs
app2-def : *app2 f as bs* == *map* (*split f*) (*zip as bs*)

6.4 Splitting and concatenation

— rcat and rsplit
consts
bin-rcat :: *nat* => *int list* => *int*
bin-rsplit-aux :: *nat* * *int list* * *nat* * *int* => *int list*
bin-rsplit :: *nat* => (*nat* * *int*) => *int list*
bin-rsplitl-aux :: *nat* * *int list* * *nat* * *int* => *int list*
bin-rsplitl :: *nat* => (*nat* * *int*) => *int list*

recdef *bin-rsplit-aux measure* (*fst o snd o snd*)
  bin-rsplit-aux (*n*, *bs*, (*m*, *c*)) =
    (*if m* = *0* | *n* = *0 then bs else*
      let (*a*, *b*) = *bin-split n c*
      in bin-rsplit-aux (*n*, *b* # *bs*, (*m* − *n*, *a*)))

recdef *bin-rsplitl-aux measure* (*fst o snd o snd*)
  bin-rsplitl-aux (*n*, *bs*, (*m*, *c*)) =
    (*if m* = *0* | *n* = *0 then bs else*
      let (*a*, *b*) = *bin-split* (*min m n*) *c*
      in bin-rsplitl-aux (*n*, *b* # *bs*, (*m* − *n*, *a*)))

defs
  bin-rcat-def : *bin-rcat n bs* == *foldl* (%*u v*. *bin-cat u n v*) *Numeral.Pls bs*
  bin-rsplit-def : *bin-rsplit n w* == *bin-rsplit-aux* (*n*, [], *w*)
  bin-rsplitl-def : *bin-rsplitl n w* == *bin-rsplitl-aux* (*n*, [], *w*)

declare *bin-rsplit-aux.simps* [*simp del*]
declare *bin-rsplitl-aux.simps* [*simp del*]

lemma *bin-sign-cat*:
  !!*y*. *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*:
  !!*n y*. *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*:
  !!*b c*. *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*:
  !!*z*. *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*: !!*z m*.
  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-Pls* [*simp*]:
!!w. *bin-cat Numeral.Pls n w* = *bintrunc n w*
⟨*proof*⟩

lemma *bintr-cat1*:
!!b. *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*]:
!!b. *bin-cat a n* (*bintrunc n b*) = *bin-cat a n b*
⟨*proof*⟩

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

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

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

lemma *bin-split-Pls* [*simp*]:
bin-split n Numeral.Pls = (*Numeral.Pls*, *Numeral.Pls*)
⟨*proof*⟩

lemma *bin-split-Min* [*simp*]:
bin-split n Numeral.Min = (*Numeral.Min*, *bintrunc n Numeral.Min*)
⟨*proof*⟩

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

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

lemma *bin-cat-num*:
$!!b.\ bin\text{-}cat\ a\ n\ b = a * 2\ \hat{}\ n + bintrunc\ n\ b$
⟨*proof*⟩

lemma *bin-split-num*:
$!!b.\ bin\text{-}split\ n\ b = (b\ div\ 2\ \hat{}\ n,\ b\ mod\ 2\ \hat{}\ n)$
⟨*proof*⟩

6.5 Miscellaneous lemmas

lemma *nth-2p-bin*:
$!!m.\ bin\text{-}nth\ (2\ \hat{}\ n)\ m = (m = n)$
⟨*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*⟩

end

7 BinBoolList: Bool lists and integers

theory *BinBoolList* **imports** *BinOperations* **begin**

7.1 Arithmetic in terms of bool lists

consts
rbl-succ :: *bool list* => *bool list*
rbl-pred :: *bool list* => *bool list*
rbl-add :: *bool list* => *bool list* => *bool list*
rbl-mult :: *bool list* => *bool list* => *bool list*

primrec
Nil: *rbl-succ Nil* = *Nil*
Cons: *rbl-succ* (*x* # *xs*) = (*if x then False* # *rbl-succ xs else True* # *xs*)

primrec
Nil : *rbl-pred Nil* = *Nil*
Cons : *rbl-pred* (*x* # *xs*) = (*if x then False* # *xs else True* # *rbl-pred xs*)

primrec
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
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 *tl-take*: *tl (take n l) = take (n − 1) (tl l)*
⟨*proof*⟩

lemma *take-butlast [rule-format] :*
ALL n. n < length l --> take n (butlast l) = take n l
⟨*proof*⟩

lemma *butlast-take [rule-format] :*
ALL n. n <= length l --> butlast (take n l) = take (n − 1) l
⟨*proof*⟩

lemma *butlast-drop [rule-format] :*
ALL n. butlast (drop n l) = drop n (butlast l)
⟨*proof*⟩

lemma *butlast-power*:
(butlast ^ n) bl = take (length bl − n) bl
⟨*proof*⟩

lemma *bin-to-bl-aux-Pls-minus-simp*:
0 < n ==> bin-to-bl-aux n Numeral.Pls bl =
bin-to-bl-aux (n − 1) Numeral.Pls (False # bl)
⟨*proof*⟩

lemma *bin-to-bl-aux-Min-minus-simp*:
0 < n ==> bin-to-bl-aux n Numeral.Min bl =
bin-to-bl-aux (n − 1) Numeral.Min (True # bl)
⟨*proof*⟩

lemma *bin-to-bl-aux-Bit-minus-simp*:
0 < n ==> bin-to-bl-aux n (w BIT b) bl =
bin-to-bl-aux (n − 1) w ((b = bit.B1) # bl)
⟨*proof*⟩

declare *bin-to-bl-aux-Pls-minus-simp [simp]*
bin-to-bl-aux-Min-minus-simp [simp]
bin-to-bl-aux-Bit-minus-simp [simp]

lemma *bl-to-bin-aux-append [rule-format] :*
ALL w. bl-to-bin-aux w (bs @ cs) = bl-to-bin-aux (bl-to-bin-aux w bs) cs

⟨*proof*⟩

lemma *bin-to-bl-aux-append* [*rule-format*] :
ALL w bs. 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 (bl-to-bin bs) cs
⟨*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*: *bin-to-bl 0 bs = []*
⟨*proof*⟩

lemma *size-bin-to-bl-aux* [*rule-format*] :
ALL w bs. size (bin-to-bl-aux n w bs) = n + length bs
⟨*proof*⟩

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

lemma *bin-bl-bin′* [*rule-format*] :
ALL w bs. bl-to-bin (bin-to-bl-aux n w bs) =
bl-to-bin-aux (bintrunc n w) bs
⟨*proof*⟩

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

lemma *bl-bin-bl′* [*rule-format*] :
ALL w n. bin-to-bl (n + length bs) (bl-to-bin-aux w bs) =
bin-to-bl-aux n w bs
⟨*proof*⟩

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

declare
bin-to-bl-0 [*simp*]
size-bin-to-bl [*simp*]
bin-bl-bin [*simp*]
bl-bin-bl [*simp*]

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*: *bl-to-bin* (*False* # *bl*) = *bl-to-bin bl*
⟨*proof*⟩

lemma *bl-to-bin-Nil*: *bl-to-bin* [] = *Numeral.Pls*
⟨*proof*⟩

lemma *bin-to-bl-Pls-aux* [*rule-format*] :
ALL bl. bin-to-bl-aux n Numeral.Pls bl = *replicate n False* @ *bl*
⟨*proof*⟩

lemma *bin-to-bl-Pls*: *bin-to-bl n Numeral.Pls* = *replicate n False*
⟨*proof*⟩

lemma *bin-to-bl-Min-aux* [*rule-format*] :
ALL bl. bin-to-bl-aux n Numeral.Min bl = *replicate n True* @ *bl*
⟨*proof*⟩

lemma *bin-to-bl-Min*: *bin-to-bl n Numeral.Min* = *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*:
n <= *m* ==> *bin-to-bl n* (*bintrunc m w*) = *bin-to-bl n w*
⟨*proof*⟩

declare
bin-to-bl-trunc [*simp*]
bl-to-bin-False [*simp*]
bl-to-bin-Nil [*simp*]

lemma *bin-to-bl-aux-bintr* [*rule-format*] :
ALL m bin bl. bin-to-bl-aux n (*bintrunc m bin*) *bl* =
replicate (*n* − *m*) *False* @ *bin-to-bl-aux* (*min n m*) *bin bl*
⟨*proof*⟩

lemmas *bin-to-bl-bintr* =
bin-to-bl-aux-bintr [**where** *bl* = [], *folded bin-to-bl-def*]

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

lemma *len-bin-to-bl-aux* [*rule-format*] :
ALL w bs. length (*bin-to-bl-aux n w bs*) = *n* + *length bs*
⟨*proof*⟩

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

lemma *sign-bl-bin'* [*rule-format*] :
ALL w. bin-sign (*bl-to-bin-aux w bs*) = *bin-sign w*
⟨*proof*⟩

lemma *sign-bl-bin*: *bin-sign* (*bl-to-bin bs*) = *Numeral.Pls*
⟨*proof*⟩

lemma *bl-sbin-sign-aux* [*rule-format*] :
ALL w bs. hd (*bin-to-bl-aux* (*Suc n*) *w bs*) =
(*bin-sign* (*sbintrunc n w*) = *Numeral.Min*)
⟨*proof*⟩

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

lemma *bin-nth-of-bl-aux* [*rule-format*] :
ALL w. bin-nth (*bl-to-bin-aux w bl*) *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* [*rule-format*] : *ALL m w. n* < *m* −−>
bin-nth w n = *nth* (*rev* (*bin-to-bl m w*)) *n*
⟨*proof*⟩

lemma *nth-rev* [*rule-format*] :
n < *length xs* −−> *rev xs* ! *n* = *xs* ! (*length xs* − *1* − *n*)
⟨*proof*⟩

lemmas *nth-rev-alt* = *nth-rev* [**where** *xs* = *rev ys*, *simplified*, *standard*]

lemma *nth-bin-to-bl-aux* [*rule-format*] :
ALL w n bl. 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* ==> (*bin-to-bl m w*) ! *n* = *bin-nth w* (*m* − *Suc n*)
⟨*proof*⟩

lemma *bl-to-bin-lt2p-aux* [*rule-format*] :
ALL w. bl-to-bin-aux w bs < (*w* + *1*) ∗ (*2* ^ *length bs*)
⟨*proof*⟩

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

lemma *bl-to-bin-ge2p-aux* [*rule-format*] :
*ALL w. bl-to-bin-aux w bs >= 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*⟩

lemmas *butlast-bin-rest = butlast-rest-bin*
[**where** *w=bl-to-bin bl* **and** *n=length bl*, *simplified*, *standard*]

lemma *butlast-rest-bl2bin-aux* [*rule-format*] :
ALL w. bl ~= [] −−>
bl-to-bin-aux w (butlast bl) = bin-rest (bl-to-bin-aux w bl)
⟨*proof*⟩

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

lemma *trunc-bl2bin-aux* [*rule-format*] :
ALL w. bintrunc m (bl-to-bin-aux w bl) =
bl-to-bin-aux (bintrunc (m − length bl) w) (drop (length bl − m) bl)
⟨*proof*⟩

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

lemmas *trunc-bl2bin-len* [*simp*] =
trunc-bl2bin [*of length bl bl*, *simplified*, *standard*]

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

lemma *nth-rest-power-bin* [*rule-format*] :
ALL n. 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'* [*rule-format*] :
ALL w. *size xs* > *0* −−> *last xs* = (*bin-last* (*bl-to-bin-aux w xs*) = *bit*.*B1*)
⟨*proof*⟩

lemma *last-bin-last*:
size xs > *0* ==> *last xs* = (*bin-last* (*bl-to-bin xs*) = *bit*.*B1*)
⟨*proof*⟩

lemma *bin-last-last*:
bin-last w = (*if last* (*bin-to-bl* (*Suc n*) *w*) *then bit*.*B1 else bit*.*B0*)
⟨*proof*⟩

lemma *app2-Nil* [*simp*]: *app2 f* [] *ys* = []
⟨*proof*⟩

lemma *app2-Cons* [*simp*]:
app2 f (*x* # *xs*) (*y* # *ys*) = *f x y* # *app2 f xs ys*
⟨*proof*⟩

lemma *bl-xor-aux-bin* [*rule-format*] : *ALL v w bs cs*.
app2 (%*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*) (*app2* (%*x y*. *x* ~= *y*) *bs cs*)
⟨*proof*⟩

lemma *bl-or-aux-bin* [*rule-format*] : *ALL v w bs cs*.
app2 (*op* |) (*bin-to-bl-aux n v bs*) (*bin-to-bl-aux n w cs*) =
bin-to-bl-aux n (*v OR w*) (*app2* (*op* |) *bs cs*)
⟨*proof*⟩

lemma *bl-and-aux-bin* [*rule-format*] : *ALL v w bs cs*.
app2 (*op* &) (*bin-to-bl-aux n v bs*) (*bin-to-bl-aux n w cs*) =
bin-to-bl-aux n (*v AND w*) (*app2* (*op* &) *bs cs*)
⟨*proof*⟩

lemma *bl-not-aux-bin* [*rule-format*] :
ALL w cs. *map Not* (*bin-to-bl-aux n w cs*) =
bin-to-bl-aux n (*NOT w*) (*map Not cs*)
⟨*proof*⟩

lemmas *bl-not-bin* = *bl-not-aux-bin*
[**where** *cs* = [], *unfolded bin-to-bl-def* [*symmetric*] *map*.*simps*]

lemmas *bl-and-bin* = *bl-and-aux-bin* [**where** *bs*=[] **and** *cs*=[], *unfolded app2-Nil, folded bin-to-bl-def*]

lemmas *bl-or-bin* = *bl-or-aux-bin* [**where** *bs*=[] **and** *cs*=[],
unfolded app2-Nil, folded bin-to-bl-def]

lemmas *bl-xor-bin* = *bl-xor-aux-bin* [**where** *bs*=[] **and** *cs*=[],
unfolded app2-Nil, folded bin-to-bl-def]

lemma *drop-bin2bl-aux* [*rule-format*] :
ALL m bin bs. 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* [*rule-format*] :
ALL w bs. take m (bin-to-bl-aux m w bs) = bin-to-bl m w
⟨*proof*⟩

lemma *take-bin2bl-lem* [*rule-format*] :
ALL w bs. take m (bin-to-bl-aux (m + n) w bs) =
take m (bin-to-bl (m + n) w)
⟨*proof*⟩

lemma *bin-split-take* [*rule-format*] :
ALL b c. 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* [*rule-format*] : *ALL m l. m < n −−>*
takefill fill n l ! m = (if m < length l then l ! m else fill)
⟨*proof*⟩

lemma *takefill-alt* [*rule-format*] :
ALL l. 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′* [*rule-format*] :
ALL l n. 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*]:
$m \leq n \Longrightarrow$ *takefill x m* (*takefill x n l*) = *takefill x m l*
⟨*proof*⟩

lemma *take-takefill* [*simp*]:
$m \leq n \Longrightarrow$ *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*⟩

lemmas *bl-bin-bl-rep-drop* =
bl-bin-bl-rtf [*simplified takefill-alt*,
simplified, *simplified rev-take*, *simplified*]

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*], *standard*]

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*], *standard*]

lemmas *takefill-pred-simps* [*simp*] =
takefill-minus-simps [**where** *n=number-of bin*, *simplified nobm1*, *standard*]

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

lemmas *bl-to-bin-aux-alt* =
bl-to-bin-aux-cat [**where** *nv* = *0* **and** *v* = *Numeral.Pls*,
simplified bl-to-bin-def [*symmetric*], *simplified*]

lemmas *bin-to-bl-cat* =
bin-to-bl-aux-cat [**where** *bs* = [], *folded bin-to-bl-def*]

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*]

lemmas *bl-to-bin-app-cat* = *bl-to-bin-aux-app-cat*
[**where** *w* = *Numeral.Pls*, *folded bl-to-bin-def*]

lemmas *bin-to-bl-cat-app* = *bin-to-bl-aux-cat-app*
[**where** *bs* = [], *folded bin-to-bl-def*]

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*: $(bl\text{-}to\text{-}bin\ (True\ \#\ replicate\ n\ False)) =$
$Numeral.succ\ (bl\text{-}to\text{-}bin\ (replicate\ n\ True))$
⟨*proof*⟩

lemma *length-bl-of-nth* [*simp*]: $length\ (bl\text{-}of\text{-}nth\ n\ f) = n$
⟨*proof*⟩

lemma *nth-bl-of-nth* [*simp*]:
$m < n \Longrightarrow rev\ (bl\text{-}of\text{-}nth\ n\ f)\ !\ m = f\ m$
⟨*proof*⟩

lemma *bl-of-nth-inj*:
$(!!k.\ k < n ==> f\ k = g\ k) ==> bl\text{-}of\text{-}nth\ n\ f = bl\text{-}of\text{-}nth\ n\ g$
⟨*proof*⟩

lemma *bl-of-nth-nth-le* [*rule-format*] : $ALL\ xs.$
$length\ xs >= n --> bl\text{-}of\text{-}nth\ n\ (nth\ (rev\ xs)) = drop\ (length\ xs - n)\ xs$
⟨*proof*⟩

lemmas *bl-of-nth-nth* [*simp*] = *order-refl* [*THEN bl-of-nth-nth-le*, *simplified*]

lemma *size-rbl-pred*: $length\ (rbl\text{-}pred\ bl) = length\ bl$
⟨*proof*⟩

lemma *size-rbl-succ*: $length\ (rbl\text{-}succ\ bl) = length\ bl$
⟨*proof*⟩

lemma *size-rbl-add*:
$!!cl.\ length\ (rbl\text{-}add\ bl\ cl) = length\ bl$
⟨*proof*⟩

lemma *size-rbl-mult*:
$!!cl.\ length\ (rbl\text{-}mult\ bl\ cl) = length\ bl$
⟨*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*:
$!!bin.\ rbl\text{-}pred\ (rev\ (bin\text{-}to\text{-}bl\ n\ bin)) = rev\ (bin\text{-}to\text{-}bl\ n\ (Numeral.pred\ bin))$
⟨*proof*⟩

lemma *rbl-succ*:

!!*bin. rbl-succ* (*rev* (*bin-to-bl n bin*)) = *rev* (*bin-to-bl n* (*Numeral.succ bin*))
⟨*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*:
!!*blb. length blb* >= *length bla* ==>
rbl-add bla (*blb* @ *blc*) = *rbl-add bla blb*
⟨*proof*⟩

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

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

lemma *rbl-mult-app2*:
!!*blb. length blb* >= *length bla* ==>
rbl-mult bla (*blb* @ *blc*) = *rbl-mult bla blb*
⟨*proof*⟩

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

lemma *rbl-mult-gt1*:
m >= *length bl* ==> *rbl-mult bl* (*rev* (*bin-to-bl m binb*)) =
rbl-mult bl (*rev* (*bin-to-bl* (*length bl*) *binb*))
⟨*proof*⟩

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

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

lemma *rbbl-Cons*:
b # *rev* (*bin-to-bl n x*) = *rev* (*bin-to-bl* (*Suc n*) (*x BIT If b bit.B1 bit.B0*))
⟨*proof*⟩

lemma *rbl-mult*: !!*bina binb*.
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*)) =
(*ALL 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*) =
(*ALL ws*. *length ws* = *Suc* (*length ys*) −−> *ws* = *False* # *rbl-mult ys xs* −−>

(*y* −−> *P* (*rbl-add ws xs*)) & (~ *y* −−> *P ws*))
⟨*proof*⟩

lemma *and-len*: *xs* = *ys* ==> *xs* = *ys* & *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*, *standard*]

lemmas *tl-Nil* = *tl.simps* (*1*)
lemmas *tl-Cons* = *tl.simps* (*2*)

7.2 Repeated splitting or concatenation

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

lemma *bin-cat-foldl-lem* [*rule-format*] :
ALL x. *foldl* (%*u*. *bin-cat u n*) *x xs* =
bin-cat x (*size xs* * *n*) (*foldl* (%*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* = *split-if* [**where** *P* = *op* = *l*,
THEN iffD1, *THEN conjunct1*, *THEN mp*, *standard*]
lemmas *th-if-simp2* = *split-if* [**where** *P* = *op* = *l*,
THEN iffD1, *THEN conjunct2*, *THEN mp*, *standard*]

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*, *bs* @ *cs*, *m*, *c*) = *bin-rsplit-aux* (*n*, *bs*, *m*, *c*) @ *cs*
⟨*proof*⟩

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

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

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

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

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

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

lemma *bin-split-pred-simp* [*simp*]:
(*0*::*nat*) < *number-of bin* ⟹
bin-split (*number-of bin*) *w* =
(*let* (*w1*, *w2*) = *bin-split* (*number-of* (*Numeral.pred bin*)) (*bin-rest w*)
in (*w1*, *w2 BIT bin-last w*))
⟨*proof*⟩

declare *bin-split-pred-simp* [*simp*]

lemma *bin-rsplit-aux-simp-alt*:
bin-rsplit-aux (*n*, *bs*, *m*, *c*) =
(*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* [*THEN meta-eq-to-obj-eq*]
bin-rsplit-aux-simp-alt, *standard*]

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

lemma *bin-rsplit-size-sign′* [*rule-format*] :
n > *0* ==> (*ALL nw w*. *rev sw* = *bin-rsplit n* (*nw*, *w*) −−>)

(*ALL 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*]],
standard]

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

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

lemma *bin-rsplit-l* [*rule-format*] :
ALL 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*] :
ALL ws m. *n* ≠ *0* −−> *ws* = *bin-rsplit-aux* (*n*, *bs*, *nw*, *w*) −−>
(*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* [*rule-format*] :
n≠*0* −−> *length* (*bin-rsplit-aux* (*n*, *cs*, *nw*, *w*)) =
(*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* [*rule-format*] :
n≠*0* ==> (*ALL v bs*. *length bs* = *length cs* −−>
length (*bin-rsplit-aux* (*n*, *bs*, *nw*, *v*)) =
length (*bin-rsplit-aux* (*n*, *cs*, *nw*, *w*)))
⟨*proof*⟩

lemma *bin-rsplit-len-indep*:
$n{\neq}0$ ==> *length* (*bin-rsplit* n (*nw*, v)) = *length* (*bin-rsplit* n (*nw*, w))
⟨*proof*⟩

end

8 TdThs: Type Definition Theorems

theory *TdThs* **imports** *Main* **begin**

9 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

lemmas *Rep'* [*iff*] = *Rep* [*simplified*]

declare *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 o f = g ==> *Abs* o 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 o Abs = g ==> g o Rep = f
⟨proof⟩

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

lemma *set-Rep-Abs*: A = range (Rep o 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 o fa = fr o Rep | fa o Abs = Abs o fr ==> Abs o fr o Rep = fa
⟨proof⟩

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

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

end

interpretation *nat-int*: type-definition [int nat Collect (op <= 0)]
⟨proof⟩
declare *Nat.induct* [case-names 0 Suc, induct type]
declare *Nat.exhaust* [case-names 0 Suc, cases type]

9.1 Extended form of type definition predicate

lemma *td-conds*:
norm o norm = norm ==> (fr o norm = norm o fr) =
(norm o fr o norm = fr o norm & norm o fr o norm = norm o fr)
⟨proof⟩

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

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

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

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

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

lemma *eq-norm′*: *Rep o 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 o norm = *Abs*
norm o Rep = *Rep*
norm o norm = *norm*
⟨*proof*⟩

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

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

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

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

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

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

end

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

end

10 WordDefinition: Definition of Word Type

theory *WordDefinition* **imports** *Size BinBoolList TdThs* **begin**

typedef (**open** *word*) *'a word*
  = {(*0::int*) ..< *2^len-of TYPE('a::len0)*} ⟨*proof*⟩

instance *word* :: (*len0*) *number* ⟨*proof*⟩
instance *word* :: (*type*) *minus* ⟨*proof*⟩
instance *word* :: (*type*) *plus* ⟨*proof*⟩
instance *word* :: (*type*) *one* ⟨*proof*⟩
instance *word* :: (*type*) *zero* ⟨*proof*⟩
instance *word* :: (*type*) *times* ⟨*proof*⟩
instance *word* :: (*type*) *Divides.div* ⟨*proof*⟩
instance *word* :: (*type*) *power* ⟨*proof*⟩
instance *word* :: (*type*) *ord* ⟨*proof*⟩
instance *word* :: (*type*) *size* ⟨*proof*⟩
instance *word* :: (*type*) *inverse* ⟨*proof*⟩
instance *word* :: (*type*) *bit* ⟨*proof*⟩

10.1 Type conversions and casting

constdefs

— representation of words using unsigned or signed bins, only difference in these is the type class

word-of-int :: *int* => *'a* :: *len0 word*
word-of-int w == *Abs-word* (*bintrunc* (*len-of TYPE* (*'a*)) *w*)

— uint and sint cast a word to an integer, uint treats the word as unsigned, sint treats the most-significant-bit as a sign bit

uint :: *'a* :: *len0 word* => *int*
uint w == *Rep-word w*
sint :: *'a* :: *len word* => *int*
sint-uint: *sint w* == *sbintrunc* (*len-of TYPE* (*'a*) − *1*) (*uint w*)
unat :: *'a* :: *len0 word* => *nat*
unat w == *nat* (*uint w*)

— the sets of integers representing the words

uints :: *nat* => *int set*
uints n == *range* (*bintrunc n*)
sints :: *nat* => *int set*
sints n == *range* (*sbintrunc* (*n* − *1*))
unats :: *nat* => *nat set*
unats n == {*i*. *i* < *2* ^ *n*}
norm-sint :: *nat* => *int* => *int*
norm-sint n w == (*w* + *2* ^ (*n* − *1*)) *mod 2* ^ *n* − *2* ^ (*n* − *1*)

— cast a word to a different length

scast :: *'a* :: *len word* => *'b* :: *len word*
scast w == *word-of-int* (*sint w*)
ucast :: *'a* :: *len0 word* => *'b* :: *len0 word*
ucast w == *word-of-int* (*uint w*)

— whether a cast (or other) function is to a longer or shorter length

source-size :: (*'a* :: *len0 word* => *'b*) => *nat*
source-size c == *let arb* = *arbitrary* ; *x* = *c arb in size arb*
target-size :: (*'a* => *'b* :: *len0 word*) => *nat*
target-size c == *size* (*c arbitrary*)
is-up :: (*'a* :: *len0 word* => *'b* :: *len0 word*) => *bool*
is-up c == *source-size c* <= *target-size c*
is-down :: (*'a* :: *len0 word* => *'b* :: *len0 word*) => *bool*
is-down c == *target-size c* <= *source-size c*

constdefs

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

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

defs (**overloaded**)
word-size: *size* (*w* :: *'a* :: *len0 word*) == *len-of TYPE*(*'a*)
word-number-of-def: *number-of w* == *word-of-int w*

constdefs
word-int-case :: (*int* => *'b*) => (*'a* :: *len0 word*) => *'b*
word-int-case f w == *f* (*uint w*)

syntax
of-int :: *int* => *'a*
translations
case x of of-int y => *b* == *word-int-case* (%*y*. *b*) *x*

10.2 Arithmetic operations

defs (**overloaded**)
word-1-wi: (*1* :: (*'a* :: *len0*) *word*) == *word-of-int 1*
word-0-wi: (*0* :: (*'a* :: *len0*) *word*) == *word-of-int 0*

word-le-def: *a* <= *b* == *uint a* <= *uint b*
word-less-def: *x* < *y* == *x* <= *y* & *x* ~= (*y* :: *'a* :: *len0 word*)

constdefs
word-succ :: *'a* :: *len0 word* => *'a word*
word-succ a == *word-of-int* (*Numeral.succ* (*uint a*))

word-pred :: *'a* :: *len0 word* => *'a word*
word-pred a == *word-of-int* (*Numeral.pred* (*uint a*))

udvd :: *'a*::*len word* => *'a*::*len word* => *bool* (**infixl** *udvd 50*)
a udvd b == *EX n*>=*0*. *uint b* = *n* * *uint a*

word-sle :: *'a* :: *len word* => *'a word* => *bool* ((-/ <=*s* -) [*50*, *51*] *50*)
a <=*s b* == *sint a* <= *sint b*

word-sless :: *'a* :: *len word* => *'a word* => *bool* ((-/ <*s* -) [*50*, *51*] *50*)
(*x* <*s y*) == (*x* <=*s y* & *x* ~= *y*)

consts
word-power :: *'a* :: *len0 word* => *nat* => *'a word*
primrec
word-power a 0 = *1*
word-power a (*Suc n*) = *a* * *word-power a n*

defs (**overloaded**)
word-pow: *power* == *word-power*

word-add-def: $a + b == word\text{-}of\text{-}int\ (uint\ a + uint\ b)$
word-sub-wi: $a - b == word\text{-}of\text{-}int\ (uint\ a - uint\ b)$
word-minus-def: $-\ a == word\text{-}of\text{-}int\ (-\ uint\ a)$
word-mult-def: $a * b == word\text{-}of\text{-}int\ (uint\ a * uint\ b)$
word-div-def: $a\ div\ b == word\text{-}of\text{-}int\ (uint\ a\ div\ uint\ b)$
word-mod-def: $a\ mod\ b == word\text{-}of\text{-}int\ (uint\ a\ mod\ uint\ b)$

10.3 Bit-wise operations

defs (overloaded)
word-and-def:
(a::'a::len0 word) AND b == word-of-int (uint a AND uint b)

word-or-def:
(a::'a::len0 word) OR b == word-of-int (uint a OR uint b)

word-xor-def:
(a::'a::len0 word) XOR b == word-of-int (uint a XOR uint b)

word-not-def:
NOT (a::'a::len0 word) == word-of-int (NOT (uint a))

word-test-bit-def:
test-bit (a::'a::len0 word) == bin-nth (uint a)

word-set-bit-def:
set-bit (a::'a::len0 word) n x ==
word-of-int (bin-sc n (If x bit.B1 bit.B0) (uint a))

word-set-bits-def:
(BITS n. f n)::'a::len0 word == of-bl (bl-of-nth (len-of TYPE ('a)) f)

word-lsb-def:
lsb (a::'a::len0 word) == bin-last (uint a) = bit.B1

word-msb-def:
msb (a::'a::len word) == bin-sign (sint a) = Numeral.Min

constdefs
setBit :: 'a :: len0 word => nat => 'a word
setBit w n == set-bit w n True

clearBit :: 'a :: len0 word => nat => 'a word
clearBit w n == set-bit w n False

10.4 Shift operations

constdefs
shiftl1 :: 'a :: len0 word => 'a word

shiftl1 w == word-of-int (uint w BIT bit.B0)

— shift right as unsigned or as signed, ie logical or arithmetic
shiftr1 :: 'a :: len0 word => 'a word
shiftr1 w == word-of-int (bin-rest (uint w))

sshiftr1 :: 'a :: len word => 'a word
sshiftr1 w == word-of-int (bin-rest (sint w))

bshiftr1 :: bool => 'a :: len word => 'a word
bshiftr1 b w == of-bl (b # butlast (to-bl w))

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

mask :: nat => 'a::len word
mask n == (1 << n) − 1

revcast :: 'a :: len0 word => 'b :: len0 word
revcast w == of-bl (takefill False (len-of TYPE('b)) (to-bl w))

slice1 :: nat => 'a :: len0 word => 'b :: len0 word
slice1 n w == of-bl (takefill False n (to-bl w))

slice :: nat => 'a :: len0 word => 'b :: len0 word
slice n w == slice1 (size w − n) w

defs (**overloaded**)
shiftl-def: (w::'a::len0 word) << n == (shiftl1 ^ n) w
shiftr-def: (w::'a::len0 word) >> n == (shiftr1 ^ n) w

10.5 Rotation

constdefs
rotater1 :: 'a list => 'a list
rotater1 ys ==
  case ys of [] => [] | x # xs => last ys # butlast ys

rotater :: nat => 'a list => 'a list
rotater n == rotater1 ^ n

word-rotr :: nat => 'a :: len0 word => 'a :: len0 word
word-rotr n w == of-bl (rotater n (to-bl w))

word-rotl :: nat => 'a :: len0 word => 'a :: len0 word
word-rotl n w == of-bl (rotate n (to-bl w))

word-roti :: int => 'a :: len0 word => 'a :: len0 word

word-roti i w == if i >= 0 then word-rotr (nat i) w
else word-rotl (nat (− i)) w

10.6 Split and cat operations

constdefs
word-cat :: 'a :: len0 word => 'b :: len0 word => 'c :: len0 word
word-cat a b == word-of-int (bin-cat (uint a) (len-of TYPE ('b)) (uint b))

*word-split :: 'a :: len0 word => ('b :: len0 word) * ('c :: len0 word)*
word-split a ==
case bin-split (len-of TYPE ('c)) (uint a) of
(u, v) => (word-of-int u, word-of-int v)

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

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

constdefs
— Largest representable machine integer.
max-word :: 'a::len word
max-word ≡ word-of-int (2^len-of TYPE('a) − 1)

consts
of-bool :: bool ⇒ 'a::len word
primrec
of-bool False = 0
of-bool True = 1

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

lemmas *word-size-gt-0 [iff] =*
xtr1 [OF word-size [THEN meta-eq-to-obj-eq] len-gt-0, standard]
lemmas *lens-gt-0 = word-size-gt-0 len-gt-0*
lemmas *lens-not-0 [iff] = lens-gt-0 [THEN gr-implies-not0, standard]*

lemma *uints-num*: $uints\ n = \{i.\ 0 \le i \wedge i < 2\ \hat{}\ n\}$
⟨*proof*⟩

lemma *sints-num*: $sints\ n = \{i.\ -(2\ \hat{}\ (n - 1)) \le i \wedge i < 2\ \hat{}\ (n - 1)\}$
⟨*proof*⟩

lemmas *atLeastLessThan-alt = atLeastLessThan-def [unfolded* *atLeast-def lessThan-def Collect-conj-eq* [*symmetric*]]

lemma *mod-in-reps*: *m* > *0* ==> *y mod m* : {*0*::*int* ..< *m*}
⟨*proof*⟩

lemma
Rep-word-0:*0* <= *Rep-word x* **and**
Rep-word-lt: *Rep-word* (*x*::′*a*::*len0 word*) < *2* ˆ *len-of TYPE*(′*a*)
⟨*proof*⟩

lemma *Rep-word-mod-same*:
Rep-word x mod 2 ˆ *len-of TYPE*(′*a*) = *Rep-word* (*x*::′*a*::*len0 word*)
⟨*proof*⟩

lemma *td-ext-uint*:
td-ext (*uint* :: ′*a word* => *int*) *word-of-int* (*uints* (*len-of TYPE*(′*a*::*len0*)))
(%*w*::*int*. *w mod 2* ˆ *len-of TYPE*(′*a*))
⟨*proof*⟩

lemmas *int-word-uint* = *td-ext-uint* [*THEN td-ext.eq-norm*, *standard*]

interpretation *word-uint*:
td-ext [*uint*::′*a*::*len0 word* ⇒ *int*
word-of-int
uints (*len-of TYPE*(′*a*::*len0*))
λ*w*. *w mod 2* ˆ *len-of TYPE*(′*a*::*len0*)]
⟨*proof*⟩

lemmas *td-uint* = *word-uint.td-thm*

lemmas *td-ext-ubin* = *td-ext-uint*
[*simplified len-gt-0 no-bintr-alt1* [*symmetric*]]

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*⟩

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* :: ′*a* :: *len word*))
⟨*proof*⟩

lemma *bintr-uint'*:
n >= size w ==> bintrunc n (uint w) = uint w
⟨*proof*⟩

lemma *wi-bintr'*:
wb = word-of-int bin ==> n >= size wb ==>
word-of-int (bintrunc n bin) = wb
⟨*proof*⟩

lemmas *bintr-uint = bintr-uint' [unfolded word-size]*
lemmas *wi-bintr = wi-bintr' [unfolded word-size]*

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*⟩

lemmas *td-ext-sint = td-ext-sbin*
[simplified len-gt-0 no-sbintr-alt2 Suc-pred' [symmetric]]

interpretation *word-sint*:
td-ext [sint ::'a::len word => int
word-of-int
sints (len-of TYPE('a::len))
%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, standard]*

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

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

lemma *word-no-wi*: *number-of = word-of-int*
⟨*proof*⟩

lemma *to-bl-def'*:
(to-bl :: 'a :: len0 word => bool list) =
bin-to-bl (len-of TYPE('a)) o uint

⟨*proof*⟩

lemmas *word-reverse-no-def* [*simp*] = *word-reverse-def* [*of number-of w*, *standard*]

lemmas *uints-mod* = *uints-def* [*unfolded no-bintr-alt1*]

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

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

lemma *unat-bintrunc*:
unat (*number-of bin* :: *'a* :: *len0 word*) =
  number-of (*bintrunc* (*len-of TYPE*(*'a*)) *bin*)
⟨*proof*⟩

declare
uint-bintrunc [*simp*]
sint-sbintrunc [*simp*]
unat-bintrunc [*simp*]

lemma *size-0-eq*: *size* (*w* :: *'a* :: *len0 word*) = *0* ==> *v* = *w*
⟨*proof*⟩

lemmas *uint-lem* = *word-uint.Rep* [*unfolded uints-num mem-Collect-eq*]
lemmas *sint-lem* = *word-sint.Rep* [*unfolded sints-num mem-Collect-eq*]
lemmas *uint-ge-0* [*iff*] = *uint-lem* [*THEN conjunct1*, *standard*]
lemmas *uint-lt2p* [*iff*] = *uint-lem* [*THEN conjunct2*, *standard*]
lemmas *sint-ge* = *sint-lem* [*THEN conjunct1*, *standard*]
lemmas *sint-lt* = *sint-lem* [*THEN conjunct2*, *standard*]

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

lemmas *uint-m2p-neg* = *iffD2* [*OF diff-less-0-iff-less uint-lt2p*, *standard*]
lemmas *uint-m2p-not-non-neg* =
iffD2 [*OF linorder-not-le uint-m2p-neg*, *standard*]

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

lemmas *uint-le-0-iff* [*simp*] =
uint-ge-0 [*THEN leD*, *THEN linorder-antisym-conv1*, *standard*]

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

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

lemma *unat-number-of*:
bin-sign b = Numeral.Pls ==>
unat (number-of b::'a::len0 word) = number-of b mod 2 ^ len-of TYPE ('a)
⟨*proof*⟩

lemma *sint-number-of*: *sint (number-of b :: 'a :: len word) = (number-of b +*
2 ^ (len-of TYPE('a) − 1)) mod 2 ^ len-of TYPE('a) −
2 ^ (len-of TYPE('a) − 1)
⟨*proof*⟩

lemma *word-of-int-bin* [*simp*] :
(word-of-int (number-of bin) :: 'a :: len0 word) = (number-of 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) =
(ALL 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) =
(~ (EX 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*, *standard*]
lemmas *sint-range' = word-sint.Rep* [*unfolded One-nat-def*
sints-num mem-Collect-eq, *standard*]

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*⟩

lemmas *sint-above-size* = *sint-range-size*
[*THEN conjunct2*, *THEN* [*2*] *xtr8*, *folded One-nat-def*, *standard*]

lemmas *sint-below-size* = *sint-range-size*
[*THEN conjunct1*, *THEN* [*2*] *order-trans*, *folded One-nat-def*, *standard*]

lemma *test-bit-eq-iff*: (*test-bit* (*u*::′*a*::*len0 word*) = *test-bit v*) = (*u* = *v*)
⟨*proof*⟩

lemma *test-bit-size* [*rule-format*] : (*w*::′*a*::*len0 word*) !! *n* −−> *n* < *size w*
⟨*proof*⟩

lemma *word-eqI* [*rule-format*] :
fixes *u* :: ′*a*::*len0 word*
shows (*ALL n*. *n* < *size u* −−> *u* !! *n* = *v* !! *n*) ==> *u* = *v*
⟨*proof*⟩

lemmas *word-eqD* = *test-bit-eq-iff* [*THEN iffD2*, *THEN fun-cong*, *standard*]

lemma *test-bit-bin*′: *w* !! *n* = (*n* < *size w* & *bin-nth* (*uint w*) *n*)
⟨*proof*⟩

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

lemma *bin-nth-uint-imp*′: *bin-nth* (*uint w*) *n* −−> *n* < *size w*
⟨*proof*⟩

lemma *bin-nth-sint*′:
n >= *size w* −−> *bin-nth* (*sint w*) *n* = *bin-nth* (*sint w*) (*size w* − *1*)
⟨*proof*⟩

lemmas *bin-nth-uint-imp* = *bin-nth-uint-imp*′ [*rule-format*, *unfolded word-size*]
lemmas *bin-nth-sint* = *bin-nth-sint*′ [*rule-format*, *unfolded word-size*]

lemma *td-bl*:
type-definition (*to-bl* :: ′*a*::*len0 word* => *bool list*)
of-bl
{*bl*. *length bl* = *len-of TYPE*(′*a*)}
⟨*proof*⟩

interpretation *word-bl*:
type-definition [*to-bl* :: ′*a*::*len0 word* => *bool list*
of-bl
{*bl*. *length bl* = *len-of TYPE*(′*a*::*len0*)}]
⟨*proof*⟩

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

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

lemma *to-bl-word-rev*: *to-bl (word-reverse w) = rev (to-bl w)*
⟨*proof*⟩

lemma *word-rev-rev* [*simp*] : *word-reverse (word-reverse w) = w*
⟨*proof*⟩

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

lemmas *word-rev-gal′ = sym [THEN word-rev-gal, symmetric, standard]*

lemmas *length-bl-gt-0 [iff] = xtr1 [OF word-bl.Rep′ len-gt-0, standard]*
lemmas *bl-not-Nil [iff] =*
length-bl-gt-0 [THEN length-greater-0-conv [THEN iffD1], standard]
lemmas *length-bl-neq-0 [iff] = length-bl-gt-0 [THEN gr-implies-not0]*

lemma *hd-bl-sign-sint*: *hd (to-bl w) = (bin-sign (sint w) = Numeral.Min)*
⟨*proof*⟩

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

lemmas *of-bl-no = of-bl-def [folded word-number-of-def]*

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

lemma *no-of-bl*:
(number-of bin ::′a::len0 word) = of-bl (bin-to-bl (len-of TYPE (′a)) 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*⟩

lemmas *to-bl-no-bin* [*simp*] = *to-bl-of-bin* [*folded word-number-of-def*]

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

lemmas *uint-bl-bin* [*simp*] = *trans* [*OF bin-bl-bin word-ubin.norm-Rep*, *standard*]

lemmas *num-AB-u* [*simp*] = *word-uint.Rep-inverse*
[*unfolded o-def word-number-of-def* [*symmetric*], *standard*]
lemmas *num-AB-s* [*simp*] = *word-sint.Rep-inverse*
[*unfolded o-def word-number-of-def* [*symmetric*], *standard*]

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*
[*symmetric*, *folded word-number-of-def*, *standard*]
lemmas *sbintr-num* = *word-sbin.norm-eq-iff*
[*symmetric*, *folded word-number-of-def*, *standard*]

lemmas *num-of-bintr* = *word-ubin.Abs-norm* [*folded word-number-of-def*, *standard*]
lemmas *num-of-sbintr* = *word-sbin.Abs-norm* [*folded word-number-of-def*, *standard*]

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

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

lemmas *num-abs-bintr* = *sym* [*THEN trans*,
OF num-of-bintr word-number-of-def [*THEN meta-eq-to-obj-eq*], *standard*]
lemmas *num-abs-sbintr* = *sym* [*THEN trans*,
OF num-of-sbintr word-number-of-def [*THEN meta-eq-to-obj-eq*], *standard*]

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 (*number-of* w ::′*a*::*len0 word*) =
number-of (*bintrunc* (*len-of TYPE*(′*a*)) w)
⟨*proof*⟩

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

lemmas *source-size* = *source-size-def* [*unfolded Let-def word-size*]
lemmas *target-size* = *target-size-def* [*unfolded Let-def word-size*]
lemmas *is-down* = *is-down-def* [*unfolded source-size target-size*]
lemmas *is-up* = *is-up-def* [*unfolded source-size target-size*]

lemmas *is-up-down* =
trans [*OF is-up* [*THEN meta-eq-to-obj-eq*]
is-down [*THEN meta-eq-to-obj-eq*, *symmetric*],
standard]

lemma *down-cast-same*′: *uc* = *ucast* ==> *is-down uc* ==> *uc* = *scast*
⟨*proof*⟩

lemma *word-rev-tf*′:
r = *to-bl* (*of-bl bl*) ==> r = *rev* (*takefill False* (*length* r) (*rev bl*))
⟨*proof*⟩

lemmas *word-rev-tf* = *refl* [*THEN word-rev-tf*′, *unfolded word-bl.Rep*′, *standard*]

lemmas *word-rep-drop* = *word-rev-tf* [*simplified takefill-alt*,
simplified, *simplified rev-take*, *simplified*]

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*':
uc = *ucast* ==> *source-size uc* + *n* = *target-size uc* ==>
to-bl (*uc w*) = *replicate n False* @ (*to-bl w*)
⟨*proof*⟩

lemma *ucast-down-drop*':
uc = *ucast* ==> *source-size uc* = *target-size uc* + *n* ==>
to-bl (*uc w*) = *drop n* (*to-bl w*)
⟨*proof*⟩

lemma *scast-down-drop*':
sc = *scast* ==> *source-size sc* = *target-size sc* + *n* ==>
to-bl (*sc w*) = *drop n* (*to-bl w*)
⟨*proof*⟩

lemma *sint-up-scast*':
sc = *scast* ==> *is-up sc* ==> *sint* (*sc w*) = *sint w*
⟨*proof*⟩

lemma *uint-up-ucast*':
uc = *ucast* ==> *is-up uc* ==> *uint* (*uc w*) = *uint w*
⟨*proof*⟩

lemmas *down-cast-same* = *refl* [*THEN down-cast-same*']
lemmas *ucast-up-app* = *refl* [*THEN ucast-up-app*']
lemmas *ucast-down-drop* = *refl* [*THEN ucast-down-drop*']
lemmas *scast-down-drop* = *refl* [*THEN scast-down-drop*']
lemmas *uint-up-ucast* = *refl* [*THEN uint-up-ucast*']
lemmas *sint-up-scast* = *refl* [*THEN sint-up-scast*']

lemma *ucast-up-ucast*': *uc* = *ucast* ==> *is-up uc* ==> *ucast* (*uc w*) = *ucast w*
⟨*proof*⟩

lemma *scast-up-scast*': *sc* = *scast* ==> *is-up sc* ==> *scast* (*sc w*) = *scast w*
⟨*proof*⟩

lemma *ucast-of-bl-up*':
w = *of-bl bl* ==> *size bl* <= *size w* ==> *ucast w* = *of-bl bl*
⟨*proof*⟩

lemmas *ucast-up-ucast* = *refl* [*THEN ucast-up-ucast*']
lemmas *scast-up-scast* = *refl* [*THEN scast-up-scast*']
lemmas *ucast-of-bl-up* = *refl* [*THEN ucast-of-bl-up*']

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-no'*:
uc = *ucast* ==> *is-down uc* ==> *uc* (*number-of bin*) = *number-of bin*
⟨*proof*⟩

lemmas *ucast-down-no* = *ucast-down-no'* [*OF refl*]

lemma *ucast-down-bl'*: *uc* = *ucast* ==> *is-down uc* ==> *uc* (*of-bl bl*) = *of-bl bl*
⟨*proof*⟩

lemmas *ucast-down-bl* = *ucast-down-bl'* [*OF refl*]

lemmas *slice-def'* = *slice-def* [*unfolded word-size*]
lemmas *test-bit-def'* = *word-test-bit-def* [*THEN meta-eq-to-obj-eq*, *THEN fun-cong*]

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

end

11 WordArith: Word Arithmetic

theory *WordArith* **imports** *WordDefinition* **begin**

lemma *word-less-alt*: $(a < b) = (uint\ a < uint\ b)$
⟨*proof*⟩

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

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

lemmas *word-arith-wis* [*THEN meta-eq-to-obj-eq*] =
word-add-def word-mult-def word-minus-def
word-succ-def word-pred-def word-0-wi word-1-wi

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

lemmas *word-div-no* [*simp*] =
word-div-def [*of number-of a number-of b*, *standard*]

lemmas *word-mod-no* [*simp*] =
word-mod-def [*of number-of a number-of b*, *standard*]

lemmas *word-less-no* [*simp*] =
word-less-def [*of number-of a number-of b*, *standard*]

lemmas *word-le-no* [*simp*] =
word-le-def [*of number-of a number-of b*, *standard*]

lemmas *word-sless-no* [*simp*] =
word-sless-def [*of number-of a number-of b*, *standard*]

lemmas *word-sle-no* [*simp*] =
word-sle-def [*of number-of a number-of b*, *standard*]

lemmas *word-0-wi-Pls* = *word-0-wi* [*folded Pls-def*]
lemmas *word-0-no* = *word-0-wi-Pls* [*folded word-no-wi*]

lemma *int-one-bin*: (*1* :: *int*) == (*Numeral.Pls BIT bit.B1*)
⟨*proof*⟩

lemma *word-1-no*:
$(1 :: 'a :: len0\ word) == number\text{-}of\ (Numeral.Pls\ BIT\ bit.B1)$
⟨*proof*⟩

lemma *word-m1-wi*: $-1 == word\text{-}of\text{-}int\ -1$
⟨*proof*⟩

lemma *word-m1-wi-Min*: $-1 = word\text{-}of\text{-}int\ Numeral.Min$
⟨*proof*⟩

lemma *word-0-bl*: $of\text{-}bl\ [] = 0$
⟨*proof*⟩

lemma *word-1-bl*: $of\text{-}bl\ [True] = 1$
⟨*proof*⟩

lemma *uint-0* [*simp*] : $(uint\ 0 = 0)$
⟨*proof*⟩

lemma *of-bl-0* [*simp*] : $of\text{-}bl\ (replicate\ n\ False) = 0$
⟨*proof*⟩

lemma *to-bl-0*:
$to\text{-}bl\ (0::'a::len0\ word) = replicate\ (len\text{-}of\ TYPE('a))\ False$
⟨*proof*⟩

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

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

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

lemma *size-0-same'*: $size\ w = 0 ==> w = (v :: 'a :: len0\ word)$
⟨*proof*⟩

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

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

lemma *unat-gt-0*: $(0 < unat\ x) = (x\ \tilde{}= 0)$
⟨*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*⟩

lemmas *arths* =
bintr-ariths [*THEN word-ubin.norm-eq-iff* [*THEN iffD1*],
folded word-ubin.eq-norm, *standard*]

lemma *wi-homs*:
shows
wi-hom-add: *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* (*Numeral.succ a*) **and**
wi-hom-pred: *word-pred* (*word-of-int a*) = *word-of-int* (*Numeral.pred a*)
⟨*proof*⟩

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

lemma *word-sub-def*: *a* − *b* == *a* + − (*b* :: ′*a* :: *len0 word*)
⟨*proof*⟩

lemmas *word-diff-minus* = *word-sub-def* [*THEN meta-eq-to-obj-eq*, *standard*]

lemma *word-of-int-sub-hom*:
(*word-of-int a*) − *word-of-int b* = *word-of-int* (*a* − *b*)
⟨*proof*⟩

lemmas *new-word-of-int-homs* =

word-of-int-sub-hom wi-homs word-0-wi word-1-wi

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

lemmas *word-of-int-hom-syms* =
new-word-of-int-hom-syms [*unfolded succ-def pred-def*]

lemmas *word-of-int-homs* =
new-word-of-int-homs [*unfolded succ-def pred-def*]

lemmas *word-of-int-add-hom* = *word-of-int-homs* (*2*)
lemmas *word-of-int-mult-hom* = *word-of-int-homs* (*3*)
lemmas *word-of-int-minus-hom* = *word-of-int-homs* (*4*)
lemmas *word-of-int-succ-hom* = *word-of-int-homs* (*5*)
lemmas *word-of-int-pred-hom* = *word-of-int-homs* (*6*)
lemmas *word-of-int-0-hom* = *word-of-int-homs* (*7*)
lemmas *word-of-int-1-hom* = *word-of-int-homs* (*8*)

lemmas *word-arith-alts* =
word-sub-wi [*unfolded succ-def pred-def*, *THEN meta-eq-to-obj-eq*, *standard*]
word-arith-wis [*unfolded succ-def pred-def*, *standard*]

lemmas *word-sub-alt* = *word-arith-alts* (*1*)
lemmas *word-add-alt* = *word-arith-alts* (*2*)
lemmas *word-mult-alt* = *word-arith-alts* (*3*)
lemmas *word-minus-alt* = *word-arith-alts* (*4*)
lemmas *word-succ-alt* = *word-arith-alts* (*5*)
lemmas *word-pred-alt* = *word-arith-alts* (*6*)
lemmas *word-0-alt* = *word-arith-alts* (*7*)
lemmas *word-1-alt* = *word-arith-alts* (*8*)

11.1 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*⟩

lemmas *uint-cong* = *arg-cong* [**where** *f* = *uint*]

lemmas *uint-word-ariths* =
word-arith-alts [*THEN trans* [*OF uint-cong int-word-uint*], *standard*]

lemmas *uint-word-arith-bintrs* = *uint-word-ariths* [*folded bintrunc-mod2p*]

lemmas *sint-word-ariths* = *uint-word-arith-bintrs*
[*THEN uint-sint* [*symmetric*, *THEN trans*],
unfolded uint-sint bintr-arith1s bintr-ariths
len-gt-0 [*THEN bin-sbin-eq-iff'*] *word-sbin.norm-Rep*, *standard*]

lemmas *uint-div-alt* = *word-div-def*
[*THEN meta-eq-to-obj-eq* [*THEN trans* [*OF uint-cong int-word-uint*]], *standard*]
lemmas *uint-mod-alt* = *word-mod-def*
[*THEN meta-eq-to-obj-eq* [*THEN trans* [*OF uint-cong int-word-uint*]], *standard*]

lemma *word-pred-0-n1*: *word-pred 0* = *word-of-int* −*1*
⟨*proof*⟩

lemma *word-pred-0-Min*: *word-pred 0* = *word-of-int Numeral.Min*
⟨*proof*⟩

lemma *word-m1-Min*: − *1* = *word-of-int Numeral.Min*
⟨*proof*⟩

lemma *succ-pred-no* [*simp*]:
word-succ (*number-of bin*) = *number-of* (*Numeral.succ bin*) &
word-pred (*number-of bin*) = *number-of* (*Numeral.pred bin*)
⟨*proof*⟩

lemma *word-sp-01* [*simp*] :
word-succ −*1* = *0* & *word-succ 0* = *1* & *word-pred 0* = −*1* & *word-pred 1* = *0*
⟨*proof*⟩

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

lemma *word-arith-eqs*:
fixes *a* :: *'a*::*len0 word*
fixes *b* :: *'a*::*len0 word*
shows
word-add-0: *0* + *a* = *a* **and**
word-add-0-right: *a* + *0* = *a* **and**
word-mult-1: *1* ∗ *a* = *a* **and**
word-mult-1-right: *a* ∗ *1* = *a* **and**
word-add-commute: *a* + *b* = *b* + *a* **and**
word-add-assoc: *a* + *b* + *c* = *a* + (*b* + *c*) **and**
word-add-left-commute: *a* + (*b* + *c*) = *b* + (*a* + *c*) **and**
word-mult-commute: *a* ∗ *b* = *b* ∗ *a* **and**
word-mult-assoc: *a* ∗ *b* ∗ *c* = *a* ∗ (*b* ∗ *c*) **and**

word-mult-left-commute: $a * (b * c) = b * (a * c)$ **and**
word-left-distrib: $(a + b) * c = a * c + b * c$ **and**
word-right-distrib: $a * (b + c) = a * b + a * c$ **and**
word-left-minus: $- a + a = 0$ **and**
word-diff-0-right: $a - 0 = a$ **and**
word-diff-self: $a - a = 0$
⟨*proof*⟩

lemmas *word-add-ac = word-add-commute word-add-assoc word-add-left-commute*
lemmas *word-mult-ac = word-mult-commute word-mult-assoc word-mult-left-commute*

lemmas *word-plus-ac0 = word-add-0 word-add-0-right word-add-ac*
lemmas *word-times-ac1 = word-mult-1 word-mult-1-right word-mult-ac*

11.2 Order on fixed-length words

lemma *word-order-trans*: $x <= y ==> y <= z ==> x <= (z :: 'a :: len0\ word)$
⟨*proof*⟩

lemma *word-order-refl*: $z <= (z :: 'a :: len0\ word)$
⟨*proof*⟩

lemma *word-order-antisym*: $x <= y ==> y <= x ==> x = (y :: 'a :: len0\ word)$
⟨*proof*⟩

lemma *word-order-linear*:
$y <= x \mid x <= (y :: 'a :: len0\ word)$
⟨*proof*⟩

lemma *word-zero-le* [*simp*] :
$0 <= (y :: 'a :: len0\ word)$
⟨*proof*⟩

instance *word* :: (*len0*) *semigroup-add*
⟨*proof*⟩

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

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

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

lemmas *word-n1-ge* [*simp*] = *word-m1-ge* [*simplified word-sp-01*]

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 :: 'a :: len0 word))
⟨*proof*⟩

lemmas *word-gt-0-no* [*simp*] = *word-gt-0* [*of number-of y*, *standard*]

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 = (EX 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*, *standard*]

lemma *word-zero-neq-one*: 0 < len-of TYPE ('a :: len0) ==> (0 :: 'a word) ~= 1
⟨*proof*⟩

lemmas *lenw1-zero-neq-one* = *len-gt-0* [*THEN word-zero-neq-one*]

lemma *no-no* [*simp*] : number-of (number-of b) = number-of b
⟨*proof*⟩

lemma *unat-minus-one*: x ~= 0 ==> unat (x - 1) = unat x - 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*, *standard*]
lemmas *uint-mult-ge0* [*simp*] =
mult-nonneg-nonneg [*OF uint-ge-0 uint-ge-0*, *standard*]

lemma *uint-sub-lt2p* [*simp*]:
$uint\ (x :: 'a :: len0\ word) - uint\ (y :: 'b :: len0\ word) <$
$2\ \hat{}\ len\text{-}of\ TYPE('a)$
⟨*proof*⟩

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

lemma *uint-add-lem*:
$(uint\ x + uint\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a)) =$
$(uint\ (x + y :: 'a :: len0\ word) = uint\ x + uint\ y)$
⟨*proof*⟩

lemma *uint-mult-lem*:
$(uint\ x * uint\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a)) =$
$(uint\ (x * y :: 'a :: len0\ word) = uint\ x * uint\ y)$
⟨*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*⟩

lemmas *uint-sub-if'* =
trans [*OF uint-word-ariths*(*1*) *mod-sub-if-z*, *simplified*, *standard*]
lemmas *uint-plus-if'* =
trans [*OF uint-word-ariths*(*2*) *mod-add-if-z*, *simplified*, *standard*]

11.4 Definition of uint_arith

lemma *word-of-int-inverse*:
$word\text{-}of\text{-}int\ r = a ==> 0 <= r ==> r < 2\ \hat{}\ len\text{-}of\ TYPE('a) ==>$
$uint\ (a::'a::len0\ word) = r$
⟨*proof*⟩

lemma *uint-split*:
fixes $x::'a::len0\ word$
shows $P\ (uint\ x) =$
$(ALL\ i.\ word\text{-}of\text{-}int\ i = x\ \&\ 0 <= i\ \&\ i < 2\hat{}len\text{-}of\ TYPE('a) --> P\ i)$
⟨*proof*⟩

lemma *uint-split-asm*:
  fixes $x::'a::len0\ word$
  shows $P\ (uint\ x) =$
  $(\sim(EX\ i.\ word\text{-}of\text{-}int\ i = x\ \&\ 0 <= i\ \&\ i < 2\hat{}len\text{-}of\ TYPE('a)\ \&\ \sim\ P\ i))$
  ⟨*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*: $False ==> a\ \hat{}\ b = c\ \hat{}\ d$
  ⟨*proof*⟩

⟨*ML*⟩

11.5 More on overflows and monotonicity

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

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

lemma *no-ulen-sub*: $((x :: 'a :: len0\ word) >= x - y) = (uint\ y <= uint\ x)$
  ⟨*proof*⟩

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

lemmas *olen-add-eqv* = *trans* [*OF no-olen-add no-olen-add'* [*symmetric*], *standard*]

lemmas *uint-plus-simple-iff* = *trans* [*OF no-olen-add uint-add-lem*, *standard*]
lemmas *uint-plus-simple* = *uint-plus-simple-iff* [*THEN iffD1*, *standard*]
lemmas *uint-minus-simple-iff* = *trans* [*OF no-ulen-sub uint-sub-lem*, *standard*]
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*, *standard*]

lemma *word-less-sub1*:
  $(x :: 'a :: len\ word) \sim= 0 ==> (1 < x) = (0 < x - 1)$

⟨*proof*⟩

lemma *word-le-sub1*:
$(x :: 'a :: len\ word)$ ~= 0 ==> (1 <= x) = (0 <= x − 1)
⟨*proof*⟩

lemma *sub-wrap-lt*:
((x :: 'a :: len0 word) < x − z) = (x < z)
⟨*proof*⟩

lemma *sub-wrap*:
((x :: 'a :: len0 word) <= x − z) = (z = 0 | x < z)
⟨*proof*⟩

lemma *plus-minus-not-NULL-ab*:
(x :: 'a :: len0 word) <= ab − c ==> c <= ab ==> c ~= 0 ==> x + c ~= 0
⟨*proof*⟩

lemma *plus-minus-no-overflow-ab*:
(x :: 'a :: len0 word) <= ab − c ==> c <= ab ==> x <= x + c
⟨*proof*⟩

lemma *le-minus'*:
(a :: 'a :: len0 word) + c <= b ==> a <= a + c ==> c <= b − a
⟨*proof*⟩

lemma *le-plus'*:
(a :: 'a :: len0 word) <= b ==> c <= b − a ==> a + c <= b
⟨*proof*⟩

lemmas *le-plus* = *le-plus'* [*rotated*]

lemmas *le-minus* = *leD* [*THEN thin-rl*, *THEN le-minus'*, *standard*]

lemma *word-plus-mono-right*:
(y :: 'a :: len0 word) <= z ==> x <= x + z ==> x + y <= x + z
⟨*proof*⟩

lemma *word-less-minus-cancel*:
y − x < z − x ==> x <= z ==> (y :: 'a :: len0 word) < z
⟨*proof*⟩

lemma *word-less-minus-mono-left*:
(y :: 'a :: len0 word) < z ==> x <= y ==> y − x < z − x
⟨*proof*⟩

lemma *word-less-minus-mono*:
a < c ==> d < b ==> a − b < a ==> c − d < c
==> a − b < c − (d::'a::len word)

⟨*proof*⟩

lemma *word-le-minus-cancel*:
$y - x <= z - x ==> x <= z ==> (y :: 'a :: len0\ word) <= z$
⟨*proof*⟩

lemma *word-le-minus-mono-left*:
$(y :: 'a :: len0\ word) <= z ==> x <= y ==> y - x <= z - x$
⟨*proof*⟩

lemma *word-le-minus-mono*:
$a <= c ==> d <= b ==> a - b <= a ==> c - d <= c$
$==> a - b <= c - (d::'a::len\ word)$
⟨*proof*⟩

lemma *plus-le-left-cancel-wrap*:
$(x :: 'a :: len0\ word) + y' < x ==> x + y < x ==> (x + y' < x + y) = (y' < y)$
⟨*proof*⟩

lemma *plus-le-left-cancel-nowrap*:
$(x :: 'a :: len0\ word) <= x + y' ==> x <= x + y ==>$
$(x + y' < x + y) = (y' < y)$
⟨*proof*⟩

lemma *word-plus-mono-right2*:
$(a :: 'a :: len0\ word) <= a + b ==> c <= b ==> a <= a + c$
⟨*proof*⟩

lemma *word-less-add-right*:
$(x :: 'a :: len0\ word) < y - z ==> z <= y ==> x + z < y$
⟨*proof*⟩

lemma *word-less-sub-right*:
$(x :: 'a :: len0\ word) < y + z ==> y <= x ==> x - y < z$
⟨*proof*⟩

lemma *word-le-plus-either*:
$(x :: 'a :: len0\ word) <= y \mid x <= z ==> y <= y + z ==> x <= y + z$
⟨*proof*⟩

lemma *word-less-nowrapI*:
$(x :: 'a :: len0\ word) < z - k ==> k <= z ==> 0 < k ==> x < x + k$
⟨*proof*⟩

lemma *inc-le*: $(i :: 'a :: len\ word) < m ==> i + 1 <= m$
⟨*proof*⟩

lemma *inc-i*:

(1 :: 'a :: len word) <= i ==> i < m ==> 1 <= (i + 1) & i + 1 <= m
⟨proof⟩

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

lemma *udvd-incr'*:
p < q ==> uint p = ua + n * uint K ==>
uint q = ua + n' * uint K ==> p + K <= q
⟨proof⟩

lemma *udvd-decr'*:
p < q ==> uint p = ua + n * uint K ==>
uint q = ua + n' * uint K ==> p <= q − K
⟨proof⟩

lemmas *udvd-incr-lem0* = *udvd-incr-lem* [**where** *ua=0*, *simplified*]
lemmas *udvd-incr0* = *udvd-incr'* [**where** *ua=0*, *simplified*]
lemmas *udvd-decr0* = *udvd-decr'* [**where** *ua=0*, *simplified*]

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*⟩

11.6 Arithmetic type class instantiations

instance *word* :: (*len0*) *comm-monoid-add* ⟨*proof*⟩

instance *word* :: (*len0*) *comm-monoid-mult*
⟨*proof*⟩

instance *word* :: (*len0*) *comm-semiring*
⟨*proof*⟩

instance *word* :: (*len0*) *ab-group-add* ⟨*proof*⟩

instance *word* :: (*len0*) *comm-ring* ⟨*proof*⟩

instance *word* :: (*len*) *comm-semiring-1*
⟨*proof*⟩

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

instance *word* :: (*len0*) *comm-semiring-0* ⟨*proof*⟩

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

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

lemma *zero-bintrunc*:
iszero (*number-of* *x* :: *'a* :: *len* *word*) =
(*bintrunc* (*len-of* *TYPE*(*'a*)) *x* = *Numeral.Pls*)
⟨*proof*⟩

lemmas *word-le-0-iff* [*simp*] =
word-zero-le [*THEN* *leD*, *THEN* *linorder-antisym-conv1*]

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

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

lemma *word-of-int-nat*:
0 <= x ==> word-of-int x = of-nat (nat x)
⟨*proof*⟩

lemma *word-number-of-eq*:
number-of w = (of-int w :: 'a :: len word)
⟨*proof*⟩

instance *word* :: (*len*) *number-ring*
⟨*proof*⟩

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

11.7 Word and nat

lemma *td-ext-unat'*:
n = len-of TYPE ('a :: len) ==>
td-ext (unat :: 'a word => nat) of-nat
(unats n) (%i. i mod 2 ^ n)
⟨*proof*⟩

lemmas *td-ext-unat = refl [THEN td-ext-unat']*
lemmas *unat-of-nat = td-ext-unat [THEN td-ext.eq-norm, standard]*

interpretation *word-unat*:
td-ext [unat::'a::len word => nat
of-nat
unats (len-of TYPE('a::len))
%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 :: 'a :: len word) ==> y : unats (len-of TYPE ('a))*
⟨*proof*⟩

lemma *word-nchotomy*:

ALL w. EX n. (w :: 'a :: len word) = of-nat n & n < 2 ^ len-of TYPE ('a)
⟨*proof*⟩

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

lemma *of-nat-eq-size*:
(of-nat n = w) = (EX 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*⟩

lemmas *of-nat-2p = mult-1 [symmetric, THEN iffD2 [OF of-nat-0 exI]]*

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

lemmas *unat-cong* = *arg-cong* [**where** *f* = *unat*]

lemmas *unat-word-ariths* = *word-arith-nat-defs*
[*THEN trans* [*OF unat-cong unat-of-nat*], *standard*]

lemmas *word-sub-less-iff* = *word-sub-le-iff*
[*simplified linorder-not-less* [*symmetric*], *simplified*]

lemma *unat-add-lem*:
(*unat x + unat y < 2 ˆ len-of TYPE('a)*) =
(*unat (x + y :: 'a :: len word) = unat x + unat y*)
⟨*proof*⟩

lemma *unat-mult-lem*:
(*unat x ∗ unat y < 2 ˆ len-of TYPE('a)*) =
(*unat (x ∗ y :: 'a :: len word) = unat x ∗ unat y*)
⟨*proof*⟩

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

lemma *le-no-overflow*:
x <= b ==> a <= a + b ==> x <= a + (b :: 'a :: len0 word)
⟨*proof*⟩

lemmas *un-ui-le* = *trans*
[*OF word-le-nat-alt* [*symmetric*]
word-le-def [*THEN meta-eq-to-obj-eq*],
standard]

lemma *unat-sub-if-size*:
unat $(x - y)$ = (*if unat y* <= *unat x*
then unat x − *unat y*
else unat x + *2* ^ *size x* − *unat y*)
⟨*proof*⟩

lemmas *unat-sub-if'* = *unat-sub-if-size* [*unfolded word-size*]

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

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

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

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

11.8 Definition of unat_arith tactic

lemma *unat-split*:
fixes *x*::'*a*::*len word*
shows *P* (*unat x*) =
(*ALL n. of-nat n* = *x* & *n* < *2*^*len-of TYPE*('*a*) −−> *P n*)
⟨*proof*⟩

lemma *unat-split-asm*:
fixes *x*::'*a*::*len word*
shows *P* (*unat x*) =
(~(*EX n. of-nat n* = *x* & *n* < *2*^*len-of TYPE*('*a*) & ~ *P n*))
⟨*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*:
$((x :: 'a :: len\ word) <= x + y) = (unat\ x + unat\ y < 2\ \hat{}\ size\ x)$
⟨*proof*⟩

lemma *unat-sub*: $b <= a ==> unat\ (a - b) = unat\ a - unat\ (b :: 'a :: len\ word)$
⟨*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*, *standard*]

lemma *word-div-mult*:
$(0 :: 'a :: len\ word) < y ==> unat\ x * unat\ y < 2\ \hat{}\ len\text{-}of\ TYPE('a) ==>$
$x * y\ div\ y = x$
⟨*proof*⟩

lemma *div-lt'*: $(i :: 'a :: len\ word) <= k\ div\ x ==>$
$unat\ i * unat\ x < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemmas *div-lt''* = *order-less-imp-le* [*THEN div-lt'*]

lemma *div-lt-mult*: $(i :: 'a :: len\ word) < k\ div\ x ==> 0 < x ==> i * x < k$
⟨*proof*⟩

lemma *div-le-mult*:
$(i :: 'a :: len\ word) <= k\ div\ x ==> 0 < x ==> i * x <= k$
⟨*proof*⟩

lemma *div-lt-uint'*:
$(i :: 'a :: len\ word) <= k\ div\ x ==> uint\ i * uint\ x < 2\ \hat{}\ len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemmas *div-lt-uint''* = *order-less-imp-le* [*THEN div-lt-uint'*]

lemma *word-le-exists'*:
$(x :: 'a :: len0\ word) <= y ==>$
$(EX\ z.\ y = x + z\ \&\ uint\ x + uint\ z < 2\ \hat{}\ len\text{-}of\ TYPE('a))$
⟨*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*, *standard*]
lemmas *word-diff-ls* = *mcs* [**where** $z = w + x$, *unfolded add-diff-cancel*, *standard*]
lemmas *word-plus-mcs* = *word-diff-ls*
[**where** $y = v + x$, *unfolded add-diff-cancel*, *standard*]

lemmas *le-unat-uoi* = *unat-le* [*THEN word-unat.Abs-inverse*]

lemmas *thd* = *refl* [*THEN* [*2*] *split-div-lemma* [*THEN iffD2*], *THEN conjunct1*]

lemma *thd1*:
$a\ div\ b * b \leq (a::nat)$
⟨*proof*⟩

lemmas *uno-simps* [*THEN le-unat-uoi*, *standard*] =
mod-le-divisor div-le-dividend thd1

lemma *word-mod-div-equality*:
$(n\ div\ b) * b + (n\ mod\ b) = (n :: 'a :: len\ word)$
⟨*proof*⟩

lemma *word-div-mult-le*: $a\ div\ b * b <= (a::'a::len\ word)$
⟨*proof*⟩

lemma *word-mod-less-divisor*: $0 < n ==> m\ mod\ n < (n :: 'a :: len\ word)$
⟨*proof*⟩

lemma *word-of-int-power-hom*:
$word\text{-}of\text{-}int\ a\ \hat{}\ n = (word\text{-}of\text{-}int\ (a\ \hat{}\ n) :: 'a :: len\ word)$
⟨*proof*⟩

lemma *word-arith-power-alt*:
$a\ \hat{}\ n = (word\text{-}of\text{-}int\ (uint\ a\ \hat{}\ n) :: 'a :: len\ word)$
⟨*proof*⟩

lemma *of-bl-length-less*:
$length\ x = k ==> k < len\text{-}of\ TYPE('a) ==> (of\text{-}bl\ x :: 'a :: len\ word) < 2\ \hat{}\ k$
⟨*proof*⟩

11.9 Cardinality, finiteness of set of words

lemmas *card-lessThan'* = *card-lessThan* [*unfolded lessThan-def*]

lemmas *card-eq* = *word-unat.Abs-inj-on* [*THEN card-image*,
unfolded word-unat.image, *unfolded unats-def*, *standard*]

lemmas *card-word* = *trans* [*OF card-eq card-lessThan'*, *standard*]

lemma *finite-word-UNIV*: *finite* (*UNIV* :: *'a* :: *len word set*)
⟨*proof*⟩

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

end

12 WordBitwise: Bitwise Operations on Words

theory *WordBitwise* **imports** *WordArith* **begin**

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*, *standard*]

lemmas *word-log-binary-defs* =
word-and-def word-or-def word-xor-def

lemmas *word-no-log-defs* [*simp*] =
word-not-def [**where** *a*=*number-of a*,
unfolded word-no-wi wils1, *folded word-no-wi*, *standard*]
word-log-binary-defs [**where** *a*=*number-of a* **and** *b*=*number-of b*,
unfolded word-no-wi wils1, *folded word-no-wi*, *standard*]

lemmas *word-wi-log-defs* = *word-no-log-defs* [*unfolded word-no-wi*]

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 *word-ops-nth-size*:
n < *size* (*x*::*'a*::*len0 word*) ==>
(*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*)
⟨*proof*⟩

lemma *word-ao-nth*:
fixes $x :: 'a::len0\ word$
shows $(x\ OR\ y)\ !!\ n = (x\ !!\ n \mid y\ !!\ n)\ \&$
$(x\ AND\ y)\ !!\ n = (x\ !!\ n\ \&\ y\ !!\ n)$
⟨*proof*⟩

lemmas *bwsimps* =
word-of-int-homs(*2*)
word-0-wi-Pls
word-m1-wi-Min
word-wi-log-defs

lemma *word-bw-assocs*:
fixes $x :: 'a::len0\ word$
shows
$(x\ AND\ y)\ AND\ z = x\ AND\ y\ AND\ z$
$(x\ OR\ y)\ OR\ z = x\ OR\ y\ OR\ z$
$(x\ XOR\ y)\ XOR\ z = x\ XOR\ y\ XOR\ z$
⟨*proof*⟩

lemma *word-bw-comms*:
fixes $x :: 'a::len0\ word$
shows
$x\ AND\ y = y\ AND\ x$
$x\ OR\ y = y\ OR\ x$
$x\ XOR\ y = y\ XOR\ x$
⟨*proof*⟩

lemma *word-bw-lcs*:
fixes $x :: 'a::len0\ word$
shows
$y\ AND\ x\ AND\ z = x\ AND\ y\ AND\ z$
$y\ OR\ x\ OR\ z = x\ OR\ y\ OR\ z$
$y\ XOR\ x\ XOR\ z = x\ XOR\ y\ XOR\ z$
⟨*proof*⟩

lemma *word-log-esimps* [*simp*]:
fixes $x :: 'a::len0\ word$
shows
$x\ AND\ 0 = 0$
$x\ AND\ -1 = x$
$x\ OR\ 0 = x$
$x\ OR\ -1 = -1$
$x\ XOR\ 0 = x$
$x\ XOR\ -1 = NOT\ x$
$0\ AND\ x = 0$
$-1\ AND\ x = x$

$0\ OR\ x = x$
$-1\ OR\ x = -1$
$0\ XOR\ x = x$
$-1\ XOR\ x = NOT\ x$
⟨*proof*⟩

lemma *word-not-dist*:
fixes $x :: 'a::len0\ word$
shows
$NOT\ (x\ OR\ y) = NOT\ x\ AND\ NOT\ y$
$NOT\ (x\ AND\ y) = NOT\ x\ OR\ NOT\ y$
⟨*proof*⟩

lemma *word-bw-same*:
fixes $x :: 'a::len0\ word$
shows
$x\ AND\ x = x$
$x\ OR\ x = x$
$x\ XOR\ x = 0$
⟨*proof*⟩

lemma *word-ao-absorbs* [*simp*]:
fixes $x :: 'a::len0\ word$
shows
$x\ AND\ (y\ OR\ x) = x$
$x\ OR\ y\ AND\ x = x$
$x\ AND\ (x\ OR\ y) = x$
$y\ AND\ x\ OR\ x = x$
$(y\ OR\ x)\ AND\ x = x$
$x\ OR\ x\ AND\ y = x$
$(x\ OR\ y)\ AND\ x = x$
$x\ AND\ y\ OR\ x = x$
⟨*proof*⟩

lemma *word-not-not* [*simp*]:
$NOT\ NOT\ (x::'a::len0\ word) = x$
⟨*proof*⟩

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

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

lemma *word-add-not* [*simp*]:

fixes x :: $'a$::*len0 word*
shows $x + NOT\ x = -1$
⟨*proof*⟩

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

lemma *leoa*:
fixes x :: $'a$::*len0 word*
shows $(w = (x\ OR\ y)) ==> (y = (w\ AND\ y))$ ⟨*proof*⟩
lemma *leao*:
fixes x' :: $'a$::*len0 word*
shows $(w' = (x'\ AND\ y')) ==> (x' = (x'\ OR\ w'))$ ⟨*proof*⟩

lemmas *word-ao-equiv* = *leao* [*COMP leoa* [*COMP iffI*]]

lemma *le-word-or2*: $x <= x\ OR\ (y::'a::len0\ word)$
⟨*proof*⟩

lemmas *le-word-or1* = *xtr3* [*OF word-bw-comms* (*2*) *le-word-or2*, *standard*]
lemmas *word-and-le1* =
xtr3 [*OF word-ao-absorbs* (*4*) [*symmetric*] *le-word-or2*, *standard*]
lemmas *word-and-le2* =
xtr3 [*OF word-ao-absorbs* (*8*) [*symmetric*] *le-word-or2*, *standard*]

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

lemma *bl-word-xor*: *to-bl* (*v XOR w*) = *app2 op* ~= (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *bl-word-or*: *to-bl* (*v OR w*) = *app2 op* | (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *bl-word-and*: *to-bl* (*v AND w*) = *app2 op* & (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

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

lemma *word-lsb-1-0*: *lsb* (1::$'a$::*len word*) & ~ *lsb* (0::$'b$::*len0 word*)
⟨*proof*⟩

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

lemma *word-lsb-int*: *lsb w* = (*uint w mod 2 = 1*)

⟨*proof*⟩

lemma *word-msb-sint*: $msb\ w = (sint\ w < 0)$
⟨*proof*⟩

lemma *word-msb-no'*:
$w = number\text{-}of\ bin \Longrightarrow msb\ (w::'a::len\ word) = bin\text{-}nth\ bin\ (size\ w - 1)$
⟨*proof*⟩

lemmas *word-msb-no* = *refl* [*THEN word-msb-no'*, *unfolded word-size*]

lemma *word-msb-nth'*: $msb\ (w::'a::len\ word) = bin\text{-}nth\ (uint\ w)\ (size\ w - 1)$
⟨*proof*⟩

lemmas *word-msb-nth* = *word-msb-nth'* [*unfolded word-size*]

lemma *word-msb-alt*: $msb\ (w::'a::len\ word) = hd\ (to\text{-}bl\ w)$
⟨*proof*⟩

lemma *word-set-nth*:
$set\text{-}bit\ w\ n\ (test\text{-}bit\ w\ n) = (w::'a::len0\ word)$
⟨*proof*⟩

lemma *bin-nth-uint'*:
$bin\text{-}nth\ (uint\ w)\ n = (rev\ (bin\text{-}to\text{-}bl\ (size\ w)\ (uint\ w))\ !\ n\ \&\ n < size\ w)$
⟨*proof*⟩

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

lemma *test-bit-bl*: $w\ !!\ n = (rev\ (to\text{-}bl\ w)\ !\ n\ \&\ n < size\ w)$
⟨*proof*⟩

lemma *to-bl-nth*: $n < size\ w \Longrightarrow to\text{-}bl\ w\ !\ n = w\ !!\ (size\ w - Suc\ n)$
⟨*proof*⟩

lemma *test-bit-set*:
fixes $w :: 'a::len0\ word$
shows $(set\text{-}bit\ w\ n\ x)\ !!\ n = (n < size\ w\ \&\ x)$
⟨*proof*⟩

lemma *test-bit-set-gen*:
fixes $w :: 'a::len0\ word$
shows $test\text{-}bit\ (set\text{-}bit\ w\ n\ x)\ m =$
$(if\ m = n\ then\ n < size\ w\ \&\ x\ else\ test\text{-}bit\ w\ m)$
⟨*proof*⟩

lemma *of-bl-rep-False*: $of\text{-}bl\ (replicate\ n\ False\ @\ bs) = of\text{-}bl\ bs$
⟨*proof*⟩

lemma *msb-nth'*:
  fixes *w* :: *'a::len word*
  shows *msb w = w !! (size w − 1)*
  ⟨*proof*⟩

lemmas *msb-nth = msb-nth'* [*unfolded word-size*]

lemmas *msb0 = len-gt-0* [*THEN diff-Suc-less, THEN*
  word-ops-nth-size [*unfolded word-size*], *standard*]
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*], *standard*]
lemmas *word-ops-lsb = lsb0* [*unfolded word-lsb-alt*]

lemma *td-ext-nth'*:
  n = size (w::'a::len0 word) ==> ofn = set-bits ==> [*w, ofn g*] *= l ==>*
    td-ext test-bit ofn {f. ALL i. f i −−> i < n} (%h i. h i & i < n)
  ⟨*proof*⟩

lemmas *td-ext-nth = td-ext-nth'* [*OF refl refl refl, unfolded word-size*]

interpretation *test-bit*:
  td-ext [*op !! :: 'a::len0 word => nat => bool*
          set-bits
          $\{f.\ \forall i.\ f\ i \longrightarrow i < \textit{len-of TYPE('a::len0)}\}$
          $(\lambda h\ i.\ h\ i \wedge i < \textit{len-of TYPE('a::len0)})$]
  ⟨*proof*⟩

declare *test-bit.Rep'* [*simp del*]
declare *test-bit.Rep'* [*rule del*]

lemmas *td-nth = test-bit.td-thm*

lemma *word-set-set-same*:
  fixes *w* :: *'a::len0 word*
  shows *set-bit (set-bit w n x) n y = set-bit w n y*
  ⟨*proof*⟩

lemma *word-set-set-diff*:
  fixes *w* :: *'a::len0 word*
  assumes *m ~= n*
  shows *set-bit (set-bit w m x) n y = set-bit (set-bit w n y) m x*
  ⟨*proof*⟩

lemma *test-bit-no'*:
  fixes *w* :: *'a::len0 word*
  shows *w = number-of bin ==> test-bit w n = (n < size w & bin-nth bin n)*
  ⟨*proof*⟩

lemmas *test-bit-no* =
refl [*THEN test-bit-no'*, *unfolded word-size*, *THEN eq-reflection*, *standard*]

lemma *nth-0*: ~ (*0*::'*a*::*len0 word*) !! *n*
⟨*proof*⟩

lemma *nth-sint*:
fixes *w* :: '*a*::*len word*
defines $l \equiv$ *len-of TYPE* ('*a*)
shows *bin-nth* (*sint w*) *n* = (*if n* < *l* − *1 then w* !! *n else w* !! (*l* − *1*))
⟨*proof*⟩

lemma *word-lsb-no*:
lsb (*number-of bin* :: '*a* :: *len word*) = (*bin-last bin* = *bit.B1*)
⟨*proof*⟩

lemma *word-set-no*:
set-bit (*number-of bin*::'*a*::*len0 word*) *n b* =
number-of (*bin-sc n* (*if b then bit.B1 else bit.B0*) *bin*)
⟨*proof*⟩

lemmas *setBit-no* = *setBit-def* [*THEN trans* [*OF meta-eq-to-obj-eq word-set-no*], *simplified if-simps*, *THEN eq-reflection*, *standard*]
lemmas *clearBit-no* = *clearBit-def* [*THEN trans* [*OF meta-eq-to-obj-eq word-set-no*], *simplified if-simps*, *THEN eq-reflection*, *standard*]

lemma *to-bl-n1*:
to-bl (−*1*::'*a*::*len0 word*) = *replicate* (*len-of TYPE* ('*a*)) *True*
⟨*proof*⟩

lemma *word-msb-n1*: *msb* (−*1*::'*a*::*len word*)
⟨*proof*⟩

declare *word-set-set-same* [*simp*] *word-set-nth* [*simp*]
test-bit-no [*simp*] *word-set-no* [*simp*] *nth-0* [*simp*]
setBit-no [*simp*] *clearBit-no* [*simp*]
word-lsb-no [*simp*] *word-msb-no* [*simp*] *word-msb-n1* [*simp*] *word-lsb-1-0* [*simp*]

lemma *word-set-nth-iff*:
(*set-bit w n b* = *w*) = (*w* !! *n* = *b* | *n* >= *size* (*w*::'*a*::*len0 word*))
⟨*proof*⟩

lemma *test-bit-2p'*:
w = *word-of-int* (*2* ^ *n*) ==>
w !! *m* = (*m* = *n* & *m* < *size* (*w* :: '*a* :: *len word*))
⟨*proof*⟩

lemmas *test-bit-2p* = *refl* [*THEN test-bit-2p'*, *unfolded word-size*]

lemmas *nth-w2p* = *test-bit-2p* [*unfolded of-int-number-of-eq*
word-of-int [*symmetric*] *of-int-power*]

lemma *uint-2p*:
$(0::'a::len\ word) < 2\ \hat{}\ n ==> uint\ (2\ \hat{}\ n::'a::len\ word) = 2\ \hat{}\ n$
⟨*proof*⟩

lemma *word-of-int-2p*: $(word\text{-}of\text{-}int\ (2\ \hat{}\ n) :: 'a :: len\ word) = 2\ \hat{}\ n$
⟨*proof*⟩

lemma *bang-is-le*: $x\ !!\ m ==> 2\ \hat{}\ m <= (x :: 'a :: len\ word)$
⟨*proof*⟩

lemma *word-clr-le*:
fixes $w :: 'a::len0\ word$
shows $w >= set\text{-}bit\ w\ n\ False$
⟨*proof*⟩

lemma *word-set-ge*:
fixes $w :: 'a::len\ word$
shows $w <= set\text{-}bit\ w\ n\ True$
⟨*proof*⟩

end

13 WordShift: Shifting, Rotating, and Splitting Words

theory *WordShift* **imports** *WordBitwise* **begin**

13.1 Bit shifting

lemma *shiftl1-number* [*simp*] :
$shiftl1\ (number\text{-}of\ w) = number\text{-}of\ (w\ BIT\ bit.B0)$
⟨*proof*⟩

lemma *shiftl1-0* [*simp*] : $shiftl1\ 0 = 0$
⟨*proof*⟩

lemmas *shiftl1-def-u* = *shiftl1-def* [*folded word-number-of-def*]

lemma *shiftl1-def-s*: $shiftl1\ w = number\text{-}of\ (sint\ w\ BIT\ bit.B0)$
⟨*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'* [*rule-format*]:
ALL n. ((w::'a::len0 word) << m) !! n = (n < size w & n >= m & w !! (n − m))
⟨*proof*⟩

lemmas *nth-shiftl = nth-shiftl'* [*unfolded word-size*]

lemma *nth-shiftr1*: *shiftr1 w !! n = w !! Suc n*
⟨*proof*⟩

lemma *nth-shiftr*:
⋀n. ((w::'a::len0 word) >> m) !! n = w !! (n + m)
⟨*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*] :
ALL 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*⟩

13.1.1 shift functions in terms of lists of bools

lemmas *bshiftr1-no-bin* [*simp*] =
bshiftr1-def [**where** *w=number-of w*, *unfolded to-bl-no-bin*, *standard*]

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::'a::len0 word*) = *of-bl* (*to-bl w* @ [*False*])
⟨*proof*⟩

lemma *bl-shiftl1*:
to-bl (*shiftl1* (*w* :: *'a* :: *len word*)) = *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* :: *'a* :: *len word*)) = *False* # *butlast* (*to-bl w*)
⟨*proof*⟩

lemma *shiftl1-rev*:
shiftl1 (*w* :: *'a* :: *len word*) = *word-reverse* (*shiftr1* (*word-reverse w*))
⟨*proof*⟩

lemma *shiftl-rev*:
shiftl (*w* :: *'a* :: *len word*) *n* = *word-reverse* (*shiftr* (*word-reverse w*) *n*)
⟨*proof*⟩

lemmas *rev-shiftl* =
shiftl-rev [**where** w = *word-reverse* w, *simplified*, *standard*]

lemmas *shiftr-rev* = *rev-shiftl* [*THEN word-rev-gal'*, *standard*]
lemmas *rev-shiftr* = *shiftl-rev* [*THEN word-rev-gal'*, *standard*]

lemma *bl-sshiftr1*:
to-bl (*sshiftr1* (w :: $'a$ :: *len word*)) = *hd* (*to-bl* w) # *butlast* (*to-bl* w)
⟨*proof*⟩

lemma *drop-shiftr*:
drop n (*to-bl* ((w :: $'a$ :: *len word*) >> n)) = *take* (*size* w − n) (*to-bl* w)
⟨*proof*⟩

lemma *drop-sshiftr*:
drop n (*to-bl* ((w :: $'a$ :: *len word*) >>> n)) = *take* (*size* w − n) (*to-bl* w)
⟨*proof*⟩

lemma *take-shiftr* [*rule-format*] :
n <= *size* (w :: $'a$ :: *len word*) −−> *take* n (*to-bl* (w >> n)) =
replicate n *False*
⟨*proof*⟩

lemma *take-sshiftr'* [*rule-format*] :
n <= *size* (w :: $'a$ :: *len word*) −−> *hd* (*to-bl* (w >>> n)) = *hd* (*to-bl* w) &
take n (*to-bl* (w >>> n)) = *replicate* n (*hd* (*to-bl* w))
⟨*proof*⟩

lemmas *hd-sshiftr* = *take-sshiftr'* [*THEN conjunct1*, *standard*]
lemmas *take-sshiftr* = *take-sshiftr'* [*THEN conjunct2*, *standard*]

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::$'a$::*len0 word*) << (n::*nat*) = *of-bl* (*to-bl* w @ *replicate* n *False*)
⟨*proof*⟩

lemmas *shiftl-number* [*simp*] = *shiftl-def* [**where** w=*number-of* w, *standard*]

lemma *bl-shiftl*:
to-bl (w << n) = *drop* n (*to-bl* w) @ *replicate* (*min* (*size* w) n) *False*
⟨*proof*⟩

lemma *shiftl-zero-size*:
fixes *x* :: *'a::len0 word*
shows *size x <= n ==> x << n = 0*
⟨*proof*⟩

lemma *shiftl1-2t*: *shiftl1* (*w* :: *'a* :: *len word*) = *2* ∗ *w*
⟨*proof*⟩

lemma *shiftl1-p*: *shiftl1* (*w* :: *'a* :: *len word*) = *w* + *w*
⟨*proof*⟩

lemma *shiftl-t2n*: *shiftl* (*w* :: *'a* :: *len word*) *n* = *2* ˆ *n* ∗ *w*
⟨*proof*⟩

lemma *shiftr1-bintr* [*simp*]:
(*shiftr1* (*number-of w*) :: *'a* :: *len0 word*) =
number-of (*bin-rest* (*bintrunc* (*len-of TYPE* (*'a*)) *w*))
⟨*proof*⟩

lemma *sshiftr1-sbintr* [*simp*] :
(*sshiftr1* (*number-of w*) :: *'a* :: *len word*) =
number-of (*bin-rest* (*sbintrunc* (*len-of TYPE* (*'a*) − *1*) *w*))
⟨*proof*⟩

lemma *shiftr-no'*:
w = *number-of bin* ==>
(*w::'a::len0 word*) >> *n* = *number-of* ((*bin-rest* ˆ *n*) (*bintrunc* (*size w*) *bin*))
⟨*proof*⟩

lemma *sshiftr-no'*:
w = *number-of bin* ==> *w* >>> *n* = *number-of* ((*bin-rest* ˆ *n*)
(*sbintrunc* (*size w* − *1*) *bin*))
⟨*proof*⟩

lemmas *sshiftr-no* [*simp*] =
sshiftr-no' [**where** *w* = *number-of w*, *OF refl*, *unfolded word-size*, *standard*]

lemmas *shiftr-no* [*simp*] =
shiftr-no' [**where** *w* = *number-of w*, *OF refl*, *unfolded word-size*, *standard*]

lemma *shiftr1-bl-of'*:
us = *shiftr1* (*of-bl bl*) ==> *length bl* <= *size us* ==>
us = *of-bl* (*butlast bl*)
⟨*proof*⟩

lemmas *shiftr1-bl-of* = *refl* [*THEN shiftr1-bl-of'*, *unfolded word-size*]

lemma *shiftr-bl-of′* [*rule-format*]:
us = *of-bl bl* >> *n* ==> *length bl* <= *size us* −−>
us = *of-bl* (*take* (*length bl* − *n*) *bl*)
⟨*proof*⟩

lemmas *shiftr-bl-of* = *refl* [*THEN shiftr-bl-of′*, *unfolded word-size*]

lemmas *shiftr-bl* = *word-bl.Rep′* [*THEN eq-imp-le*, *THEN shiftr-bl-of*,
simplified word-size, *simplified*, *THEN eq-reflection*, *standard*]

lemma *msb-shift′*: *msb* (*w*::′*a*::*len word*) <−> (*w* >> (*size w* − *1*)) ~= *0*
⟨*proof*⟩

lemmas *msb-shift* = *msb-shift′* [*unfolded word-size*]

lemma *align-lem-or* [*rule-format*] :
ALL x m. length x = *n* + *m* −−> *length y* = *n* + *m* −−>
drop m x = *replicate n False* −−> *take m y* = *replicate m False* −−>
app2 op | *x y* = *take m x* @ *drop m y*
⟨*proof*⟩

lemma *align-lem-and* [*rule-format*] :
ALL x m. length x = *n* + *m* −−> *length y* = *n* + *m* −−>
drop m x = *replicate n False* −−> *take m y* = *replicate m False* −−>
app2 op & *x y* = *replicate* (*n* + *m*) *False*
⟨*proof*⟩

lemma *aligned-bl-add-size′*:
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*⟩

lemmas *aligned-bl-add-size* = *refl* [*THEN aligned-bl-add-size′*]

13.1.2 Mask

lemma *nth-mask′*: *m* = *mask n* ==> *test-bit m i* = (*i* < *n* & *i* < *size m*)
⟨*proof*⟩

lemmas *nth-mask* [*simp*] = *refl* [*THEN nth-mask′*]

lemma *mask-bl*: *mask n* = *of-bl* (*replicate n True*)
⟨*proof*⟩

lemma *mask-bin*: *mask n* = *number-of* (*bintrunc n Numeral.Min*)
⟨*proof*⟩

lemma *and-mask-bintr*: *w AND mask n* = *number-of* (*bintrunc n* (*uint w*))
⟨*proof*⟩

lemma *and-mask-no*: *number-of i AND mask n* = *number-of* (*bintrunc n i*)
⟨*proof*⟩

lemmas *and-mask-wi* = *and-mask-no* [*unfolded word-number-of-def*]

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*⟩

lemmas *and-mask-mod-2p* =
and-mask-bintr [*unfolded word-number-of-alt no-bintr-alt*]

lemma *and-mask-lt-2p*: *uint* (*w AND mask n*) < *2* ˆ *n*
⟨*proof*⟩

lemmas *eq-mod-iff* = *trans* [*symmetric*, *OF int-mod-lem eq-sym-conv*]

lemma *mask-eq-iff*: (*w AND mask n*) = *w* <−> *uint w* < *2* ˆ *n*
⟨*proof*⟩

lemma *and-mask-dvd*: *2* ˆ *n dvd uint w* = (*w AND mask n* = *0*)
⟨*proof*⟩

lemma *and-mask-dvd-nat*: *2* ˆ *n dvd unat w* = (*w AND mask n* = *0*)
⟨*proof*⟩

lemma *word-2p-lem*:
n < *size w* ==> *w* < *2* ˆ *n* = (*uint* (*w* :: *'a* :: *len word*) < *2* ˆ *n*)
⟨*proof*⟩

lemma *less-mask-eq*: *x* < *2* ˆ *n* ==> *x AND mask n* = (*x* :: *'a* :: *len word*)
⟨*proof*⟩

lemmas *mask-eq-iff-w2p* =
trans [*OF mask-eq-iff word-2p-lem* [*symmetric*], *standard*]

lemmas *and-mask-less'* =
iffD2 [*OF word-2p-lem and-mask-lt-2p*, *simplified word-size*, *standard*]

lemma *and-mask-less-size*: *n* < *size x* ==> *x AND mask n* < *2*ˆ*n*
⟨*proof*⟩

lemma *word-mod-2p-is-mask'*:

$c = 2 \hat{}\ n ==> c > 0 ==> x \bmod c = (x :: 'a :: len\ word)\ AND\ mask\ n$
⟨*proof*⟩

lemmas *word-mod-2p-is-mask* = *refl* [*THEN word-mod-2p-is-mask'*]

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*⟩

13.1.3 Revcast

lemmas *revcast-def'* = *revcast-def* [*simplified*]
lemmas *revcast-def''* = *revcast-def'* [*simplified word-size*]
lemmas *revcast-no-def* [*simp*] =
revcast-def' [**where** *w=number-of w*, *unfolded word-size*, *standard*]

lemma *to-bl-revcast*:
to-bl (*revcast w* :: *'a* :: *len0 word*) =
takefill False (*len-of TYPE* (*'a*)) (*to-bl w*)
⟨*proof*⟩

lemma *revcast-rev-ucast'*:
cs = [*rc*, *uc*] ==> *rc* = *revcast* (*word-reverse w*) ==> *uc* = *ucast w* ==>
rc = *word-reverse uc*
⟨*proof*⟩

lemmas *revcast-rev-ucast* = *revcast-rev-ucast'* [*OF refl refl refl*]

lemmas *revcast-ucast* = *revcast-rev-ucast*
[**where** *w* = *word-reverse w*, *simplified word-rev-rev*, *standard*]

lemmas *ucast-revcast* = *revcast-rev-ucast* [*THEN word-rev-gal'*, *standard*]
lemmas *ucast-rev-revcast* = *revcast-ucast* [*THEN word-rev-gal'*, *standard*]

— linking revcast and cast via shift

lemmas *wsst-TYs* = *source-size target-size word-size*

lemma *revcast-down-uu'*:
$rc = revcast \implies source\text{-}size\ rc = target\text{-}size\ rc + n \implies$
$rc\ (w :: 'a :: len\ word) = ucast\ (w >> n)$
⟨*proof*⟩

lemma *revcast-down-us'*:
$rc = revcast \implies source\text{-}size\ rc = target\text{-}size\ rc + n \implies$
$rc\ (w :: 'a :: len\ word) = ucast\ (w >>> n)$
⟨*proof*⟩

lemma *revcast-down-su'*:
$rc = revcast \implies source\text{-}size\ rc = target\text{-}size\ rc + n \implies$
$rc\ (w :: 'a :: len\ word) = scast\ (w >> n)$
⟨*proof*⟩

lemma *revcast-down-ss'*:
$rc = revcast \implies source\text{-}size\ rc = target\text{-}size\ rc + n \implies$
$rc\ (w :: 'a :: len\ word) = scast\ (w >>> n)$
⟨*proof*⟩

lemmas *revcast-down-uu* = *refl* [*THEN revcast-down-uu'*]
lemmas *revcast-down-us* = *refl* [*THEN revcast-down-us'*]
lemmas *revcast-down-su* = *refl* [*THEN revcast-down-su'*]
lemmas *revcast-down-ss* = *refl* [*THEN revcast-down-ss'*]

lemma *cast-down-rev*:
$uc = ucast \implies source\text{-}size\ uc = target\text{-}size\ uc + n \implies$
$uc\ w = revcast\ ((w :: 'a :: len\ word) << n)$
⟨*proof*⟩

lemma *revcast-up'*:
$rc = revcast \implies source\text{-}size\ rc + n = target\text{-}size\ rc \implies$
$rc\ w = (ucast\ w :: 'a :: len\ word) << n$
⟨*proof*⟩

lemmas *revcast-up* = *refl* [*THEN revcast-up'*]

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*]]

13.1.4 Slices

lemmas *slice1-no-bin* [*simp*] =
slice1-def [**where** *w*=*number-of w*, *unfolded to-bl-no-bin*, *standard*]

lemmas *slice-no-bin* [*simp*] =
trans [*OF slice-def* [*THEN meta-eq-to-obj-eq*]
slice1-no-bin [*THEN meta-eq-to-obj-eq*],
unfolded word-size, *standard*]

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*], *standard*]

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*⟩

lemmas *slice-id* = *trans* [*OF ucast-slice* [*symmetric*] *ucast-id*]

lemma *revcast-slice1'*:
rc = *revcast w* ==> *slice1* (*size rc*) *w* = *rc*
⟨*proof*⟩

lemmas *revcast-slice1* = *refl* [*THEN revcast-slice1'*]

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*, *standard*]

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'*:
res = *slice n* (*word-reverse w*) ==> $n + k +$ *size res* = *size w* ==>
res = *word-reverse* (*slice k w*)
⟨*proof*⟩

lemmas *rev-slice* = *refl* [*THEN rev-slice'*, *unfolded word-size*]

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* :: *'a* :: *len word*) + *sint y* = *sint* (*x* + *y*)) =
((((*x*+*y*) *XOR x*) *AND* ((*x*+*y*) *XOR y*)) >> (*size x* − *1*) = *0*)
⟨*proof*⟩

13.2 Split and cat

lemmas *word-split-bin′* = *word-split-def* [*THEN meta-eq-to-obj-eq*, *standard*]
lemmas *word-cat-bin′* = *word-cat-def* [*THEN meta-eq-to-obj-eq*, *standard*]

lemma *word-rsplit-no*:
(*word-rsplit* (*number-of bin* :: ′*b* :: *len0 word*) :: ′*a word list*) =
map number-of (*bin-rsplit* (*len-of TYPE*(′*a* :: *len*))
(*len-of TYPE*(′*b*), *bintrunc* (*len-of TYPE*(′*b*)) *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*ˆ(*length ys*) + *of-bl ys*
⟨*proof*⟩

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

lemma *of-bl-True*:
(*of-bl* (*True*#*xs*)::′*a*::*len word*) = *2*ˆ*length xs* + *of-bl xs*
⟨*proof*⟩

lemma *of-bl-Cons*:
of-bl (*x*#*xs*) = *of-bool x* ∗ *2*ˆ*length xs* + *of-bl xs*
⟨*proof*⟩

lemma *split-uint-lem*: *bin-split n* (*uint* (*w* :: ′*a* :: *len0 word*)) = (*a*, *b*) ==>
a = *bintrunc* (*len-of TYPE*(′*a*) − *n*) *a* & *b* = *bintrunc* (*len-of TYPE*(′*a*)) *b*
⟨*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::'a::len word) :: ('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)))
⟨proof⟩
lemma test-bit-split':
word-split c = (a, b) −−> (ALL n m. b !! n = (n < size b & c !! n) &
a !! m = (m < size a & c !! (m + size b)))
⟨proof⟩

lemmas test-bit-split =
test-bit-split' [THEN mp, simplified all-simps, standard]

lemma test-bit-split-eq: word-split c = (a, b) <−>
((ALL n::nat. b !! n = (n < size b & c !! n)) &
(ALL 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], standard]

13.2.1 Split and slice

lemma split-slices:
word-split w = (u, v) ==> u = slice (size v) w & v = slice 0 w
⟨proof⟩

lemma slice-cat1':
wc = word-cat a b ==> size wc >= size a + size b ==> slice (size b) wc = a
⟨proof⟩

lemmas slice-cat1 = refl [THEN slice-cat1']
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=number-of a*
    and *b=number-of b*,
    unfolded to-bl-no-bin, *standard*]

lemmas *word-split-bl-no-bin* [*simp*] =
  word-split-bl-eq [**where** *c=number-of c*, *unfolded to-bl-no-bin*, *standard*]

— 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 :: 'a :: len word) ==>
    *k < length sw ==> (rev sw ! k) !! m = (w !! (k * size (hd sw) + m))*
  ⟨*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*, *standard*]

lemma *nth-rcat-lem'* [*rule-format*] :
  *sw = size (hd wl :: 'a :: len word) ==> (ALL n. n < size wl * sw -->*
    rev (concat (map to-bl wl)) ! n =
    rev (to-bl (rev wl ! (n div sw))) ! (n mod sw))
  ⟨*proof*⟩

lemmas *nth-rcat-lem* = *refl* [*THEN nth-rcat-lem'*, *unfolded word-size*]

lemma *test-bit-rcat*:
sw = *size* (*hd wl* :: *'a* :: *len word*) ==> *rc* = *word-rcat wl* ==> *rc* !! *n* =
(*n* < *size rc* & *n div sw* < *size wl* & (*rev wl*) ! (*n div sw*) !! (*n mod sw*))
⟨*proof*⟩

lemma *foldl-eq-foldr* [*rule-format*] :
ALL x. *foldl op* + *x xs* = *foldr op* + (*x* # *xs*) (*0* :: *'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'*:
[*u*,*v*] = *p* ==> [*su*,*sv*] = *q* ==> *word-rsplit u* = *su* ==>
word-rsplit v = *sv* ==> *length su* = *length sv*
⟨*proof*⟩

lemmas *word-rsplit-len-indep* = *word-rsplit-len-indep'* [*OF refl refl refl refl*]

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* :: *'a* :: *len word list*) = *frcw* ==>
size frcw = *length ws* ∗ *len-of TYPE* (*'a*) ==>
size (*hd* [*word-rsplit frcw*, *ws*]) = *size ws*
⟨*proof*⟩

lemmas *size-word-rsplit-rcat-size* =
size-word-rsplit-rcat-size' [*simplified*]

lemma *msrevs*:
fixes *n*::*nat*
shows $0 < n \Longrightarrow (k * n + m)$ *div* $n = m$ *div* $n + k$
and $(k * n + m)$ *mod* $n = m$ *mod* n
⟨*proof*⟩

lemma *word-rsplit-rcat-size'*:
word-rcat (*ws* :: *'a* :: *len word list*) = *frcw* ==>
size frcw = *length ws* ∗ *len-of TYPE* (*'a*) ==> *word-rsplit frcw* = *ws*
⟨*proof*⟩

lemmas *word-rsplit-rcat-size* = *refl* [*THEN word-rsplit-rcat-size'*]

13.3 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* [*standard*] =
trans [*OF rotate0* [*THEN fun-cong*] *id-apply*]
rotate-rotate [*symmetric*]
rotate-id
rotate-conv-mod
rotate-eq-mod

13.3.1 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*⟩

lemmas *rotater-rev* = *rotater-rev'* [**where** *xs* = *rev ys*, *simplified*, *standard*]

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*] :
ALL 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*, *standard*]
lemmas *rotate-rl* [*simp*] =
rotate-inv-eq [*THEN conjunct2*, *THEN conjunct1*, *standard*]

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*⟩

lemmas *rrs0* = *rotate-eqs* [*THEN restrict-to-left*,
simplified rotate-gal [*symmetric*] *rotate-gal'* [*symmetric*], *standard*]

lemmas *rrs1* = *rrs0* [*THEN refl* [*THEN rev-iffD1*]]
lemmas *rotater-eqs* = *rrs1* [*simplified length-rotater*, *standard*]
lemmas *rotater-0* = *rotater-eqs* (*1*)
lemmas *rotater-add* = *rotater-eqs* (*2*)

13.3.2 map, app2, commuting with rotate(r)

lemma *last-map*: *xs* ~= [] ==> *last* (*map f xs*) = *f* (*last xs*)
⟨*proof*⟩

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*] :
ALL 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-app2* [*rule-format*] :
ALL ys. *length xs* = *length ys* --> *xs* ~= [] -->
last (*app2 f xs ys*) = *f* (*last xs*) (*last ys*) &
butlast (*app2 f xs ys*) = *app2 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-app2*:
length xs = *length ys* ==>
rotater1 (*app2 f xs ys*) = *app2 f* (*rotater1 xs*) (*rotater1 ys*)
⟨*proof*⟩

lemmas *lrth* =
box-equals [*OF asm-rl length-rotater* [*symmetric*]
length-rotater [*symmetric*],
THEN rotater1-app2]

lemma *rotater-app2*:

length xs = length ys ==>
rotater n (app2 f xs ys) = app2 f (rotater n xs) (rotater n ys)
⟨*proof*⟩

lemma *rotate1-app2*:
length xs = length ys ==>
rotate1 (app2 f xs ys) = app2 f (rotate1 xs) (rotate1 ys)
⟨*proof*⟩

lemmas *lth = box-equals [OF asm-rl length-rotate [symmetric]*
length-rotate [symmetric], THEN rotate1-app2]

lemma *rotate-app2*:
length xs = length ys ==>
rotate n (app2 f xs ys) = app2 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′:

$(w = word\text{-}rotr\ n\ v) = (v = word\text{-}rotl\ n\ w)$
⟨*proof*⟩

lemma *word-rotr-rev*:
$word\text{-}rotr\ n\ w = word\text{-}reverse\ (word\text{-}rotl\ n\ (word\text{-}reverse\ w))$
⟨*proof*⟩

lemma *word-roti-0* [*simp*]: $word\text{-}roti\ 0\ w = w$
⟨*proof*⟩

lemmas *abl-cong* = *arg-cong* [**where** f = *of-bl*]

lemma *word-roti-add*:
$word\text{-}roti\ (m + n)\ w = word\text{-}roti\ m\ (word\text{-}roti\ n\ w)$
⟨*proof*⟩

lemma *word-roti-conv-mod'*: $word\text{-}roti\ n\ w = word\text{-}roti\ (n\ mod\ int\ (size\ w))\ w$
⟨*proof*⟩

lemmas *word-roti-conv-mod* = *word-roti-conv-mod'* [*unfolded word-size*]

13.3.3 Word rotation commutes with bit-wise operations

locale *word-rotate*

context *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-app2* [*OF lbl-lbl*] = *rotate-app2 rotater-app2*

lemmas *ths-map* [**where** *xs* = *to-bl v*] = *rotate-map rotater-map*

lemmas *th1s* [*simplified word-rot-defs'* [*symmetric*]] = *ths-app2 ths-map*

lemma *word-rot-logs*:
$word\text{-}rotl\ n\ (NOT\ v) = NOT\ word\text{-}rotl\ n\ v$
$word\text{-}rotr\ n\ (NOT\ v) = NOT\ word\text{-}rotr\ n\ v$
$word\text{-}rotl\ n\ (x\ AND\ y) = word\text{-}rotl\ n\ x\ AND\ word\text{-}rotl\ n\ y$
$word\text{-}rotr\ n\ (x\ AND\ y) = word\text{-}rotr\ n\ x\ AND\ word\text{-}rotr\ n\ y$
$word\text{-}rotl\ n\ (x\ OR\ y) = word\text{-}rotl\ n\ x\ OR\ word\text{-}rotl\ n\ y$
$word\text{-}rotr\ n\ (x\ OR\ y) = word\text{-}rotr\ n\ x\ OR\ word\text{-}rotr\ n\ y$
$word\text{-}rotl\ n\ (x\ XOR\ y) = word\text{-}rotl\ n\ x\ XOR\ word\text{-}rotl\ n\ y$
$word\text{-}rotr\ n\ (x\ XOR\ y) = word\text{-}rotr\ n\ x\ XOR\ word\text{-}rotr\ n\ y$

⟨*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′, *standard*]

lemmas *bl-word-rotr-dt* = *trans* [*OF to-bl-rotr rotater-drop-take*,
simplified word-bl.Rep′, *standard*]

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*], *standard*]
lemmas *word-rotr-dt* = *bl-word-rotr-dt*
[*THEN word-bl.Rep-inverse′* [*symmetric*], *standard*]
lemmas *word-roti-dt* = *bl-word-roti-dt*
[*THEN word-bl.Rep-inverse′* [*symmetric*], *standard*]

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*=*number-of w*, *unfolded to-bl-no-bin*, *standard*]

lemmas *word-rotl-dt-no-bin′* [*simp*] =
word-rotl-dt [**where** *w*=*number-of w*, *unfolded to-bl-no-bin*, *standard*]

declare *word-roti-def* [*simp*]

end

14 Boolean-Algebra: Boolean Algebras

theory *Boolean-Algebra*
imports *Main*
begin

locale *boolean* =
  fixes *conj* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\sqcap$ *70*)
  fixes *disj* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixr** $\sqcup$ *65*)
  fixes *compl* :: $'a \Rightarrow 'a$ ($\sim$ - [*81*] *80*)
  fixes *zero* :: $'a$ (**0**)
  fixes *one* :: $'a$ (**1**)
  assumes *conj-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
  assumes *disj-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
  assumes *conj-commute*: $x \sqcap y = y \sqcap x$
  assumes *disj-commute*: $x \sqcup y = y \sqcup x$
  assumes *conj-disj-distrib*: $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
  assumes *disj-conj-distrib*: $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
  assumes *conj-one-right* [*simp*]: $x \sqcap \mathbf{1} = x$
  assumes *disj-zero-right* [*simp*]: $x \sqcup \mathbf{0} = x$
  assumes *conj-cancel-right* [*simp*]: $x \sqcap \sim x = \mathbf{0}$
  assumes *disj-cancel-right* [*simp*]: $x \sqcup \sim x = \mathbf{1}$
begin

lemmas *disj-ac* =
  disj-assoc disj-commute
  mk-left-commute [**where** $'a = 'a$, *of disj*, *OF disj-assoc disj-commute*]

lemmas *conj-ac* =
  conj-assoc conj-commute
  mk-left-commute [**where** $'a = 'a$, *of conj*, *OF conj-assoc conj-commute*]

lemma *dual*: *boolean disj conj compl one zero*
⟨*proof*⟩

14.1 Complement

lemma *complement-unique*:
  assumes *1*: $a \sqcap x = \mathbf{0}$
  assumes *2*: $a \sqcup x = \mathbf{1}$
  assumes *3*: $a \sqcap y = \mathbf{0}$
  assumes *4*: $a \sqcup y = \mathbf{1}$
  shows $x = y$
⟨*proof*⟩

lemma *compl-unique*: $[\![x \sqcap y = \mathbf{0}; x \sqcup y = \mathbf{1}]\!] \Longrightarrow \sim x = y$
⟨*proof*⟩

lemma *double-compl* [*simp*]: $\sim (\sim x) = x$
⟨*proof*⟩

lemma *compl-eq-compl-iff* [*simp*]: $(\sim x = \sim y) = (x = y)$
⟨*proof*⟩

14.2 Conjunction

lemma *conj-absorb* [*simp*]: $x \sqcap x = x$
⟨*proof*⟩

lemma *conj-zero-right* [*simp*]: $x \sqcap \mathbf{0} = \mathbf{0}$
⟨*proof*⟩

lemma *compl-one* [*simp*]: $\sim \mathbf{1} = \mathbf{0}$
⟨*proof*⟩

lemma *conj-zero-left* [*simp*]: $\mathbf{0} \sqcap x = \mathbf{0}$
⟨*proof*⟩

lemma *conj-one-left* [*simp*]: $\mathbf{1} \sqcap x = x$
⟨*proof*⟩

lemma *conj-cancel-left* [*simp*]: $\sim x \sqcap x = \mathbf{0}$
⟨*proof*⟩

lemma *conj-left-absorb* [*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$
⟨*proof*⟩

lemma *conj-disj-distrib2*:
$(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
⟨*proof*⟩

lemmas *conj-disj-distribs* =
conj-disj-distrib conj-disj-distrib2

14.3 Disjunction

lemma *disj-absorb* [*simp*]: $x \sqcup x = x$
⟨*proof*⟩

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$
⟨*proof*⟩

lemma *compl-zero* [*simp*]: $\sim \mathbf{0} = \mathbf{1}$
⟨*proof*⟩

lemma *disj-zero-left* [*simp*]: $\mathbf{0} \sqcup x = x$
⟨*proof*⟩

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
⟨*proof*⟩

lemma *disj-cancel-left* [*simp*]: $\sim x \sqcup x = \mathbf{1}$
⟨*proof*⟩

lemma *disj-left-absorb* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
⟨*proof*⟩

lemma *disj-conj-distrib2*:
$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
⟨*proof*⟩

lemmas *disj-conj-distribs* =
disj-conj-distrib disj-conj-distrib2

14.4 De Morgan's Laws

lemma *de-Morgan-conj* [*simp*]: $\sim (x \sqcap y) = \sim x \sqcup \sim y$
⟨*proof*⟩

lemma *de-Morgan-disj* [*simp*]: $\sim (x \sqcup y) = \sim x \sqcap \sim y$
⟨*proof*⟩

end

14.5 Symmetric Difference

locale *boolean-xor* = *boolean* +
fixes *xor* :: *'a => 'a => 'a* (**infixr** ⊕ *65*)
assumes *xor-def*: $x \oplus y = (x \sqcap \sim y) \sqcup (\sim x \sqcap y)$
begin

lemma *xor-def2*:
$x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$
⟨*proof*⟩

lemma *xor-commute*: $x \oplus y = y \oplus x$
⟨*proof*⟩

lemma *xor-assoc*: $(x \oplus y) \oplus z = x \oplus (y \oplus z)$
⟨*proof*⟩

lemmas *xor-ac* =
xor-assoc xor-commute
mk-left-commute [**where** *'a* = *'a*, *of xor*, *OF xor-assoc xor-commute*]

lemma *xor-zero-right* [*simp*]: $x \oplus \mathbf{0} = x$
⟨*proof*⟩

lemma *xor-zero-left* [*simp*]: $\mathbf{0} \oplus x = x$
⟨*proof*⟩

lemma *xor-one-right* [*simp*]: $x \oplus \mathbf{1} = \sim x$
⟨*proof*⟩

lemma *xor-one-left* [*simp*]: $\mathbf{1} \oplus x = \sim x$
⟨*proof*⟩

lemma *xor-self* [*simp*]: $x \oplus x = \mathbf{0}$
⟨*proof*⟩

lemma *xor-left-self* [*simp*]: $x \oplus (x \oplus y) = y$
⟨*proof*⟩

lemma *xor-compl-left*: $\sim x \oplus y = \sim (x \oplus y)$
⟨*proof*⟩

lemma *xor-compl-right*: $x \oplus \sim y = \sim (x \oplus y)$
⟨*proof*⟩

lemma *xor-cancel-right* [*simp*]: $x \oplus \sim x = \mathbf{1}$
⟨*proof*⟩

lemma *xor-cancel-left* [*simp*]: $\sim x \oplus x = \mathbf{1}$
⟨*proof*⟩

lemma *conj-xor-distrib*: $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$
⟨*proof*⟩

lemma *conj-xor-distrib2*:
$(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$
⟨*proof*⟩

lemmas *conj-xor-distribs* =
conj-xor-distrib conj-xor-distrib2

end

end

15 WordGenLib: Miscellaneous Library for Words

theory *WordGenLib* **imports** *WordShift Boolean-Algebra*
begin

declare *of-nat-2p* [*simp*]

lemma *word-int-cases*:
$[\![\bigwedge n.\ [\![(x :: 'a::len0\ word) = word\text{-}of\text{-}int\ n;\ 0 \le n;\ n < 2\hat{\ }len\text{-}of\ TYPE('a)]\!] \Longrightarrow P]\!]$
$\Longrightarrow P$
⟨*proof*⟩

lemma *word-nat-cases* [*cases type*: *word*]:
$[\![\bigwedge n.\ [\![(x :: 'a::len\ word) = of\text{-}nat\ n;\ n < 2\hat{}len\text{-}of\ TYPE('a)]\!] \Longrightarrow P]\!]$
$\Longrightarrow P$
⟨*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*]:
$(max\text{-}word::'a::len\ word)\ !!\ n = (n < len\text{-}of\ TYPE('a))$
⟨*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*:
$x\ AND\ (y\ OR\ z) = x\ AND\ y\ OR\ x\ AND\ (z::'a::len0\ word)$
⟨*proof*⟩

lemma *word-oa-dist2*:

$x\ OR\ y\ AND\ z = (x\ OR\ y)\ AND\ (x\ OR\ (z::'a::len0\ word))$
⟨*proof*⟩

lemma *word-and-not* [*simp*]:
$x\ AND\ NOT\ x = (0::'a::len0\ word)$
⟨*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*:
$boolean\ [op\ AND\ op\ OR\ bitNOT\ 0\ max\text{-}word]$
⟨*proof*⟩

lemma *word-xor-and-or*:
$x\ XOR\ y = x\ AND\ NOT\ y\ OR\ NOT\ x\ AND\ (y::'a::len0\ word)$
⟨*proof*⟩

interpretation *word-bool-alg*:
$boolean\text{-}xor\ [op\ AND\ op\ OR\ bitNOT\ 0\ max\text{-}word\ op\ XOR]$
⟨*proof*⟩

lemma *shiftr-0* [*iff*]:
$(x::'a::len0\ word) >> 0 = x$
⟨*proof*⟩

lemma *shiftl-0* [*simp*]:
$(x :: 'a :: len\ word) << 0 = x$
⟨*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*]:
$((x::'a::len\ word) < 1) = (x = 0)$
⟨*proof*⟩

lemma *to-bl-mask*:
to-bl (*mask n* :: *'a::len word*) =
replicate (*len-of TYPE*(*'a*) − *n*) *False* @
replicate (*min* (*len-of TYPE*(*'a*)) *n*) *True*
⟨*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*
fixes *n*
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::'a::len word*)) =
(*if unat x* + *unat y* < *2^size x then*

$unat\ x + unat\ y$
$else$
$unat\ x + unat\ y - 2\hat{\ }size\ x)$
⟨proof⟩

lemma *word-neq-0-conv* [*simp*]:
fixes $w :: 'a :: len\ word$
shows $(w \neq 0) = (0 < w)$
⟨proof⟩

lemma *max-lt*:
$unat\ (max\ a\ b\ div\ c) = unat\ (max\ a\ b)\ div\ unat\ (c{::}\ 'a :: len\ word)$
⟨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-simple*:
$x \leq y \Longrightarrow unat\ (y - x) = unat\ y - unat\ x$
⟨proof⟩

lemmas *unat-sub* = *unat-sub-simple*

lemma *word-less-sub1*:
fixes $x :: 'a :: len\ word$
shows $x \neq 0 \Longrightarrow 1 < x = (0 < x - 1)$
⟨proof⟩

lemma *word-le-sub1*:
fixes $x :: 'a :: len\ word$
shows $x \neq 0 \Longrightarrow 1 \leq x = (0 \leq x - 1)$
⟨proof⟩

lemmas *word-less-sub1-numberof* [*simp*] =
word-less-sub1 [*of number-of w*, *standard*]
lemmas *word-le-sub1-numberof* [*simp*] =
word-le-sub1 [*of number-of w*, *standard*]

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

lemmas *word-of-int-inj* =
word-uint.Abs-inject [*unfolded uints-num*, *simplified*]

lemma *word-le-less-eq*:
$(x :: 'z::len\ word) \leq y = (x = y \vee x < y)$
$\langle proof \rangle$

lemma *mod-plus-cong*:
assumes *1*: $(b::int) = b'$
and *2*: $x\ mod\ b' = x'\ mod\ b'$
and *3*: $y\ mod\ b' = y'\ mod\ b'$
and *4*: $x' + y' = z'$
shows $(x + y)\ mod\ b = z'\ mod\ b'$
$\langle proof \rangle$

lemma *mod-minus-cong*:
assumes *1*: $(b::int) = b'$
and *2*: $x\ mod\ b' = x'\ mod\ b'$
and *3*: $y\ mod\ b' = y'\ mod\ b'$
and *4*: $x' - y' = z'$
shows $(x - y)\ mod\ b = z'\ mod\ b'$
$\langle proof \rangle$

lemma *word-induct-less*:
$[\![P\ (0::'a::len\ word);\ \bigwedge n.\ [\![n < m;\ P\ n]\!] \Longrightarrow P\ (1 + n)]\!] \Longrightarrow P\ m$
$\langle proof \rangle$

lemma *word-induct*:
$[\![P\ (0::'a::len\ word);\ \bigwedge n.\ P\ n \Longrightarrow P\ (1 + n)]\!] \Longrightarrow P\ m$
$\langle proof \rangle$

lemma *word-induct2* [*induct type*]:
$[\![P\ 0;\ \bigwedge n.\ [\![1 + n \neq 0;\ P\ n]\!] \Longrightarrow P\ (1 + n)]\!] \Longrightarrow P\ (n::'b::len\ word)$
$\langle proof \rangle$

constdefs
$word\text{-}rec :: 'a \Rightarrow ('b::len\ word \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b\ word \Rightarrow 'a$
$word\text{-}rec\ forZero\ forSuc\ n \equiv nat\text{-}rec\ forZero\ (forSuc \circ of\text{-}nat)\ (unat\ n)$

lemma *word-rec-0*: $word\text{-}rec\ z\ s\ 0 = z$
$\langle proof \rangle$

lemma *word-rec-Suc*:
$1 + n \neq (0::'a::len\ word) \Longrightarrow word\text{-}rec\ z\ s\ (1 + n) = s\ n\ (word\text{-}rec\ z\ s\ n)$
$\langle proof \rangle$

lemma *word-rec-Pred*:
$n \neq 0 \Longrightarrow word\text{-}rec\ z\ s\ n = s\ (n - 1)\ (word\text{-}rec\ z\ s\ (n - 1))$
$\langle proof \rangle$

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) ($f \circ op + 1$) n
⟨*proof*⟩

lemma *word-rec-twice*:
$m \leq n \Longrightarrow$ *word-rec* z f n = *word-rec* (*word-rec* z f $(n - m)$) ($f \circ op + (n - m)$) 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*:
$1 + n \neq (0::'a::len\ word) \Longrightarrow unat\ (1 + n) = Suc\ (unat\ n)$
⟨*proof*⟩

end

16 WordMain: Main Word Library

theory *WordMain* **imports** *WordGenLib*
begin

lemmas *word-no-1* [*simp*] = *word-1-no* [*symmetric*]
lemmas *word-no-0* [*simp*] = *word-0-no* [*symmetric*]

declare *word-0-bl* [*simp*]
declare *bin-to-bl-def* [*simp*]
declare *to-bl-0* [*simp*]
declare *of-bl-True* [*simp*]

Examples

types *word32* = *32 word*
types *word8* = *8 word*
types *byte* = *word8*

for more see WordExampes.thy

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.