

# Machine Words in Isabelle/HOL

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

May 22, 2012

## Abstract

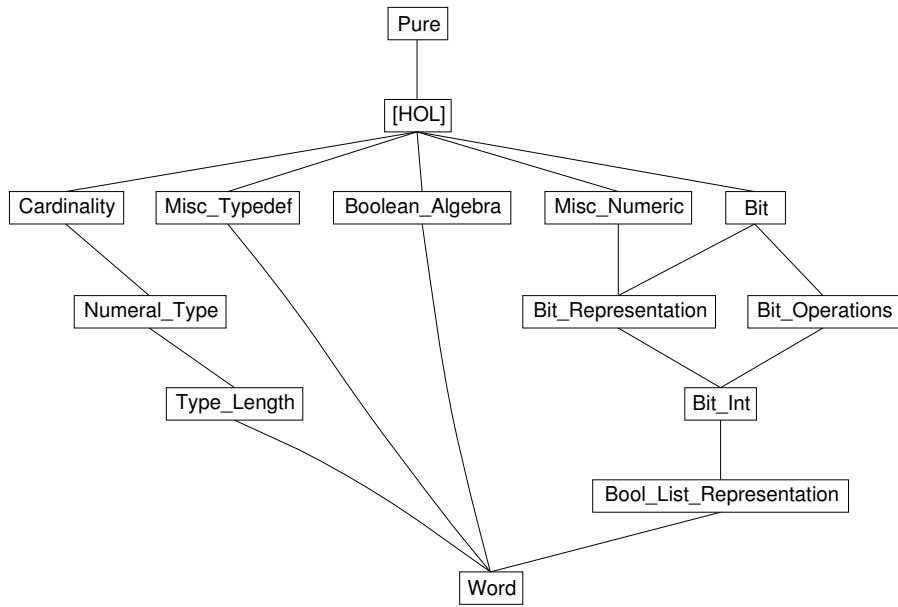
A formalisation of generic, fixed size machine words in Isabelle/HOL.  
An earlier version of this formalisation is described in [1].

## Contents

<b>1</b>	<b>Cardinality: Cardinality of types</b>	<b>5</b>
1.1	Preliminary lemmas . . . . .	5
1.2	Cardinalities of types . . . . .	5
1.3	Classes with at least 1 and 2 . . . . .	5
<b>2</b>	<b>Numeral-Type: Numeral Syntax for Types</b>	<b>6</b>
2.1	Numeral Types . . . . .	6
2.2	Locales for modular arithmetic subtypes . . . . .	7
2.3	Ring class instances . . . . .	9
2.4	Syntax . . . . .	10
2.5	Examples . . . . .	11
<b>3</b>	<b>Type-Length: Assigning lengths to types by typeclasses</b>	<b>11</b>
<b>4</b>	<b>Misc-Typedef: Type Definition Theorems</b>	<b>12</b>
<b>5</b>	<b>More lemmas about normal type definitions</b>	<b>12</b>
5.1	Extended form of type definition predicate . . . . .	14
<b>6</b>	<b>Boolean-Algebra: Boolean Algebras</b>	<b>15</b>
6.1	Complement . . . . .	16
6.2	Conjunction . . . . .	17
6.3	Disjunction . . . . .	17
6.4	De Morgan's Laws . . . . .	18
6.5	Symmetric Difference . . . . .	18
<b>7</b>	<b>Misc-Numeric: Useful Numerical Lemmas</b>	<b>19</b>

<b>8 Bit: The Field of Integers mod 2</b>	<b>25</b>
8.1 Bits as a datatype . . . . .	25
8.2 Type <i>bit</i> forms a field . . . . .	25
8.3 Numerals at type <i>bit</i> . . . . .	26
<b>9 Bit-Representation: Basic Definitions for Binary Integers</b>	<b>27</b>
9.1 Further properties of numerals . . . . .	27
9.2 Destructors for binary integers . . . . .	29
9.3 Truncating binary integers . . . . .	30
9.4 Simplifications for (s)bintrunc . . . . .	31
9.5 Splitting and concatenation . . . . .	39
9.6 Miscellaneous lemmas . . . . .	39
<b>10 Bit-Operations: Syntactic classes for bitwise operations</b>	<b>40</b>
10.1 Bitwise operations on <i>bit</i> . . . . .	41
<b>11 Bit-Int: Bitwise Operations on Binary Integers</b>	<b>42</b>
11.1 Logical operations . . . . .	42
11.1.1 Basic simplification rules . . . . .	43
11.1.2 Binary destructors . . . . .	44
11.1.3 Derived properties . . . . .	45
11.1.4 Simplification with numerals . . . . .	47
11.1.5 Interactions with arithmetic . . . . .	50
11.1.6 Truncating results of bit-wise operations . . . . .	50
11.2 Setting and clearing bits . . . . .	51
11.3 Splitting and concatenation . . . . .	52
11.4 Miscellaneous lemmas . . . . .	54
<b>12 Bool-List-Representation: Bool lists and integers</b>	<b>55</b>
12.1 Operations on lists of booleans . . . . .	55
12.2 Arithmetic in terms of bool lists . . . . .	56
12.3 Repeated splitting or concatenation . . . . .	69
<b>13 Word: A type of finite bit strings</b>	<b>72</b>
13.1 Type definition . . . . .	72
13.2 Basic code generation setup . . . . .	73
13.3 Type conversions and casting . . . . .	73
13.4 Type-definition locale instantiations . . . . .	75
13.5 Correspondence relation for theorem transfer . . . . .	76
13.6 Arithmetic operations . . . . .	76
13.7 Ordering . . . . .	78
13.8 Bit-wise operations . . . . .	79
13.9 Shift operations . . . . .	80
13.10 Rotation . . . . .	80

13.11	Split and cat operations . . . . .	81
13.12	Theorems about typedefs . . . . .	81
13.13	Testing bits . . . . .	86
13.14	Word Arithmetic . . . . .	92
13.15	Transferring goals from words to ints . . . . .	94
13.16	Order on fixed-length words . . . . .	95
13.17	Conditions for the addition (etc) of two words to overflow . . . . .	96
13.18	Definition of uint_arith . . . . .	97
13.19	More on overflows and monotonicity . . . . .	97
13.20	Arithmetic type class instantiations . . . . .	101
13.21	Word and nat . . . . .	101
13.22	Definition of unat_arith tactic . . . . .	104
13.23	Cardinality, finiteness of set of words . . . . .	107
13.24	Bitwise Operations on Words . . . . .	107
13.25	Shifting, Rotating, and Splitting Words . . . . .	116
	13.25.1 shift functions in terms of lists of bools . . . . .	117
	13.25.2 Mask . . . . .	121
	13.25.3 Revcast . . . . .	122
	13.25.4 Slices . . . . .	124
13.26	Split and cat . . . . .	126
	13.26.1 Split and slice . . . . .	128
13.27	Rotation . . . . .	130
	13.27.1 Rotation of list to right . . . . .	130
	13.27.2 map, map2, commuting with rotate(r) . . . . .	132
	13.27.3 Word rotation commutes with bit-wise operations . . . . .	134
13.28	Maximum machine word . . . . .	135
13.29	Recursion combinator for words . . . . .	140



## 1 Cardinality: Cardinality of types

```
theory Cardinality
imports ~~/src/HOL/Main
begin
```

### 1.1 Preliminary lemmas

```
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) => CONST card (CONST UNIV :: 't set)
```

```
<ML>
```

```
lemma card-unit [simp]: CARD(unit) = 1
<proof>
```

```
lemma card-prod [simp]: CARD('a × 'b) = CARD('a::finite) * CARD('b::finite)
<proof>
```

```
lemma card-sum [simp]: CARD('a + 'b) = CARD('a::finite) + CARD('b::finite)
<proof>
```

```
lemma card-option [simp]: CARD('a option) = Suc CARD('a::finite)
<proof>
```

```
lemma card-set [simp]: CARD('a set) = 2 ^ CARD('a::finite)
<proof>
```

```
lemma card-nat [simp]: CARD(nat) = 0
<proof>
```

### 1.3 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⟩
```

```
lemma one-less-int-card: 1 < int CARD('a::card2)
  ⟨proof⟩
```

```
end
```

## 2 Numeral-Type: Numeral Syntax for Types

```
theory Numeral-Type
imports Cardinality
begin
```

### 2.1 Numeral Types

```
typedef (open) num0 = UNIV :: nat set ⟨proof⟩
typedef (open) num1 = UNIV :: unit set ⟨proof⟩
```

```
typedef (open) 'a bit0 = {0 ..< 2 * int CARD('a::finite)}
  ⟨proof⟩
```

```
typedef (open) 'a bit1 = {0 ..< 1 + 2 * int CARD('a::finite)}
  ⟨proof⟩
```

```
lemma card-num0 [simp]: CARD (num0) = 0
  ⟨proof⟩
```

```
lemma card-num1 [simp]: CARD(num1) = 1
  ⟨proof⟩
```

```
lemma card-bit0 [simp]: CARD('a bit0) = 2 * CARD('a::finite)
  ⟨proof⟩
```

```
lemma card-bit1 [simp]: CARD('a bit1) = Suc (2 * CARD('a::finite))
  ⟨proof⟩
```

```
instance num1 :: finite
  ⟨proof⟩
```

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

```
instance bit1 :: (finite) card2
```

*<proof>*

## 2.2 Locales for modular arithmetic subtypes

```

locale mod-type =
  fixes n :: int
  and Rep :: 'a::{zero,one,plus,times,uminus,minus}  $\Rightarrow$  int
  and Abs :: int  $\Rightarrow$  'a::{zero,one,plus,times,uminus,minus}
  assumes type: type-definition Rep Abs {0.. $n$ }
  and size1: 1 < n
  and zero-def: 0 = Abs 0
  and one-def: 1 = Abs 1
  and add-def: x + y = Abs ((Rep x + Rep y) mod n)
  and mult-def: x * y = Abs ((Rep x * Rep y) mod n)
  and diff-def: x - y = Abs ((Rep x - Rep y) mod n)
  and minus-def: - x = Abs ((- Rep x) mod n)
begin

```

```

lemma size0: 0 < n
<proof>

```

```

lemmas definitions =
  zero-def one-def add-def mult-def minus-def diff-def

```

```

lemma Rep-less-n: Rep x < n
<proof>

```

```

lemma Rep-le-n: Rep x  $\leq$  n
<proof>

```

```

lemma Rep-inject-sym: x = y  $\longleftrightarrow$  Rep x = Rep y
<proof>

```

```

lemma Rep-inverse: Abs (Rep x) = x
<proof>

```

```

lemma Abs-inverse: m  $\in$  {0.. $n$ }  $\Longrightarrow$  Rep (Abs m) = m
<proof>

```

```

lemma Rep-Abs-mod: Rep (Abs (m mod n)) = m mod n
<proof>

```

```

lemma Rep-Abs-0: Rep (Abs 0) = 0
<proof>

```

```

lemma Rep-0: Rep 0 = 0
<proof>

```

```

lemma Rep-Abs-1: Rep (Abs 1) = 1

```

*<proof>*

**lemma** *Rep-1*:  $Rep\ 1 = 1$

*<proof>*

**lemma** *Rep-mod*:  $Rep\ x\ mod\ n = Rep\ x$

*<proof>*

**lemmas** *Rep-simps* =

*Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1*

**lemma** *comm-ring-1*: *OFCLASS*('a, *comm-ring-1-class*)

*<proof>*

**end**

**locale** *mod-ring* = *mod-type* n *Rep* *Abs*

**for** *n* :: *int*

**and** *Rep* :: 'a::{*comm-ring-1*}  $\Rightarrow$  *int*

**and** *Abs* :: *int*  $\Rightarrow$  'a::{*comm-ring-1*}

**begin**

**lemma** *of-nat-eq*:  $of\ nat\ k = Abs\ (int\ k\ mod\ n)$

*<proof>*

**lemma** *of-int-eq*:  $of\ int\ z = Abs\ (z\ mod\ n)$

*<proof>*

**lemma** *Rep-numeral*:

$Rep\ (numeral\ w) = numeral\ w\ mod\ n$

*<proof>*

**lemma** *iszero-numeral*:

$iszero\ (numeral\ w::'a) \longleftrightarrow numeral\ w\ mod\ n = 0$

*<proof>*

**lemma** *cases*:

**assumes** *1*:  $\bigwedge z. \llbracket (x::'a) = of\ int\ z; 0 \leq z; z < n \rrbracket \Longrightarrow P$

**shows** *P*

*<proof>*

**lemma** *induct*:

$(\bigwedge z. \llbracket 0 \leq z; z < n \rrbracket \Longrightarrow P\ (of\ int\ z)) \Longrightarrow P\ (x::'a)$

*<proof>*

**end**

### 2.3 Ring class instances

Unfortunately *ring-1* instance is not possible for *num1*, since 0 and 1 are not distinct.

**instantiation** *num1* :: {*comm-ring,comm-monoid-mult,numeral*}  
**begin**

**lemma** *num1-eq-iff*: (*x*::*num1*) = (*y*::*num1*)  $\longleftrightarrow$  *True*  
 ⟨*proof*⟩

**instance** ⟨*proof*⟩

**end**

**instantiation**

*bit0* and *bit1* :: (*finite*) {*zero,one,plus,times,uminus,minus*}  
**begin**

**definition** *Abs-bit0'* :: *int*  $\Rightarrow$  '*a bit0* **where**  
*Abs-bit0'* *x* = *Abs-bit0* (*x mod int CARD('a bit0)*)

**definition** *Abs-bit1'* :: *int*  $\Rightarrow$  '*a bit1* **where**  
*Abs-bit1'* *x* = *Abs-bit1* (*x mod int CARD('a bit1)*)

**definition** *0* = *Abs-bit0 0*

**definition** *1* = *Abs-bit0 1*

**definition** *x + y* = *Abs-bit0'* (*Rep-bit0 x + Rep-bit0 y*)

**definition** *x \* y* = *Abs-bit0'* (*Rep-bit0 x \* Rep-bit0 y*)

**definition** *x - y* = *Abs-bit0'* (*Rep-bit0 x - Rep-bit0 y*)

**definition**  $- x$  = *Abs-bit0'* ( $- \text{Rep-bit0 } x$ )

**definition** *0* = *Abs-bit1 0*

**definition** *1* = *Abs-bit1 1*

**definition** *x + y* = *Abs-bit1'* (*Rep-bit1 x + Rep-bit1 y*)

**definition** *x \* y* = *Abs-bit1'* (*Rep-bit1 x \* Rep-bit1 y*)

**definition** *x - y* = *Abs-bit1'* (*Rep-bit1 x - Rep-bit1 y*)

**definition**  $- x$  = *Abs-bit1'* ( $- \text{Rep-bit1 } x$ )

**instance** ⟨*proof*⟩

**end**

**interpretation** *bit0*:

*mod-type int CARD('a::finite bit0)*

*Rep-bit0* :: '*a::finite bit0*  $\Rightarrow$  *int*

*Abs-bit0* :: *int*  $\Rightarrow$  '*a::finite bit0*

⟨*proof*⟩

**interpretation** *bit1*:

```

  mod-type int CARD('a::finite bit1)
    Rep-bit1 :: 'a::finite bit1 => int
    Abs-bit1 :: int => 'a::finite bit1
  <proof>

```

```

instance bit0 :: (finite) comm-ring-1
  <proof>

```

```

instance bit1 :: (finite) comm-ring-1
  <proof>

```

```

interpretation bit0:
  mod-ring int CARD('a::finite bit0)
    Rep-bit0 :: 'a::finite bit0 => int
    Abs-bit0 :: int => 'a::finite bit0
  <proof>

```

```

interpretation bit1:
  mod-ring int CARD('a::finite bit1)
    Rep-bit1 :: 'a::finite bit1 => int
    Abs-bit1 :: int => 'a::finite bit1
  <proof>

```

Set up cases, induction, and arithmetic

```

lemmas bit0-cases [case-names of-int, cases type: bit0] = bit0.cases
lemmas bit1-cases [case-names of-int, cases type: bit1] = bit1.cases

```

```

lemmas bit0-induct [case-names of-int, induct type: bit0] = bit0.induct
lemmas bit1-induct [case-names of-int, induct type: bit1] = bit1.induct

```

```

lemmas bit0-iszero-numeral [simp] = bit0.iszero-numeral
lemmas bit1-iszero-numeral [simp] = bit1.iszero-numeral

```

```

declare eq-numeral-iff-iszero [where 'a=( 'a::finite) bit0, standard, simp]
declare eq-numeral-iff-iszero [where 'a=( 'a::finite) bit1, standard, simp]

```

## 2.4 Syntax

```

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

```

```

translations
  (type) 1 == (type) num1
  (type) 0 == (type) num0

```

<ML>

## 2.5 Examples

```

lemma  $CARD(0) = 0$  <proof>
lemma  $CARD(17) = 17$  <proof>
lemma  $8 * 11 ^ 3 - 6 = (2::5)$  <proof>

end

```

## 3 Type-Length: Assigning lengths to types by type-classes

```

theory Type-Length
imports ~/src/HOL/Library/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*.

```

class len0 =
  fixes len-of :: 'a itself  $\Rightarrow$  nat

```

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

```

class len = len0 +
  assumes len-gt-0 [iff]:  $0 < \text{len-of } TYPE ('a)$ 

```

```

instantiation num0 and num1 :: len0
begin

```

```

definition
  len-num0:  $\text{len-of } (x::\text{num0 } \textit{itself}) = 0$ 

```

```

definition
  len-num1:  $\text{len-of } (x::\text{num1 } \textit{itself}) = 1$ 

```

```

instance <proof>

```

```

end

```

```

instantiation bit0 and bit1 :: (len0) len0
begin

```

```

definition
  len-bit0:  $\text{len-of } (x::'a::\text{len0 } \textit{bit0 } \textit{itself}) = 2 * \text{len-of } TYPE ('a)$ 

```

```

definition
  len-bit1:  $\text{len-of } (x::'a::\text{len0 } \textit{bit1 } \textit{itself}) = 2 * \text{len-of } TYPE ('a) + 1$ 

```

```

instance ⟨proof⟩

end

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⟩

end

```

## 4 Misc-Typedef: Type Definition Theorems

```

theory Misc-Typedef
imports Main
begin

```

## 5 More lemmas about normal type definitions

```

lemma
  tdD1: type-definition Rep Abs A  $\implies \forall x. \text{Rep } x \in A$  and
  tdD2: type-definition Rep Abs A  $\implies \forall x. \text{Abs } (\text{Rep } x) = x$  and
  tdD3: type-definition Rep Abs A  $\implies \forall y. y \in A \longrightarrow \text{Rep } (\text{Abs } y) = y$ 
  ⟨proof⟩

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

context type-definition
begin

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

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

lemma Abs-inverse':
  r : A  $\implies \text{Abs } r = a \implies \text{Rep } a = r$ 
  ⟨proof⟩

lemma Rep-comp-inverse:
  Rep o f = g  $\implies \text{Abs } o g = f$ 
  ⟨proof⟩

```

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

**lemma** *Rep-inverse'*:  $Rep\ a = r \implies Abs\ r = a$   
 ⟨*proof*⟩

**lemma** *comp-Abs-inverse*:  
 $f\ o\ Abs = g \implies g\ o\ Rep = f$   
 ⟨*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*:  
 $Rep\ o\ fa = fr\ o\ Rep \mid fa\ o\ Abs = Abs\ o\ fr \implies Abs\ o\ fr\ o\ Rep = fa$   
 ⟨*proof*⟩

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

**lemma** *fns4*:  
 $Rep\ o\ fa\ o\ Abs = fr \implies$   
 $Rep\ o\ fa = fr\ o\ Rep \ \& \ fa\ o\ Abs = Abs\ o\ fr$   
 ⟨*proof*⟩

**end**

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

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

## 5.1 Extended form of type definition predicate

**lemma** *td-conds*:

$norm \ o \ norm = norm \ ==> \ (fr \ o \ norm = norm \ o \ fr) =$   
 $(norm \ o \ fr \ o \ norm = fr \ o \ norm \ \& \ norm \ o \ fr \ o \ norm = norm \ o \ fr)$   
 $\langle proof \rangle$

**lemma** *fn-comm-power*:

$fa \ o \ tr = tr \ o \ fr \ ==> \ fa \ \wedge \wedge \ n \ o \ tr = tr \ o \ fr \ \wedge \wedge \ n$   
 $\langle proof \rangle$

**lemmas** *fn-comm-power'* =

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

**locale** *td-ext* = *type-definition* +

**fixes** *norm*

**assumes** *eq-norm*:  $\bigwedge x. Rep \ (Abs \ x) = norm \ x$

**begin**

**lemma** *Abs-norm* [*simp*]:

$Abs \ (norm \ x) = Abs \ x$   
 $\langle proof \rangle$

**lemma** *td-th*:

$g \ o \ Abs = f \ ==> \ f \ (Rep \ x) = g \ x$   
 $\langle proof \rangle$

**lemma** *eq-norm'*:  $Rep \ o \ Abs = norm$

$\langle proof \rangle$

**lemma** *norm-Rep* [*simp*]:  $norm \ (Rep \ x) = Rep \ x$

$\langle proof \rangle$

**lemmas** *td* = *td-thm*

**lemma** *set-iff-norm*:  $w : A \ <-> \ w = norm \ w$

$\langle proof \rangle$

**lemma** *inverse-norm*:

$(Abs \ n = w) = (Rep \ w = norm \ n)$   
 $\langle proof \rangle$

**lemma** *norm-eq-iff*:

$(norm \ x = norm \ y) = (Abs \ x = Abs \ y)$   
 $\langle proof \rangle$

**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**

## 6 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  $\sqcap$  1 = x
assumes disj-zero-right [simp]: x  $\sqcup$  0 = x
assumes conj-cancel-right [simp]: x  $\sqcap$   $\sim$  x = 0
assumes disj-cancel-right [simp]: x  $\sqcup$   $\sim$  x = 1

```

**sublocale** boolean < conj!: abel-semigroup conj <proof>

**sublocale** boolean < disj!: abel-semigroup disj <proof>

**context** boolean  
**begin**

**lemmas** conj-left-commute = conj.left-commute

**lemmas** disj-left-commute = disj.left-commute

**lemmas** conj-ac = conj.assoc conj.commute conj.left-commute

**lemmas** disj-ac = disj.assoc disj.commute disj.left-commute

**lemma** dual: boolean disj conj compl one zero  
<proof>

## 6.1 Complement

**lemma** complement-unique:

**assumes** 1: a  $\sqcap$  x = **0**

**assumes** 2: a  $\sqcup$  x = **1**

**assumes** 3: a  $\sqcap$  y = **0**

**assumes** 4: a  $\sqcup$  y = **1**

**shows** x = y

<proof>

**lemma** compl-unique:  $\llbracket x \sqcap y = \mathbf{0}; x \sqcup y = \mathbf{1} \rrbracket \implies \sim x = y$

<proof>

**lemma** double-compl [simp]:  $\sim (\sim x) = x$

<proof>

**lemma** compl-eq-compl-iff [simp]:  $(\sim x = \sim y) = (x = y)$

<proof>

## 6.2 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-distrib* =  
*conj-disj-distrib conj-disj-distrib2*

## 6.3 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-distrib* =  
*disj-conj-distrib disj-conj-distrib2*

## 6.4 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**

## 6.5 Symmetric Difference

**locale** *boolean-xor* = *boolean* +  
 fixes *xor* :: 'a => 'a => 'a (**infixr**  $\oplus$  65)  
 assumes *xor-def*:  $x \oplus y = (x \sqcap \sim y) \sqcup (\sim x \sqcap y)$

**sublocale** *boolean-xor* < *xor!*: *abel-semigroup xor* ⟨proof⟩

**context** *boolean-xor*  
**begin**

**lemmas** *xor-assoc* = *xor.assoc*  
**lemmas** *xor-commute* = *xor.commute*  
**lemmas** *xor-left-commute* = *xor.left-commute*

**lemmas** *xor-ac* = *xor.assoc xor.commute xor.left-commute*

**lemma** *xor-def2*:  
 $x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$   
 ⟨proof⟩

**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*]:  $\mathbf{1} \oplus x = \sim x$   
 $\langle proof \rangle$

**lemma** *xor-self* [*simp*]:  $x \oplus x = \mathbf{0}$   
 $\langle proof \rangle$

**lemma** *xor-left-self* [*simp*]:  $x \oplus (x \oplus y) = y$   
 $\langle proof \rangle$

**lemma** *xor-compl-left* [*simp*]:  $\sim x \oplus y = \sim (x \oplus y)$   
 $\langle proof \rangle$

**lemma** *xor-compl-right* [*simp*]:  $x \oplus \sim y = \sim (x \oplus y)$   
 $\langle proof \rangle$

**lemma** *xor-cancel-right*:  $x \oplus \sim x = \mathbf{1}$   
 $\langle proof \rangle$

**lemma** *xor-cancel-left*:  $\sim x \oplus x = \mathbf{1}$   
 $\langle proof \rangle$

**lemma** *conj-xor-distrib*:  $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$   
 $\langle proof \rangle$

**lemma** *conj-xor-distrib2*:  
 $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$   
 $\langle proof \rangle$

**lemmas** *conj-xor-distribs* =  
*conj-xor-distrib conj-xor-distrib2*

**end**

**end**

## 7 Misc-Numeric: Useful Numerical Lemmas

**theory** *Misc-Numeric*  
**imports**  $\sim\sim/src/HOL/Main \sim\sim/src/HOL/Parity$   
**begin**

**lemma** *the-elimI*:  $y = \{x\} \implies the-elim\ y = x$   
 $\langle proof \rangle$

**lemma** *nonemptyE*:  $S \sim = \{\} \implies (!x. x : S \implies R) \implies R$   $\langle proof \rangle$

**lemma** *gt-or-eq-0*:  $0 < y \vee 0 = (y::nat)$   $\langle proof \rangle$

**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)

**lemmas** *nat-simps* = *diff-add-inverse2* *diff-add-inverse*

**lemmas** *nat-iffs* = *le-add1* *le-add2*

**lemma** *sum-imp-diff*:  $j = k + i \implies j - i = (k :: \text{nat})$  *<proof>*

**lemma** *zless2*:  $0 < (2 :: \text{int})$  *<proof>*

**lemmas** *zless2p* = *zless2* [*THEN* *zero-less-power*]

**lemmas** *zle2p* = *zless2p* [*THEN* *order-less-imp-le*]

**lemmas** *pos-mod-sign2* = *zless2* [*THEN* *pos-mod-sign* [**where**  $b = 2 :: \text{int}$ ]]

**lemmas** *pos-mod-bound2* = *zless2* [*THEN* *pos-mod-bound* [**where**  $b = 2 :: \text{int}$ ]]

**lemma** *nmod2*:  $n \bmod (2 :: \text{int}) = 0 \mid n \bmod 2 = 1$  *<proof>*

**lemma** *emep1*:

*even*  $n \implies \text{even } d \implies 0 \leq d \implies (n + 1) \bmod (d :: \text{int}) = (n \bmod d) + 1$   
*<proof>*

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

**lemma** *le-diff-eq'*:  $(a \leq c - b) = (b + a \leq (c :: \text{int}))$  *<proof>*

**lemma** *less-diff-eq'*:  $(a < c - b) = (b + a < (c :: \text{int}))$  *<proof>*

**lemma** *diff-le-eq'*:  $(a - b \leq c) = (a \leq b + (c :: \text{int}))$  *<proof>*

**lemma** *diff-less-eq'*:  $(a - b < c) = (a < b + (c :: \text{int}))$  *<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*:

$(2 * b + 1) \bmod (2 * 2 ^ n) = 2 * (b \bmod 2 ^ n) + (1 :: \text{int})$

*<proof>*

**lemma** *z1pmod2*:

$(2 * b + 1) \bmod 2 = (1 :: \text{int})$  *<proof>*

**lemma** *z1pdiv2*:

$(2 * b + 1) \text{ div } 2 = (b :: \text{int})$  *<proof>*

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

**lemma** *axbbyy*:

$a + m + m = b + n + n \implies (a = 0 \mid a = 1) \implies (b = 0 \mid b = 1) \implies a = b \ \& \ m = (n :: \text{int})$  *<proof>*

**lemma** *axxmod2*:

$(1 + x + x) \bmod 2 = (1 :: \text{int}) \ \& \ (0 + x + x) \bmod 2 = (0 :: \text{int})$  *<proof>*

**lemma** *axxdiv2*:

$(1 + x + x) \text{ div } 2 = (x :: \text{int}) \ \& \ (0 + x + x) \text{ div } 2 = (x :: \text{int})$  *<proof>*

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

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

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

**lemma** *zmod-zsub-self [simp]*:

$((b :: \text{int}) - a) \bmod a = b \bmod a$   
*<proof>*

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

**lemma** *mod-plus-right*:

$((a + x) \bmod m = (b + x) \bmod m) = (a \bmod m = b \bmod (m :: \text{nat}))$   
*<proof>*

**lemma** *nat-minus-mod*:  $(n - n \bmod m) \bmod m = (0 :: \text{nat})$

*<proof>*

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

**lemmas** *push-mods' = mod-add-eq*

*mod-mult-eq mod-diff-eq  
mod-minus-eq*

**lemmas** *push-mods* = *push-mods'* [*THEN eq-reflection*]  
**lemmas** *pull-mods* = *push-mods* [*symmetric*] *rdmods* [*THEN eq-reflection*]  
**lemmas** *mod-simps* =  
*mod-mult-self2-is-0* [*THEN eq-reflection*]  
*mod-mult-self1-is-0* [*THEN eq-reflection*]  
*mod-mod-trivial* [*THEN eq-reflection*]

**lemma** *nat-mod-eq*:  
 $!!b. b < n \implies a \bmod n = b \bmod n \implies a \bmod n = (b :: \text{nat})$   
*<proof>*

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

**lemma** *nat-mod-lem*:  
 $(0 :: \text{nat}) < n \implies b < n = (b \bmod n = b)$   
*<proof>*

**lemma** *mod-nat-add*:  
 $(x :: \text{nat}) < z \implies y < z \implies$   
 $(x + y) \bmod z = (\text{if } x + y < z \text{ then } x + y \text{ else } x + y - z)$   
*<proof>*

**lemma** *mod-nat-sub*:  
 $(x :: \text{nat}) < z \implies (x - y) \bmod z = x - y$   
*<proof>*

**lemma** *int-mod-lem*:  
 $(0 :: \text{int}) < n \implies (0 \leq b \ \& \ b < n) = (b \bmod n = b)$   
*<proof>*

**lemma** *int-mod-eq*:  
 $(0 :: \text{int}) \leq b \implies b < n \implies a \bmod n = b \bmod n \implies a \bmod n = b$   
*<proof>*

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

**lemma** *int-mod-le*:  $(0 :: \text{int}) \leq a \implies a \bmod n \leq a$   
*<proof>*

**lemma** *int-mod-le'*:  $(0 :: \text{int}) \leq b - n \implies b \bmod n \leq b - n$   
*<proof>*

**lemma** *int-mod-ge*:  $a < n \implies 0 < (n :: \text{int}) \implies a \leq a \bmod n$   
*<proof>*

**lemma** *int-mod-ge'*:  $b < 0 \implies 0 < (n :: \text{int}) \implies b + n \leq b \bmod n$   
*<proof>*

**lemma** *mod-add-if-z*:

$(x :: \text{int}) < z \implies y < z \implies 0 \leq y \implies 0 \leq x \implies 0 \leq z \implies$   
 $(x + y) \bmod z = (\text{if } x + y < z \text{ then } x + y \text{ else } x + y - z)$   
 $\langle \text{proof} \rangle$

**lemma** *mod-sub-if-z*:

$(x :: \text{int}) < z \implies y < z \implies 0 \leq y \implies 0 \leq x \implies 0 \leq z \implies$   
 $(x - y) \bmod z = (\text{if } y \leq x \text{ then } x - y \text{ else } x - y + z)$   
 $\langle \text{proof} \rangle$

**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*:  $m \sim = (0 :: \text{int}) \implies (a + m * n) \text{ div } m = a \text{ div } m + n$   
 $\langle \text{proof} \rangle$

**lemma** *mod-power-lem*:

$a > 1 \implies a ^ n \bmod a ^ m = (\text{if } m \leq n \text{ then } 0 \text{ else } (a :: \text{int}) ^ n)$   
 $\langle \text{proof} \rangle$

**lemma** *min-pm* [*simp*]:  $\min a b + (a - b) = (a :: \text{nat})$   $\langle \text{proof} \rangle$

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

**lemma** *rev-min-pm* [*simp*]:  $\min b a + (a - b) = (a :: \text{nat})$   $\langle \text{proof} \rangle$

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

**lemma** *pl-pl-rels*:

$a + b = c + d \implies$   
 $a \geq c \ \& \ b \leq d \mid a \leq c \ \& \ b \geq d \implies (d :: \text{nat})$   $\langle \text{proof} \rangle$

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

**lemma** *minus-eq*:  $(m - k = m) = (k = 0 \mid m = (0 :: \text{nat}))$   $\langle \text{proof} \rangle$

**lemma** *pl-pl-mm*:  $(a :: \text{nat}) + b = c + d \implies a - c = d - b$   $\langle \text{proof} \rangle$

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

**lemma** *min-minus* [*simp*]:  $\min m (m - k) = (m - k :: \text{nat})$   $\langle \text{proof} \rangle$

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

**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*:

$0 < c \implies (a \geq b * c) = (a \text{ div } c \geq (b :: \text{nat}))$   
 ⟨proof⟩

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

**lemma** *div-mult-le*:  $(a :: \text{nat}) \text{ div } b * b \leq a$

⟨proof⟩

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

**lemma** *given-quot*:  $f > (0 :: \text{nat}) \implies (f * l + (f - 1)) \text{ div } f = l$

⟨proof⟩

**lemma** *given-quot-alt*:  $f > (0 :: \text{nat}) \implies (l * f + f - \text{Suc } 0) \text{ div } f = l$

⟨proof⟩

**lemma** *diff-mod-le*:  $(a :: \text{nat}) < d \implies b \text{ dvd } d \implies a - a \text{ mod } b \leq d - b$

⟨proof⟩

**lemma** *less-le-mult'*:

$w * c < b * c \implies 0 \leq c \implies (w + 1) * c \leq b * (c :: \text{int})$

⟨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*]

**lemma** *lrlem'*:

**assumes** *d*:  $(i :: \text{nat}) \leq j \vee m < j'$

**assumes** *R1*:  $i * k \leq j * k \implies R$

**assumes** *R2*:  $\text{Suc } m * k' \leq j' * k' \implies R$

**shows** *R* ⟨proof⟩

**lemma** *lrlem*:  $(0 :: \text{nat}) < sc \implies$

$(sc - n + (n + lb * n) <= m * n) = (sc + lb * n <= m * n)$

⟨proof⟩

**lemma** *gen-minus*:  $0 < n \implies f n = f (\text{Suc } (n - 1))$

⟨proof⟩

**lemma** *mpl-lem*:  $j \leq (i :: \text{nat}) \implies k < j \implies i - j + k < i$  ⟨proof⟩

**lemma** *nonneg-mod-div*:

$0 \leq a \implies 0 \leq b \implies 0 \leq (a \text{ mod } b :: \text{int}) \ \& \ 0 \leq a \text{ div } b$

⟨proof⟩

**end**

## 8 Bit: The Field of Integers mod 2

```
theory Bit
imports Main
begin
```

### 8.1 Bits as a datatype

```
typedef (open) bit = UNIV :: bool set <proof>
```

```
instantiation bit :: {zero, one}
begin
```

```
definition zero-bit-def:
  0 = Abs-bit False
```

```
definition one-bit-def:
  1 = Abs-bit True
```

```
instance <proof>
```

```
end
```

```
rep-datatype 0::bit 1::bit
<proof>
```

```
lemma bit-not-0-iff [iff]: (x::bit) ≠ 0 ⟷ x = 1
  <proof>
```

```
lemma bit-not-1-iff [iff]: (x::bit) ≠ 1 ⟷ x = 0
  <proof>
```

### 8.2 Type *bit* forms a field

```
instantiation bit :: field-inverse-zero
begin
```

```
definition plus-bit-def:
  x + y = bit-case y (bit-case 1 0 y) x
```

```
definition times-bit-def:
  x * y = bit-case 0 y x
```

```
definition uminus-bit-def [simp]:
  - x = (x :: bit)
```

```
definition minus-bit-def [simp]:
```

$$x - y = (x + y :: \text{bit})$$

**definition** *inverse-bit-def* [*simp*]:  
*inverse*  $x = (x :: \text{bit})$

**definition** *divide-bit-def* [*simp*]:  
 $x / y = (x * y :: \text{bit})$

**lemmas** *field-bit-defs* =  
*plus-bit-def times-bit-def minus-bit-def uminus-bit-def*  
*divide-bit-def inverse-bit-def*

**instance**  $\langle \text{proof} \rangle$

**end**

**lemma** *bit-add-self*:  $x + x = (0 :: \text{bit})$   
 $\langle \text{proof} \rangle$

**lemma** *bit-mult-eq-1-iff* [*simp*]:  $x * y = (1 :: \text{bit}) \longleftrightarrow x = 1 \wedge y = 1$   
 $\langle \text{proof} \rangle$

Not sure whether the next two should be simp rules.

**lemma** *bit-add-eq-0-iff*:  $x + y = (0 :: \text{bit}) \longleftrightarrow x = y$   
 $\langle \text{proof} \rangle$

**lemma** *bit-add-eq-1-iff*:  $x + y = (1 :: \text{bit}) \longleftrightarrow x \neq y$   
 $\langle \text{proof} \rangle$

### 8.3 Numerals at type *bit*

All numerals reduce to either 0 or 1.

**lemma** *bit-minus1* [*simp*]:  $-1 = (1 :: \text{bit})$   
 $\langle \text{proof} \rangle$

**lemma** *bit-neg-numeral* [*simp*]:  $(\text{neg-numeral } w :: \text{bit}) = \text{numeral } w$   
 $\langle \text{proof} \rangle$

**lemma** *bit-numeral-even* [*simp*]:  $\text{numeral } (\text{Num.Bit0 } w) = (0 :: \text{bit})$   
 $\langle \text{proof} \rangle$

**lemma** *bit-numeral-odd* [*simp*]:  $\text{numeral } (\text{Num.Bit1 } w) = (1 :: \text{bit})$   
 $\langle \text{proof} \rangle$

**end**

## 9 Bit-Representation: Basic Definitions for Binary Integers

**theory** *Bit-Representation*  
**imports** *Misc-Numeric*  $\sim\sim$  /src/HOL/Library/Bit  
**begin**

### 9.1 Further properties of numerals

**definition** *bitval* :: *bit*  $\Rightarrow$  '*a::zero-neq-one* **where**  
*bitval* = *bit-case* 0 1

**lemma** *bitval-simps* [*simp*]:  
*bitval* 0 = 0  
*bitval* 1 = 1  
 $\langle$ *proof* $\rangle$

**definition** *Bit* :: *int*  $\Rightarrow$  *bit*  $\Rightarrow$  *int* (**infixl** *BIT* 90) **where**  
*k BIT b* = *bitval* b + *k* + *k*

**definition** *bin-last* :: *int*  $\Rightarrow$  *bit* **where**  
*bin-last* w = (if w mod 2 = 0 then (0::bit) else (1::bit))

**definition** *bin-rest* :: *int*  $\Rightarrow$  *int* **where**  
*bin-rest* w = w div 2

**lemma** *bin-rl-simp* [*simp*]:  
*bin-rest* w *BIT* *bin-last* w = w  
 $\langle$ *proof* $\rangle$

**lemma** *bin-rest-BIT* [*simp*]: *bin-rest* (x *BIT* b) = x  
 $\langle$ *proof* $\rangle$

**lemma** *bin-last-BIT* [*simp*]: *bin-last* (x *BIT* b) = b  
 $\langle$ *proof* $\rangle$

**lemma** *BIT-eq-iff* [*iff*]: u *BIT* b = v *BIT* c  $\longleftrightarrow$  u = v  $\wedge$  b = c  
 $\langle$ *proof* $\rangle$

**lemma** *BIT-bin-simps* [*simp*]:  
*numeral* k *BIT* 0 = *numeral* (Num.Bit0 k)  
*numeral* k *BIT* 1 = *numeral* (Num.Bit1 k)  
*neg-numeral* k *BIT* 0 = *neg-numeral* (Num.Bit0 k)  
*neg-numeral* k *BIT* 1 = *neg-numeral* (Num.BitM k)  
 $\langle$ *proof* $\rangle$

**lemma** *BIT-special-simps* [*simp*]:  
**shows** 0 *BIT* 0 = 0 **and** 0 *BIT* 1 = 1 **and** 1 *BIT* 0 = 2 **and** 1 *BIT* 1 = 3  
 $\langle$ *proof* $\rangle$

**lemma** *BitM-inc*:  $\text{Num.BitM} (\text{Num.inc } w) = \text{Num.Bit1 } w$   
 ⟨proof⟩

**lemma** *expand-BIT*:

$\text{numeral} (\text{Num.Bit0 } w) = \text{numeral } w \text{ BIT } 0$

$\text{numeral} (\text{Num.Bit1 } w) = \text{numeral } w \text{ BIT } 1$

$\text{neg-numeral} (\text{Num.Bit0 } w) = \text{neg-numeral } w \text{ BIT } 0$

$\text{neg-numeral} (\text{Num.Bit1 } w) = \text{neg-numeral} (w + \text{Num.One}) \text{ BIT } 1$

⟨proof⟩

**lemma** *bin-last-numeral-simps* [simp]:

$\text{bin-last } 0 = 0$

$\text{bin-last } 1 = 1$

$\text{bin-last } -1 = 1$

$\text{bin-last } \text{Numeral1} = 1$

$\text{bin-last} (\text{numeral} (\text{Num.Bit0 } w)) = 0$

$\text{bin-last} (\text{numeral} (\text{Num.Bit1 } w)) = 1$

$\text{bin-last} (\text{neg-numeral} (\text{Num.Bit0 } w)) = 0$

$\text{bin-last} (\text{neg-numeral} (\text{Num.Bit1 } w)) = 1$

⟨proof⟩

**lemma** *bin-rest-numeral-simps* [simp]:

$\text{bin-rest } 0 = 0$

$\text{bin-rest } 1 = 0$

$\text{bin-rest } -1 = -1$

$\text{bin-rest } \text{Numeral1} = 0$

$\text{bin-rest} (\text{numeral} (\text{Num.Bit0 } w)) = \text{numeral } w$

$\text{bin-rest} (\text{numeral} (\text{Num.Bit1 } w)) = \text{numeral } w$

$\text{bin-rest} (\text{neg-numeral} (\text{Num.Bit0 } w)) = \text{neg-numeral } w$

$\text{bin-rest} (\text{neg-numeral} (\text{Num.Bit1 } w)) = \text{neg-numeral} (w + \text{Num.One})$

⟨proof⟩

**lemma** *less-Bits*:

$(v \text{ BIT } b < w \text{ BIT } c) = (v < w \mid v \leq w \ \& \ b = (0::\text{bit}) \ \& \ c = (1::\text{bit}))$

⟨proof⟩

**lemma** *le-Bits*:

$(v \text{ BIT } b \leq w \text{ BIT } c) = (v < w \mid v \leq w \ \& \ (b \sim = (1::\text{bit}) \mid c \sim = (0::\text{bit})))$

⟨proof⟩

**lemma** *Bit-B0*:

$k \text{ BIT } (0::\text{bit}) = k + k$

⟨proof⟩

**lemma** *Bit-B1*:

$k \text{ BIT } (1::\text{bit}) = k + k + 1$

⟨proof⟩

**lemma** *Bit-B0-2t*:  $k \text{ BIT } (0::\text{bit}) = 2 * k$   
 ⟨proof⟩

**lemma** *Bit-B1-2t*:  $k \text{ BIT } (1::\text{bit}) = 2 * k + 1$   
 ⟨proof⟩

**lemma** *B-mod-2'*:  
 $X = 2 \implies (w \text{ BIT } (1::\text{bit})) \text{ mod } X = 1 \ \& \ (w \text{ BIT } (0::\text{bit})) \text{ mod } X = 0$   
 ⟨proof⟩

**lemma** *neB1E* [*elim!*]:  
 assumes *ne*:  $y \neq (1::\text{bit})$   
 assumes *y*:  $y = (0::\text{bit}) \implies P$   
 shows *P*  
 ⟨proof⟩

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

**lemma** *bin-exhaust*:  
 assumes *Q*:  $\bigwedge x \ b. \ \text{bin} = x \ \text{BIT} \ b \implies Q$   
 shows *Q*  
 ⟨proof⟩

## 9.2 Destructors for binary integers

**primrec** *bin-nth* **where**  
 $Z$ :  $\text{bin-nth } w \ 0 = (\text{bin-last } w = (1::\text{bit}))$   
 $| \text{Suc}$ :  $\text{bin-nth } w \ (\text{Suc } n) = \text{bin-nth } (\text{bin-rest } w) \ n$

**lemma** *bin-abs-lem*:  
 $\text{bin} = (w \ \text{BIT} \ b) \implies \text{bin} \ \sim = \ -1 \ \dashrightarrow \ \text{bin} \ \sim = \ 0 \ \dashrightarrow$   
 $\text{nat } (\text{abs } w) < \text{nat } (\text{abs } \text{bin})$   
 ⟨proof⟩

**lemma** *bin-induct*:  
 assumes *PPls*:  $P \ 0$   
 and *PMin*:  $P \ -1$   
 and *PBit*:  $\forall \text{bin } \text{bit}. \ P \ \text{bin} \implies P \ (\text{bin} \ \text{BIT} \ \text{bit})$   
 shows  $P \ \text{bin}$   
 ⟨proof⟩

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

**lemma** *bin-nth-lem* [*rule-format*]:  
 $\text{ALL } y. \ \text{bin-nth } x = \text{bin-nth } y \ \dashrightarrow \ x = y$   
 ⟨proof⟩

**lemma** *bin-nth-eq-iff*:  $(\text{bin-nth } x = \text{bin-nth } y) = (x = y)$   
 ⟨proof⟩

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

**lemma** *bin-eq-iff*:  $x = y \longleftrightarrow (\forall n. \text{bin-nth } x \ n = \text{bin-nth } y \ n)$   
 ⟨proof⟩

**lemma** *bin-nth-zero* [simp]:  $\neg \text{bin-nth } 0 \ n$   
 ⟨proof⟩

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

**lemma** *bin-nth-minus1* [simp]:  $\text{bin-nth } -1 \ n$   
 ⟨proof⟩

**lemma** *bin-nth-0-BIT*:  $\text{bin-nth } (w \ \text{BIT } b) \ 0 = (b = (1::\text{bit}))$   
 ⟨proof⟩

**lemma** *bin-nth-Suc-BIT*:  $\text{bin-nth } (w \ \text{BIT } b) \ (\text{Suc } n) = \text{bin-nth } w \ n$   
 ⟨proof⟩

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

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

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

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

### 9.3 Truncating binary integers

**definition** *bin-sign* ::  $\text{int} \Rightarrow \text{int}$  **where**  
*bin-sign-def*:  $\text{bin-sign } k = (\text{if } k \geq 0 \text{ then } 0 \text{ else } -1)$

**lemma** *bin-sign-simps* [simp]:  
 $\text{bin-sign } 0 = 0$

$bin\text{-}sign\ 1 = 0$   
 $bin\text{-}sign\ (numeral\ k) = 0$   
 $bin\text{-}sign\ (neg\text{-}numeral\ k) = -1$   
 $bin\text{-}sign\ (w\ BIT\ b) = bin\text{-}sign\ w$   
 ⟨proof⟩

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

**primrec** *bintrunc* ::  $nat \Rightarrow int \Rightarrow int$  **where**  
 $Z : bintrunc\ 0\ bin = 0$   
 $| Suc : bintrunc\ (Suc\ n)\ bin = bintrunc\ n\ (bin\text{-}rest\ bin)\ BIT\ (bin\text{-}last\ bin)$

**primrec** *sbintrunc* ::  $nat \Rightarrow int \Rightarrow int$  **where**  
 $Z : sbintrunc\ 0\ bin = (case\ bin\text{-}last\ bin\ of\ (1::bit) \Rightarrow -1\ | (0::bit) \Rightarrow 0)$   
 $| Suc : sbintrunc\ (Suc\ n)\ bin = sbintrunc\ n\ (bin\text{-}rest\ bin)\ BIT\ (bin\text{-}last\ bin)$

**lemma** *sign-bintr*:  $bin\text{-}sign\ (bintrunc\ n\ w) = 0$   
 ⟨proof⟩

**lemma** *bintrunc-mod2p*:  $bintrunc\ n\ w = (w\ mod\ 2^{\wedge}\ n)$   
 ⟨proof⟩

**lemma** *sbintrunc-mod2p*:  $sbintrunc\ n\ w = (w + 2^{\wedge}\ n)\ mod\ 2^{\wedge}\ (Suc\ n) - 2^{\wedge}\ n$   
 ⟨proof⟩

## 9.4 Simplifications for (s)bintrunc

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

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

**lemma** *sbintrunc-n-minus1* [simp]:  $sbintrunc\ n\ -1 = -1$   
 ⟨proof⟩

**lemma** *bintrunc-Suc-numeral*:  
 $bintrunc\ (Suc\ n)\ 1 = 1$   
 $bintrunc\ (Suc\ n)\ -1 = bintrunc\ n\ -1\ BIT\ 1$   
 $bintrunc\ (Suc\ n)\ (numeral\ (Num.Bit0\ w)) = bintrunc\ n\ (numeral\ w)\ BIT\ 0$   
 $bintrunc\ (Suc\ n)\ (numeral\ (Num.Bit1\ w)) = bintrunc\ n\ (numeral\ w)\ BIT\ 1$   
 $bintrunc\ (Suc\ n)\ (neg\text{-}numeral\ (Num.Bit0\ w)) =$   
 $bintrunc\ n\ (neg\text{-}numeral\ w)\ BIT\ 0$   
 $bintrunc\ (Suc\ n)\ (neg\text{-}numeral\ (Num.Bit1\ w)) =$   
 $bintrunc\ n\ (neg\text{-}numeral\ (w + Num.One))\ BIT\ 1$   
 ⟨proof⟩

**lemma** *sbintrunc-0-numeral* [simp]:

$$\begin{aligned} & \text{sbintrunc } 0 \ 1 = -1 \\ & \text{sbintrunc } 0 \ (\text{numeral } (\text{Num.Bit0 } w)) = 0 \\ & \text{sbintrunc } 0 \ (\text{numeral } (\text{Num.Bit1 } w)) = -1 \\ & \text{sbintrunc } 0 \ (\text{neg-numeral } (\text{Num.Bit0 } w)) = 0 \\ & \text{sbintrunc } 0 \ (\text{neg-numeral } (\text{Num.Bit1 } w)) = -1 \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *sbintrunc-Suc-numeral*:

$$\begin{aligned} & \text{sbintrunc } (\text{Suc } n) \ 1 = 1 \\ & \text{sbintrunc } (\text{Suc } n) \ (\text{numeral } (\text{Num.Bit0 } w)) = \\ & \quad \text{sbintrunc } n \ (\text{numeral } w) \ \text{BIT } 0 \\ & \text{sbintrunc } (\text{Suc } n) \ (\text{numeral } (\text{Num.Bit1 } w)) = \\ & \quad \text{sbintrunc } n \ (\text{numeral } w) \ \text{BIT } 1 \\ & \text{sbintrunc } (\text{Suc } n) \ (\text{neg-numeral } (\text{Num.Bit0 } w)) = \\ & \quad \text{sbintrunc } n \ (\text{neg-numeral } w) \ \text{BIT } 0 \\ & \text{sbintrunc } (\text{Suc } n) \ (\text{neg-numeral } (\text{Num.Bit1 } w)) = \\ & \quad \text{sbintrunc } n \ (\text{neg-numeral } (w + \text{Num.One})) \ \text{BIT } 1 \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *bit-bool*:

$$(b = (b' = (1::\text{bit}))) = (b' = (\text{if } b \text{ then } (1::\text{bit}) \text{ else } (0::\text{bit})))$$

$\langle \text{proof} \rangle$

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

**lemma** *bin-sign-lem*:  $(\text{bin-sign } (\text{sbintrunc } n \ \text{bin}) = -1) = \text{bin-nth } \text{bin } n$   
 $\langle \text{proof} \rangle$

**lemma** *nth-bintr*:  $\text{bin-nth } (\text{bintrunc } m \ w) \ n = (n < m \ \& \ \text{bin-nth } w \ n)$   
 $\langle \text{proof} \rangle$

**lemma** *nth-sbintr*:

$$\begin{aligned} & \text{bin-nth } (\text{sbintrunc } m \ w) \ n = \\ & \quad (\text{if } n < m \ \text{then } \text{bin-nth } w \ n \ \text{else } \text{bin-nth } w \ m) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *bin-nth-Bit*:

$$\text{bin-nth } (w \ \text{BIT } b) \ n = (n = 0 \ \& \ b = (1::\text{bit}) \mid (\text{EX } m. \ n = \text{Suc } m \ \& \ \text{bin-nth } w \ m))$$

$\langle \text{proof} \rangle$

**lemma** *bin-nth-Bit0*:

$$\begin{aligned} & \text{bin-nth } (\text{numeral } (\text{Num.Bit0 } w)) \ n \longleftrightarrow \\ & \quad (\exists m. \ n = \text{Suc } m \ \wedge \ \text{bin-nth } (\text{numeral } w) \ m) \\ & \langle \text{proof} \rangle \end{aligned}$$

**lemma** *bin-nth-Bit1*:

$$\text{bin-nth } (\text{numeral } (\text{Num.Bit1 } w)) \ n \longleftrightarrow$$

$n = 0 \vee (\exists m. n = \text{Suc } m \wedge \text{bin-nth } (\text{numeral } w) m)$   
 ⟨proof⟩

**lemma** *bintrunc-bintrunc-l*:  
 $n \leq m \implies (\text{bintrunc } m (\text{bintrunc } n w) = \text{bintrunc } n w)$   
 ⟨proof⟩

**lemma** *sbintrunc-sbintrunc-l*:  
 $n \leq m \implies (\text{sbintrunc } m (\text{sbintrunc } n w) = \text{sbintrunc } n w)$   
 ⟨proof⟩

**lemma** *bintrunc-bintrunc-ge*:  
 $n \leq m \implies (\text{bintrunc } n (\text{bintrunc } m w) = \text{bintrunc } n w)$   
 ⟨proof⟩

**lemma** *bintrunc-bintrunc-min [simp]*:  
 $\text{bintrunc } m (\text{bintrunc } n w) = \text{bintrunc } (\text{min } m n) w$   
 ⟨proof⟩

**lemma** *sbintrunc-sbintrunc-min [simp]*:  
 $\text{sbintrunc } m (\text{sbintrunc } n w) = \text{sbintrunc } (\text{min } m n) w$   
 ⟨proof⟩

**lemmas** *bintrunc-Pls =*  
 $\text{bintrunc.Suc}$  [where  $\text{bin}=0$ , simplified *bin-last-numeral-simps bin-rest-numeral-simps*]

**lemmas** *bintrunc-Min [simp] =*  
 $\text{bintrunc.Suc}$  [where  $\text{bin}=-1$ , simplified *bin-last-numeral-simps bin-rest-numeral-simps*]

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

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

**lemmas** *sbintrunc-Suc-Pls =*  
 $\text{sbintrunc.Suc}$  [where  $\text{bin}=0$ , simplified *bin-last-numeral-simps bin-rest-numeral-simps*]

**lemmas** *sbintrunc-Suc-Min =*  
 $\text{sbintrunc.Suc}$  [where  $\text{bin}=-1$ , simplified *bin-last-numeral-simps bin-rest-numeral-simps*]

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

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

**lemmas** *sbintrunc-Pls =*

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

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

**lemmas** *sbintrunc-0-BIT-B0* [simp] =  
*sbintrunc.Z* [where  $bin=w$  BIT (0::bit),  
simplified bin-last-numeral-simps bin-rest-numeral-simps bit.simps] **for**  
 $w$

**lemmas** *sbintrunc-0-BIT-B1* [simp] =  
*sbintrunc.Z* [where  $bin=w$  BIT (1::bit),  
simplified bin-last-BIT bin-rest-numeral-simps bit.simps] **for**  $w$

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

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

**lemma** *bintrunc-minus*:  
 $0 < n \implies bintrunc (Suc (n - 1)) w = bintrunc n w$   
(proof)

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

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

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

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

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

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

**lemmas** *bmsts* = *bintrunc-minus-simps*(1-3) [THEN *thobini1* [THEN [2] *trans*]]

**lemmas** *bintrunc-Pls-minus-I* = *bmsts*(1)

**lemmas** *bintrunc-Min-minus-I* = *bmsts*(2)

**lemmas** *bintrunc-BIT-minus-I* = *bmsts*(3)

**lemma** *bintrunc-Suc-lem*:  
 $bintrunc (Suc n) x = y \implies m = Suc n \implies bintrunc m x = y$   
(proof)

**lemmas** *bintrunc-Suc-Ialts* =

*bintrunc-Min-I* [THEN *bintrunc-Suc-lem*]

*bintrunc-BIT-I* [THEN *bintrunc-Suc-lem*]

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

**lemmas** *sbintrunc-Suc-Is* =

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

**lemmas** *sbintrunc-Suc-minus-Is* =

*sbintrunc-minus-simps(1-3)* [THEN *thobini1* [THEN [2] *trans*]]

**lemma** *sbintrunc-Suc-lem*:

$\text{sbintrunc } (\text{Suc } n) \ x = y \implies m = \text{Suc } n \implies \text{sbintrunc } m \ x = y$

*<proof>*

**lemmas** *sbintrunc-Suc-Ialts* =

*sbintrunc-Suc-Is* [THEN *sbintrunc-Suc-lem*]

**lemma** *sbintrunc-bintrunc-lt*:

$m > n \implies \text{sbintrunc } n \ (\text{bintrunc } m \ w) = \text{sbintrunc } n \ w$

*<proof>*

**lemma** *bintrunc-sbintrunc-le*:

$m \leq \text{Suc } n \implies \text{bintrunc } m \ (\text{sbintrunc } n \ w) = \text{bintrunc } m \ w$

*<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]:

$0 < n \implies \text{bintrunc } n \ (\text{sbintrunc } (n - 1) \ w) = \text{bintrunc } n \ w$

*<proof>*

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

$0 < n \implies \text{sbintrunc } (n - 1) \ (\text{bintrunc } n \ w) = \text{sbintrunc } (n - 1) \ w$

*<proof>*

**lemma** *bin-sbin-eq-iff*:

$\text{bintrunc } (\text{Suc } n) \ x = \text{bintrunc } (\text{Suc } n) \ y \iff$

$\text{sbintrunc } n \ x = \text{sbintrunc } n \ y$

*<proof>*

**lemma** *bin-sbin-eq-iff'*:

$0 < n \implies \text{bintrunc } n \ x = \text{bintrunc } n \ y \iff$

$\text{sbintrunc } (n - 1) \ x = \text{sbintrunc } (n - 1) \ y$

*<proof>*

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

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

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

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

**lemmas** *nat-non0-gr* =

*trans* [*OF iszero-def* [THEN *Not-eq-iff* [THEN *iffD2*]] *refl*]

**lemma** *bintrunc-numeral*:

*bintrunc* (numeral *k*) *x* =  
*bintrunc* (pred-numeral *k*) (bin-rest *x*) BIT bin-last *x*  
 ⟨proof⟩

**lemma** *sbintrunc-numeral*:

*sbintrunc* (numeral *k*) *x* =  
*sbintrunc* (pred-numeral *k*) (bin-rest *x*) BIT bin-last *x*  
 ⟨proof⟩

**lemma** *bintrunc-numeral-simps* [simp]:

*bintrunc* (numeral *k*) (numeral (Num.Bit0 *w*)) =  
*bintrunc* (pred-numeral *k*) (numeral *w*) BIT 0  
*bintrunc* (numeral *k*) (numeral (Num.Bit1 *w*)) =  
*bintrunc* (pred-numeral *k*) (numeral *w*) BIT 1  
*bintrunc* (numeral *k*) (neg-numeral (Num.Bit0 *w*)) =  
*bintrunc* (pred-numeral *k*) (neg-numeral *w*) BIT 0  
*bintrunc* (numeral *k*) (neg-numeral (Num.Bit1 *w*)) =  
*bintrunc* (pred-numeral *k*) (neg-numeral (*w* + Num.One)) BIT 1  
*bintrunc* (numeral *k*) 1 = 1  
 ⟨proof⟩

**lemma** *sbintrunc-numeral-simps* [simp]:

*sbintrunc* (numeral *k*) (numeral (Num.Bit0 *w*)) =  
*sbintrunc* (pred-numeral *k*) (numeral *w*) BIT 0  
*sbintrunc* (numeral *k*) (numeral (Num.Bit1 *w*)) =  
*sbintrunc* (pred-numeral *k*) (numeral *w*) BIT 1  
*sbintrunc* (numeral *k*) (neg-numeral (Num.Bit0 *w*)) =  
*sbintrunc* (pred-numeral *k*) (neg-numeral *w*) BIT 0  
*sbintrunc* (numeral *k*) (neg-numeral (Num.Bit1 *w*)) =  
*sbintrunc* (pred-numeral *k*) (neg-numeral (*w* + Num.One)) BIT 1  
*sbintrunc* (numeral *k*) 1 = 1  
 ⟨proof⟩

**lemma** *no-bintr-alt1*: *bintrunc* *n* = (%*w*. *w* mod 2 ^ *n* :: int)

⟨proof⟩

**lemma** *range-bintrunc*:  $\text{range } (\text{bintrunc } n) = \{i. 0 \leq i \ \& \ i < 2^n\}$   
 ⟨proof⟩

**lemma** *no-sbintr-alt2*:  
 $\text{sbintrunc } n = (\%w. (w + 2^n) \text{ mod } 2^{\text{Suc } n} - 2^n :: \text{int})$   
 ⟨proof⟩

**lemma** *range-sbintrunc*:  
 $\text{range } (\text{sbintrunc } n) = \{i. -(2^n) \leq i \ \& \ i < 2^n\}$   
 ⟨proof⟩

**lemma** *sb-inc-lem*:  
 $(a::\text{int}) + 2^k < 0 \implies a + 2^k + 2^{\text{Suc } k} \leq (a + 2^k) \text{ mod } 2^{\text{Suc } k}$   
 ⟨proof⟩

**lemma** *sb-inc-lem'*:  
 $(a::\text{int}) < -(2^k) \implies a + 2^k + 2^{\text{Suc } k} \leq (a + 2^k) \text{ mod } 2^{\text{Suc } k}$   
 ⟨proof⟩

**lemma** *sbintrunc-inc*:  
 $x < -(2^n) \implies x + 2^{\text{Suc } n} \leq \text{sbintrunc } n \ x$   
 ⟨proof⟩

**lemma** *sb-dec-lem*:  
 $(0::\text{int}) \leq -(2^k) + a \implies (a + 2^k) \text{ mod } (2 * 2^k) \leq -(2^k) + a$   
 ⟨proof⟩

**lemma** *sb-dec-lem'*:  
 $(2::\text{int})^k \leq a \implies (a + 2^k) \text{ mod } (2 * 2^k) \leq -(2^k) + a$   
 ⟨proof⟩

**lemma** *sbintrunc-dec*:  
 $x \geq (2^n) \implies x - 2^{\text{Suc } n} \geq \text{sbintrunc } n \ x$   
 ⟨proof⟩

**lemmas** *zmod-uminus'* = *zminus-zmod* [where  $m=c$ ] for  $c$

**lemmas** *zpower-zmod'* = *power-mod* [where  $b=c$  and  $n=k$ ] for  $c \ k$

**lemmas** *brdmod1s'* [symmetric] =  
*mod-add-left-eq mod-add-right-eq*  
*mod-diff-left-eq mod-diff-right-eq*  
*mod-mult-left-eq mod-mult-right-eq*

**lemmas** *brdmods'* [symmetric] =  
*zpower-zmod'* [symmetric]  
*trans [OF mod-add-left-eq mod-add-right-eq]*  
*trans [OF mod-diff-left-eq mod-diff-right-eq]*  
*trans [OF mod-mult-right-eq mod-mult-left-eq]*  
*zmod-uminus'* [symmetric]

*mod-add-left-eq* [where  $b = 1::int$ ]  
*mod-diff-left-eq* [where  $b = 1::int$ ]

**lemmas** *bintr-arith1s* =  
*brdmod1s'* [where  $c=2^n::int$ , folded *bintrunc-mod2p*] **for**  $n$   
**lemmas** *bintr-ariths* =  
*brdmods'* [where  $c=2^n::int$ , folded *bintrunc-mod2p*] **for**  $n$

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

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

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

**lemma** *bintr-Min*:  $\text{bintrunc } n \ -1 = 2^n - 1$   
 ⟨*proof*⟩

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

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

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

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

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

**lemma** *bin-rest-power-trunc* [*rule-format*] :  
 $(\text{bin-rest } ^k) (\text{bintrunc } n \ \text{bin}) =$   
 $\text{bintrunc } (n - k) ((\text{bin-rest } ^k) \ \text{bin})$   
 ⟨*proof*⟩

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

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

**lemma** *bintrunc-rest* [*simp*]:

$$\text{bintrunc } n \ (\text{bin-rest } (\text{bintrunc } n \ \text{bin})) = \text{bin-rest } (\text{bintrunc } n \ \text{bin})$$

*<proof>*

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

$$\text{sbintrunc } n \ (\text{bin-rest } (\text{sbintrunc } n \ \text{bin})) = \text{bin-rest } (\text{sbintrunc } n \ \text{bin})$$

*<proof>*

**lemma** *bintrunc-rest'*:

$$\text{bintrunc } n \ o \ \text{bin-rest } o \ \text{bintrunc } n = \text{bin-rest } o \ \text{bintrunc } n$$

*<proof>*

**lemma** *sbintrunc-rest'*:

$$\text{sbintrunc } n \ o \ \text{bin-rest } o \ \text{sbintrunc } n = \text{bin-rest } o \ \text{sbintrunc } n$$

*<proof>*

**lemma** *rco-lem*:

$$f \ o \ g \ o \ f = g \ o \ f \implies f \ o \ (g \ o \ f) \ \wedge \wedge \ n = g \ \wedge \wedge \ n \ o \ f$$

*<proof>*

**lemma** *rco-alt*:  $(f \ o \ g) \ \wedge \wedge \ n \ o \ f = f \ o \ (g \ o \ f) \ \wedge \wedge \ n$

*<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*]

## 9.5 Splitting and concatenation

**primrec** *bin-split* ::  $\text{nat} \Rightarrow \text{int} \Rightarrow \text{int} \times \text{int}$  **where**

$$Z: \text{bin-split } 0 \ w = (w, 0)$$

$$| \text{Suc}: \text{bin-split } (\text{Suc } n) \ w = (\text{let } (w1, w2) = \text{bin-split } n \ (\text{bin-rest } w) \\ \text{in } (w1, w2 \ \text{BIT } \text{bin-last } w))$$

**lemma** [*code*]:

$$\text{bin-split } (\text{Suc } n) \ w = (\text{let } (w1, w2) = \text{bin-split } n \ (\text{bin-rest } w) \ \text{in } (w1, w2 \ \text{BIT } \text{bin-last } w))$$

$$\text{bin-split } 0 \ w = (w, 0)$$

*<proof>*

**primrec** *bin-cat* ::  $\text{int} \Rightarrow \text{nat} \Rightarrow \text{int} \Rightarrow \text{int}$  **where**

$$Z: \text{bin-cat } w \ 0 \ v = w$$

$$| \text{Suc}: \text{bin-cat } w \ (\text{Suc } n) \ v = \text{bin-cat } w \ n \ (\text{bin-rest } v) \ \text{BIT } \text{bin-last } v$$

## 9.6 Miscellaneous lemmas

**lemma** *funpow-minus-simp*:

$$0 < n \implies f \ \wedge \wedge \ n = f \ o \ f \ \wedge \wedge \ (n - 1)$$

*<proof>*

**lemma** *funpow-numeral* [*simp*]:  
 $f \text{ ^^ } \text{numeral } k = f \circ f \text{ ^^ } (\text{pred-numeral } k)$   
*<proof>*

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

**lemmas** *power-minus-simp* =  
 $\text{trans } [OF \ \text{gen-minus } [\mathbf{where} \ f = \text{power } f] \ \text{power-Suc}] \ \mathbf{for} \ f$

**lemma** *list-exhaust-size-gt0*:  
**assumes**  $y: \bigwedge a \ \text{list}. \ y = a \ \# \ \text{list} \implies P$   
**shows**  $0 < \text{length } y \implies P$   
*<proof>*

**lemma** *list-exhaust-size-eq0*:  
**assumes**  $y: y = [] \implies P$   
**shows**  $\text{length } y = 0 \implies P$   
*<proof>*

**lemma** *size-Cons-lem-eq*:  
 $y = xa \ \# \ \text{list} \implies \text{size } y = \text{Suc } k \implies \text{size } \text{list} = k$   
*<proof>*

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

**lemma** *not-B1-is-B0*:  $y \neq (1::\text{bit}) \implies y = (0::\text{bit})$   
*<proof>*

**lemma** *B1-ass-B0*:  
**assumes**  $y: y = (0::\text{bit}) \implies y = (1::\text{bit})$   
**shows**  $y = (1::\text{bit})$   
*<proof>*

**lemmas** *n2s-ths* [*THEN eq-reflection*] = *add-2-eq-Suc add-2-eq-Suc'*

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

**end**

## 10 Bit-Operations: Syntactic classes for bitwise operations

**theory** *Bit-Operations*  
**imports**  $\sim\sim / \text{src} / \text{HOL} / \text{Library} / \text{Bit}$

**begin**

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

We want the bitwise operations to bind slightly weaker than  $+$  and  $-$ , but  $\sim\sim$  to bind slightly stronger than  $*$ .

Testing and shifting operations.

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

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

## 10.1 Bitwise operations on *bit*

**instantiation** bit :: bit

**begin**

**primrec** bitNOT-bit **where**

```
NOT 0 = (1::bit)
| NOT 1 = (0::bit)
```

**primrec** bitAND-bit **where**

```
0 AND y = (0::bit)
| 1 AND y = (y::bit)
```

**primrec** bitOR-bit **where**

```
0 OR y = (y::bit)
| 1 OR y = (1::bit)
```

**primrec** bitXOR-bit **where**

```
0 XOR y = (y::bit)
| 1 XOR y = (NOT y :: bit)
```

**instance** ⟨proof⟩

**end**

**lemmas** bit-simps =

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

**lemma** *bit-extra-simps* [*simp*]:

*x AND 0 = (0::bit)*  
*x AND 1 = (x::bit)*  
*x OR 1 = (1::bit)*  
*x OR 0 = (x::bit)*  
*x XOR 1 = NOT (x::bit)*  
*x XOR 0 = (x::bit)*  
 ⟨*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 = 0*  
 ⟨*proof*⟩

**lemma** *bit-not-not* [*simp*]: *NOT (NOT (x::bit)) = x*

⟨*proof*⟩

**end**

## 11 Bit-Int: Bitwise Operations on Binary Integers

**theory** *Bit-Int*

**imports** *Bit-Representation Bit-Operations*

**begin**

### 11.1 Logical operations

bit-wise logical operations on the int type

**instantiation** *int :: bit*

**begin**

**definition** *int-not-def*:

*bitNOT = (λx::int. - x - 1)*

**function** *bitAND-int* **where**

*bitAND-int x y =*  
*(if x = 0 then 0 else if x = -1 then y else*  
*(bin-rest x AND bin-rest y) BIT (bin-last x AND bin-last y))*

$\langle proof \rangle$

**termination**

$\langle proof \rangle$

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

**definition** *int-or-def*:

$bitOR = (\lambda x y::int. NOT (NOT x AND NOT y))$

**definition** *int-xor-def*:

$bitXOR = (\lambda x y::int. (x AND NOT y) OR (NOT x AND y))$

**instance**  $\langle proof \rangle$

**end**

### 11.1.1 Basic simplification rules

**lemma** *int-not-BIT* [*simp*]:

$NOT (w BIT b) = (NOT w) BIT (NOT b)$

$\langle proof \rangle$

**lemma** *int-not-simps* [*simp*]:

$NOT (0::int) = -1$

$NOT (1::int) = -2$

$NOT (-1::int) = 0$

$NOT (numeral w::int) = neg-numeral (w + Num.One)$

$NOT (neg-numeral (Num.Bit0 w)::int) = numeral (Num.BitM w)$

$NOT (neg-numeral (Num.Bit1 w)::int) = numeral (Num.Bit0 w)$

$\langle proof \rangle$

**lemma** *int-not-not* [*simp*]:  $NOT (NOT (x::int)) = x$

$\langle proof \rangle$

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

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *Bit-eq-0-iff*:  $w BIT b = 0 \longleftrightarrow w = 0 \wedge b = 0$

$\langle proof \rangle$

**lemma** *Bit-eq-m1-iff*:  $w BIT b = -1 \longleftrightarrow w = -1 \wedge b = 1$

$\langle proof \rangle$

**lemma** *int-and-Bits* [*simp*]:

$(x BIT b) AND (y BIT c) = (x AND y) BIT (b AND c)$

*<proof>*

**lemma** *int-or-zero* [*simp*]:  $(0::int) \text{ OR } x = x$   
*<proof>*

**lemma** *int-or-minus1* [*simp*]:  $(-1::int) \text{ OR } x = -1$   
*<proof>*

**lemma** *bit-or-def*:  $(b::bit) \text{ OR } c = \text{NOT } (\text{NOT } b \text{ AND } \text{NOT } c)$   
*<proof>*

**lemma** *int-or-Bits* [*simp*]:  
 $(x \text{ BIT } b) \text{ OR } (y \text{ BIT } c) = (x \text{ OR } y) \text{ BIT } (b \text{ OR } c)$   
*<proof>*

**lemma** *int-xor-zero* [*simp*]:  $(0::int) \text{ XOR } x = x$   
*<proof>*

**lemma** *bit-xor-def*:  $(b::bit) \text{ XOR } c = (b \text{ AND } \text{NOT } c) \text{ OR } (\text{NOT } b \text{ AND } c)$   
*<proof>*

**lemma** *int-xor-Bits* [*simp*]:  
 $(x \text{ BIT } b) \text{ XOR } (y \text{ BIT } c) = (x \text{ XOR } y) \text{ BIT } (b \text{ XOR } c)$   
*<proof>*

### 11.1.2 Binary destructors

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

**lemma** *bin-last-NOT* [*simp*]:  $\text{bin-last } (\text{NOT } x) = \text{NOT } (\text{bin-last } x)$   
*<proof>*

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

**lemma** *bin-last-AND* [*simp*]:  $\text{bin-last } (x \text{ AND } y) = \text{bin-last } x \text{ AND } \text{bin-last } y$   
*<proof>*

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

**lemma** *bin-last-OR* [*simp*]:  $\text{bin-last } (x \text{ OR } y) = \text{bin-last } x \text{ OR } \text{bin-last } y$   
*<proof>*

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

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

$\langle proof \rangle$

**lemma** *bit-NOT-eq-1-iff* [simp]:  $NOT (b::bit) = 1 \longleftrightarrow b = 0$   
 $\langle proof \rangle$

**lemma** *bit-AND-eq-1-iff* [simp]:  $(a::bit) AND b = 1 \longleftrightarrow a = 1 \wedge b = 1$   
 $\langle proof \rangle$

**lemma** *bin-nth-ops*:

$!!x y. bin\_nth (x AND y) n = (bin\_nth x n \& bin\_nth y n)$   
 $!!x y. bin\_nth (x OR y) n = (bin\_nth x n | bin\_nth y n)$   
 $!!x y. bin\_nth (x XOR y) n = (bin\_nth x n \sim bin\_nth y n)$   
 $!!x. bin\_nth (NOT x) n = (\sim bin\_nth x n)$   
 $\langle proof \rangle$

### 11.1.3 Derived properties

**lemma** *int-xor-minus1* [simp]:  $(-1::int) XOR x = NOT x$   
 $\langle proof \rangle$

**lemma** *int-xor-extra-simps* [simp]:  
 $w XOR (0::int) = w$   
 $w XOR (-1::int) = NOT w$   
 $\langle proof \rangle$

**lemma** *int-or-extra-simps* [simp]:  
 $w OR (0::int) = w$   
 $w OR (-1::int) = -1$   
 $\langle proof \rangle$

**lemma** *int-and-extra-simps* [simp]:  
 $w AND (0::int) = 0$   
 $w AND (-1::int) = w$   
 $\langle proof \rangle$

**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*:  $!!y::int. x XOR y = y XOR x$

$\langle proof \rangle$

**lemma** *bin-ops-same* [simp]:  
 $(x::int) AND x = x$   
 $(x::int) OR x = x$   
 $(x::int) XOR x = 0$   
 $\langle proof \rangle$

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

**lemma** *bbw-ao-absorb*:  
 $!!y::int. x \text{ AND } (y \text{ OR } x) = x \ \& \ x \text{ OR } (y \text{ AND } x) = x$   
 $\langle \text{proof} \rangle$

**lemma** *bbw-ao-absorbs-other*:  
 $x \text{ AND } (x \text{ OR } y) = x \ \wedge \ (y \text{ AND } x) \text{ OR } x = (x::int)$   
 $(y \text{ OR } x) \text{ AND } x = x \ \wedge \ x \text{ OR } (x \text{ AND } y) = (x::int)$   
 $(x \text{ OR } y) \text{ AND } x = x \ \wedge \ (x \text{ AND } y) \text{ OR } x = (x::int)$   
 $\langle \text{proof} \rangle$

**lemmas** *bbw-ao-absorbs [simp]* = *bbw-ao-absorb bbw-ao-absorbs-other*

**lemma** *int-xor-not*:  
 $!!y::int. (\text{NOT } x) \text{ XOR } y = \text{NOT } (x \text{ XOR } y) \ \&$   
 $x \text{ XOR } (\text{NOT } y) = \text{NOT } (x \text{ XOR } y)$   
 $\langle \text{proof} \rangle$

**lemma** *int-and-assoc*:  
 $(x \text{ AND } y) \text{ AND } (z::int) = x \text{ AND } (y \text{ AND } z)$   
 $\langle \text{proof} \rangle$

**lemma** *int-or-assoc*:  
 $(x \text{ OR } y) \text{ OR } (z::int) = x \text{ OR } (y \text{ OR } z)$   
 $\langle \text{proof} \rangle$

**lemma** *int-xor-assoc*:  
 $(x \text{ XOR } y) \text{ XOR } (z::int) = x \text{ XOR } (y \text{ XOR } z)$   
 $\langle \text{proof} \rangle$

**lemmas** *bbw-assocs* = *int-and-assoc int-or-assoc int-xor-assoc*

**lemma** *bbw-lcs [simp]*:  
 $(y::int) \text{ AND } (x \text{ AND } z) = x \text{ AND } (y \text{ AND } z)$   
 $(y::int) \text{ OR } (x \text{ OR } z) = x \text{ OR } (y \text{ OR } z)$   
 $(y::int) \text{ XOR } (x \text{ XOR } z) = x \text{ XOR } (y \text{ XOR } z)$   
 $\langle \text{proof} \rangle$

**lemma** *bbw-not-dist*:  
 $!!y::int. \text{NOT } (x \text{ OR } y) = (\text{NOT } x) \text{ AND } (\text{NOT } y)$   
 $!!y::int. \text{NOT } (x \text{ AND } y) = (\text{NOT } x) \text{ OR } (\text{NOT } y)$   
 $\langle \text{proof} \rangle$

**lemma** *bbw-oa-dist*:

$$\begin{aligned} &!!y z::int. (x \text{ AND } y) \text{ OR } z = \\ &\quad (x \text{ OR } z) \text{ AND } (y \text{ OR } z) \\ &\langle \text{proof} \rangle \end{aligned}$$

**lemma** *bbw-ao-dist*:

$$\begin{aligned} &!!y z::int. (x \text{ OR } y) \text{ AND } z = \\ &\quad (x \text{ AND } z) \text{ OR } (y \text{ AND } z) \\ &\langle \text{proof} \rangle \end{aligned}$$

#### 11.1.4 Simplification with numerals

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

**lemma** *bin-rl-eqI*:  $\llbracket \text{bin-rest } x = \text{bin-rest } y; \text{bin-last } x = \text{bin-last } y \rrbracket \implies x = y$   
 $\langle \text{proof} \rangle$

**lemma** *bin-rest-neg-numeral-BitM* [simp]:

$$\begin{aligned} &\text{bin-rest } (\text{neg-numeral } (\text{Num.BitM } w)) = \text{neg-numeral } w \\ &\langle \text{proof} \rangle \end{aligned}$$

**lemma** *bin-last-neg-numeral-BitM* [simp]:

$$\begin{aligned} &\text{bin-last } (\text{neg-numeral } (\text{Num.BitM } w)) = 1 \\ &\langle \text{proof} \rangle \end{aligned}$$

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

**lemma** *int-and-numerals* [simp]:

$$\begin{aligned} &\text{numeral } (\text{Num.Bit0 } x) \text{ AND numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ AND numeral } y) \text{ BIT } 0 \\ &\text{numeral } (\text{Num.Bit0 } x) \text{ AND numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ AND numeral } y) \text{ BIT } 0 \\ &\text{numeral } (\text{Num.Bit1 } x) \text{ AND numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ AND numeral } y) \text{ BIT } 0 \\ &\text{numeral } (\text{Num.Bit1 } x) \text{ AND numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ AND numeral } y) \text{ BIT } 1 \\ &\text{numeral } (\text{Num.Bit0 } x) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ AND neg-numeral } y) \text{ BIT } 0 \\ &\text{numeral } (\text{Num.Bit0 } x) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ AND neg-numeral } (y + \text{Num.One})) \text{ BIT } 0 \\ &\text{numeral } (\text{Num.Bit1 } x) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ AND neg-numeral } y) \text{ BIT } 0 \\ &\text{numeral } (\text{Num.Bit1 } x) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ AND neg-numeral } (y + \text{Num.One})) \text{ BIT } 1 \\ &\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } x \text{ AND numeral } y) \text{ BIT } 0 \\ &\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } x \text{ AND numeral } y) \text{ BIT } 0 \\ &\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ AND numeral } y) \text{ BIT } 0 \end{aligned}$$

$\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ AND numeral } y) \text{ BIT 1}$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } x \text{ AND neg-numeral } y) \text{ BIT 0}$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } x \text{ AND neg-numeral } (y + \text{Num.One})) \text{ BIT 0}$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ AND neg-numeral } y) \text{ BIT 0}$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ AND neg-numeral } (y + \text{Num.One})) \text{ BIT 1}$   
 $(1::\text{int}) \text{ AND numeral } (\text{Num.Bit0 } y) = 0$   
 $(1::\text{int}) \text{ AND numeral } (\text{Num.Bit1 } y) = 1$   
 $(1::\text{int}) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = 0$   
 $(1::\text{int}) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = 1$   
 $\text{numeral } (\text{Num.Bit0 } x) \text{ AND } (1::\text{int}) = 0$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ AND } (1::\text{int}) = 1$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND } (1::\text{int}) = 0$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND } (1::\text{int}) = 1$   
 {proof}

**lemma** *int-or-numerals* [simp]:

$\text{numeral } (\text{Num.Bit0 } x) \text{ OR numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ OR numeral } y) \text{ BIT 0}$   
 $\text{numeral } (\text{Num.Bit0 } x) \text{ OR numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ OR numeral } y) \text{ BIT 1}$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ OR numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ OR numeral } y) \text{ BIT 1}$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ OR numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ OR numeral } y) \text{ BIT 1}$   
 $\text{numeral } (\text{Num.Bit0 } x) \text{ OR neg-numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ OR neg-numeral } y) \text{ BIT 0}$   
 $\text{numeral } (\text{Num.Bit0 } x) \text{ OR neg-numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ OR neg-numeral } (y + \text{Num.One})) \text{ BIT 1}$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ OR neg-numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ OR neg-numeral } y) \text{ BIT 1}$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ OR neg-numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ OR neg-numeral } (y + \text{Num.One})) \text{ BIT 1}$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ OR numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } x \text{ OR numeral } y) \text{ BIT 0}$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ OR numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } x \text{ OR numeral } y) \text{ BIT 1}$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ OR numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ OR numeral } y) \text{ BIT 1}$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ OR numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ OR numeral } y) \text{ BIT 1}$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ OR neg-numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } x \text{ OR neg-numeral } y) \text{ BIT 0}$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ OR neg-numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } x \text{ OR neg-numeral } (y + \text{Num.One})) \text{ BIT 1}$

$\text{neg-numeral } (\text{Num.Bit1 } x) \text{ OR neg-numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ OR neg-numeral } y) \text{ BIT } 1$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ OR neg-numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ OR neg-numeral } (y + \text{Num.One})) \text{ BIT } 1$   
 $(1::\text{int}) \text{ OR numeral } (\text{Num.Bit0 } y) = \text{numeral } (\text{Num.Bit1 } y)$   
 $(1::\text{int}) \text{ OR numeral } (\text{Num.Bit1 } y) = \text{numeral } (\text{Num.Bit1 } y)$   
 $(1::\text{int}) \text{ OR neg-numeral } (\text{Num.Bit0 } y) = \text{neg-numeral } (\text{Num.BitM } y)$   
 $(1::\text{int}) \text{ OR neg-numeral } (\text{Num.Bit1 } y) = \text{neg-numeral } (\text{Num.Bit1 } y)$   
 $\text{numeral } (\text{Num.Bit0 } x) \text{ OR } (1::\text{int}) = \text{numeral } (\text{Num.Bit1 } x)$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ OR } (1::\text{int}) = \text{numeral } (\text{Num.Bit1 } x)$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ OR } (1::\text{int}) = \text{neg-numeral } (\text{Num.BitM } x)$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ OR } (1::\text{int}) = \text{neg-numeral } (\text{Num.Bit1 } x)$   
 {proof}

**lemma** *int-xor-numerals [simp]*:

$\text{numeral } (\text{Num.Bit0 } x) \text{ XOR numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ XOR numeral } y) \text{ BIT } 0$   
 $\text{numeral } (\text{Num.Bit0 } x) \text{ XOR numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ XOR numeral } y) \text{ BIT } 1$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ XOR numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ XOR numeral } y) \text{ BIT } 1$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ XOR numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ XOR numeral } y) \text{ BIT } 0$   
 $\text{numeral } (\text{Num.Bit0 } x) \text{ XOR neg-numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ XOR neg-numeral } y) \text{ BIT } 0$   
 $\text{numeral } (\text{Num.Bit0 } x) \text{ XOR neg-numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ XOR neg-numeral } (y + \text{Num.One})) \text{ BIT } 1$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ XOR neg-numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ XOR neg-numeral } y) \text{ BIT } 1$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ XOR neg-numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ XOR neg-numeral } (y + \text{Num.One})) \text{ BIT } 0$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ XOR numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } x \text{ XOR numeral } y) \text{ BIT } 0$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ XOR numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } x \text{ XOR numeral } y) \text{ BIT } 1$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ XOR numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ XOR numeral } y) \text{ BIT } 1$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ XOR numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ XOR numeral } y) \text{ BIT } 0$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ XOR neg-numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } x \text{ XOR neg-numeral } y) \text{ BIT } 0$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ XOR neg-numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } x \text{ XOR neg-numeral } (y + \text{Num.One})) \text{ BIT } 1$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ XOR neg-numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ XOR neg-numeral } y) \text{ BIT } 1$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ XOR neg-numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ XOR neg-numeral } (y + \text{Num.One})) \text{ BIT } 0$   
 $(1::\text{int}) \text{ XOR numeral } (\text{Num.Bit0 } y) = \text{numeral } (\text{Num.Bit1 } y)$   
 $(1::\text{int}) \text{ XOR numeral } (\text{Num.Bit1 } y) = \text{numeral } (\text{Num.Bit0 } y)$

$(1::int) \text{ XOR } \text{neg-numeral } (\text{Num.Bit0 } y) = \text{neg-numeral } (\text{Num.BitM } y)$   
 $(1::int) \text{ XOR } \text{neg-numeral } (\text{Num.Bit1 } y) = \text{neg-numeral } (\text{Num.Bit0 } (y + \text{Num.One}))$   
 $\text{numeral } (\text{Num.Bit0 } x) \text{ XOR } (1::int) = \text{numeral } (\text{Num.Bit1 } x)$   
 $\text{numeral } (\text{Num.Bit1 } x) \text{ XOR } (1::int) = \text{numeral } (\text{Num.Bit0 } x)$   
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ XOR } (1::int) = \text{neg-numeral } (\text{Num.BitM } x)$   
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ XOR } (1::int) = \text{neg-numeral } (\text{Num.Bit0 } (x + \text{Num.One}))$   
 ⟨proof⟩

### 11.1.5 Interactions with arithmetic

**lemma** *plus-and-or* [rule-format]:

$ALL y::int. (x \text{ AND } y) + (x \text{ OR } y) = x + y$   
 ⟨proof⟩

**lemma** *le-int-or*:

$\text{bin-sign } (y::int) = 0 ==> x \leq x \text{ OR } y$   
 ⟨proof⟩

**lemmas** *int-and-le =*

$\text{itr3 } [\text{OF } \text{bbw-ao-absorbs } (2) [\text{THEN } \text{conjunct2, symmetric}] \text{ le-int-or}]$

**lemma** *add-BIT-simps* [simp]:

$x \text{ BIT } 0 + y \text{ BIT } 0 = (x + y) \text{ BIT } 0$   
 $x \text{ BIT } 0 + y \text{ BIT } 1 = (x + y) \text{ BIT } 1$   
 $x \text{ BIT } 1 + y \text{ BIT } 0 = (x + y) \text{ BIT } 1$   
 $x \text{ BIT } 1 + y \text{ BIT } 1 = (x + y + 1) \text{ BIT } 0$   
 ⟨proof⟩

**lemma** *bin-add-not*:  $x + \text{NOT } x = (-1::int)$

⟨proof⟩

### 11.1.6 Truncating results of bit-wise operations

**lemma** *bin-trunc-ao*:

$!!x y. (\text{bintrunc } n x) \text{ AND } (\text{bintrunc } n y) = \text{bintrunc } n (x \text{ AND } y)$   
 $!!x y. (\text{bintrunc } n x) \text{ OR } (\text{bintrunc } n y) = \text{bintrunc } n (x \text{ OR } y)$   
 ⟨proof⟩

**lemma** *bin-trunc-xor*:

$!!x y. \text{bintrunc } n (\text{bintrunc } n x \text{ XOR } \text{bintrunc } n y) =$   
 $\text{bintrunc } n (x \text{ XOR } y)$   
 ⟨proof⟩

**lemma** *bin-trunc-not*:

$!!x. \text{bintrunc } n (\text{NOT } (\text{bintrunc } n x)) = \text{bintrunc } n (\text{NOT } x)$   
 ⟨proof⟩

**lemma** *bintr-bintr-i*:

$$x = \text{bintrunc } n \ y \implies \text{bintrunc } n \ x = \text{bintrunc } n \ y$$

*<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*]

## 11.2 Setting and clearing bits

**primrec**

$$\text{bin-sc} :: \text{nat} \Rightarrow \text{bit} \Rightarrow \text{int} \Rightarrow \text{int}$$

**where**

$$Z: \text{bin-sc } 0 \ b \ w = \text{bin-rest } w \ \text{BIT } b$$

$$| \text{Suc}: \text{bin-sc } (\text{Suc } n) \ b \ w = \text{bin-sc } n \ b \ (\text{bin-rest } w) \ \text{BIT } \text{bin-last } w$$

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

$$\text{bin-nth } (\text{bin-sc } n \ b \ w) \ n = (b = 1)$$

*<proof>*

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

$$\text{bin-sc } n \ c \ (\text{bin-sc } n \ b \ w) = \text{bin-sc } n \ c \ w$$

*<proof>*

**lemma** *bin-sc-sc-diff*:

$$m \sim = n \implies$$

$$\text{bin-sc } m \ c \ (\text{bin-sc } n \ b \ w) = \text{bin-sc } n \ b \ (\text{bin-sc } m \ c \ w)$$

*<proof>*

**lemma** *bin-nth-sc-gen*:

$$\text{bin-nth } (\text{bin-sc } n \ b \ w) \ m = (\text{if } m = n \ \text{then } b = 1 \ \text{else } \text{bin-nth } w \ m)$$

*<proof>*

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

$$(\text{bin-sc } n \ (\text{If } (\text{bin-nth } w \ n) \ 1 \ 0) \ w) = w$$

*<proof>*

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

$$\text{bin-sign } (\text{bin-sc } n \ b \ w) = \text{bin-sign } w$$

*<proof>*

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

$$\text{bintrunc } m \ (\text{bin-sc } n \ x \ (\text{bintrunc } m \ (w))) = \text{bintrunc } m \ (\text{bin-sc } n \ x \ w)$$

*<proof>*

**lemma** *bin-clr-le*:

$$\text{bin-sc } n \ 0 \ w \leq w$$

*<proof>*

**lemma** *bin-set-ge*:

*bin-sc n 1 w >= w*  
 ⟨proof⟩

**lemma** *bintr-bin-clr-le*:

*bintrunc n (bin-sc m 0 w) <= bintrunc n w*  
 ⟨proof⟩

**lemma** *bintr-bin-set-ge*:

*bintrunc n (bin-sc m 1 w) >= bintrunc n w*  
 ⟨proof⟩

**lemma** *bin-sc-FP [simp]*: *bin-sc n 0 0 = 0*

⟨proof⟩

**lemma** *bin-sc-TM [simp]*: *bin-sc n 1 -1 = -1*

⟨proof⟩

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

**lemma** *bin-sc-minus*:

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

**lemmas** *bin-sc-Suc-minus =*

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

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

*bin-sc (numeral k) b w =*  
*bin-sc (pred-numeral k) b (bin-rest w) BIT bin-last w*  
 ⟨proof⟩

### 11.3 Splitting and concatenation

**definition** *bin-rcat* :: *nat ⇒ int list ⇒ int* **where**

*bin-rcat n = foldl (λu v. bin-cat u n v) 0*

**fun** *bin-rsplit-aux* :: *nat ⇒ nat ⇒ int ⇒ int list ⇒ int list* **where**

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

**definition** *bin-rsplit* :: *nat ⇒ nat × int ⇒ int list* **where**

*bin-rsplit n w = bin-rsplit-aux n (fst w) (snd w) []*

**fun** *bin-rsplitl-aux* :: *nat ⇒ nat ⇒ int ⇒ int list ⇒ int list* **where**

*bin-rsplitl-aux n m c bs =*

(if  $m = 0 \mid n = 0$  then  $bs$  else  
 let  $(a, b) = \text{bin-split } (\text{min } m \ n) \ c$   
 in  $\text{bin-rsplittl-aux } n \ (m - n) \ a \ (b \ \# \ bs)$ )

**definition**  $\text{bin-rsplittl} :: \text{nat} \Rightarrow \text{nat} \times \text{int} \Rightarrow \text{int list}$  **where**  
 $\text{bin-rsplittl } n \ w = \text{bin-rsplittl-aux } n \ (\text{fst } w) \ (\text{snd } w) \ []$

**declare**  $\text{bin-rsplit-aux.simps}$  [simp del]  
**declare**  $\text{bin-rsplittl-aux.simps}$  [simp del]

**lemma**  $\text{bin-sign-cat}$ :  
 $\text{bin-sign } (\text{bin-cat } x \ n \ y) = \text{bin-sign } x$   
 ⟨proof⟩

**lemma**  $\text{bin-cat-Suc-Bit}$ :  
 $\text{bin-cat } w \ (\text{Suc } n) \ (v \ \text{BIT } b) = \text{bin-cat } w \ n \ v \ \text{BIT } b$   
 ⟨proof⟩

**lemma**  $\text{bin-nth-cat}$ :  
 $\text{bin-nth } (\text{bin-cat } x \ k \ y) \ n =$   
 (if  $n < k$  then  $\text{bin-nth } y \ n$  else  $\text{bin-nth } x \ (n - k)$ )  
 ⟨proof⟩

**lemma**  $\text{bin-nth-split}$ :  
 $\text{bin-split } n \ c = (a, b) \ ==>$   
 (ALL  $k$ .  $\text{bin-nth } a \ k = \text{bin-nth } c \ (n + k)$ ) &  
 (ALL  $k$ .  $\text{bin-nth } b \ k = (k < n \ \& \ \text{bin-nth } c \ k)$ )  
 ⟨proof⟩

**lemma**  $\text{bin-cat-assoc}$ :  
 $\text{bin-cat } (\text{bin-cat } x \ m \ y) \ n \ z = \text{bin-cat } x \ (m + n) \ (\text{bin-cat } y \ n \ z)$   
 ⟨proof⟩

**lemma**  $\text{bin-cat-assoc-sym}$ :  
 $\text{bin-cat } x \ m \ (\text{bin-cat } y \ n \ z) = \text{bin-cat } (\text{bin-cat } x \ (m - n) \ y) \ (\text{min } m \ n) \ z$   
 ⟨proof⟩

**lemma**  $\text{bin-cat-zero}$  [simp]:  $\text{bin-cat } 0 \ n \ w = \text{bintrunc } n \ w$   
 ⟨proof⟩

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

**lemma**  $\text{bintr-cat}$ :  $\text{bintrunc } m \ (\text{bin-cat } a \ n \ b) =$   
 $\text{bin-cat } (\text{bintrunc } (m - n) \ a) \ n \ (\text{bintrunc } (\text{min } m \ n) \ b)$   
 ⟨proof⟩

**lemma**  $\text{bintr-cat-same}$  [simp]:

$\text{bintrunc } n \ (\text{bin-cat } a \ n \ b) = \text{bintrunc } n \ b$   
 ⟨proof⟩

**lemma** *cat-bintr* [simp]:  
 $\text{bin-cat } a \ n \ (\text{bintrunc } n \ b) = \text{bin-cat } a \ n \ b$   
 ⟨proof⟩

**lemma** *split-bintrunc*:  
 $\text{bin-split } n \ c = (a, b) \implies b = \text{bintrunc } n \ c$   
 ⟨proof⟩

**lemma** *bin-cat-split*:  
 $\text{bin-split } n \ w = (u, v) \implies w = \text{bin-cat } u \ n \ v$   
 ⟨proof⟩

**lemma** *bin-split-cat*:  
 $\text{bin-split } n \ (\text{bin-cat } v \ n \ w) = (v, \text{bintrunc } n \ w)$   
 ⟨proof⟩

**lemma** *bin-split-zero* [simp]:  $\text{bin-split } n \ 0 = (0, 0)$   
 ⟨proof⟩

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

**lemma** *bin-split-trunc*:  
 $\text{bin-split } (\text{min } m \ n) \ c = (a, b) \implies$   
 $\text{bin-split } n \ (\text{bintrunc } m \ c) = (\text{bintrunc } (m - n) \ a, b)$   
 ⟨proof⟩

**lemma** *bin-split-trunc1*:  
 $\text{bin-split } n \ c = (a, b) \implies$   
 $\text{bin-split } n \ (\text{bintrunc } m \ c) = (\text{bintrunc } (m - n) \ a, \text{bintrunc } m \ b)$   
 ⟨proof⟩

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

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

## 11.4 Miscellaneous lemmas

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

**lemma** *ex-eq-or*:

$(EX\ m.\ n = Suc\ m \ \& \ (m = k \mid P\ m)) = (n = Suc\ k \mid (EX\ m.\ n = Suc\ m \ \& \ P\ m))$   
 ⟨proof⟩

**end**

## 12 Bool-List-Representation: Bool lists and integers

**theory** *Bool-List-Representation*

**imports** *Bit-Int*

**begin**

### 12.1 Operations on lists of booleans

**primrec** *bl-to-bin-aux* :: *bool list*  $\Rightarrow$  *int*  $\Rightarrow$  *int* **where**

*Nil*: *bl-to-bin-aux* [] *w* = *w*  
 | *Cons*: *bl-to-bin-aux* (*b* # *bs*) *w* =  
   *bl-to-bin-aux* *bs* (*w* BIT (if *b* then 1 else 0))

**definition** *bl-to-bin* :: *bool list*  $\Rightarrow$  *int* **where**

*bl-to-bin-def*: *bl-to-bin* *bs* = *bl-to-bin-aux* *bs* 0

**primrec** *bin-to-bl-aux* :: *nat*  $\Rightarrow$  *int*  $\Rightarrow$  *bool list*  $\Rightarrow$  *bool list* **where**

*Z*: *bin-to-bl-aux* 0 *w* *bl* = *bl*  
 | *Suc*: *bin-to-bl-aux* (*Suc* *n*) *w* *bl* =  
   *bin-to-bl-aux* *n* (*bin-rest* *w*) ((*bin-last* *w* = 1) # *bl*)

**definition** *bin-to-bl* :: *nat*  $\Rightarrow$  *int*  $\Rightarrow$  *bool list* **where**

*bin-to-bl-def* : *bin-to-bl* *n* *w* = *bin-to-bl-aux* *n* *w* []

**primrec** *bl-of-nth* :: *nat*  $\Rightarrow$  (*nat*  $\Rightarrow$  *bool*)  $\Rightarrow$  *bool list* **where**

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

**primrec** *takefill* :: '*a*  $\Rightarrow$  *nat*  $\Rightarrow$  '*a* *list*  $\Rightarrow$  '*a* *list* **where**

*Z*: *takefill* *fill* 0 *xs* = []  
 | *Suc*: *takefill* *fill* (*Suc* *n*) *xs* = (  
   case *xs* of [] => *fill* # *takefill* *fill* *n* *xs*  
   | *y* # *ys* => *y* # *takefill* *fill* *n* *ys*)

**definition** *map2* :: ('*a*  $\Rightarrow$  '*b*  $\Rightarrow$  '*c*)  $\Rightarrow$  '*a* *list*  $\Rightarrow$  '*b* *list*  $\Rightarrow$  '*c* *list* **where**

*map2* *f* *as* *bs* = *map* (*split* *f*) (*zip* *as* *bs*)

**lemma** *map2-Nil* [*simp*]:  $\text{map2 } f \ [] \ ys = []$   
 ⟨*proof*⟩

**lemma** *map2-Nil2* [*simp*]:  $\text{map2 } f \ xs \ [] = []$   
 ⟨*proof*⟩

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

## 12.2 Arithmetic in terms of bool lists

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

**primrec** *rbl-succ* :: *bool list* => *bool list* **where**  
*Nil*: *rbl-succ Nil* = *Nil*  
 | *Cons*: *rbl-succ* (*x* # *xs*) = (if *x* then *False* # *rbl-succ xs* else *True* # *xs*)

**primrec** *rbl-pred* :: *bool list* => *bool list* **where**  
*Nil*: *rbl-pred Nil* = *Nil*  
 | *Cons*: *rbl-pred* (*x* # *xs*) = (if *x* then *False* # *xs* else *True* # *rbl-pred xs*)

**primrec** *rbl-add* :: *bool list* => *bool list* => *bool list* **where**  
 — result is length of first arg, second arg may be longer  
*Nil*: *rbl-add Nil x* = *Nil*  
 | *Cons*: *rbl-add* (*y* # *ys*) *x* = (let *ws* = *rbl-add ys (tl x)* in  
 (*y* ~ = *hd x*) # (if *hd x* & *y* then *rbl-succ ws* else *ws*))

**primrec** *rbl-mult* :: *bool list* => *bool list* => *bool list* **where**  
 — result is length of first arg, second arg may be longer  
*Nil*: *rbl-mult Nil x* = *Nil*  
 | *Cons*: *rbl-mult* (*y* # *ys*) *x* = (let *ws* = *False* # *rbl-mult ys x* in  
 if *y* then *rbl-add ws x* else *ws*)

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

**lemma** *bin-to-bl-aux-zero-minus-simp* [*simp*]:  
 $0 < n \implies \text{bin-to-bl-aux } n \ 0 \ bl =$   
 $\text{bin-to-bl-aux } (n - 1) \ 0 \ (\text{False} \# \ bl)$   
 ⟨*proof*⟩

**lemma** *bin-to-bl-aux-minus1-minus-simp* [*simp*]:  
 $0 < n \implies \text{bin-to-bl-aux } n \ -1 \ bl =$   
 $\text{bin-to-bl-aux } (n - 1) \ -1 \ (\text{True} \# \ bl)$   
 ⟨*proof*⟩

**lemma** *bin-to-bl-aux-one-minus-simp* [simp]:

$$0 < n \implies \text{bin-to-bl-aux } n \ 1 \ \text{bl} = \\ \text{bin-to-bl-aux } (n - 1) \ 0 \ (\text{True} \ \# \ \text{bl}) \\ \langle \text{proof} \rangle$$

**lemma** *bin-to-bl-aux-Bit-minus-simp* [simp]:

$$0 < n \implies \text{bin-to-bl-aux } n \ (w \ \text{BIT} \ b) \ \text{bl} = \\ \text{bin-to-bl-aux } (n - 1) \ w \ ((b = 1) \ \# \ \text{bl}) \\ \langle \text{proof} \rangle$$

**lemma** *bin-to-bl-aux-Bit0-minus-simp* [simp]:

$$0 < n \implies \text{bin-to-bl-aux } n \ (\text{numeral } (\text{Num.Bit0 } w)) \ \text{bl} = \\ \text{bin-to-bl-aux } (n - 1) \ (\text{numeral } w) \ (\text{False} \ \# \ \text{bl}) \\ \langle \text{proof} \rangle$$

**lemma** *bin-to-bl-aux-Bit1-minus-simp* [simp]:

$$0 < n \implies \text{bin-to-bl-aux } n \ (\text{numeral } (\text{Num.Bit1 } w)) \ \text{bl} = \\ \text{bin-to-bl-aux } (n - 1) \ (\text{numeral } w) \ (\text{True} \ \# \ \text{bl}) \\ \langle \text{proof} \rangle$$

Link between bin and bool list.

**lemma** *bl-to-bin-aux-append*:

$$\text{bl-to-bin-aux } (bs \ @ \ cs) \ w = \text{bl-to-bin-aux } cs \ (\text{bl-to-bin-aux } bs \ w) \\ \langle \text{proof} \rangle$$

**lemma** *bin-to-bl-aux-append*:

$$\text{bin-to-bl-aux } n \ w \ bs \ @ \ cs = \text{bin-to-bl-aux } n \ w \ (bs \ @ \ cs) \\ \langle \text{proof} \rangle$$

**lemma** *bl-to-bin-append*:

$$\text{bl-to-bin } (bs \ @ \ cs) = \text{bl-to-bin-aux } cs \ (\text{bl-to-bin } bs) \\ \langle \text{proof} \rangle$$

**lemma** *bin-to-bl-aux-alt*:

$$\text{bin-to-bl-aux } n \ w \ bs = \text{bin-to-bl } n \ w \ @ \ bs \\ \langle \text{proof} \rangle$$

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

$\langle \text{proof} \rangle$

**lemma** *size-bin-to-bl-aux*:

$$\text{size } (\text{bin-to-bl-aux } n \ w \ bs) = n + \text{length } bs \\ \langle \text{proof} \rangle$$

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

$\langle \text{proof} \rangle$

**lemma** *bin-bl-bin'*:

$$\text{bl-to-bin } (\text{bin-to-bl-aux } n \ w \ bs) =$$

*bl-to-bin-aux* *bs* (*bintrunc* *n w*)  
 ⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

**lemma** *bin-to-bl-aux-bintr*:  
*bin-to-bl-aux* *n* (*bintrunc* *m bin*) *bl* =  
*replicate* (*n - m*) *False* @ *bin-to-bl-aux* (*min* *n m*) *bin bl*

*<proof>*

**lemma** *bin-to-bl-bintr*:

$bin-to-bl\ n\ (bintrunc\ m\ bin) =$   
 $replicate\ (n - m)\ False\ @\ bin-to-bl\ (min\ n\ m)\ bin$   
*<proof>*

**lemma** *bl-to-bin-rep-False*:  $bl-to-bin\ (replicate\ n\ False) = 0$

*<proof>*

**lemma** *len-bin-to-bl-aux*:

$length\ (bin-to-bl-aux\ n\ w\ bs) = n + length\ bs$   
*<proof>*

**lemma** *len-bin-to-bl [simp]*:  $length\ (bin-to-bl\ n\ w) = n$

*<proof>*

**lemma** *sign-bl-bin'*:

$bin-sign\ (bl-to-bin-aux\ bs\ w) = bin-sign\ w$   
*<proof>*

**lemma** *sign-bl-bin*:  $bin-sign\ (bl-to-bin\ bs) = 0$

*<proof>*

**lemma** *bl-sbin-sign-aux*:

$hd\ (bin-to-bl-aux\ (Suc\ n)\ w\ bs) =$   
 $(bin-sign\ (sbintrunc\ n\ w) = -1)$   
*<proof>*

**lemma** *bl-sbin-sign*:

$hd\ (bin-to-bl\ (Suc\ n)\ w) = (bin-sign\ (sbintrunc\ n\ w) = -1)$   
*<proof>*

**lemma** *bin-nth-of-bl-aux*:

$bin-nth\ (bl-to-bin-aux\ bl\ w)\ n =$   
 $(n < size\ bl\ \&\ rev\ bl\ !\ n\ | \ n \geq length\ bl\ \&\ bin-nth\ w\ (n - size\ bl))$   
*<proof>*

**lemma** *bin-nth-of-bl*:  $bin-nth\ (bl-to-bin\ bl)\ n = (n < length\ bl\ \&\ rev\ bl\ !\ n)$

*<proof>*

**lemma** *bin-nth-bl*:  $n < m \implies bin-nth\ w\ n = nth\ (rev\ (bin-to-bl\ m\ w))\ n$

*<proof>*

**lemma** *nth-rev*:

$n < length\ xs \implies rev\ xs\ !\ n = xs\ !\ (length\ xs - 1 - n)$   
*<proof>*

**lemma** *nth-rev-alt*:  $n < length\ ys \implies ys\ !\ n = rev\ ys\ !\ (length\ ys - Suc\ n)$

*<proof>*

**lemma** *nth-bin-to-bl-aux*:

$n < m + \text{length } bl \implies (\text{bin-to-bl-aux } m \ w \ bl) ! n =$   
*(if*  $n < m$  *then*  $\text{bin-nth } w \ (m - 1 - n)$  *else*  $bl ! (n - m)$ *)*  
*<proof>*

**lemma** *nth-bin-to-bl*:  $n < m \implies (\text{bin-to-bl } m \ w) ! n = \text{bin-nth } w \ (m - \text{Suc } n)$

*<proof>*

**lemma** *bl-to-bin-lt2p-aux*:

$\text{bl-to-bin-aux } bs \ w < (w + 1) * (2 \wedge \text{length } bs)$   
*<proof>*

**lemma** *bl-to-bin-lt2p*:  $\text{bl-to-bin } bs < (2 \wedge \text{length } bs)$

*<proof>*

**lemma** *bl-to-bin-ge2p-aux*:

$\text{bl-to-bin-aux } bs \ w \geq w * (2 \wedge \text{length } bs)$   
*<proof>*

**lemma** *bl-to-bin-ge0*:  $\text{bl-to-bin } bs \geq 0$

*<proof>*

**lemma** *butlast-rest-bin*:

$\text{butlast } (\text{bin-to-bl } n \ w) = \text{bin-to-bl } (n - 1) \ (\text{bin-rest } w)$   
*<proof>*

**lemma** *butlast-bin-rest*:

$\text{butlast } bl = \text{bin-to-bl } (\text{length } bl - \text{Suc } 0) \ (\text{bin-rest } (\text{bl-to-bin } bl))$   
*<proof>*

**lemma** *butlast-rest-bl2bin-aux*:

$bl \sim [] \implies$   
 $\text{bl-to-bin-aux } (\text{butlast } bl) \ w = \text{bin-rest } (\text{bl-to-bin-aux } bl \ w)$   
*<proof>*

**lemma** *butlast-rest-bl2bin*:

$\text{bl-to-bin } (\text{butlast } bl) = \text{bin-rest } (\text{bl-to-bin } bl)$   
*<proof>*

**lemma** *trunc-bl2bin-aux*:

$\text{bintrunc } m \ (\text{bl-to-bin-aux } bl \ w) =$   
 $\text{bl-to-bin-aux } (\text{drop } (\text{length } bl - m) \ bl) \ (\text{bintrunc } (m - \text{length } bl) \ w)$   
*<proof>*

**lemma** *trunc-bl2bin*:

$\text{bintrunc } m \ (\text{bl-to-bin } bl) = \text{bl-to-bin } (\text{drop } (\text{length } bl - m) \ bl)$   
*<proof>*

**lemma** *trunc-bl2bin-len* [simp]:

$$\text{bintrunc } (\text{length } \text{bl}) (\text{bl-to-bin } \text{bl}) = \text{bl-to-bin } \text{bl}$$

⟨proof⟩

**lemma** *bl2bin-drop*:

$$\text{bl-to-bin } (\text{drop } k \text{ bl}) = \text{bintrunc } (\text{length } \text{bl} - k) (\text{bl-to-bin } \text{bl})$$

⟨proof⟩

**lemma** *nth-rest-power-bin*:

$$\text{bin-nth } ((\text{bin-rest } \wedge^k) w) n = \text{bin-nth } w (n + k)$$

⟨proof⟩

**lemma** *take-rest-power-bin*:

$$m \leq n \implies \text{take } m (\text{bin-to-bl } n w) = \text{bin-to-bl } m ((\text{bin-rest } \wedge^{(n - m)}) w)$$

⟨proof⟩

**lemma** *hd-butlast*:  $\text{size } xs > 1 \implies \text{hd } (\text{butlast } xs) = \text{hd } xs$

⟨proof⟩

**lemma** *last-bin-last'*:

$$\text{size } xs > 0 \implies \text{last } xs = (\text{bin-last } (\text{bl-to-bin-aux } xs w) = 1)$$

⟨proof⟩

**lemma** *last-bin-last*:

$$\text{size } xs > 0 \implies \text{last } xs = (\text{bin-last } (\text{bl-to-bin } xs) = 1)$$

⟨proof⟩

**lemma** *bin-last-last*:

$$\text{bin-last } w = (\text{if last } (\text{bin-to-bl } (\text{Suc } n) w) \text{ then } 1 \text{ else } 0)$$

⟨proof⟩

**lemma** *bl-xor-aux-bin*:

$$\text{map2 } (\%x y. x \sim y) (\text{bin-to-bl-aux } n v bs) (\text{bin-to-bl-aux } n w cs) =$$

$$\text{bin-to-bl-aux } n (v \text{ XOR } w) (\text{map2 } (\%x y. x \sim y) bs cs)$$

⟨proof⟩

**lemma** *bl-or-aux-bin*:

$$\text{map2 } (\text{op } |) (\text{bin-to-bl-aux } n v bs) (\text{bin-to-bl-aux } n w cs) =$$

$$\text{bin-to-bl-aux } n (v \text{ OR } w) (\text{map2 } (\text{op } |) bs cs)$$

⟨proof⟩

**lemma** *bl-and-aux-bin*:

$$\text{map2 } (\text{op } \&) (\text{bin-to-bl-aux } n v bs) (\text{bin-to-bl-aux } n w cs) =$$

$$\text{bin-to-bl-aux } n (v \text{ AND } w) (\text{map2 } (\text{op } \&) bs cs)$$

⟨proof⟩

**lemma** *bl-not-aux-bin*:

$$\begin{aligned} \text{map } \text{Not } (\text{bin-to-bl-aux } n \ w \ cs) = \\ \text{bin-to-bl-aux } n \ (\text{NOT } w) \ (\text{map } \text{Not } cs) \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *bl-not-bin*:  $\text{map } \text{Not } (\text{bin-to-bl } n \ w) = \text{bin-to-bl } n \ (\text{NOT } w)$

$\langle \text{proof} \rangle$

**lemma** *bl-and-bin*:

$$\begin{aligned} \text{map2 } (\text{op } \wedge) (\text{bin-to-bl } n \ v) (\text{bin-to-bl } n \ w) = \text{bin-to-bl } n \ (v \ \text{AND } w) \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *bl-or-bin*:

$$\begin{aligned} \text{map2 } (\text{op } \vee) (\text{bin-to-bl } n \ v) (\text{bin-to-bl } n \ w) = \text{bin-to-bl } n \ (v \ \text{OR } w) \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *bl-xor-bin*:

$$\begin{aligned} \text{map2 } (\lambda x \ y. \ x \neq y) (\text{bin-to-bl } n \ v) (\text{bin-to-bl } n \ w) = \text{bin-to-bl } n \ (v \ \text{XOR } w) \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *drop-bin2bl-aux*:

$$\begin{aligned} \text{drop } m \ (\text{bin-to-bl-aux } n \ \text{bin } bs) = \\ \text{bin-to-bl-aux } (n - m) \ \text{bin } (\text{drop } (m - n) \ bs) \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *drop-bin2bl*:  $\text{drop } m \ (\text{bin-to-bl } n \ \text{bin}) = \text{bin-to-bl } (n - m) \ \text{bin}$

$\langle \text{proof} \rangle$

**lemma** *take-bin2bl-lem1*:

$$\begin{aligned} \text{take } m \ (\text{bin-to-bl-aux } m \ w \ bs) = \text{bin-to-bl } m \ w \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *take-bin2bl-lem*:

$$\begin{aligned} \text{take } m \ (\text{bin-to-bl-aux } (m + n) \ w \ bs) = \\ \text{take } m \ (\text{bin-to-bl } (m + n) \ w) \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *bin-split-take*:

$$\begin{aligned} \text{bin-split } n \ c = (a, b) \implies \\ \text{bin-to-bl } m \ a = \text{take } m \ (\text{bin-to-bl } (m + n) \ c) \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *bin-split-take1*:

$$\begin{aligned} k = m + n \implies \text{bin-split } n \ c = (a, b) \implies \\ \text{bin-to-bl } m \ a = \text{take } m \ (\text{bin-to-bl } k \ c) \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *nth-takefill*:  $m < n \implies$

$\text{takefill } \text{fill } n \ l \ ! \ m = (\text{if } m < \text{length } l \ \text{then } l \ ! \ m \ \text{else } \text{fill})$

*<proof>*

**lemma** *takefill-alt*:

$takefill\ fill\ n\ l = take\ n\ l\ @\ replicate\ (n - length\ l)\ fill$   
*<proof>*

**lemma** *takefill-replicate* [*simp*]:

$takefill\ fill\ n\ (replicate\ m\ fill) = replicate\ n\ fill$   
*<proof>*

**lemma** *takefill-le'*:

$n = m + k \implies takefill\ x\ m\ (takefill\ x\ n\ l) = takefill\ x\ m\ l$   
*<proof>*

**lemma** *length-takefill* [*simp*]:  $length\ (takefill\ fill\ n\ l) = n$

*<proof>*

**lemma** *take-takefill'*:

$!!w\ n.\ n = k + m \implies take\ k\ (takefill\ fill\ n\ w) = takefill\ fill\ k\ w$   
*<proof>*

**lemma** *drop-takefill*:

$!!w.\ drop\ k\ (takefill\ fill\ (m + k)\ w) = takefill\ fill\ m\ (drop\ k\ w)$   
*<proof>*

**lemma** *takefill-le* [*simp*]:

$m \leq n \implies takefill\ x\ m\ (takefill\ x\ n\ l) = takefill\ x\ m\ l$   
*<proof>*

**lemma** *take-takefill* [*simp*]:

$m \leq n \implies take\ m\ (takefill\ fill\ n\ w) = takefill\ fill\ m\ w$   
*<proof>*

**lemma** *takefill-append*:

$takefill\ fill\ (m + length\ xs)\ (xs\ @\ w) = xs\ @\ (takefill\ fill\ m\ w)$   
*<proof>*

**lemma** *takefill-same'*:

$l = length\ xs \implies takefill\ fill\ l\ xs = xs$   
*<proof>*

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

**lemma** *takefill-bintrunc*:

$takefill\ False\ n\ bl = rev\ (bin-to-bl\ n\ (bl-to-bin\ (rev\ bl)))$   
*<proof>*

**lemma** *bl-bin-bl-rtf*:

$bin-to-bl\ n\ (bl-to-bin\ bl) = rev\ (takefill\ False\ n\ (rev\ bl))$

*<proof>*

**lemma** *bl-bin-bl-rep-drop*:

*bin-to-bl n (bl-to-bin bl) =*  
*replicate (n - length bl) False @ drop (length bl - n) bl*  
*<proof>*

**lemma** *tf-rev*:

*n + k = m + length bl ==> takefill x m (rev (takefill y n bl)) =*  
*rev (takefill y m (rev (takefill x k (rev bl))))*  
*<proof>*

**lemma** *takefill-minus*:

*0 < n ==> takefill fill (Suc (n - 1)) w = takefill fill n w*  
*<proof>*

**lemmas** *takefill-Suc-cases =*

*list.cases [THEN takefill.Suc [THEN trans]]*

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

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

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

*takefill-minus [symmetric, THEN trans]]*

**lemma** *takefill-numeral-Nil [simp]*:

*takefill fill (numeral k) [] = fill # takefill fill (pred-numeral k) []*  
*<proof>*

**lemma** *takefill-numeral-Cons [simp]*:

*takefill fill (numeral k) (x # xs) = x # takefill fill (pred-numeral k) xs*  
*<proof>*

**lemma** *bl-to-bin-aux-cat*:

*!!nv v. bl-to-bin-aux bs (bin-cat w nv v) =*  
*bin-cat w (nv + length bs) (bl-to-bin-aux bs v)*  
*<proof>*

**lemma** *bin-to-bl-aux-cat*:

*!!w bs. bin-to-bl-aux (nv + nw) (bin-cat v nw w) bs =*  
*bin-to-bl-aux nv v (bin-to-bl-aux nw w bs)*  
*<proof>*

**lemma** *bl-to-bin-aux-alt*:

*bl-to-bin-aux bs w = bin-cat w (length bs) (bl-to-bin bs)*  
*<proof>*

**lemma** *bin-to-bl-cat*:

$$\begin{aligned} \text{bin-to-bl } (nv + nw) \text{ (bin-cat } v \text{ } nw \text{ } w) &= \\ \text{bin-to-bl-aux } nv \text{ } v \text{ (bin-to-bl } nw \text{ } w) & \\ \langle \text{proof} \rangle & \end{aligned}$$

**lemmas** *bl-to-bin-aux-app-cat* =

$$\text{trans [OF bl-to-bin-aux-append bl-to-bin-aux-alt]}$$

**lemmas** *bin-to-bl-aux-cat-app* =

$$\text{trans [OF bin-to-bl-aux-cat bin-to-bl-aux-alt]}$$

**lemma** *bl-to-bin-app-cat*:

$$\begin{aligned} \text{bl-to-bin } (bsa \text{ @ } bs) &= \text{bin-cat } (\text{bl-to-bin } bsa) \text{ (length } bs) \text{ (bl-to-bin } bs) \\ \langle \text{proof} \rangle & \end{aligned}$$

**lemma** *bin-to-bl-cat-app*:

$$\begin{aligned} \text{bin-to-bl } (n + nw) \text{ (bin-cat } w \text{ } nw \text{ } wa) &= \text{bin-to-bl } n \text{ } w \text{ @ } \text{bin-to-bl } nw \text{ } wa \\ \langle \text{proof} \rangle & \end{aligned}$$

**lemma** *bl-to-bin-app-cat-alt*:

$$\begin{aligned} \text{bin-cat } (\text{bl-to-bin } cs) \text{ } n \text{ } w &= \text{bl-to-bin } (cs \text{ @ } \text{bin-to-bl } n \text{ } w) \\ \langle \text{proof} \rangle & \end{aligned}$$

**lemma** *mask-lem*:  $(\text{bl-to-bin } (\text{True} \# \text{replicate } n \text{ False})) =$

$$\begin{aligned} (\text{bl-to-bin } (\text{replicate } n \text{ True})) + 1 & \\ \langle \text{proof} \rangle & \end{aligned}$$

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

$$\langle \text{proof} \rangle$$

**lemma** *nth-bl-of-nth [simp]*:

$$\begin{aligned} m < n \implies \text{rev } (\text{bl-of-nth } n \text{ } f) ! m &= f \text{ } m \\ \langle \text{proof} \rangle & \end{aligned}$$

**lemma** *bl-of-nth-inj*:

$$\begin{aligned} (!!k. k < n \implies f \text{ } k = g \text{ } k) \implies \text{bl-of-nth } n \text{ } f &= \text{bl-of-nth } n \text{ } g \\ \langle \text{proof} \rangle & \end{aligned}$$

**lemma** *bl-of-nth-nth-le*:

$$\begin{aligned} n \leq \text{length } xs \implies \text{bl-of-nth } n \text{ (nth } (\text{rev } xs)) &= \text{drop } (\text{length } xs - n) \text{ } xs \\ \langle \text{proof} \rangle & \end{aligned}$$

**lemma** *bl-of-nth-nth [simp]*:  $\text{bl-of-nth } (\text{length } xs) \text{ (op } ! \text{ (rev } xs)) = xs$

$$\langle \text{proof} \rangle$$

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

$$\langle \text{proof} \rangle$$

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

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

**lemma** *size-rbl-mult*:  
 !!cl.  $\text{length } (\text{rbl-mult } bl \ cl) = \text{length } bl$   
 ⟨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** *pred-BIT-simps* [simp]:  
 $x \text{ BIT } 0 - 1 = (x - 1) \text{ BIT } 1$   
 $x \text{ BIT } 1 - 1 = x \text{ BIT } 0$   
 ⟨proof⟩

**lemma** *rbl-pred*:  
 $\text{rbl-pred } (\text{rev } (\text{bin-to-bl } n \ bin)) = \text{rev } (\text{bin-to-bl } n \ (bin - 1))$   
 ⟨proof⟩

**lemma** *succ-BIT-simps* [simp]:  
 $x \text{ BIT } 0 + 1 = x \text{ BIT } 1$   
 $x \text{ BIT } 1 + 1 = (x + 1) \text{ BIT } 0$   
 ⟨proof⟩

**lemma** *rbl-succ*:  
 $\text{rbl-succ } (\text{rev } (\text{bin-to-bl } n \ bin)) = \text{rev } (\text{bin-to-bl } n \ (bin + 1))$   
 ⟨proof⟩

**lemma** *add-BIT-simps* [simp]:  
 $x \text{ BIT } 0 + y \text{ BIT } 0 = (x + y) \text{ BIT } 0$   
 $x \text{ BIT } 0 + y \text{ BIT } 1 = (x + y) \text{ BIT } 1$   
 $x \text{ BIT } 1 + y \text{ BIT } 0 = (x + y) \text{ BIT } 1$   
 $x \text{ BIT } 1 + y \text{ BIT } 1 = (x + y + 1) \text{ BIT } 0$   
 ⟨proof⟩

**lemma** *rbl-add*:  
 !!bina binb.  $\text{rbl-add } (\text{rev } (\text{bin-to-bl } n \ bina)) \ (\text{rev } (\text{bin-to-bl } n \ binb)) =$   
 $\text{rev } (\text{bin-to-bl } n \ (bina + binb))$   
 ⟨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** rdbl-Cons:

b # rev (bin-to-bl n x) = rev (bin-to-bl (Suc n) (x BIT If b 1 0))  
 <proof>

**lemma** mult-BIT-simps [simp]:

x BIT 0 \* y = (x \* y) BIT 0  
 x \* y BIT 0 = (x \* y) BIT 0  
 x BIT 1 \* y = (x \* y) BIT 0 + y  
 <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*:

$$P (\text{rbl-add } (y \# ys) (x \# xs)) =$$

$$(\text{ALL } ws. \text{length } ws = \text{length } ys \longrightarrow ws = \text{rbl-add } ys \ xs \longrightarrow$$

$$(y \longrightarrow ((x \longrightarrow P (\text{False} \# \text{rbl-succ } ws)) \ \& \ (\sim x \longrightarrow P (\text{True} \# ws)))) \ \&$$

$$(\sim y \longrightarrow P (x \# ws)))$$

*<proof>*

**lemma** *rbl-mult-split*:

$$P (\text{rbl-mult } (y \# ys) \ xs) =$$

$$(\text{ALL } ws. \text{length } ws = \text{Suc } (\text{length } ys) \longrightarrow ws = \text{False} \# \text{rbl-mult } ys \ xs \longrightarrow$$

$$(y \longrightarrow P (\text{rbl-add } ws \ xs)) \ \& \ (\sim y \longrightarrow P \ ws))$$

*<proof>*

**lemma** *and-len*:  $xs = ys \implies xs = ys \ \& \ \text{length } xs = \text{length } ys$

*<proof>*

**lemma** *size-if*:  $\text{size } (\text{if } p \ \text{then } xs \ \text{else } ys) = (\text{if } p \ \text{then } \text{size } xs \ \text{else } \text{size } ys)$

*<proof>*

**lemma** *tl-if*:  $\text{tl } (\text{if } p \ \text{then } xs \ \text{else } ys) = (\text{if } p \ \text{then } \text{tl } xs \ \text{else } \text{tl } ys)$

*<proof>*

**lemma** *hd-if*:  $\text{hd } (\text{if } p \ \text{then } xs \ \text{else } ys) = (\text{if } p \ \text{then } \text{hd } xs \ \text{else } \text{hd } ys)$

*<proof>*

**lemma** *if-Not-x*:  $(\text{if } p \ \text{then } \sim x \ \text{else } x) = (p = (\sim x))$

*<proof>*

**lemma** *if-x-Not*:  $(\text{if } p \ \text{then } x \ \text{else } \sim x) = (p = x)$

*<proof>*

**lemma** *if-same-and*:  $(\text{If } p \ x \ y \ \& \ \text{If } p \ u \ v) = (\text{if } p \ \text{then } x \ \& \ u \ \text{else } y \ \& \ v)$

*<proof>*

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

*<proof>*

**lemma** *if-same-eq-not*:

$$(\text{If } p \ x \ y = (\sim \text{If } p \ u \ v)) = (\text{if } p \ \text{then } x = (\sim u) \ \text{else } y = (\sim v))$$

*<proof>*

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

*<proof>*

**lemma** *if-single*:

(if  $xc$  then  $[xab]$  else  $[an]$ ) =  $[if\ xc\ then\ xab\ else\ an]$   
 ⟨proof⟩

**lemma** *if-bool-simps*:

If  $p$  True  $y = (p \mid y) \ \&$  If  $p$  False  $y = (\sim p \ \& \ y) \ \&$   
 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** *segr* = *eq-reflection* [where  $x = size\ w$ ] for  $w$

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

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

### 12.3 Repeated splitting or concatenation

**lemma** *sclem*:

$size\ (concat\ (map\ (bin-to-bl\ n)\ xs)) = length\ xs * n$   
 ⟨proof⟩

**lemma** *bin-cat-foldl-lem*:

$foldl\ (\%u.\ bin-cat\ u\ n)\ x\ xs =$   
 $bin-cat\ x\ (size\ xs * n)\ (foldl\ (\%u.\ bin-cat\ u\ n)\ y\ xs)$   
 ⟨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*] for  $l$

**lemmas** *th-if-simp2* = *split-if* [where  $P = op = l$ , THEN *iffD1*, THEN *conjunct2*, THEN *mp*] for  $l$

**lemmas** *rsplit-aux-simp1s* = *rsplit-aux-simps* [THEN *th-if-simp1*]

**lemmas** *rsplit-aux-simp2ls* = *rsplit-aux-simps* [THEN *th-if-simp2*]

**lemmas** *bin-rsplit-aux-simp2s* [*simp*] = *rsplit-aux-simp2ls* [*unfolded Let-def*]

**lemmas** *rbscl* = *bin-rsplit-aux-simp2s* (2)

**lemmas** *rsplit-aux-0-simps* [*simp*] =

*rsplit-aux-simp1s* [*OF disjI1*] *rsplit-aux-simp1s* [*OF disjI2*]

**lemma** *bin-rsplit-aux-append*:

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

**lemma** *bin-rsplittl-aux-append*:

$bin-rsplittl-aux\ n\ m\ c\ (bs\ @\ cs) = bin-rsplittl-aux\ n\ m\ c\ bs\ @\ cs$   
 ⟨proof⟩

**lemmas** *rsplit-aux-apps* [where  $bs = []$ ] =

*bin-rsplit-aux-append bin-rsplittl-aux-append*

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

**lemmas** *rsplit-aux-alt*s = *rsplit-aux-apps*

[*unfolded append-Nil rsplit-def-auxs [symmetric]*]

**lemma** *bin-split-minus*:  $0 < n \implies bin-split\ (Suc\ (n - 1))\ w = bin-split\ n\ w$

⟨proof⟩

**lemmas** *bin-split-minus-simp* =

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

**lemma** *bin-split-pred-simp* [*simp*]:

$(0::nat) < numeral\ bin \implies$   
 $bin-split\ (numeral\ bin)\ w =$   
 $(let\ (w1,\ w2) = bin-split\ (numeral\ bin - 1)\ (bin-rest\ w)$   
 $in\ (w1,\ w2\ BIT\ bin-last\ w))$   
 ⟨proof⟩

**lemma** *bin-rsplit-aux-simp-alt*:

$bin-rsplit-aux\ n\ m\ c\ bs =$   
 $(if\ m = 0 \vee n = 0$   
 $then\ bs$   
 $else\ let\ (a,\ b) = bin-split\ n\ c\ in\ bin-rsplit\ n\ (m - n,\ a)\ @\ b\ \# bs)$   
 ⟨proof⟩

**lemmas** *bin-rsplit-simp-alt* =

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

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

**lemma** *bin-rsplit-size-sign'* [*rule-format*] :

$\llbracket n > 0; rev\ sw = bin-rsplit\ n\ (nw,\ w) \rrbracket \implies$   
 $(ALL\ v: set\ sw.\ bintrunc\ n\ v = v)$   
 ⟨proof⟩

**lemmas** *bin-rsplit-size-sign* = *bin-rsplit-size-sign'* [*OF asm-rl*

*rev-rev-ident [THEN trans] set-rev [THEN equalityD2 [THEN subsetD]]]*

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

$n > 0 \implies m < n \implies (ALL\ w\ k\ nw.\ rev\ sw = bin-rsplit\ n\ (nw, w) \dashrightarrow$   
 $k < size\ sw \dashrightarrow bin-nth\ (sw\ !\ k)\ m = bin-nth\ w\ (k * n + m))$   
 ⟨proof⟩

**lemma** *bin-rsplit-all*:

$0 < nw \implies nw \leq n \implies bin-rsplit\ n\ (nw, w) = [bintrunc\ n\ w]$   
 ⟨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] :

$n > 0 \dashrightarrow bin-rsplit\ n\ (n * size\ ws, bin-rcat\ n\ ws) = map\ (bintrunc\ n)\ ws$   
 ⟨proof⟩

**lemma** *bin-rsplit-aux-len-le* [rule-format] :

$\forall\ ws\ m.\ n \neq 0 \rightarrow ws = bin-rsplit-aux\ n\ nw\ w\ bs \rightarrow$   
 $length\ ws \leq m \leftrightarrow nw + length\ bs * n \leq m * n$   
 ⟨proof⟩

**lemma** *bin-rsplit-len-le*:

$n \neq 0 \dashrightarrow ws = bin-rsplit\ n\ (nw, w) \dashrightarrow (length\ ws \leq m) = (nw \leq m * n)$   
 ⟨proof⟩

**lemma** *bin-rsplit-aux-len*:

$n \neq 0 \implies length\ (bin-rsplit-aux\ n\ nw\ w\ cs) =$   
 $(nw + n - 1) \div n + length\ cs$   
 ⟨proof⟩

**lemma** *bin-rsplit-len*:

$n \neq 0 \implies length\ (bin-rsplit\ n\ (nw, w)) = (nw + n - 1) \div n$   
 ⟨proof⟩

**lemma** *bin-rsplit-aux-len-indep*:

$n \neq 0 \implies length\ bs = length\ cs \implies$   
 $length\ (bin-rsplit-aux\ n\ nw\ v\ bs) =$   
 $length\ (bin-rsplit-aux\ n\ nw\ w\ cs)$   
 ⟨proof⟩

**lemma** *bin-rsplit-len-indep*:

$n \neq 0 \implies length\ (bin-rsplit\ n\ (nw, v)) = length\ (bin-rsplit\ n\ (nw, w))$   
 ⟨proof⟩

**end**

## 13 Word: A type of finite bit strings

```

theory Word
imports
  Type-Length
  Misc-Typedef
  ~~/src/HOL/Library/Boolean-Algebra
  Bool-List-Representation
uses
  (~/src/HOL/Word/Tools/smt-word.ML)
  (~/src/HOL/Word/Tools/word-lib.ML)
begin

see Examples/WordExamples.thy for examples

```

### 13.1 Type definition

```

typedef (open) 'a word = {(0::int) ..< 2^len-of TYPE('a::len0)}
morphisms uint Abs-word <proof>

```

**lemma** *uint-nonnegative*:

```

0 ≤ uint w
<proof>

```

**lemma** *uint-bounded*:

```

fixes w :: 'a::len0 word
shows uint w < 2 ^ len-of TYPE('a)
<proof>

```

**lemma** *uint-idem*:

```

fixes w :: 'a::len0 word
shows uint w mod 2 ^ len-of TYPE('a) = uint w
<proof>

```

**definition** *word-of-int* :: int ⇒ 'a::len0 word **where**

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

```

word-of-int k = Abs-word (k mod 2 ^ len-of TYPE('a))

```

**lemma** *uint-word-of-int*:

```

uint (word-of-int k :: 'a::len0 word) = k mod 2 ^ len-of TYPE('a)
<proof>

```

**lemma** *word-of-int-uint*:

```

word-of-int (uint w) = w
<proof>

```

**lemma** *word-uint-eq-iff*:

```

a = b ↔ uint a = uint b
<proof>

```

**lemma** *word-uint-eqI*:  
 $\text{uint } a = \text{uint } b \implies a = b$   
 ⟨*proof*⟩

### 13.2 Basic code generation setup

**definition** *Word* ::  $\text{int} \Rightarrow 'a::\text{len0 word}$   
**where**  
 [*code-post*]: *Word* = *word-of-int*

**lemma** [*code abstype*]:  
 $\text{Word } (\text{uint } w) = w$   
 ⟨*proof*⟩

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

**instantiation** *word* ::  $(\text{len0}) \text{ equal}$   
**begin**

**definition** *equal-word* ::  $'a \text{ word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool}$  **where**  
 $\text{equal-word } k \ l \longleftrightarrow \text{HOL.equal } (\text{uint } k) (\text{uint } l)$

**instance** ⟨*proof*⟩

**end**

**notation** *fcomp* (**infixl**  $\circ > 60$ )  
**notation** *scomp* (**infixl**  $\circ \rightarrow 60$ )

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

**definition**  
 $\text{random-word } i = \text{Random.range } i \circ \rightarrow (\lambda k. \text{Pair } ($   
 $\text{let } j = \text{word-of-int } (\text{Code-Numeral.int-of } k) :: 'a \text{ word}$   
 $\text{in } (j, \lambda :: \text{unit. Code-Evaluation.term-of } j)))$

**instance** ⟨*proof*⟩

**end**

**no-notation** *fcomp* (**infixl**  $\circ > 60$ )  
**no-notation** *scomp* (**infixl**  $\circ \rightarrow 60$ )

### 13.3 Type conversions and casting

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

**definition** *unat* :: 'a :: len0 word => nat **where**  
*unat* w = nat (uint w)

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

**definition** *sints* :: nat => int set **where**  
*sints* n = range (sbintrunc (n - 1))

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

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

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

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

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

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

**instance** <proof>

**end**

**definition** *source-size* :: ('a :: len0 word => 'b) => nat **where**  
 — whether a cast (or other) function is to a longer or shorter length  
*source-size* c = (let arb = undefined ; x = c arb in size arb)

**definition** *target-size* :: ('a => 'b :: len0 word) => nat **where**  
*target-size* c = size (c undefined)

**definition** *is-up* :: ('a :: len0 word => 'b :: len0 word) => bool **where**  
*is-up* c <math>\longleftrightarrow</math> *source-size* c <= *target-size* c

**definition** *is-down* :: ('a :: len0 word => 'b :: len0 word) => bool **where**  
*is-down* c <math>\longleftrightarrow</math> *target-size* c <= *source-size* c

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

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

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

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

**translations**

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

### 13.4 Type-definition locale instantiations

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

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

**lemmas** *lens-not-0* [iff] = *lens-gt-0* [THEN *gr-implies-not0*]

**lemma** *uints-num*: uints n = {i. 0 ≤ i ∧ i < 2 ^ n}  
 ⟨proof⟩

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

**lemma**

*uint-0:0* ≤ uint x **and**  
*uint-lt*: uint (x::'a::len0 word) < 2 ^ len-of TYPE('a)  
 ⟨proof⟩

**lemma** *uint-mod-same*:

uint x mod 2 ^ len-of TYPE('a) = uint (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 ^ len-of TYPE('a))  
 ⟨proof⟩

**interpretation** *word-uint*:

*td-ext* uint::'a::len0 word ⇒ int  
*word-of-int*  
 uints (len-of TYPE('a::len0))  
 λw. w mod 2 ^ len-of TYPE('a::len0)  
 ⟨proof⟩

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

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

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

**interpretation** *word-ubin*:  
*td-ext uint :: 'a::len0 word*  $\Rightarrow$  *int*  
*word-of-int*  
*uints* (*len-of TYPE('a::len0)*)  
*bintrunc* (*len-of TYPE('a::len0)*)  
*<proof>*

**lemma** *split-word-all*:  
 ( $\bigwedge x :: 'a::len0$  *word*. *PROP P x*)  $\equiv$  ( $\bigwedge x$ . *PROP P* (*word-of-int x*))  
*<proof>*

### 13.5 Correspondence relation for theorem transfer

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

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

**lemma** *reflp-word*:  
*reflp* ( $\lambda x y$ . *bintrunc* (*len-of TYPE('a::len0)*) *x = bintrunc* (*len-of TYPE('a)*)  
*y*)  
*<proof>*

**setup-lifting** *Quotient-word reflp-word*

TODO: The next lemma could be generated automatically.

**lemma** *uint-transfer* [*transfer-rule*]:  
 (*fun-rel cr-word op =*) (*bintrunc* (*len-of TYPE('a)*))  
 (*uint* :: *'a::len0 word*  $\Rightarrow$  *int*)  
*<proof>*

### 13.6 Arithmetic operations

**lift-definition** *word-succ* :: *'a::len0 word*  $\Rightarrow$  *'a word* **is**  $\lambda x$ . *x + 1*  
*<proof>*

**lift-definition** *word-pred* :: *'a::len0 word*  $\Rightarrow$  *'a word* **is**  $\lambda x$ . *x - 1*  
*<proof>*

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

**lift-definition** *zero-word* :: 'a word is 0 <proof>

**lift-definition** *one-word* :: 'a word is 1 <proof>

**lift-definition** *plus-word* :: 'a word  $\Rightarrow$  'a word  $\Rightarrow$  'a word is op +  
 <proof>

**lift-definition** *minus-word* :: 'a word  $\Rightarrow$  'a word  $\Rightarrow$  'a word is op -  
 <proof>

**lift-definition** *uminus-word* :: 'a word  $\Rightarrow$  'a word is *uminus*  
 <proof>

**lift-definition** *times-word* :: 'a word  $\Rightarrow$  'a word  $\Rightarrow$  'a word is op \*  
 <proof>

**definition**

*word-div-def*:  $a \text{ div } b = \text{word-of-int } (\text{uint } a \text{ div uint } b)$

**definition**

*word-mod-def*:  $a \text{ mod } b = \text{word-of-int } (\text{uint } a \text{ mod uint } b)$

**instance**

<proof>

**end**

Legacy theorems:

**lemma** *word-arith-wis* [*code*]: **shows**

*word-add-def*:  $a + b = \text{word-of-int } (\text{uint } a + \text{uint } b)$  **and**

*word-sub-wi*:  $a - b = \text{word-of-int } (\text{uint } a - \text{uint } b)$  **and**

*word-mult-def*:  $a * b = \text{word-of-int } (\text{uint } a * \text{uint } b)$  **and**

*word-minus-def*:  $- a = \text{word-of-int } (- \text{uint } a)$  **and**

*word-succ-alt*:  $\text{word-succ } a = \text{word-of-int } (\text{uint } a + 1)$  **and**

*word-pred-alt*:  $\text{word-pred } a = \text{word-of-int } (\text{uint } a - 1)$  **and**

*word-0-wi*:  $0 = \text{word-of-int } 0$  **and**

*word-1-wi*:  $1 = \text{word-of-int } 1$

<proof>

**lemmas** *ariths* =

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

**lemma** *wi-homs*:

**shows**

*wi-hom-add*:  $\text{word-of-int } a + \text{word-of-int } b = \text{word-of-int } (a + b)$  **and**

*wi-hom-sub*:  $\text{word-of-int } a - \text{word-of-int } b = \text{word-of-int } (a - b)$  **and**

*wi-hom-mult*:  $\text{word-of-int } a * \text{word-of-int } b = \text{word-of-int } (a * b)$  **and**  
*wi-hom-neg*:  $-\text{word-of-int } a = \text{word-of-int } (-a)$  **and**  
*wi-hom-succ*:  $\text{word-succ } (\text{word-of-int } a) = \text{word-of-int } (a + 1)$  **and**  
*wi-hom-pred*:  $\text{word-pred } (\text{word-of-int } a) = \text{word-of-int } (a - 1)$   
 ⟨*proof*⟩

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

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

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

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

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

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

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

### 13.7 Ordering

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

**definition**  
*word-le-def*:  $a \leq b \iff \text{uint } a \leq \text{uint } b$

**definition**  
*word-less-def*:  $a < b \iff \text{uint } a < \text{uint } b$

**instance**  
 ⟨*proof*⟩

**end**

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

**definition** *word-sless* ::  $'a :: \text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool}$  ((-/ <s -) [50, 51] 50)  
**where**  
 $(x <s y) = (x <=s y \ \& \ x \sim y)$

### 13.8 Bit-wise operations

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

**lift-definition** *bitNOT-word* :: 'a word  $\Rightarrow$  'a word **is** *bitNOT*  
 ⟨*proof*⟩

**lift-definition** *bitAND-word* :: 'a word  $\Rightarrow$  'a word  $\Rightarrow$  'a word **is** *bitAND*  
 ⟨*proof*⟩

**lift-definition** *bitOR-word* :: 'a word  $\Rightarrow$  'a word  $\Rightarrow$  'a word **is** *bitOR*  
 ⟨*proof*⟩

**lift-definition** *bitXOR-word* :: 'a word  $\Rightarrow$  'a word  $\Rightarrow$  'a word **is** *bitXOR*  
 ⟨*proof*⟩

**definition**

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

**definition**

*word-set-bit-def*: *set-bit a n x* =  
*word-of-int (bin-sc n (If x 1 0) (uint a))*

**definition**

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

**definition**

*word-lsb-def*: *lsb a*  $\longleftrightarrow$  *bin-last (uint a)* = 1

**definition** *shiftl1* :: 'a word  $\Rightarrow$  'a word **where**

*shiftl1 w* = *word-of-int (uint w BIT 0)*

**definition** *shiftr1* :: 'a word  $\Rightarrow$  'a word **where**

— shift right as unsigned or as signed, ie logical or arithmetic

*shiftr1 w* = *word-of-int (bin-rest (uint w))*

**definition**

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

**definition**

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

**instance** ⟨*proof*⟩

**end**

**lemma** [*code*]: **shows**

*word-not-def*: *NOT (a::'a::len0 word)* = *word-of-int (NOT (uint a))* **and**  
*word-and-def*: (*a::'a word*) *AND b* = *word-of-int (uint a AND uint b)* **and**

*word-or-def*: ( $a :: 'a \text{ word}$ )  $OR$   $b = \text{word-of-int } (\text{uint } a \text{ } OR \text{ } \text{uint } b)$  **and**  
*word-xor-def*: ( $a :: 'a \text{ word}$ )  $XOR$   $b = \text{word-of-int } (\text{uint } a \text{ } XOR \text{ } \text{uint } b)$   
 ⟨*proof*⟩

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

**definition**

*word-msb-def*:  
 $msb \ a \longleftrightarrow \text{bin-sign } (\text{sint } a) = -1$

**instance** ⟨*proof*⟩

**end**

**definition** *setBit* :: ' $a$  ::  $len0 \text{ word} \Rightarrow \text{nat} \Rightarrow 'a \text{ word}$  **where**  
 $setBit \ w \ n = \text{set-bit } w \ n \ \text{True}$

**definition** *clearBit* :: ' $a$  ::  $len0 \text{ word} \Rightarrow \text{nat} \Rightarrow 'a \text{ word}$  **where**  
 $clearBit \ w \ n = \text{set-bit } w \ n \ \text{False}$

### 13.9 Shift operations

**definition** *sshiftr1* :: ' $a$  ::  $len \text{ word} \Rightarrow 'a \text{ word}$  **where**  
 $sshiftr1 \ w = \text{word-of-int } (\text{bin-rest } (\text{sint } w))$

**definition** *bshiftr1* ::  $bool \Rightarrow 'a$  ::  $len \text{ word} \Rightarrow 'a \text{ word}$  **where**  
 $bshiftr1 \ b \ w = \text{of-bl } (b \ \# \ \text{butlast } (\text{to-bl } w))$

**definition** *sshiftr* :: ' $a$  ::  $len \text{ word} \Rightarrow \text{nat} \Rightarrow 'a \text{ word}$  (**infixl**  $>>>$  55) **where**  
 $w \ >>> \ n = (\text{sshiftr1 } \wedge \wedge \ n) \ w$

**definition** *mask* ::  $\text{nat} \Rightarrow 'a :: len \text{ word}$  **where**  
 $mask \ n = (1 \ \ll \ n) - 1$

**definition** *revcast* :: ' $a$  ::  $len0 \text{ word} \Rightarrow 'b$  ::  $len0 \text{ word}$  **where**  
 $revcast \ w = \text{of-bl } (\text{takefill } \text{False } (\text{len-of } \text{TYPE}('b)) (\text{to-bl } w))$

**definition** *slice1* ::  $\text{nat} \Rightarrow 'a$  ::  $len0 \text{ word} \Rightarrow 'b$  ::  $len0 \text{ word}$  **where**  
 $slice1 \ n \ w = \text{of-bl } (\text{takefill } \text{False } n (\text{to-bl } w))$

**definition** *slice* ::  $\text{nat} \Rightarrow 'a$  ::  $len0 \text{ word} \Rightarrow 'b$  ::  $len0 \text{ word}$  **where**  
 $slice \ n \ w = slice1 \ (\text{size } w - n) \ w$

### 13.10 Rotation

**definition** *rotater1* :: ' $a \text{ list} \Rightarrow 'a \text{ list}$  **where**

$rotater1 \ ys =$   
 (case of  $\square \Rightarrow \square \mid x \ \# \ xs \Rightarrow \text{last } ys \ \# \ \text{butlast } ys$ )

**definition** *rotater* :: *nat* => 'a list => 'a list **where**  
*rotater* *n* = *rotater1* ^^ *n*

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

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

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

### 13.11 Split and cat operations

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

**definition** *word-split* :: 'a :: len0 word => ('b :: len0 word) \* ('c :: len0 word)  
**where**

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

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

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

**definition** *max-word* :: 'a::len word — Largest representable machine integer.  
**where**

*max-word* = *word-of-int* ( $2^{\text{len-of TYPE}('a)} - 1$ )

**primrec** *of-bool* :: *bool* => 'a::len word **where**

*of-bool* *False* = 0  
 | *of-bool* *True* = 1

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

### 13.12 Theorems about typedefs

**lemma** *sint-sbintrunc'*:

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

**lemma** *uint-sint*:

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

**lemma** *bintr-uint*:

**fixes**  $w :: 'a::\text{len0 word}$   
**shows**  $\text{len-of } \text{TYPE}('a) \leq n \implies \text{bintrunc } n (\text{uint } w) = \text{uint } w$   
 ⟨proof⟩

**lemma** *wi-bintr*:

$\text{len-of } \text{TYPE}('a::\text{len0}) \leq n \implies$   
 $\text{word-of-int } (\text{bintrunc } n w) = (\text{word-of-int } w :: 'a \text{ word})$   
 ⟨proof⟩

**lemma** *td-ext-sbin*:

$\text{td-ext } (\text{sint} :: 'a \text{ word} \implies \text{int}) \text{ word-of-int } (\text{sints } (\text{len-of } \text{TYPE}('a::\text{len})))$   
 $(\text{sbintrunc } (\text{len-of } \text{TYPE}('a) - 1))$   
 ⟨proof⟩

**lemmas** *td-ext-sint = td-ext-sbin*

[*simplified len-gt-0 no-sbintr-alt2 Suc-pred' [symmetric]*]

**interpretation** *word-sint*:

$\text{td-ext } \text{sint} :: 'a::\text{len word} \implies \text{int}$   
 $\text{word-of-int}$   
 $\text{sints } (\text{len-of } \text{TYPE}('a::\text{len}))$   
 $\%w. (w + 2^{(\text{len-of } \text{TYPE}('a::\text{len}) - 1)}) \bmod 2^{(\text{len-of } \text{TYPE}('a::\text{len}) - 1)}$   
 ⟨proof⟩

**interpretation** *word-sbin*:

$\text{td-ext } \text{sint} :: 'a::\text{len word} \implies \text{int}$   
 $\text{word-of-int}$   
 $\text{sints } (\text{len-of } \text{TYPE}('a::\text{len}))$   
 $\text{sbintrunc } (\text{len-of } \text{TYPE}('a::\text{len}) - 1)$   
 ⟨proof⟩

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

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

**lemma** *to-bl-def'*:

$(\text{to-bl} :: 'a :: \text{len0 word} \implies \text{bool list}) =$   
 $\text{bin-to-bl } (\text{len-of } \text{TYPE}('a)) \circ \text{uint}$   
 ⟨proof⟩

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

**lemma** *uints-mod*:  $uints\ n = range\ (\lambda w. w\ mod\ 2\ ^\ n)$   
 ⟨proof⟩

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

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

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

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

**lemma** *word-numeral-transfer* [*transfer-rule*]:  
 $(fun-rel\ op = cr-word)\ numeral\ numeral$   
 $(fun-rel\ op = cr-word)\ neg-numeral\ neg-numeral$   
 ⟨proof⟩

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

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

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

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

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

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

**lemma** *size-0-eq*:  $size\ (w :: 'a :: len0\ word) = 0 \implies v = w$

*<proof>*

**lemma** *uint-ge-0* [*iff*]:  $0 \leq \text{uint } (x :: 'a :: \text{len0 word})$   
*<proof>*

**lemma** *uint-lt2p* [*iff*]:  $\text{uint } (x :: 'a :: \text{len0 word}) < 2^{\text{len-of TYPE('a)}}$   
*<proof>*

**lemma** *sint-ge*:  $-(2^{\text{len-of TYPE('a)} - 1}) \leq \text{sint } (x :: 'a :: \text{len word})$   
*<proof>*

**lemma** *sint-lt*:  $\text{sint } (x :: 'a :: \text{len word}) < 2^{\text{len-of TYPE('a)} - 1}$   
*<proof>*

**lemma** *sign-uint-Pls* [*simp*]:  
 $\text{bin-sign } (\text{uint } x) = 0$   
*<proof>*

**lemma** *uint-m2p-neg*:  $\text{uint } (x :: 'a :: \text{len0 word}) - 2^{\text{len-of TYPE('a)}} < 0$   
*<proof>*

**lemma** *uint-m2p-not-non-neg*:  
 $\neg 0 \leq \text{uint } (x :: 'a :: \text{len0 word}) - 2^{\text{len-of TYPE('a)}}$   
*<proof>*

**lemma** *lt2p-lem*:  
 $\text{len-of TYPE('a)} \leq n \implies \text{uint } (w :: 'a :: \text{len0 word}) < 2^n$   
*<proof>*

**lemma** *uint-le-0-iff* [*simp*]:  $\text{uint } x \leq 0 \iff \text{uint } x = 0$   
*<proof>*

**lemma** *uint-nat*:  $\text{uint } w = \text{int } (\text{unat } w)$   
*<proof>*

**lemma** *uint-numeral*:  
 $\text{uint } (\text{numeral } b :: 'a :: \text{len0 word}) = \text{numeral } b \bmod 2^{\text{len-of TYPE('a)}}$   
*<proof>*

**lemma** *uint-neg-numeral*:  
 $\text{uint } (\text{neg-numeral } b :: 'a :: \text{len0 word}) = \text{neg-numeral } b \bmod 2^{\text{len-of TYPE('a)}}$   
*<proof>*

**lemma** *unat-numeral*:  
 $\text{unat } (\text{numeral } b :: 'a :: \text{len0 word}) = \text{numeral } b \bmod 2^{\text{len-of TYPE('a)}}$   
*<proof>*

**lemma** *sint-numeral*:  $\text{sint } (\text{numeral } b :: 'a :: \text{len word}) = (\text{numeral } b + 2^{\text{len-of TYPE('a)} - 1}) \bmod 2^{\text{len-of TYPE('a)} - 1}$

$2 \wedge (\text{len-of TYPE}('a) - 1)$   
 ⟨proof⟩

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

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

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

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

**lemma** *word-int-case-wi*:  
*word-int-case f (word-of-int i :: 'b word) =*  
*f (i mod  $2 \wedge \text{len-of TYPE}('b::\text{len0})$ )*  
 ⟨proof⟩

**lemma** *word-int-split*:  
*P (word-int-case f x) =*  
*(ALL i. x = (word-of-int i :: 'b :: len0 word) &*  
*0 <= i & i <  $2 \wedge \text{len-of TYPE}('b)$  --> 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 <  $2 \wedge \text{len-of TYPE}('b::\text{len0})$  & ~ P (f n)))*  
 ⟨proof⟩

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

**lemma** *uint-range-size*:  $0 \leq \text{uint } w \ \& \ \text{uint } w < 2 \wedge \text{size } w$   
 ⟨proof⟩

**lemma** *sint-range-size*:  
 $-(2 \wedge (\text{size } w - \text{Suc } 0)) \leq \text{sint } w \ \& \ \text{sint } w < 2 \wedge (\text{size } w - \text{Suc } 0)$   
 ⟨proof⟩

**lemma** *sint-above-size*:  $2 \wedge (\text{size } (w::'a::\text{len } \text{word}) - 1) \leq x \implies \text{sint } w < x$   
 ⟨proof⟩

**lemma** *sint-below-size*:  
 $x \leq -(2 \wedge (\text{size } (w::'a::\text{len } \text{word}) - 1)) \implies x \leq \text{sint } w$

*<proof>*

### 13.13 Testing bits

**lemma** *test-bit-eq-iff*:  $(\text{test-bit } (u::'a::\text{len0 word}) = \text{test-bit } v) = (u = v)$   
*<proof>*

**lemma** *test-bit-size* [rule-format] :  $(w::'a::\text{len0 word}) !! n \longrightarrow n < \text{size } w$   
*<proof>*

**lemma** *word-eq-iff*:  
**fixes**  $x y :: 'a::\text{len0 word}$   
**shows**  $x = y \longleftrightarrow (\forall n < \text{len-of TYPE('a)}. x !! n = y !! n)$   
*<proof>*

**lemma** *word-eqI* [rule-format]:  
**fixes**  $u :: 'a::\text{len0 word}$   
**shows**  $(\text{ALL } n. n < \text{size } u \longrightarrow u !! n = v !! n) \Longrightarrow u = v$   
*<proof>*

**lemma** *word-eqD*:  $(u::'a::\text{len0 word}) = v \Longrightarrow u !! x = v !! x$   
*<proof>*

**lemma** *test-bit-bin'*:  $w !! n = (n < \text{size } w \ \& \ \text{bin-nth } (\text{uint } w) \ n)$   
*<proof>*

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

**lemma** *bin-nth-uint-imp*:  
 $\text{bin-nth } (\text{uint } (w::'a::\text{len0 word})) \ n \Longrightarrow n < \text{len-of TYPE('a)}$   
*<proof>*

**lemma** *bin-nth-sint*:  
**fixes**  $w :: 'a::\text{len word}$   
**shows**  $\text{len-of TYPE('a)} \leq n \Longrightarrow$   
 $\text{bin-nth } (\text{sint } w) \ n = \text{bin-nth } (\text{sint } w) \ (\text{len-of TYPE('a)} - 1)$   
*<proof>*

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

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

*<proof>*

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

**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  $\implies$  word-reverse u = w*

*<proof>*

**lemma** *word-rev-gal': u = word-reverse w  $\implies$  w = word-reverse u*

*<proof>*

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

*<proof>*

**lemma** *bl-not-Nil [iff]: to-bl (x::'a::len word)  $\neq$  []*

*<proof>*

**lemma** *length-bl-neq-0 [iff]: length (to-bl (x::'a::len word))  $\neq$  0*

*<proof>*

**lemma** *hd-bl-sign-sint: hd (to-bl w) = (bin-sign (sint w) = -1)*

*<proof>*

**lemma** *of-bl-drop':*

*lend = length bl - len-of TYPE ('a :: len0)  $\implies$*

*of-bl (drop lend bl) = (of-bl bl :: 'a word)*

*<proof>*

**lemma** *test-bit-of-bl:*

*(of-bl bl::'a::len0 word) !! n = (rev bl ! n  $\wedge$  n < len-of TYPE('a)  $\wedge$  n < length bl)*

*<proof>*

**lemma** *no-of-bl:*

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

*<proof>*

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

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

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

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

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

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

**lemma** *uint-bl-bin*:  
**fixes**  $x :: 'a::len0\ word$   
**shows**  $bl-to-bin\ (bin-to-bl\ (len-of\ TYPE('a))\ (uint\ x)) = uint\ x$   
 ⟨*proof*⟩

**lemma** *uints-unats*:  $uints\ n = int\ 'unats\ n$   
 ⟨*proof*⟩

**lemma** *unats-uints*:  $unats\ n = nat\ 'uints\ n$   
 ⟨*proof*⟩

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

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

**lemma** *num-of-bintr'*:  
 $bintrunc\ (len-of\ TYPE('a :: len0))\ (numeral\ a) = (numeral\ b) \implies$   
 $numeral\ a = (numeral\ b :: 'a\ word)$   
 ⟨*proof*⟩

**lemma** *num-of-sbintr'*:  
 $sbintrunc\ (len-of\ TYPE('a :: len) - 1)\ (numeral\ a) = (numeral\ b) \implies$   
 $numeral\ a = (numeral\ b :: 'a\ word)$   
 ⟨*proof*⟩

**lemma** *num-abs-bintr*:

$(\text{numeral } x :: 'a \text{ word}) =$   
 $\text{word-of-int } (\text{bintrunc } (\text{len-of TYPE}('a::\text{len0})) (\text{numeral } x))$   
 $\langle \text{proof} \rangle$

**lemma** *num-abs-sbintr*:

$(\text{numeral } x :: 'a \text{ word}) =$   
 $\text{word-of-int } (\text{sbintrunc } (\text{len-of TYPE}('a::\text{len}) - 1) (\text{numeral } x))$   
 $\langle \text{proof} \rangle$

**lemma** *ucast-id*:  $\text{ucast } w = w$

$\langle \text{proof} \rangle$

**lemma** *scast-id*:  $\text{scast } w = w$

$\langle \text{proof} \rangle$

**lemma** *ucast-bl*:  $\text{ucast } w = \text{of-bl } (\text{to-bl } w)$

$\langle \text{proof} \rangle$

**lemma** *nth-ucast*:

$(\text{ucast } w :: 'a::\text{len0} \text{ word}) !! n = (w !! n \ \& \ n < \text{len-of TYPE}('a))$   
 $\langle \text{proof} \rangle$

**lemma** *ucast-bintr* [*simp*]:

$\text{ucast } (\text{numeral } w :: 'a::\text{len0} \text{ word}) =$   
 $\text{word-of-int } (\text{bintrunc } (\text{len-of TYPE}('a)) (\text{numeral } w))$   
 $\langle \text{proof} \rangle$

**lemma** *scast-sbintr* [*simp*]:

$\text{scast } (\text{numeral } w :: 'a::\text{len} \text{ word}) =$   
 $\text{word-of-int } (\text{sbintrunc } (\text{len-of TYPE}('a) - \text{Suc } 0) (\text{numeral } w))$   
 $\langle \text{proof} \rangle$

**lemma** *source-size*:  $\text{source-size } (c :: 'a::\text{len0} \text{ word} \Rightarrow -) = \text{len-of TYPE}('a)$

$\langle \text{proof} \rangle$

**lemma** *target-size*:  $\text{target-size } (c :: - \Rightarrow 'b::\text{len0} \text{ word}) = \text{len-of TYPE}('b)$

$\langle \text{proof} \rangle$

**lemma** *is-down*:

**fixes**  $c :: 'a::\text{len0} \text{ word} \Rightarrow 'b::\text{len0} \text{ word}$   
**shows**  $\text{is-down } c \longleftrightarrow \text{len-of TYPE}('b) \leq \text{len-of TYPE}('a)$   
 $\langle \text{proof} \rangle$

**lemma** *is-up*:

**fixes**  $c :: 'a::len0\ word \Rightarrow 'b::len0\ word$   
**shows**  $is-up\ c \longleftrightarrow len-of\ TYPE('a) \leq len-of\ TYPE('b)$   
 $\langle proof \rangle$

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

**lemma** *down-cast-same* [*OF refl*]:  $uc = ucast \Longrightarrow is-down\ uc \Longrightarrow uc = scast$   
 $\langle proof \rangle$

**lemma** *word-rev-tf*:

$to-bl\ (of-bl\ bl::'a::len0\ word) =$   
 $rev\ (takefill\ False\ (len-of\ TYPE('a))\ (rev\ bl))$   
 $\langle proof \rangle$

**lemma** *word-rep-drop*:

$to-bl\ (of-bl\ bl::'a::len0\ word) =$   
 $replicate\ (len-of\ TYPE('a) - length\ bl)\ False\ @$   
 $drop\ (length\ bl - len-of\ TYPE('a))\ bl$   
 $\langle proof \rangle$

**lemma** *to-bl-ucast*:

$to-bl\ (ucast\ (w::'b::len0\ word) :: 'a::len0\ word) =$   
 $replicate\ (len-of\ TYPE('a) - len-of\ TYPE('b))\ False\ @$   
 $drop\ (len-of\ TYPE('b) - len-of\ TYPE('a))\ (to-bl\ w)$   
 $\langle proof \rangle$

**lemma** *ucast-up-app* [*OF refl*]:

$uc = ucast \Longrightarrow source-size\ uc + n = target-size\ uc \Longrightarrow$   
 $to-bl\ (uc\ w) = replicate\ n\ False\ @\ (to-bl\ w)$   
 $\langle proof \rangle$

**lemma** *ucast-down-drop* [*OF refl*]:

$uc = ucast \Longrightarrow source-size\ uc = target-size\ uc + n \Longrightarrow$   
 $to-bl\ (uc\ w) = drop\ n\ (to-bl\ w)$   
 $\langle proof \rangle$

**lemma** *scast-down-drop* [*OF refl*]:

$sc = scast \Longrightarrow source-size\ sc = target-size\ sc + n \Longrightarrow$   
 $to-bl\ (sc\ w) = drop\ n\ (to-bl\ w)$   
 $\langle proof \rangle$

**lemma** *sint-up-scast* [*OF refl*]:

$sc = scast \Longrightarrow is-up\ sc \Longrightarrow sint\ (sc\ w) = sint\ w$   
 $\langle proof \rangle$

**lemma** *uint-up-ucast* [*OF refl*]:

$uc = ucast \Longrightarrow is-up\ uc \Longrightarrow uint\ (uc\ w) = uint\ w$

*<proof>*

**lemma** *ucast-up-ucast* [*OF refl*]:

$uc = ucast \implies is-up\ uc \implies ucast\ (uc\ w) = ucast\ w$   
*<proof>*

**lemma** *scast-up-scast* [*OF refl*]:

$sc = scast \implies is-up\ sc \implies scast\ (sc\ w) = scast\ w$   
*<proof>*

**lemma** *ucast-of-bl-up* [*OF refl*]:

$w = of-bl\ bl \implies size\ bl \leq size\ w \implies ucast\ w = of-bl\ bl$   
*<proof>*

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

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

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

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

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

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

**lemma** *up-ucast-surj*:

$is-up\ (ucast :: 'b::len0\ word \Rightarrow 'a::len0\ word) \implies$   
 $surj\ (ucast :: 'a\ word \Rightarrow 'b\ word)$   
*<proof>*

**lemma** *up-scast-surj*:

$is-up\ (scast :: 'b::len\ word \Rightarrow 'a::len\ word) \implies$   
 $surj\ (scast :: 'a\ word \Rightarrow 'b\ word)$   
*<proof>*

**lemma** *down-scast-inj*:

$is-down\ (scast :: 'b::len\ word \Rightarrow 'a::len\ word) \implies$   
 $inj-on\ (ucast :: 'a\ word \Rightarrow 'b\ word)\ A$   
*<proof>*

**lemma** *down-ucast-inj*:

$is-down\ (ucast :: 'b::len0\ word \Rightarrow 'a::len0\ word) \implies$   
 $inj-on\ (ucast :: 'a\ word \Rightarrow 'b\ word)\ A$   
*<proof>*

**lemma** *of-bl-append-same*:  $of-bl\ (X\ @\ to-bl\ w) = w$

*<proof>*

**lemma** *ucast-down-wi* [*OF refl*]:

$uc = ucast \implies is-down\ uc \implies uc\ (word-of-int\ x) = word-of-int\ x$   
*<proof>*

**lemma** *ucast-down-no* [*OF refl*]:

$uc = ucast \implies is\_down\ uc \implies uc\ (numeral\ bin) = numeral\ bin$   
 ⟨*proof*⟩

**lemma** *ucast-down-bl* [*OF refl*]:

$uc = ucast \implies is\_down\ uc \implies uc\ (of\_bl\ bl) = of\_bl\ bl$   
 ⟨*proof*⟩

**lemmas** *slice-def' = slice-def* [*unfolded word-size*]

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

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

### 13.14 Word Arithmetic

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

⟨*proof*⟩

**lemma** *signed-linorder*: *class.linorder word-sle word-sless*

⟨*proof*⟩

**interpretation** *signed*: *linorder word-sle word-sless*

⟨*proof*⟩

**lemma** *udvdI*:

$0 \leq n \implies uint\ b = n * uint\ a \implies a\ udvd\ b$   
 ⟨*proof*⟩

**lemmas** *word-div-no* [*simp*] = *word-div-def* [*of numeral a numeral b*] **for** *a b*

**lemmas** *word-mod-no* [*simp*] = *word-mod-def* [*of numeral a numeral b*] **for** *a b*

**lemmas** *word-less-no* [*simp*] = *word-less-def* [*of numeral a numeral b*] **for** *a b*

**lemmas** *word-le-no* [*simp*] = *word-le-def* [*of numeral a numeral b*] **for** *a b*

**lemmas** *word-sless-no* [*simp*] = *word-sless-def* [*of numeral a numeral b*] **for** *a b*

**lemmas** *word-sle-no* [*simp*] = *word-sle-def* [*of numeral a numeral b*] **for** *a b*

**lemma** *word-1-no*:  $(1::'a::len0\ word) = Numeral1$

⟨*proof*⟩

**lemma** *word-m1-wi*:  $-1 = word\_of\_int\ -1$

⟨*proof*⟩

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

⟨*proof*⟩

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

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

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

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

**lemma** *uint-0-iff*: *(uint x = 0) = (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  $\implies$  w = (v :: 'a :: len0 word)*  
 ⟨*proof*⟩

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

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

**lemma** *unat-gt-0*: *(0 < unat x) = (x  $\sim$  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** *word-arith-alt* =  
*word-sub-wi*  
*word-arith-wis*

### 13.15 Transferring goals from words to ints

**lemma** *word-ths*:

**shows**

*word-succ-p1*: *word-succ a = a + 1* **and**  
*word-pred-m1*: *word-pred a = a - 1* **and**  
*word-pred-succ*: *word-pred (word-succ a) = a* **and**  
*word-succ-pred*: *word-succ (word-pred a) = a* **and**  
*word-mult-succ*: *word-succ a \* b = b + a \* b*  
*<proof>*

**lemma** *uint-cong*: *x = y  $\implies$  uint x = uint y*  
*<proof>*

**lemmas** *uint-word-ariths* =  
*word-arith-alt* [*THEN trans* [*OF uint-cong int-word-uint*]]

**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*]

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

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

**lemma** *word-pred-0-n1*: *word-pred 0 = word-of-int - 1*  
*<proof>*

**lemma** *succ-pred-no* [*simp*]:

*word-succ (numeral w) = numeral w + 1*  
*word-pred (numeral w) = numeral w - 1*  
*word-succ (neg-numeral w) = neg-numeral w + 1*

$word-pred (neg-numeral w) = neg-numeral w - 1$   
 ⟨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*:  
 $\exists y. x = word-of-int y$   
 ⟨proof⟩

### 13.16 Order on fixed-length words

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

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

**lemma** *word-n1-ge* [simp]:  $y \leq (-1 :: 'a :: len0 \ word)$   
 ⟨proof⟩

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

**lemma** *word-gt-0*:  $0 < y \iff 0 \neq (y :: 'a :: len0 \ word)$   
 ⟨proof⟩

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

**lemma** *word-sless-alt*:  $(a <_s b) = (sint a < sint b)$   
 ⟨proof⟩

**lemma** *word-le-nat-alt*:  $(a \leq b) = (unat a \leq unat b)$   
 ⟨proof⟩

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

**lemma** *wi-less*:  
 $(word-of-int n < (word-of-int m :: 'a :: len0 \ word)) =$   
 $(n \bmod 2 \wedge len-of \ TYPE('a) < m \bmod 2 \wedge len-of \ TYPE('a))$   
 ⟨proof⟩

**lemma** *wi-le*:  
 $(word-of-int n \leq (word-of-int m :: 'a :: len0 \ word)) =$   
 $(n \bmod 2 \wedge len-of \ TYPE('a) \leq m \bmod 2 \wedge len-of \ TYPE('a))$

*<proof>*

**lemma** *udvd-nat-alt*:  $a \text{ udvd } b = (EX \ n \geq 0. \text{ unat } b = n * \text{ unat } a)$   
*<proof>*

**lemma** *udvd-iff-dvd*:  $x \text{ udvd } y \iff \text{ unat } x \text{ dvd } \text{ unat } y$   
*<proof>*

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

**lemma** *unat-minus-one*:  $x \sim 0 \implies \text{ unat } (x - 1) = \text{ unat } x - 1$   
*<proof>*

**lemma** *measure-unat*:  $p \sim 0 \implies \text{ unat } (p - 1) < \text{ unat } p$   
*<proof>*

**lemmas** *uint-add-ge0* [simp] = *add-nonneg-nonneg* [OF *uint-ge-0 uint-ge-0*]

**lemmas** *uint-mult-ge0* [simp] = *mult-nonneg-nonneg* [OF *uint-ge-0 uint-ge-0*]

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

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

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

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

**lemma** *uint-sub-lem*:  
 $(\text{ uint } x \geq \text{ uint } y) = (\text{ uint } (x - y) = \text{ uint } x - \text{ uint } y)$   
*<proof>*

**lemma** *uint-add-le*:  $\text{ uint } (x + y) \leq \text{ uint } x + \text{ uint } y$   
*<proof>*

**lemma** *uint-sub-ge*:  $\text{ uint } (x - y) \geq \text{ uint } x - \text{ uint } y$   
*<proof>*

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

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

**13.18 Definition of uint\_arith****lemma** *word-of-int-inverse*:
$$\text{word-of-int } r = a \implies 0 \leq r \implies r < 2^{\text{len-of TYPE('a)}} \implies$$

$$\text{uint } (a :: 'a :: \text{len0 word}) = r$$

*<proof>*

**lemma** *uint-split*:**fixes**  $x :: 'a :: \text{len0 word}$ **shows**  $P (\text{uint } x) =$ 

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

$$(\sim (\text{EX } i. \text{word-of-int } i = x \ \& \ 0 \leq i \ \& \ i < 2^{\text{len-of TYPE('a)}} \ \& \ \sim P \ i))$$
*<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*:  $\text{False} \implies a \wedge b = c \wedge d$ *<proof>**<ML>***13.19 More on overflows and monotonicity****lemma** *no-plus-overflow-uint-size*:
$$((x :: 'a :: \text{len0 word}) \leq x + y) = (\text{uint } x + \text{uint } y < 2^{\text{size } x})$$
*<proof>***lemmas** *no-olen-add* = *no-plus-overflow-uint-size [unfolded word-size]***lemma** *no-olen-sub*:  $((x :: 'a :: \text{len0 word}) \geq x - y) = (\text{uint } y \leq \text{uint } x)$ *<proof>***lemma** *no-olen-add'*:**fixes**  $x :: 'a :: \text{len0 word}$ **shows**  $(x \leq y + x) = (\text{uint } y + \text{uint } x < 2^{\text{len-of TYPE('a)}})$ *<proof>***lemmas** *olen-add-quiv* = *trans [OF no-olen-add no-olen-add' [symmetric]]*

**lemmas** *uint-plus-simple-iff* = *trans* [*OF no-olen-add uint-add-lem*]  
**lemmas** *uint-plus-simple* = *uint-plus-simple-iff* [*THEN iffD1*]  
**lemmas** *uint-minus-simple-iff* = *trans* [*OF no-ulen-sub uint-sub-lem*]  
**lemmas** *uint-minus-simple-alt* = *uint-sub-lem* [*folded word-le-def*]  
**lemmas** *word-sub-le-iff* = *no-ulen-sub* [*folded word-le-def*]  
**lemmas** *word-sub-le* = *word-sub-le-iff* [*THEN iffD2*]

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

**lemma** *word-le-sub1*:  
 $(x :: 'a :: \text{len } \text{word}) \sim = 0 \implies (1 \leq x) = (0 \leq x - 1)$   
*<proof>*

**lemma** *sub-wrap-lt*:  
 $((x :: 'a :: \text{len } 0 \text{ word}) < x - z) = (x < z)$   
*<proof>*

**lemma** *sub-wrap*:  
 $((x :: 'a :: \text{len } 0 \text{ word}) \leq x - z) = (z = 0 \mid x < z)$   
*<proof>*

**lemma** *plus-minus-not-NULL-ab*:  
 $(x :: 'a :: \text{len } 0 \text{ word}) \leq ab - c \implies c \leq ab \implies c \sim = 0 \implies x + c \sim = 0$   
*<proof>*

**lemma** *plus-minus-no-overflow-ab*:  
 $(x :: 'a :: \text{len } 0 \text{ word}) \leq ab - c \implies c \leq ab \implies x \leq x + c$   
*<proof>*

**lemma** *le-minus'*:  
 $(a :: 'a :: \text{len } 0 \text{ word}) + c \leq b \implies a \leq a + c \implies c \leq b - a$   
*<proof>*

**lemma** *le-plus'*:  
 $(a :: 'a :: \text{len } 0 \text{ word}) \leq b \implies c \leq b - a \implies a + c \leq b$   
*<proof>*

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

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

**lemma** *word-plus-mono-right*:  
 $(y :: 'a :: \text{len } 0 \text{ word}) \leq z \implies x \leq x + z \implies x + y \leq x + z$   
*<proof>*

**lemma** *word-less-minus-cancel*:

$$y - x < z - x \implies x \leq z \implies (y :: 'a :: \text{len0 word}) < z$$

*<proof>*

**lemma** *word-less-minus-mono-left:*

$$(y :: 'a :: \text{len0 word}) < z \implies x \leq y \implies y - x < z - x$$

*<proof>*

**lemma** *word-less-minus-mono:*

$$\begin{aligned} a < c \implies d < b \implies a - b < a \implies c - d < c \\ \implies a - b < c - (d :: 'a :: \text{len word}) \end{aligned}$$

*<proof>*

**lemma** *word-le-minus-cancel:*

$$y - x \leq z - x \implies x \leq z \implies (y :: 'a :: \text{len0 word}) \leq z$$

*<proof>*

**lemma** *word-le-minus-mono-left:*

$$(y :: 'a :: \text{len0 word}) \leq z \implies x \leq y \implies y - x \leq z - x$$

*<proof>*

**lemma** *word-le-minus-mono:*

$$\begin{aligned} a \leq c \implies d \leq b \implies a - b \leq a \implies c - d \leq c \\ \implies a - b \leq c - (d :: 'a :: \text{len word}) \end{aligned}$$

*<proof>*

**lemma** *plus-le-left-cancel-wrap:*

$$(x :: 'a :: \text{len0 word}) + y' < x \implies x + y < x \implies (x + y' < x + y) = (y' < y)$$

*<proof>*

**lemma** *plus-le-left-cancel-nowrap:*

$$\begin{aligned} (x :: 'a :: \text{len0 word}) \leq x + y' \implies x \leq x + y \implies \\ (x + y' < x + y) = (y' < y) \end{aligned}$$

*<proof>*

**lemma** *word-plus-mono-right2:*

$$(a :: 'a :: \text{len0 word}) \leq a + b \implies c \leq b \implies a \leq a + c$$

*<proof>*

**lemma** *word-less-add-right:*

$$(x :: 'a :: \text{len0 word}) < y - z \implies z \leq y \implies x + z < y$$

*<proof>*

**lemma** *word-less-sub-right:*

$$(x :: 'a :: \text{len0 word}) < y + z \implies y \leq x \implies x - y < z$$

*<proof>*

**lemma** *word-le-plus-either:*

$$(x :: 'a :: \text{len0 word}) \leq y \mid x \leq z \implies y \leq y + z \implies x \leq y + z$$

*<proof>*

**lemma** *word-less-nowrapI*:

$$(x :: 'a :: \text{len0 word}) < z - k \implies k \leq z \implies 0 < k \implies x < x + k$$

*<proof>*

**lemma** *inc-le*:  $(i :: 'a :: \text{len word}) < m \implies i + 1 \leq m$

*<proof>*

**lemma** *inc-i*:

$$(1 :: 'a :: \text{len word}) \leq i \implies i < m \implies 1 \leq (i + 1) \ \& \ i + 1 \leq m$$

*<proof>*

**lemma** *udvd-incr-lem*:

$$up < uq \implies up = ua + n * \text{uint } K \implies$$

$$uq = ua + n' * \text{uint } K \implies up + \text{uint } K \leq uq$$

*<proof>*

**lemma** *udvd-incr'*:

$$p < q \implies \text{uint } p = ua + n * \text{uint } K \implies$$

$$\text{uint } q = ua + n' * \text{uint } K \implies p + K \leq q$$

*<proof>*

**lemma** *udvd-decr'*:

$$p < q \implies \text{uint } p = ua + n * \text{uint } K \implies$$

$$\text{uint } q = ua + n' * \text{uint } K \implies p \leq q - K$$

*<proof>*

**lemmas** *udvd-incr-lem0* = *udvd-incr-lem* [**where** *ua=0, unfolded add-0-left*]

**lemmas** *udvd-incr0* = *udvd-incr'* [**where** *ua=0, unfolded add-0-left*]

**lemmas** *udvd-decr0* = *udvd-decr'* [**where** *ua=0, unfolded add-0-left*]

**lemma** *udvd-minus-le'*:

$$xy < k \implies z \text{ udvd } xy \implies z \text{ udvd } k \implies xy \leq k - z$$

*<proof>*

*<ML>*

**lemma** *udvd-incr2-K*:

$$p < a + s \implies a \leq a + s \implies K \text{ udvd } s \implies K \text{ udvd } p - a \implies a \leq p \implies$$

$$0 < K \implies p \leq p + K \ \& \ p + K \leq a + s$$

*<proof>*

*<ML>*

**lemma** *word-succ-rbl*:

$$\text{to-bl } w = \text{bl} \implies \text{to-bl } (\text{word-succ } w) = (\text{rev } (\text{rbl-succ } (\text{rev } \text{bl})))$$

*<proof>*

**lemma** *word-pred-rbl*:

$to-bl\ w = bl \implies to-bl\ (word-pred\ w) = (rev\ (rbl-pred\ (rev\ bl)))$   
 ⟨proof⟩

**lemma** *word-add-rbl*:

$to-bl\ v = vbl \implies to-bl\ w = wbl \implies$   
 $to-bl\ (v + w) = (rev\ (rbl-add\ (rev\ vbl)\ (rev\ wbl)))$   
 ⟨proof⟩

**lemma** *word-mult-rbl*:

$to-bl\ v = vbl \implies to-bl\ w = wbl \implies$   
 $to-bl\ (v * w) = (rev\ (rbl-mult\ (rev\ vbl)\ (rev\ wbl)))$   
 ⟨proof⟩

**lemma** *rtb-rbl-ariths*:

$rev\ (to-bl\ w) = ys \implies rev\ (to-bl\ (word-succ\ w)) = rbl-succ\ ys$   
 $rev\ (to-bl\ w) = ys \implies rev\ (to-bl\ (word-pred\ w)) = rbl-pred\ ys$   
 $rev\ (to-bl\ v) = ys \implies rev\ (to-bl\ w) = xs \implies rev\ (to-bl\ (v * w)) = rbl-mult\ ys\ xs$   
 $rev\ (to-bl\ v) = ys \implies rev\ (to-bl\ w) = xs \implies rev\ (to-bl\ (v + w)) = rbl-add\ ys\ xs$   
 ⟨proof⟩

### 13.20 Arithmetic type class instantiations

**lemmas** *word-le-0-iff* [*simp*] =

*word-zero-le* [*THEN leD, THEN linorder-antisym-conv1*]

**lemma** *word-of-int-nat*:

$0 \leq x \implies word-of-int\ x = of-nat\ (nat\ x)$   
 ⟨proof⟩

**lemma** *iszero-word-no* [*simp*]:

$iszero\ (numeral\ bin :: 'a :: len\ word) =$   
 $iszero\ (bintrunc\ (len-of\ TYPE('a))\ (numeral\ bin))$   
 ⟨proof⟩

Use *iszero* to simplify equalities between word numerals.

**lemmas** *word-eq-numeral-iff-iszero* [*simp*] =

*eq-numeral-iff-iszero* [**where** *'a='a::len word*]

### 13.21 Word and nat

**lemma** *td-ext-unat* [*OF refl*]:

$n = len-of\ TYPE\ ('a :: len) \implies$   
 $td-ext\ (unat :: 'a\ word \Rightarrow nat)\ of-nat$   
 $(unats\ n)\ (\%i.\ i\ mod\ 2 \wedge n)$   
 ⟨proof⟩

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

**interpretation** *word-unat*:

*td-ext unat::'a::len word => nat*  
*of-nat*  
*unats (len-of TYPE('a::len))*  
*%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*:  $y \leq \text{unat } (z :: 'a :: \text{len word}) \implies y : \text{unats } (\text{len-of TYPE } ('a))$   
 ⟨*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**  $w :: 'a :: \text{len word}$   
**shows**  $(\text{of-nat } n = w) = (\exists q. n = \text{unat } w + q * 2 ^ \text{len-of TYPE } ('a))$   
 ⟨*proof*⟩

**lemma** *of-nat-eq-size*:

$(\text{of-nat } n = w) = (\text{EX } q. n = \text{unat } w + q * 2 ^ \text{size } w)$   
 ⟨*proof*⟩

**lemma** *of-nat-0*:

$(\text{of-nat } m = (0 :: 'a :: \text{len word})) = (\exists q. m = q * 2 ^ \text{len-of TYPE } ('a))$   
 ⟨*proof*⟩

**lemma** *of-nat-2p [simp]*:

$\text{of-nat } (2 ^ \text{len-of TYPE } ('a)) = (0 :: 'a :: \text{len word})$   
 ⟨*proof*⟩

**lemma** *of-nat-gt-0*:  $\text{of-nat } k \sim = 0 \implies 0 < k$

⟨*proof*⟩

**lemma** *of-nat-neq-0*:

$0 < k \implies k < 2 ^ \text{len-of TYPE } ('a :: \text{len}) \implies \text{of-nat } k \sim = (0 :: 'a \text{ word})$   
 ⟨*proof*⟩

**lemma** *Abs-fnat-hom-add*:

$\text{of-nat } a + \text{of-nat } b = \text{of-nat } (a + b)$   
 ⟨*proof*⟩

**lemma** *Abs-fnat-hom-mult*:

$of\text{-}nat\ a * of\text{-}nat\ b = (of\text{-}nat\ (a * b) :: 'a :: len\ word)$   
 $\langle proof \rangle$

**lemma** *Abs-fnat-hom-Suc*:  
 $word\text{-}succ\ (of\text{-}nat\ a) = of\text{-}nat\ (Suc\ a)$   
 $\langle proof \rangle$

**lemma** *Abs-fnat-hom-0*:  $(0 :: 'a :: len\ word) = of\text{-}nat\ 0$   
 $\langle proof \rangle$

**lemma** *Abs-fnat-hom-1*:  $(1 :: 'a :: len\ word) = of\text{-}nat\ (Suc\ 0)$   
 $\langle proof \rangle$

**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*:  
 $a + b = of\text{-}nat\ (unat\ a + unat\ b)$   
 $\langle proof \rangle$

**lemma** *word-arith-nat-mult*:  
 $a * b = of\text{-}nat\ (unat\ a * unat\ b)$   
 $\langle proof \rangle$

**lemma** *word-arith-nat-Suc*:  
 $word\text{-}succ\ a = of\text{-}nat\ (Suc\ (unat\ a))$   
 $\langle proof \rangle$

**lemma** *word-arith-nat-div*:  
 $a\ div\ b = of\text{-}nat\ (unat\ a\ div\ unat\ b)$   
 $\langle proof \rangle$

**lemma** *word-arith-nat-mod*:  
 $a\ mod\ b = of\text{-}nat\ (unat\ a\ mod\ unat\ b)$   
 $\langle proof \rangle$

**lemmas** *word-arith-nat-defs* =  
*word-arith-nat-add word-arith-nat-mult*  
*word-arith-nat-Suc Abs-fnat-hom-0*  
*Abs-fnat-hom-1 word-arith-nat-div*  
*word-arith-nat-mod*

**lemma** *unat-cong*:  $x = y \implies unat\ x = unat\ y$   
 $\langle proof \rangle$

**lemmas** *unat-word-ariths* = *word-arith-nat-defs*  
 $[THEN\ trans\ [OF\ unat\text{-}cong\ unat\text{-}of\text{-}nat]]$

**lemmas** *word-sub-less-iff* = *word-sub-le-iff*  
 [unfolded linorder-not-less [symmetric] Not-eq-iff]

**lemma** *unat-add-lem*:

(*unat x + unat y < 2 ^ len-of TYPE('a)*) =  
 (*unat (x + y :: 'a :: len word) = unat x + unat y*)  
 ⟨*proof*⟩

**lemma** *unat-mult-lem*:

(*unat x \* unat y < 2 ^ 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]*

**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]*

**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*⟩

### 13.22 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**  $x :: 'a :: \text{len word}$

**shows**  $P (\text{unat } x) =$

$(\sim (EX n. \text{of-nat } n = x \ \& \ n < 2^{\text{len-of TYPE('a)}} \ \& \ \sim P \ n))$

*<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*:

$((x :: 'a :: \text{len word}) \leq x + y) = (\text{unat } x + \text{unat } y < 2^{\text{size } x})$

*<proof>*

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

**lemmas** *unat-plus-simple* = *trans* [*OF no-olen-add-nat unat-add-lem*]

**lemma** *word-div-mult*:

$(0 :: 'a :: \text{len word}) < y \implies \text{unat } x * \text{unat } y < 2^{\text{len-of TYPE('a)}} \implies$

$x * y \text{ div } y = x$

*<proof>*

**lemma** *div-lt'*:  $(i :: 'a :: \text{len word}) \leq k \text{ div } x \implies$

$\text{unat } i * \text{unat } x < 2^{\text{len-of TYPE('a)}}$

*<proof>*

**lemmas** *div-lt''* = *order-less-imp-le* [*THEN div-lt'*]

**lemma** *div-lt-mult*:  $(i :: 'a :: \text{len word}) < k \text{ div } x \implies 0 < x \implies i * x < k$

*<proof>*

**lemma** *div-le-mult*:

$(i :: 'a :: \text{len word}) \leq k \text{ div } x \implies 0 < x \implies i * x \leq k$

*<proof>*

**lemma** *div-lt-uint'*:

$(i :: 'a :: \text{len word}) \leq k \text{ div } x \implies \text{uint } i * \text{uint } x < 2^{\text{len-of TYPE('a)}}$

*<proof>*

**lemmas** *div-lt-uint''* = *order-less-imp-le* [*THEN div-lt-uint'*]

**lemma** *word-le-exists'*:

$(x :: 'a :: \text{len0 word}) \leq y \implies$   
 $(\text{EX } z. y = x + z \ \& \ \text{uint } x + \text{uint } z < 2 \wedge \text{len-of TYPE('a)})$   
 ⟨*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*] **for**  $w \ x$   
**lemmas** *word-diff-ls* = *mcs* [**where**  $z = w + x$ , *unfolded add-diff-cancel*] **for**  $w \ x$   
**lemmas** *word-plus-mcs* = *word-diff-ls* [**where**  $y = v + x$ , *unfolded add-diff-cancel*]  
**for**  $v \ x$

**lemmas** *le-unat-uo* = *unat-le* [*THEN word-unat.Abs-inverse*]

**lemmas** *thd* = *refl* [*THEN* [2] *split-div-lemma* [*THEN iffD2*], *THEN conjunct1*]

**lemma** *thd1*:

$a \ \text{div} \ b * b \leq (a :: \text{nat})$   
 ⟨*proof*⟩

**lemmas** *uno-simps* [*THEN le-unat-uo*] = *mod-le-divisor* *div-le-dividend* *thd1*

**lemma** *word-mod-div-equality*:

$(n \ \text{div} \ b) * b + (n \ \text{mod} \ b) = (n :: 'a :: \text{len word})$   
 ⟨*proof*⟩

**lemma** *word-div-mult-le*:  $a \ \text{div} \ b * b \leq (a :: 'a :: \text{len word})$   
 ⟨*proof*⟩

**lemma** *word-mod-less-divisor*:  $0 < n \implies m \ \text{mod} \ n < (n :: 'a :: \text{len word})$   
 ⟨*proof*⟩

**lemma** *word-of-int-power-hom*:

$\text{word-of-int } a \wedge n = (\text{word-of-int } (a \wedge n) :: 'a :: \text{len word})$   
 ⟨*proof*⟩

**lemma** *word-arith-power-alt*:

$a \wedge n = (\text{word-of-int } (\text{uint } a \wedge n) :: 'a :: \text{len word})$   
 ⟨*proof*⟩

**lemma** *of-bl-length-less*:

$length\ x = k \implies k < len\text{-of}\ TYPE('a) \implies (of\text{-bl}\ x :: 'a :: len\ word) < 2 \wedge k$   
 ⟨proof⟩

### 13.23 Cardinality, finiteness of set of words

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

**lemma**  $card\text{-word}: CARD('a::len0\ word) = 2 \wedge len\text{-of}\ TYPE('a)$   
 ⟨proof⟩

**lemma**  $card\text{-word}\text{-size}$ :  
 $card\ (UNIV :: 'a :: len0\ word\ set) = (2 \wedge size\ (x :: 'a\ word))$   
 ⟨proof⟩

### 13.24 Bitwise Operations on Words

**lemmas**  $bin\text{-log}\text{-bintrs} = bin\text{-trunc}\text{-not}\ bin\text{-trunc}\text{-xor}\ bin\text{-trunc}\text{-and}\ bin\text{-trunc}\text{-or}$

**lemmas**  $wils1 = bin\text{-log}\text{-bintrs}\ [THEN\ word\text{-ubin}\text{-norm}\text{-eq}\text{-iff}\ [THEN\ iffD1],$   
 $folded\ word\text{-ubin}\text{-eq}\text{-norm},\ THEN\ eq\text{-reflection}]$

**lemmas**  $word\text{-log}\text{-binary}\text{-defs} =$   
 $word\text{-and}\text{-def}\ word\text{-or}\text{-def}\ word\text{-xor}\text{-def}$

**lemma**  $word\text{-wi}\text{-log}\text{-defs}$ :  
 $NOT\ word\text{-of}\text{-int}\ a = word\text{-of}\text{-int}\ (NOT\ a)$   
 $word\text{-of}\text{-int}\ a\ AND\ word\text{-of}\text{-int}\ b = word\text{-of}\text{-int}\ (a\ AND\ b)$   
 $word\text{-of}\text{-int}\ a\ OR\ word\text{-of}\text{-int}\ b = word\text{-of}\text{-int}\ (a\ OR\ b)$   
 $word\text{-of}\text{-int}\ a\ XOR\ word\text{-of}\text{-int}\ b = word\text{-of}\text{-int}\ (a\ XOR\ b)$   
 ⟨proof⟩

**lemma**  $word\text{-no}\text{-log}\text{-defs}\ [simp]$ :  
 $NOT\ (numeral\ a) = word\text{-of}\text{-int}\ (NOT\ (numeral\ a))$   
 $NOT\ (neg\text{-numeral}\ a) = word\text{-of}\text{-int}\ (NOT\ (neg\text{-numeral}\ a))$   
 $numeral\ a\ AND\ numeral\ b = word\text{-of}\text{-int}\ (numeral\ a\ AND\ numeral\ b)$   
 $numeral\ a\ AND\ neg\text{-numeral}\ b = word\text{-of}\text{-int}\ (numeral\ a\ AND\ neg\text{-numeral}\ b)$   
 $neg\text{-numeral}\ a\ AND\ numeral\ b = word\text{-of}\text{-int}\ (neg\text{-numeral}\ a\ AND\ numeral\ b)$   
 $neg\text{-numeral}\ a\ AND\ neg\text{-numeral}\ b = word\text{-of}\text{-int}\ (neg\text{-numeral}\ a\ AND\ neg\text{-numeral}\ b)$   
 $numeral\ a\ OR\ numeral\ b = word\text{-of}\text{-int}\ (numeral\ a\ OR\ numeral\ b)$   
 $numeral\ a\ OR\ neg\text{-numeral}\ b = word\text{-of}\text{-int}\ (numeral\ a\ OR\ neg\text{-numeral}\ b)$   
 $neg\text{-numeral}\ a\ OR\ numeral\ b = word\text{-of}\text{-int}\ (neg\text{-numeral}\ a\ OR\ numeral\ b)$   
 $neg\text{-numeral}\ a\ OR\ neg\text{-numeral}\ b = word\text{-of}\text{-int}\ (neg\text{-numeral}\ a\ OR\ neg\text{-numeral}\ b)$   
 b)

*numeral a XOR numeral b = word-of-int (numeral a XOR numeral b)*  
*numeral a XOR neg-numeral b = word-of-int (numeral a XOR neg-numeral b)*  
*neg-numeral a XOR numeral b = word-of-int (neg-numeral a XOR numeral b)*  
*neg-numeral a XOR neg-numeral b = word-of-int (neg-numeral a XOR neg-numeral b)*  
 ⟨proof⟩

Special cases for when one of the arguments equals 1.

**lemma** *word-bitwise-1-simps* [simp]:

*NOT (1::'a::len0 word) = -2*  
*1 AND numeral b = word-of-int (1 AND numeral b)*  
*1 AND neg-numeral b = word-of-int (1 AND neg-numeral b)*  
*numeral a AND 1 = word-of-int (numeral a AND 1)*  
*neg-numeral a AND 1 = word-of-int (neg-numeral a AND 1)*  
*1 OR numeral b = word-of-int (1 OR numeral b)*  
*1 OR neg-numeral b = word-of-int (1 OR neg-numeral b)*  
*numeral a OR 1 = word-of-int (numeral a OR 1)*  
*neg-numeral a OR 1 = word-of-int (neg-numeral a OR 1)*  
*1 XOR numeral b = word-of-int (1 XOR numeral b)*  
*1 XOR neg-numeral b = word-of-int (1 XOR neg-numeral b)*  
*numeral a XOR 1 = word-of-int (numeral a XOR 1)*  
*neg-numeral a XOR 1 = word-of-int (neg-numeral a XOR 1)*  
 ⟨proof⟩

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

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

**lemma** *test-bit-wi* [simp]:

*(word-of-int x::'a::len0 word) !! n  $\longleftrightarrow$  n < len-of TYPE('a)  $\wedge$  bin-nth x n*  
 ⟨proof⟩

**lemma** *word-test-bit-transfer* [transfer-rule]:

*(fun-rel cr-word (fun-rel op = op =))*  
*( $\lambda x n. n < len-of TYPE('a) \wedge bin-nth x n$ ) (test-bit :: 'a::len0 word  $\Rightarrow$  -)*  
 ⟨proof⟩

**lemma** *word-ops-nth-size*:

*n < size (x::'a::len0 word)  $\implies$*   
*(x OR y) !! n = (x !! n | y !! n) &*  
*(x AND y) !! n = (x !! n & y !! n) &*  
*(x XOR y) !! n = (x !! n  $\sim$  y !! n) &*  
*(NOT x) !! n = ( $\sim$  x !! n)*  
 ⟨proof⟩

**lemma** *word-ao-nth*:

*fixes x :: 'a::len0 word*

**shows**  $(x \text{ OR } y) !! n = (x !! n | y !! n) \&$   
 $(x \text{ AND } y) !! n = (x !! n \& y !! n)$   
 ⟨proof⟩

**lemma** *test-bit-numeral* [simp]:  
 $(\text{numeral } w :: 'a::\text{len0 word}) !! n \longleftrightarrow$   
 $n < \text{len-of TYPE}('a) \wedge \text{bin-nth}(\text{numeral } w) n$   
 ⟨proof⟩

**lemma** *test-bit-neg-numeral* [simp]:  
 $(\text{neg-numeral } w :: 'a::\text{len0 word}) !! n \longleftrightarrow$   
 $n < \text{len-of TYPE}('a) \wedge \text{bin-nth}(\text{neg-numeral } w) n$   
 ⟨proof⟩

**lemma** *test-bit-1* [simp]:  $(1 :: 'a::\text{len word}) !! n \longleftrightarrow n = 0$   
 ⟨proof⟩

**lemma** *nth-0* [simp]:  $\sim (0 :: 'a::\text{len0 word}) !! n$   
 ⟨proof⟩

**lemma** *nth-minus1* [simp]:  $(-1 :: 'a::\text{len0 word}) !! n \longleftrightarrow n < \text{len-of TYPE}('a)$   
 ⟨proof⟩

**lemmas** *bwsimps* =  
*wi-hom-add*  
*word-wi-log-defs*

**lemma** *word-bw-assocs*:  
**fixes**  $x :: 'a::\text{len0 word}$   
**shows**  
 $(x \text{ AND } y) \text{ AND } z = x \text{ AND } y \text{ AND } z$   
 $(x \text{ OR } y) \text{ OR } z = x \text{ OR } y \text{ OR } z$   
 $(x \text{ XOR } y) \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$   
 ⟨proof⟩

**lemma** *word-bw-comms*:  
**fixes**  $x :: 'a::\text{len0 word}$   
**shows**  
 $x \text{ AND } y = y \text{ AND } x$   
 $x \text{ OR } y = y \text{ OR } x$   
 $x \text{ XOR } y = y \text{ XOR } x$   
 ⟨proof⟩

**lemma** *word-bw-lcs*:  
**fixes**  $x :: 'a::\text{len0 word}$   
**shows**  
 $y \text{ AND } x \text{ AND } z = x \text{ AND } y \text{ AND } z$

$y \text{ OR } x \text{ OR } z = x \text{ OR } y \text{ OR } z$   
 $y \text{ XOR } x \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$   
 ⟨proof⟩

**lemma** *word-log-esimps* [simp]:

**fixes**  $x :: 'a::len0 \text{ word}$

**shows**

$x \text{ AND } 0 = 0$   
 $x \text{ AND } -1 = x$   
 $x \text{ OR } 0 = x$   
 $x \text{ OR } -1 = -1$   
 $x \text{ XOR } 0 = x$   
 $x \text{ XOR } -1 = \text{NOT } x$   
 $0 \text{ AND } x = 0$   
 $-1 \text{ AND } x = x$   
 $0 \text{ OR } x = x$   
 $-1 \text{ OR } x = -1$   
 $0 \text{ XOR } x = x$   
 $-1 \text{ XOR } x = \text{NOT } x$   
 ⟨proof⟩

**lemma** *word-not-dist*:

**fixes**  $x :: 'a::len0 \text{ word}$

**shows**

$\text{NOT } (x \text{ OR } y) = \text{NOT } x \text{ AND } \text{NOT } y$   
 $\text{NOT } (x \text{ AND } y) = \text{NOT } x \text{ OR } \text{NOT } y$   
 ⟨proof⟩

**lemma** *word-bw-same*:

**fixes**  $x :: 'a::len0 \text{ word}$

**shows**

$x \text{ AND } x = x$   
 $x \text{ OR } x = x$   
 $x \text{ XOR } x = 0$   
 ⟨proof⟩

**lemma** *word-ao-absorbs* [simp]:

**fixes**  $x :: 'a::len0 \text{ word}$

**shows**

$x \text{ AND } (y \text{ OR } x) = x$   
 $x \text{ OR } y \text{ AND } x = x$   
 $x \text{ AND } (x \text{ OR } y) = x$   
 $y \text{ AND } x \text{ OR } x = x$   
 $(y \text{ OR } x) \text{ AND } x = x$   
 $x \text{ OR } x \text{ AND } y = x$   
 $(x \text{ OR } y) \text{ AND } x = x$   
 $x \text{ AND } y \text{ OR } x = x$   
 ⟨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**  $(w = (x\ OR\ y)) \implies (y = (w\ AND\ y))$  ⟨*proof*⟩

**lemma** *leao*:  
**fixes**  $x' :: 'a::len0\ word$   
**shows**  $(w' = (x'\ AND\ y')) \implies (x' = (x'\ OR\ w'))$  ⟨*proof*⟩

**lemmas** *word-ao-equiv* = *leao* [*COMP* *leoa* [*COMP* *iffI*]]

**lemma** *le-word-or2*:  $x \leq x\ OR\ (y::'a::len0\ word)$   
 ⟨*proof*⟩

**lemmas** *le-word-or1* = *xtr3* [*OF* *word-bw-comms* (2) *le-word-or2*]

**lemmas** *word-and-le1* = *xtr3* [*OF* *word-ao-absorbs* (4) [*symmetric*] *le-word-or2*]

**lemmas** *word-and-le2* = *xtr3* [*OF* *word-ao-absorbs* (8) [*symmetric*] *le-word-or2*]

**lemma** *bl-word-not*:  $to-bl\ (NOT\ w) = map\ Not\ (to-bl\ w)$   
 ⟨*proof*⟩

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

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

**lemma** *bl-word-and*:  $to-bl (v \text{ AND } w) = map2 \text{ op } \& (to-bl v) (to-bl w)$   
 ⟨proof⟩

**lemma** *word-lsb-alt*:  $lsb (w::'a::len0 \text{ word}) = test-bit w 0$   
 ⟨proof⟩

**lemma** *word-lsb-1-0* [simp]:  $lsb (1::'a::len \text{ word}) \& \sim lsb (0::'b::len0 \text{ word})$   
 ⟨proof⟩

**lemma** *word-lsb-last*:  $lsb (w::'a::len \text{ word}) = last (to-bl w)$   
 ⟨proof⟩

**lemma** *word-lsb-int*:  $lsb w = (uint w \bmod 2 = 1)$   
 ⟨proof⟩

**lemma** *word-msb-sint*:  $msb w = (sint w < 0)$   
 ⟨proof⟩

**lemma** *msb-word-of-int*:  
 $msb (\text{word-of-int } x::'a::len \text{ word}) = bin-nth x (\text{len-of TYPE('a)} - 1)$   
 ⟨proof⟩

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

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

**lemma** *word-msb-0* [simp]:  $\neg msb (0::'a::len \text{ word})$   
 ⟨proof⟩

**lemma** *word-msb-1* [simp]:  $msb (1::'a::len \text{ word}) \longleftrightarrow \text{len-of TYPE('a)} = 1$   
 ⟨proof⟩

**lemma** *word-msb-nth*:  
 $msb (w::'a::len \text{ word}) = bin-nth (uint w) (\text{len-of TYPE('a)} - 1)$   
 ⟨proof⟩

**lemma** *word-msb-alt*:  $msb (w::'a::len \text{ word}) = hd (to-bl w)$   
 ⟨proof⟩

**lemma** *word-set-nth* [simp]:  
 $set-bit w n (\text{test-bit } w n) = (w::'a::len0 \text{ word})$   
 ⟨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**  $bin\text{-}nth\text{-}uint = bin\text{-}nth\text{-}uint'$  [unfolded word-size]

**lemma**  $test\text{-}bit\text{-}bl: w\ !!\ n = (rev\ (to\text{-}bl\ w)\ !\ n\ \&\ n < size\ w)$   
 ⟨proof⟩

**lemma**  $to\text{-}bl\text{-}nth: n < size\ w \implies to\text{-}bl\ w\ !\ n = w\ !!\ (size\ w - Suc\ n)$   
 ⟨proof⟩

**lemma**  $test\text{-}bit\text{-}set:$   
**fixes**  $w :: 'a::len0\ word$   
**shows**  $(set\text{-}bit\ w\ n\ x)\ !!\ n = (n < size\ w\ \&\ x)$   
 ⟨proof⟩

**lemma**  $test\text{-}bit\text{-}set\text{-}gen:$   
**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**  $of\text{-}bl\text{-}rep\text{-}False: of\text{-}bl\ (replicate\ n\ False\ @\ bs) = of\text{-}bl\ bs$   
 ⟨proof⟩

**lemma**  $msb\text{-}nth:$   
**fixes**  $w :: 'a::len\ word$   
**shows**  $msb\ w = w\ !!\ (len\text{-}of\ TYPE('a) - 1)$   
 ⟨proof⟩

**lemmas**  $msb0 = len\text{-}gt\text{-}0$  [THEN  $diff\text{-}Suc\text{-}less$ , THEN  $word\text{-}ops\text{-}nth\text{-}size$  [unfolded word-size]]

**lemmas**  $msb1 = msb0$  [where  $i = 0$ ]

**lemmas**  $word\text{-}ops\text{-}msb = msb1$  [unfolded  $msb\text{-}nth$  [symmetric, unfolded  $One\text{-}nat\text{-}def$ ]]

**lemmas**  $lsb0 = len\text{-}gt\text{-}0$  [THEN  $word\text{-}ops\text{-}nth\text{-}size$  [unfolded word-size]]

**lemmas**  $word\text{-}ops\text{-}lsb = lsb0$  [unfolded word-lsb-alt]

**lemma**  $td\text{-}ext\text{-}nth$  [OF  $refl\ refl\ refl$ , unfolded word-size]:  
 $n = size\ (w :: 'a::len0\ word) \implies ofn = set\text{-}bits \implies [w, ofn\ g] = l \implies$   
 $td\text{-}ext\ test\text{-}bit\ ofn\ \{f. ALL\ i. f\ i \longrightarrow i < n\}\ (\%h\ i. h\ i\ \&\ i < n)$   
 ⟨proof⟩

**interpretation**  $test\text{-}bit:$   
 $td\text{-}ext\ op\ !! :: 'a::len0\ word \Rightarrow nat \Rightarrow bool$   
 $set\text{-}bits$   
 $\{f. \forall i. f\ i \longrightarrow i < len\text{-}of\ TYPE('a::len0)\}$   
 $(\lambda h\ i. h\ i \wedge i < len\text{-}of\ TYPE('a::len0))$   
 ⟨proof⟩

**lemmas** *td-nth = test-bit.td-thm*

**lemma** *word-set-set-same* [*simp*]:

**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** *nth-sint*:

**fixes** *w :: 'a::len word*

**defines** *l ≡ len-of TYPE ('a)*

**shows** *bin-nth (sint w) n = (if n < l - 1 then w !! n else w !! (l - 1))*

*<proof>*

**lemma** *word-lsb-numeral* [*simp*]:

*lsb (numeral bin :: 'a :: len word) = (bin-last (numeral bin) = 1)*

*<proof>*

**lemma** *word-lsb-neg-numeral* [*simp*]:

*lsb (neg-numeral bin :: 'a :: len word) = (bin-last (neg-numeral bin) = 1)*

*<proof>*

**lemma** *set-bit-word-of-int*:

*set-bit (word-of-int x) n b = word-of-int (bin-sc n (if b then 1 else 0) x)*

*<proof>*

**lemma** *word-set-numeral* [*simp*]:

*set-bit (numeral bin :: 'a::len0 word) n b =*

*word-of-int (bin-sc n (if b then 1 else 0) (numeral bin))*

*<proof>*

**lemma** *word-set-neg-numeral* [*simp*]:

*set-bit (neg-numeral bin :: 'a::len0 word) n b =*

*word-of-int (bin-sc n (if b then 1 else 0) (neg-numeral bin))*

*<proof>*

**lemma** *word-set-bit-0* [*simp*]:

*set-bit 0 n b = word-of-int (bin-sc n (if b then 1 else 0) 0)*

*<proof>*

**lemma** *word-set-bit-1* [*simp*]:

*set-bit 1 n b = word-of-int (bin-sc n (if b then 1 else 0) 1)*

*<proof>*

**lemma** *setBit-no* [simp]:

*setBit* (numeral bin) n = *word-of-int* (bin-sc n 1 (numeral bin))  
 ⟨proof⟩

**lemma** *clearBit-no* [simp]:

*clearBit* (numeral bin) n = *word-of-int* (bin-sc n 0 (numeral bin))  
 ⟨proof⟩

**lemma** *to-bl-n1*:

*to-bl* (-1::'a::len0 word) = *replicate* (len-of TYPE ('a)) True  
 ⟨proof⟩

**lemma** *word-msb-n1* [simp]: *msb* (-1::'a::len word)

⟨proof⟩

**lemma** *word-set-nth-iff*:

(*set-bit* w n b = w) = (w !! n = b | n >= size (w::'a::len0 word))  
 ⟨proof⟩

**lemma** *test-bit-2p*:

(*word-of-int* (2 ^ n)::'a::len word) !! m  $\longleftrightarrow$  m = n  $\wedge$  m < len-of TYPE('a)  
 ⟨proof⟩

**lemma** *nth-w2p*:

((2::'a::len word) ^ n) !! m  $\longleftrightarrow$  m = n  $\wedge$  m < len-of TYPE('a::len)  
 ⟨proof⟩

**lemma** *uint-2p*:

(0::'a::len word) < 2 ^ n  $\implies$  *uint* (2 ^ n::'a::len word) = 2 ^ n  
 ⟨proof⟩

**lemma** *word-of-int-2p*: (*word-of-int* (2 ^ n) :: 'a :: len word) = 2 ^ n

⟨proof⟩

**lemma** *bang-is-le*: x !! m  $\implies$  2 ^ m <= (x :: 'a :: len word)

⟨proof⟩

**lemma** *word-clr-le*:

**fixes** w :: 'a::len0 word

**shows** w >= *set-bit* w n False

⟨proof⟩

**lemma** *word-set-ge*:

**fixes** w :: 'a::len word

**shows** w <= *set-bit* w n True

⟨proof⟩

### 13.25 Shifting, Rotating, and Splitting Words

**lemma** *shiffl1-wi* [simp]: *shiffl1 (word-of-int w) = word-of-int (w BIT 0)*  
 ⟨proof⟩

**lemma** *shiffl1-numeral* [simp]:  
*shiffl1 (numeral w) = numeral (Num.Bit0 w)*  
 ⟨proof⟩

**lemma** *shiffl1-neg-numeral* [simp]:  
*shiffl1 (neg-numeral w) = neg-numeral (Num.Bit0 w)*  
 ⟨proof⟩

**lemma** *shiffl1-0* [simp] : *shiffl1 0 = 0*  
 ⟨proof⟩

**lemma** *shiffl1-def-u*: *shiffl1 w = word-of-int (uint w BIT 0)*  
 ⟨proof⟩

**lemma** *shiffl1-def-s*: *shiffl1 w = word-of-int (sint w BIT 0)*  
 ⟨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** *shiffl-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-shiffl1*: *shiffl1 w !! n = (n < size w & n > 0 & w !! (n - 1))*  
 ⟨proof⟩

**lemma** *nth-shiffl'* [rule-format]:  
 ALL n. ((w::'a::len0 word) << m) !! n = (n < size w & n >= m & w !! (n - m))  
 ⟨proof⟩

**lemmas**  $nth\text{-shiffl} = nth\text{-shiffl}'$  [unfolded word-size]

**lemma**  $nth\text{-shiftr1}$ :  $shiftr1\ w\ !!\ n = w\ !!\ Suc\ n$   
 ⟨proof⟩

**lemma**  $nth\text{-shiftr}$ :  
 $\bigwedge n. ((w::'a::len0\ word) >> m) !! n = w !! (n + m)$   
 ⟨proof⟩

**lemma**  $uint\text{-shiftr1}$ :  $uint\ (shiftr1\ w) = bin\text{-rest}\ (uint\ w)$   
 ⟨proof⟩

**lemma**  $nth\text{-sshiftr1}$ :  
 $sshiftr1\ w\ !!\ n = (if\ n = size\ w - 1\ then\ w\ !!\ n\ else\ w\ !!\ Suc\ n)$   
 ⟨proof⟩

**lemma**  $nth\text{-sshiftr}$  [rule-format] :  
 ALL  $n. sshiftr\ w\ m\ !!\ n = (n < size\ w \ \&$   
 $(if\ n + m \geq size\ w\ then\ w\ !!\ (size\ w - 1)\ else\ w\ !!\ (n + m)))$   
 ⟨proof⟩

**lemma**  $shiftr1\text{-div-2}$ :  $uint\ (shiftr1\ w) = uint\ w\ div\ 2$   
 ⟨proof⟩

**lemma**  $sshiftr1\text{-div-2}$ :  $sint\ (sshiftr1\ w) = sint\ w\ div\ 2$   
 ⟨proof⟩

**lemma**  $shiftr\text{-div-2n}$ :  $uint\ (shiftr\ w\ n) = uint\ w\ div\ 2\ ^\ n$   
 ⟨proof⟩

**lemma**  $sshiftr\text{-div-2n}$ :  $sint\ (sshiftr\ w\ n) = sint\ w\ div\ 2\ ^\ n$   
 ⟨proof⟩

### 13.25.1 shift functions in terms of lists of bools

**lemmas**  $bshiftr1\text{-numeral}$  [simp] =  
 $bshiftr1\text{-def}$  [where  $w = numeral\ w$ , unfolded to-bl-numeral] for  $w$

**lemma**  $bshiftr1\text{-bl}$ :  $to\text{-bl}\ (bshiftr1\ b\ w) = b\ \#\ butlast\ (to\text{-bl}\ w)$   
 ⟨proof⟩

**lemma**  $shiffl1\text{-of-bl}$ :  $shiffl1\ (of\text{-bl}\ bl) = of\text{-bl}\ (bl\ @\ [False])$   
 ⟨proof⟩

**lemma**  $shiffl1\text{-bl}$ :  $shiffl1\ (w::'a::len0\ word) = of\text{-bl}\ (to\text{-bl}\ w\ @\ [False])$   
 ⟨proof⟩

**lemma** *bl-shiftl1*:

$to-bl (shiftl1 (w :: 'a :: len word)) = tl (to-bl w) @ [False]$   
 ⟨proof⟩

**lemma** *bl-shiftl1'*:

$to-bl (shiftl1 w) = tl (to-bl w @ [False])$   
 ⟨proof⟩

**lemma** *shiftr1-bl*:  $shiftr1 w = of-bl (butlast (to-bl w))$

⟨proof⟩

**lemma** *bl-shiftr1*:

$to-bl (shiftr1 (w :: 'a :: len word)) = False \# butlast (to-bl w)$   
 ⟨proof⟩

**lemma** *bl-shiftr1'*:

$to-bl (shiftr1 w) = butlast (False \# to-bl w)$   
 ⟨proof⟩

**lemma** *shiftl1-rev*:

$shiftl1 w = word-reverse (shiftr1 (word-reverse w))$   
 ⟨proof⟩

**lemma** *shiftl-rev*:

$shiftl w n = word-reverse (shiftr (word-reverse w) n)$   
 ⟨proof⟩

**lemma** *rev-shiftl*:  $word-reverse w \ll n = word-reverse (w \gg n)$

⟨proof⟩

**lemma** *shiftr-rev*:  $w \gg n = word-reverse (word-reverse w \ll n)$

⟨proof⟩

**lemma** *rev-shiftr*:  $word-reverse w \gg n = word-reverse (w \ll n)$

⟨proof⟩

**lemma** *bl-sshiftr1*:

$to-bl (sshiftr1 (w :: 'a :: len word)) = hd (to-bl w) \# butlast (to-bl w)$   
 ⟨proof⟩

**lemma** *drop-shiftr*:

$drop n (to-bl ((w :: 'a :: len word) \gg n)) = take (size w - n) (to-bl w)$   
 ⟨proof⟩

**lemma** *drop-sshiftr*:

$drop n (to-bl ((w :: 'a :: len word) \ggg n)) = take (size w - n) (to-bl w)$

*<proof>*

**lemma** *take-shiftr*:

$n \leq \text{size } w \implies \text{take } n (\text{to-bl } (w \gg n)) = \text{replicate } n \text{ False}$

*<proof>*

**lemma** *take-sshiftr'* [rule-format] :

$n \leq \text{size } (w :: 'a :: \text{len } \text{word}) \dashrightarrow \text{hd } (\text{to-bl } (w \gg \gg n)) = \text{hd } (\text{to-bl } w) \ \&$   
 $\text{take } n (\text{to-bl } (w \gg \gg n)) = \text{replicate } n (\text{hd } (\text{to-bl } w))$

*<proof>*

**lemmas** *hd-sshiftr* = *take-sshiftr'* [THEN *conjunct1*]

**lemmas** *take-sshiftr* = *take-sshiftr'* [THEN *conjunct2*]

**lemma** *atd-lem*:  $\text{take } n \text{ xs} = t \implies \text{drop } n \text{ xs} = d \implies \text{xs} = t @ d$

*<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*:  $\text{of-bl } bl \ll n = \text{of-bl } (bl @ \text{replicate } n \text{ False})$

*<proof>*

**lemma** *shiftl-bl*:

$(w :: 'a :: \text{len } 0 \text{ word}) \ll (n :: \text{nat}) = \text{of-bl } (\text{to-bl } w @ \text{replicate } n \text{ False})$

*<proof>*

**lemmas** *shiftl-numeral* [simp] = *shiftl-def* [where *w=numeral w*] **for** *w*

**lemma** *bl-shiftl*:

$\text{to-bl } (w \ll n) = \text{drop } n (\text{to-bl } w) @ \text{replicate } (\text{min } (\text{size } w) \ n) \ \text{False}$

*<proof>*

**lemma** *shiftl-zero-size*:

**fixes** *x* :: 'a :: len 0 word

**shows**  $\text{size } x \leq n \implies x \ll n = 0$

*<proof>*

**lemma** *shiftl1-2t*:  $\text{shiftl1 } (w :: 'a :: \text{len } \text{word}) = 2 * w$

*<proof>*

**lemma** *shiftl1-p*:  $\text{shiftl1 } (w :: 'a :: \text{len } \text{word}) = w + w$

*<proof>*

**lemma** *shiftl-t2n*:  $\text{shiftl } (w :: 'a :: \text{len } \text{word}) \ n = 2 ^ n * w$

*<proof>*

**lemma** *shiftr1-bintr* [simp]:

(*shiftr1* (numeral *w*) :: 'a :: len0 word) =  
 word-of-int (bin-rest (bintrunc (len-of TYPE ('a)) (numeral *w*)))  
 ⟨proof⟩

**lemma** *sshiftr1-sbintr* [simp]:

(*sshiftr1* (numeral *w*) :: 'a :: len word) =  
 word-of-int (bin-rest (sbintrunc (len-of TYPE ('a) - 1) (numeral *w*)))  
 ⟨proof⟩

**lemma** *shiftr-no* [simp]:

(numeral *w*::'a::len0 word) >> *n* = word-of-int  
 ((bin-rest ^^ *n*) (bintrunc (len-of TYPE('a)) (numeral *w*)))  
 ⟨proof⟩

**lemma** *sshiftr-no* [simp]:

(numeral *w*::'a::len word) >>> *n* = word-of-int  
 ((bin-rest ^^ *n*) (sbintrunc (len-of TYPE('a) - 1) (numeral *w*)))  
 ⟨proof⟩

**lemma** *shiftr1-bl-of*:

length *bl* ≤ len-of TYPE('a) ⇒  
*shiftr1* (of-bl *bl*::'a::len0 word) = of-bl (butlast *bl*)  
 ⟨proof⟩

**lemma** *shiftr-bl-of*:

length *bl* ≤ len-of TYPE('a) ⇒  
 (of-bl *bl*::'a::len0 word) >> *n* = of-bl (take (length *bl* - *n*) *bl*)  
 ⟨proof⟩

**lemma** *shiftr-bl*:

(*x*::'a::len0 word) >> *n* ≡ of-bl (take (len-of TYPE('a) - *n*) (to-bl *x*))  
 ⟨proof⟩

**lemma** *msb-shift*:

*msb* (*w*::'a::len word) ↔ (*w* >> (len-of TYPE('a) - 1)) ≠ 0  
 ⟨proof⟩

**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 -->  
 map2 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 -->

$\text{map2 } op \ \& \ x \ y = \text{replicate } (n + m) \ \text{False}$   
 ⟨proof⟩

**lemma** *aligned-bl-add-size* [OF refl]:

$\text{size } x - n = m \implies n \leq \text{size } x \implies \text{drop } m \ (\text{to-bl } x) = \text{replicate } n \ \text{False} \implies$   
 $\text{take } m \ (\text{to-bl } y) = \text{replicate } m \ \text{False} \implies$   
 $\text{to-bl } (x + y) = \text{take } m \ (\text{to-bl } x) \ @ \ \text{drop } m \ (\text{to-bl } y)$   
 ⟨proof⟩

### 13.25.2 Mask

**lemma** *nth-mask* [OF refl, simp]:

$m = \text{mask } n \implies \text{test-bit } m \ i = (i < n \ \& \ i < \text{size } m)$   
 ⟨proof⟩

**lemma** *mask-bl*:  $\text{mask } n = \text{of-bl } (\text{replicate } n \ \text{True})$

⟨proof⟩

**lemma** *mask-bin*:  $\text{mask } n = \text{word-of-int } (\text{bintrunc } n - 1)$

⟨proof⟩

**lemma** *and-mask-bintr*:  $w \ \text{AND} \ \text{mask } n = \text{word-of-int } (\text{bintrunc } n \ (\text{uint } w))$

⟨proof⟩

**lemma** *and-mask-wi*:  $\text{word-of-int } i \ \text{AND} \ \text{mask } n = \text{word-of-int } (\text{bintrunc } n \ i)$

⟨proof⟩

**lemma** *and-mask-no*:  $\text{numeral } i \ \text{AND} \ \text{mask } n = \text{word-of-int } (\text{bintrunc } n \ (\text{numeral } i))$

⟨proof⟩

**lemma** *bl-and-mask'*:

$\text{to-bl } (w \ \text{AND} \ \text{mask } n :: 'a :: \text{len } \text{word}) =$   
 $\text{replicate } (\text{len-of } \text{TYPE}('a) - n) \ \text{False} \ @$   
 $\text{drop } (\text{len-of } \text{TYPE}('a) - n) \ (\text{to-bl } w)$   
 ⟨proof⟩

**lemma** *and-mask-mod-2p*:  $w \ \text{AND} \ \text{mask } n = \text{word-of-int } (\text{uint } w \ \text{mod } 2 \ ^n)$

⟨proof⟩

**lemma** *and-mask-lt-2p*:  $\text{uint } (w \ \text{AND} \ \text{mask } n) < 2 \ ^n$

⟨proof⟩

**lemma** *eq-mod-iff*:  $0 < (n :: \text{int}) \implies b = b \ \text{mod } n \iff 0 \leq b \ \wedge \ b < n$

⟨proof⟩

**lemma** *mask-eq-iff*:  $(w \ \text{AND} \ \text{mask } n) = w \iff \text{uint } w < 2 \ ^n$

⟨proof⟩

**lemma** *and-mask-dvd*:  $2 \wedge n \text{ dvd uint } w = (w \text{ AND mask } n = 0)$   
 ⟨proof⟩

**lemma** *and-mask-dvd-nat*:  $2 \wedge n \text{ dvd unat } w = (w \text{ AND mask } n = 0)$   
 ⟨proof⟩

**lemma** *word-2p-lem*:  
 $n < \text{size } w \implies w < 2 \wedge n = (\text{uint } (w :: 'a :: \text{len word}) < 2 \wedge n)$   
 ⟨proof⟩

**lemma** *less-mask-eq*:  $x < 2 \wedge n \implies x \text{ AND mask } n = (x :: 'a :: \text{len word})$   
 ⟨proof⟩

**lemmas** *mask-eq-iff-w2p* = *trans* [*OF* *mask-eq-iff* *word-2p-lem* [*symmetric*]]

**lemmas** *and-mask-less'* = *iffD2* [*OF* *word-2p-lem* *and-mask-lt-2p*, *simplified word-size*]

**lemma** *and-mask-less-size*:  $n < \text{size } x \implies x \text{ AND mask } n < 2 \wedge n$   
 ⟨proof⟩

**lemma** *word-mod-2p-is-mask* [*OF* *refl*]:  
 $c = 2 \wedge n \implies c > 0 \implies x \text{ mod } c = (x :: 'a :: \text{len word}) \text{ AND mask } n$   
 ⟨proof⟩

**lemma** *mask-egs*:  
 $(a \text{ AND mask } n) + b \text{ AND mask } n = a + b \text{ AND mask } n$   
 $a + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$   
 $(a \text{ AND mask } n) - b \text{ AND mask } n = a - b \text{ AND mask } n$   
 $a - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$   
 $a * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$   
 $(b \text{ AND mask } n) * a \text{ AND mask } n = b * a \text{ AND mask } n$   
 $(a \text{ AND mask } n) + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$   
 $(a \text{ AND mask } n) - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$   
 $(a \text{ AND mask } n) * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$   
 $-(a \text{ AND mask } n) \text{ AND mask } n = -a \text{ AND mask } n$   
 $\text{word-succ } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-succ } a \text{ AND mask } n$   
 $\text{word-pred } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-pred } a \text{ AND mask } n$   
 ⟨proof⟩

**lemma** *mask-power-eq*:  
 $(x \text{ AND mask } n) \wedge k \text{ AND mask } n = x \wedge k \text{ AND mask } n$   
 ⟨proof⟩

### 13.25.3 Recast

**lemmas** *revcast-def'* = *revcast-def* [*simplified*]

**lemmas** *revcast-def''* = *revcast-def'* [*simplified word-size*]

**lemmas** *revcast-no-def* [*simp*] = *revcast-def'* [**where**  $w = \text{numeral } w$ , *unfolded word-size*]  
**for**  $w$

**lemma** *to-bl-revcast*:

$$\begin{aligned} \text{to-bl } (\text{revcast } w :: 'a :: \text{len0 } \text{word}) &= \\ \text{takefill } \text{False } (\text{len-of } \text{TYPE } ('a)) (\text{to-bl } w) & \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *revcast-rev-ucast* [*OF refl refl refl*]:

$$\begin{aligned} cs = [rc, uc] \implies rc = \text{revcast } (\text{word-reverse } w) \implies uc = \text{ucast } w \implies \\ rc = \text{word-reverse } uc & \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *revcast-ucast*:  $\text{revcast } w = \text{word-reverse } (\text{ucast } (\text{word-reverse } w))$

$\langle \text{proof} \rangle$

**lemma** *ucast-revcast*:  $\text{ucast } w = \text{word-reverse } (\text{revcast } (\text{word-reverse } w))$

$\langle \text{proof} \rangle$

**lemma** *ucast-rev-revcast*:  $\text{ucast } (\text{word-reverse } w) = \text{word-reverse } (\text{revcast } w)$

$\langle \text{proof} \rangle$

**lemmas** *wsst-TYs* = *source-size target-size word-size*

**lemma** *revcast-down-uu* [*OF refl*]:

$$\begin{aligned} rc = \text{revcast} \implies \text{source-size } rc = \text{target-size } rc + n \implies \\ rc (w :: 'a :: \text{len } \text{word}) = \text{ucast } (w \gg n) & \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *revcast-down-us* [*OF refl*]:

$$\begin{aligned} rc = \text{revcast} \implies \text{source-size } rc = \text{target-size } rc + n \implies \\ rc (w :: 'a :: \text{len } \text{word}) = \text{ucast } (w \gg \gg n) & \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *revcast-down-su* [*OF refl*]:

$$\begin{aligned} rc = \text{revcast} \implies \text{source-size } rc = \text{target-size } rc + n \implies \\ rc (w :: 'a :: \text{len } \text{word}) = \text{scast } (w \gg n) & \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *revcast-down-ss* [*OF refl*]:

$$\begin{aligned} rc = \text{revcast} \implies \text{source-size } rc = \text{target-size } rc + n \implies \\ rc (w :: 'a :: \text{len } \text{word}) = \text{scast } (w \gg \gg n) & \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *cast-down-rev*:

$$\begin{aligned} uc = \text{ucast} \implies \text{source-size } uc = \text{target-size } uc + n \implies \\ uc w = \text{revcast } ((w :: 'a :: \text{len } \text{word}) \ll n) & \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *revcast-up* [*OF refl*]:

$rc = \text{revcast} \implies \text{source-size } rc + n = \text{target-size } rc \implies$   
 $rc\ w = (\text{ucast } w :: 'a :: \text{len word}) \ll n$   
 ⟨proof⟩

**lemmas**  $rc1 = \text{revcast-up}$  [THEN  
 $\text{revcast-rev-ucast}$  [symmetric, THEN trans, THEN word-rev-gal, symmetric]]  
**lemmas**  $rc2 = \text{revcast-down-uu}$  [THEN  
 $\text{revcast-rev-ucast}$  [symmetric, THEN trans, THEN word-rev-gal, symmetric]]

**lemmas**  $\text{ucast-up} =$   
 $rc1$  [simplified rev-shiftr [symmetric] revcast-ucast [symmetric]]  
**lemmas**  $\text{ucast-down} =$   
 $rc2$  [simplified rev-shiftr revcast-ucast [symmetric]]

### 13.25.4 Slices

**lemma**  $\text{slice1-no-bin}$  [simp]:  
 $\text{slice1 } n$  (numeral  $w :: 'b$  word) = of-bl (takefill False  $n$  (bin-to-bl (len-of TYPE('b  
 $:: \text{len0}$ )) (numeral  $w$ )))  
 ⟨proof⟩

**lemma**  $\text{slice-no-bin}$  [simp]:  
 $\text{slice } n$  (numeral  $w :: 'b$  word) = of-bl (takefill False (len-of TYPE('b :: len0) –  
 $n$ )  
 (bin-to-bl (len-of TYPE('b :: len0)) (numeral  $w$ )))  
 ⟨proof⟩

**lemma**  $\text{slice1-0}$  [simp] :  $\text{slice1 } n\ 0 = 0$   
 ⟨proof⟩

**lemma**  $\text{slice-0}$  [simp] :  $\text{slice } n\ 0 = 0$   
 ⟨proof⟩

**lemma**  $\text{slice-take'}$ :  $\text{slice } n\ w = \text{of-bl}$  (take (size  $w - n$ ) (to-bl  $w$ ))  
 ⟨proof⟩

**lemmas**  $\text{slice-take} = \text{slice-take'}$  [unfolded word-size]

— shiftr to a word of the same size is just slice, slice is just shiftr then ucast

**lemmas**  $\text{shiftr-slice} = \text{trans}$  [OF shiftr-bl [THEN meta-eq-to-obj-eq] slice-take [symmetric]]

**lemma**  $\text{slice-shiftr}$ :  $\text{slice } n\ w = \text{ucast}$  ( $w \gg n$ )  
 ⟨proof⟩

**lemma**  $\text{nth-slice}$ :  
 ( $\text{slice } n\ w :: 'a :: \text{len0 word}$ ) !!  $m =$   
 ( $w$  !! ( $m + n$ ) &  $m < \text{len-of TYPE ('a)}$ )  
 ⟨proof⟩

**lemma** *slice1-down-alt'*:

$sl = slice1\ n\ w \implies fs = size\ sl \implies fs + k = n \implies$   
 $to-bl\ sl = takefill\ False\ fs\ (drop\ k\ (to-bl\ w))$   
 ⟨proof⟩

**lemma** *slice1-up-alt'*:

$sl = slice1\ n\ w \implies fs = size\ sl \implies fs = n + k \implies$   
 $to-bl\ sl = takefill\ False\ fs\ (replicate\ k\ False\ @\ (to-bl\ w))$   
 ⟨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-alt =*

*le-add-diff-inverse* [symmetric, THEN su1]

*le-add-diff-inverse2* [symmetric, THEN su1]

**lemma** *ucast-slice1*:  $ucast\ w = slice1\ (size\ w)\ w$

⟨proof⟩

**lemma** *ucast-slice*:  $ucast\ w = slice\ 0\ w$

⟨proof⟩

**lemma** *slice-id*:  $slice\ 0\ t = t$

⟨proof⟩

**lemma** *revcast-slice1* [OF refl]:

$rc = revcast\ w \implies slice1\ (size\ rc)\ w = rc$   
 ⟨proof⟩

**lemma** *slice1-tf-tf'*:

$to-bl\ (slice1\ n\ w :: 'a :: len0\ word) =$   
 $rev\ (takefill\ False\ (len-of\ TYPE('a))\ (rev\ (takefill\ False\ n\ (to-bl\ w))))$   
 ⟨proof⟩

**lemmas** *slice1-tf-tf = slice1-tf-tf'* [THEN word-bl.Rep-inverse', symmetric]

**lemma** *rev-slice1*:

$n + k = len-of\ TYPE('a) + len-of\ TYPE('b) \implies$   
 $slice1\ n\ (word-reverse\ w :: 'b :: len0\ word) =$   
 $word-reverse\ (slice1\ k\ w :: 'a :: len0\ word)$   
 ⟨proof⟩

**lemma** *rev-slice*:

$n + k + len-of\ TYPE('a::len0) = len-of\ TYPE('b::len0) \implies$   
 $slice\ n\ (word-reverse\ (w::'b\ word)) = word-reverse\ (slice\ k\ w::'a\ word)$   
 ⟨proof⟩

**lemmas** *sym-notr =*

*not-iff* [THEN iffD2, THEN not-sym, THEN not-iff [THEN iffD1]]

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

**lemma** *soft-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⟩

### 13.26 Split and cat

**lemmas** *word-split-bin'* = *word-split-def*

**lemmas** *word-cat-bin'* = *word-cat-def*

**lemma** *word-rsplit-no*:

$(word\ rsplit\ (numeral\ bin :: 'b :: len0\ word) :: 'a\ word\ list) =$   
 $map\ word\ of\ int\ (bin\ rsplit\ (len\ of\ TYPE('a :: len))$   
 $(len\ of\ TYPE('b),\ bintrunc\ (len\ of\ TYPE('b))\ (numeral\ bin)))$   
 ⟨proof⟩

**lemmas** *word-rsplit-no-cl* [simp] = *word-rsplit-no*  
 [unfolded *bin-rsplitl-def* *bin-rsplit-l* [symmetric]]

**lemma** *test-bit-cat*:

$wc = word\ cat\ a\ b \implies wc\ !!\ n = (n < size\ wc \ \&$   
 $(if\ n < size\ b\ then\ b\ !!\ n\ else\ a\ !!\ (n - size\ b)))$   
 ⟨proof⟩

**lemma** *word-cat-bl*:  $word\ cat\ a\ b = of\ bl\ (to\ bl\ a\ @\ to\ bl\ b)$   
 ⟨proof⟩

**lemma** *of-bl-append*:

$(of\ bl\ (xs\ @\ ys) :: 'a :: len\ word) = of\ bl\ xs * 2^{length\ ys} + of\ bl\ ys$   
 ⟨proof⟩

**lemma** *of-bl-False* [simp]:

$of\ bl\ (False\ \#\ xs) = of\ bl\ xs$   
 ⟨proof⟩

**lemma** *of-bl-True* [simp]:

$(of\ bl\ (True\ \#\ xs) :: 'a :: len\ word) = 2^{length\ xs} + of\ bl\ xs$   
 ⟨proof⟩

**lemma** *of-bl-Cons*:

$of\ bl\ (x\ \#\ xs) = of\ bool\ x * 2^{length\ xs} + of\ bl\ xs$   
 ⟨proof⟩

**lemma** *split-uint-lem*:  $bin\ split\ n\ (uint\ (w :: 'a :: len0\ word)) = (a, b) \implies$

$a = \text{bintrunc } (\text{len-of TYPE}('a) - n) a \ \& \ b = \text{bintrunc } (\text{len-of TYPE}('a)) b$   
 ⟨proof⟩

**lemma** *word-split-bl'*:

$\text{std} = \text{size } c - \text{size } b \implies (\text{word-split } c = (a, b)) \implies$   
 $(a = \text{of-bl } (\text{take } \text{std } (\text{to-bl } c)) \ \& \ b = \text{of-bl } (\text{drop } \text{std } (\text{to-bl } c)))$   
 ⟨proof⟩

**lemma** *word-split-bl*:  $\text{std} = \text{size } c - \text{size } b \implies$

$(a = \text{of-bl } (\text{take } \text{std } (\text{to-bl } c)) \ \& \ b = \text{of-bl } (\text{drop } \text{std } (\text{to-bl } c))) <->$   
 $\text{word-split } c = (a, b)$   
 ⟨proof⟩

**lemma** *word-split-bl-eq*:

$(\text{word-split } (c::'a::\text{len } \text{word}) :: ('c :: \text{len0 } \text{word} * 'd :: \text{len0 } \text{word})) =$   
 $(\text{of-bl } (\text{take } (\text{len-of TYPE}('a::\text{len}) - \text{len-of TYPE}('d::\text{len0})) (\text{to-bl } c)),$   
 $\text{of-bl } (\text{drop } (\text{len-of TYPE}('a) - \text{len-of TYPE}('d)) (\text{to-bl } c)))$   
 ⟨proof⟩

**lemma** *test-bit-split'*:

$\text{word-split } c = (a, b) \dashrightarrow (\text{ALL } n \ m. \ b \ !! \ n = (n < \text{size } b \ \& \ c \ !! \ n) \ \&$   
 $a \ !! \ m = (m < \text{size } a \ \& \ c \ !! \ (m + \text{size } b)))$   
 ⟨proof⟩

**lemma** *test-bit-split*:

$\text{word-split } c = (a, b) \implies$   
 $(\forall n::\text{nat}. \ b \ !! \ n \longleftrightarrow n < \text{size } b \ \wedge \ c \ !! \ n) \ \wedge \ (\forall m::\text{nat}. \ a \ !! \ m \longleftrightarrow m < \text{size } a$   
 $\wedge \ c \ !! \ (m + \text{size } b))$   
 ⟨proof⟩

**lemma** *test-bit-split-eq*:  $\text{word-split } c = (a, b) <->$

$((\text{ALL } n::\text{nat}. \ b \ !! \ n = (n < \text{size } b \ \& \ c \ !! \ n)) \ \&$   
 $(\text{ALL } m::\text{nat}. \ a \ !! \ m = (m < \text{size } a \ \& \ c \ !! \ (m + \text{size } b))))$   
 ⟨proof⟩

**lemma** *word-cat-id*:  $\text{word-cat } a \ b = b$

⟨proof⟩

**lemma** *word-cat-hom*:

$\text{len-of TYPE}('a::\text{len0}) \leq \text{len-of TYPE}('b::\text{len0}) + \text{len-of TYPE}('c::\text{len0})$   
 $\implies$   
 $(\text{word-cat } (\text{word-of-int } w :: 'b \ \text{word}) (b :: 'c \ \text{word}) :: 'a \ \text{word}) =$   
 $\text{word-of-int } (\text{bin-cat } w \ (\text{size } b) \ (\text{uint } b))$   
 ⟨proof⟩

**lemma** *word-cat-split-alt*:

$\text{size } w \leq \text{size } u + \text{size } v \implies \text{word-split } w = (u, v) \implies \text{word-cat } u \ v = w$   
 ⟨proof⟩

**lemmas** *word-cat-split-size* = *sym* [THEN [2] *word-cat-split-alt* [symmetric]]

**13.26.1 Split and slice****lemma** *split-slices*:
$$\text{word-split } w = (u, v) \implies u = \text{slice } (\text{size } v) \ w \ \& \ v = \text{slice } 0 \ w$$

*<proof>*

**lemma** *slice-cat1* [*OF refl*]:
$$wc = \text{word-cat } a \ b \implies \text{size } wc \geq \text{size } a + \text{size } b \implies \text{slice } (\text{size } b) \ wc = a$$

*<proof>*

**lemmas** *slice-cat2* = *trans* [*OF slice-id word-cat-id*]**lemma** *cat-slices*:
$$a = \text{slice } n \ c \implies b = \text{slice } 0 \ c \implies n = \text{size } b \implies$$

$$\text{size } a + \text{size } b \geq \text{size } c \implies \text{word-cat } a \ b = c$$

*<proof>*

**lemma** *word-split-cat-alt*:
$$w = \text{word-cat } u \ v \implies \text{size } u + \text{size } v \leq \text{size } w \implies \text{word-split } w = (u, v)$$

*<proof>*

**lemmas** *word-cat-bl-no-bin* [*simp*] =
$$\text{word-cat-bl} \ [\mathbf{where} \ a=\text{numeral } a \ \mathbf{and} \ b=\text{numeral } b,$$

$$\text{unfolded to-bl-numeral}]$$

**for**  $a \ b$

**lemmas** *word-split-bl-no-bin* [*simp*] =
$$\text{word-split-bl-eq} \ [\mathbf{where} \ c=\text{numeral } c, \ \text{unfolded to-bl-numeral}] \ \mathbf{for} \ c$$

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

**lemma** *word-rsplit-same*:  $\text{word-rsplit } w = [w]$ *<proof>***lemma** *word-rsplit-empty-iff-size*:
$$(\text{word-rsplit } w = []) = (\text{size } w = 0)$$

*<proof>*

**lemma** *test-bit-rsplit*:
$$sw = \text{word-rsplit } w \implies m < \text{size } (\text{hd } sw :: 'a :: \text{len } \text{word}) \implies$$

$$k < \text{length } sw \implies (\text{rev } sw ! k) !! m = (w !! (k * \text{size } (\text{hd } sw) + m))$$

*<proof>*

**lemma** *word-rcat-bl*:  $\text{word-rcat } wl = \text{of-bl } (\text{concat } (\text{map } \text{to-bl } wl))$ *<proof>***lemma** *size-rcat-lem'*:
$$\text{size } (\text{concat } (\text{map } \text{to-bl } wl)) = \text{length } wl * \text{size } (\text{hd } wl)$$

$\langle \text{proof} \rangle$

**lemmas** *size-rcat-lem* = *size-rcat-lem'* [*unfolded word-size*]

**lemmas** *td-gal-lt-len* = *len-gt-0* [*THEN td-gal-lt*]

**lemma** *nth-rcat-lem*:

$n < \text{length } (wl :: 'a \text{ word list}) * \text{len-of TYPE}('a :: \text{len}) \implies$   
 $\text{rev } (\text{concat } (\text{map } \text{to-bl } wl)) ! n =$   
 $\text{rev } (\text{to-bl } (\text{rev } wl ! (n \text{ div } \text{len-of TYPE}('a)))) ! (n \text{ mod } \text{len-of TYPE}('a))$   
 $\langle \text{proof} \rangle$

**lemma** *test-bit-rcat*:

$sw = \text{size } (hd \text{ } wl :: 'a :: \text{len word}) \implies rc = \text{word-rcat } wl \implies rc !! n =$   
 $(n < \text{size } rc \ \& \ n \text{ div } sw < \text{size } wl \ \& \ (\text{rev } wl) ! (n \text{ div } sw) !! (n \text{ mod } sw))$   
 $\langle \text{proof} \rangle$

**lemma** *foldl-eq-foldr*:

$\text{foldl } op + x \ xs = \text{foldr } op + (x \ \# \ xs) \ (0 :: 'a :: \text{comm-monoid-add})$   
 $\langle \text{proof} \rangle$

**lemmas** *test-bit-cong* = *arg-cong* [**where**  $f = \text{test-bit}$ , *THEN fun-cong*]

**lemmas** *test-bit-rsplit-alt* =

*trans* [*OF nth-rev-alt* [*THEN test-bit-cong*]  
*test-bit-rsplit* [*OF refl asm-rl diff-Suc-less*]]

— lazy way of expressing that  $u$  and  $v$ , and  $su$  and  $sv$ , have same types

**lemma** *word-rsplit-len-indep* [*OF refl refl refl refl*]:

$[u, v] = p \implies [su, sv] = q \implies \text{word-rsplit } u = su \implies$   
 $\text{word-rsplit } v = sv \implies \text{length } su = \text{length } sv$   
 $\langle \text{proof} \rangle$

**lemma** *length-word-rsplit-size*:

$n = \text{len-of TYPE} ('a :: \text{len}) \implies$   
 $(\text{length } (\text{word-rsplit } w :: 'a \text{ word list}) <= m) = (\text{size } w <= m * n)$   
 $\langle \text{proof} \rangle$

**lemmas** *length-word-rsplit-lt-size* =

*length-word-rsplit-size* [*unfolded Not-eq-iff linorder-not-less* [*symmetric*]]

**lemma** *length-word-rsplit-exp-size*:

$n = \text{len-of TYPE} ('a :: \text{len}) \implies$   
 $\text{length } (\text{word-rsplit } w :: 'a \text{ word list}) = (\text{size } w + n - 1) \text{ div } n$   
 $\langle \text{proof} \rangle$

**lemma** *length-word-rsplit-even-size*:

$n = \text{len-of TYPE} ('a :: \text{len}) \implies \text{size } w = m * n \implies$   
 $\text{length } (\text{word-rsplit } w :: 'a \text{ word list}) = m$

*<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::'b::len0 word);*

*size frcw = length ws \* len-of TYPE('a)]]*

*==> length (word-rsplit frcw::'a word list) = length ws*

*<proof>*

**lemma** *msrevs:*

**fixes** *n::nat*

**shows** *0 < n ==> (k \* n + m) div n = m div n + k*

**and** *(k \* n + m) mod n = m mod n*

*<proof>*

**lemma** *word-rsplit-rcat-size [OF refl]:*

*word-rcat (ws :: 'a :: len word list) = frcw ==>*

*size frcw = length ws \* len-of TYPE ('a) ==> word-rsplit frcw = ws*

*<proof>*

## 13.27 Rotation

**lemmas** *rotater-0' [simp] = rotater-def [where n = 0, simplified]*

**lemmas** *word-rot-defs = word-roti-def word-rotr-def word-rotl-def*

**lemma** *rotate-eq-mod:*

*m mod length xs = n mod length xs ==> rotate m xs = rotate n xs*

*<proof>*

**lemmas** *rotate-eqs =*

*trans [OF rotate0 [THEN fun-cong] id-apply]*

*rotate-rotate [symmetric]*

*rotate-id*

*rotate-conv-mod*

*rotate-eq-mod*

### 13.27.1 Rotation of list to right

**lemma** *rotate1-rl': rotater1 (l @ [a]) = a # l*

*<proof>*

**lemma** *rotate1-rl* [*simp*] : *rotater1* (*rotate1* *l*) = *l*  
 ⟨*proof*⟩

**lemma** *rotate1-lr* [*simp*] : *rotate1* (*rotater1* *l*) = *l*  
 ⟨*proof*⟩

**lemma** *rotater1-rev'*: *rotater1* (*rev* *xs*) = *rev* (*rotate1* *xs*)  
 ⟨*proof*⟩

**lemma** *rotater-rev'*: *rotater* *n* (*rev* *xs*) = *rev* (*rotate* *n* *xs*)  
 ⟨*proof*⟩

**lemma** *rotater-rev*: *rotater* *n* *ys* = *rev* (*rotate* *n* (*rev* *ys*))  
 ⟨*proof*⟩

**lemma** *rotater-drop-take*:  
*rotater* *n* *xs* =  
   *drop* (*length* *xs* - *n* *mod* *length* *xs*) *xs* @  
   *take* (*length* *xs* - *n* *mod* *length* *xs*) *xs*  
 ⟨*proof*⟩

**lemma** *rotater-Suc* [*simp*] :  
*rotater* (*Suc* *n*) *xs* = *rotater1* (*rotater* *n* *xs*)  
 ⟨*proof*⟩

**lemma** *rotate-inv-plus* [*rule-format*] :  
 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*]

**lemmas** *rotate-rl* [*simp*] = *rotate-inv-eq* [*THEN* *conjunct2*, *THEN* *conjunct1*]

**lemma** *rotate-gal*: (*rotater* *n* *xs* = *ys*) = (*rotate* *n* *ys* = *xs*)  
 ⟨*proof*⟩

**lemma** *rotate-gal'*: (*ys* = *rotater* *n* *xs*) = (*xs* = *rotate* *n* *ys*)  
 ⟨*proof*⟩

**lemma** *length-rotater* [*simp*]:  
*length* (*rotater* *n* *xs*) = *length* *xs*  
 ⟨*proof*⟩

**lemma** *restrict-to-left*:

**assumes**  $x = y$

**shows**  $(x = z) = (y = z)$

*<proof>*

**lemmas**  $rrs0 = rotate\text{-}eqs$  [THEN *restrict-to-left*,

*simplified rotate-gal* [symmetric] *rotate-gal'* [symmetric]]

**lemmas**  $rrs1 = rrs0$  [THEN *refl* [THEN *rev-iffD1*]]

**lemmas**  $rotater\text{-}eqs = rrs1$  [*simplified length-rotater*]

**lemmas**  $rotater\text{-}0 = rotater\text{-}eqs$  (1)

**lemmas**  $rotater\text{-}add = rotater\text{-}eqs$  (2)

### 13.27.2 map, map2, commuting with rotate(r)

**lemma** *last-map*:  $xs \sim = [] \implies last (map f xs) = f (last xs)$

*<proof>*

**lemma** *butlast-map*:

$xs \sim = [] \implies butlast (map f xs) = map f (butlast xs)$

*<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  $ys$ .  $length\ xs = length\ ys \implies xs \sim = [] \implies$

$last (zip\ xs\ ys) = (last\ xs, last\ ys) \ \&$

$butlast (zip\ xs\ ys) = zip (butlast\ xs) (butlast\ ys)$

*<proof>*

**lemma** *but-last-map2* [rule-format] :

ALL  $ys$ .  $length\ xs = length\ ys \implies xs \sim = [] \implies$

$last (map2\ f\ xs\ ys) = f (last\ xs) (last\ ys) \ \&$

$butlast (map2\ f\ xs\ ys) = map2\ f (butlast\ xs) (butlast\ ys)$

*<proof>*

**lemma** *rotater1-zip*:

$length\ xs = length\ ys \implies$

$rotater1 (zip\ xs\ ys) = zip (rotater1\ xs) (rotater1\ ys)$

*<proof>*

**lemma** *rotater1-map2*:

$length\ xs = length\ ys \implies$

$rotater1 (map2\ f\ xs\ ys) = map2\ f (rotater1\ xs) (rotater1\ ys)$

*<proof>*

**lemmas** *lrth* =  
*box-equals* [*OF asm-rl length-rotater* [*symmetric*]  
*length-rotater* [*symmetric*],  
*THEN rotater1-map2*]

**lemma** *rotater-map2*:  
*length xs = length ys*  $\implies$   
*rotater n* (*map2 f xs ys*) = *map2 f* (*rotater n xs*) (*rotater n ys*)  
*<proof>*

**lemma** *rotate1-map2*:  
*length xs = length ys*  $\implies$   
*rotate1* (*map2 f xs ys*) = *map2 f* (*rotate1 xs*) (*rotate1 ys*)  
*<proof>*

**lemmas** *lth* = *box-equals* [*OF asm-rl length-rotate* [*symmetric*]  
*length-rotate* [*symmetric*], *THEN rotate1-map2*]

**lemma** *rotate-map2*:  
*length xs = length ys*  $\implies$   
*rotate n* (*map2 f xs ys*) = *map2 f* (*rotate n xs*) (*rotate n ys*)  
*<proof>*

**lemma** *to-bl-rotl*:  
*to-bl* (*word-rotl n w*) = *rotate n* (*to-bl w*)  
*<proof>*

**lemmas** *blrs0* = *rotate-eqs* [*THEN to-bl-rotl* [*THEN trans*]]

**lemmas** *word-rotl-eqs* =  
*blrs0* [*simplified word-bl-Rep'* *word-bl.Rep-inject to-bl-rotl* [*symmetric*]]

**lemma** *to-bl-rotr*:  
*to-bl* (*word-rotr n w*) = *rotater n* (*to-bl w*)  
*<proof>*

**lemmas** *brrs0* = *rotater-eqs* [*THEN to-bl-rotr* [*THEN trans*]]

**lemmas** *word-rotr-eqs* =  
*brrs0* [*simplified word-bl-Rep'* *word-bl.Rep-inject to-bl-rotr* [*symmetric*]]

**declare** *word-rotr-eqs* (1) [*simp*]  
**declare** *word-rotl-eqs* (1) [*simp*]

**lemma**  
*word-rot-rl* [*simp*]:  
*word-rotl k* (*word-rotr k v*) = *v* **and**  
*word-rot-lr* [*simp*]:

*word-rotr*  $k$  (*word-rotl*  $k$   $v$ ) =  $v$   
 ⟨*proof*⟩

**lemma**

*word-rot-gal*:  
 (*word-rotr*  $n$   $v$  =  $w$ ) = (*word-rotl*  $n$   $w$  =  $v$ ) **and**  
*word-rot-gal'*:  
 ( $w$  = *word-rotr*  $n$   $v$ ) = ( $v$  = *word-rotl*  $n$   $w$ )  
 ⟨*proof*⟩

**lemma** *word-rotr-rev*:

*word-rotr*  $n$   $w$  = *word-reverse* (*word-rotl*  $n$  (*word-reverse*  $w$ ))  
 ⟨*proof*⟩

**lemma** *word-roti-0* [*simp*]: *word-roti*  $0$   $w$  =  $w$

⟨*proof*⟩

**lemmas** *abl-cong* = *arg-cong* [**where**  $f$  = *of-bl*]

**lemma** *word-roti-add*:

*word-roti* ( $m + n$ )  $w$  = *word-roti*  $m$  (*word-roti*  $n$   $w$ )  
 ⟨*proof*⟩

**lemma** *word-roti-conv-mod'*: *word-roti*  $n$   $w$  = *word-roti* ( $n$  *mod* *int* (*size*  $w$ ))  $w$

⟨*proof*⟩

**lemmas** *word-roti-conv-mod* = *word-roti-conv-mod'* [*unfolded word-size*]

### 13.27.3 Word rotation commutes with bit-wise operations

**locale** *word-rotate*

**begin**

**lemmas** *word-rot-defs'* = *to-bl-rotl to-bl-rotr*

**lemmas** *blwl-syms* [*symmetric*] = *bl-word-not bl-word-and bl-word-or bl-word-xor*

**lemmas** *lbl-lbl* = *trans* [*OF* *word-bl-Rep'* *word-bl-Rep'* [*symmetric*]]

**lemmas** *ths-map2* [*OF* *lbl-lbl*] = *rotate-map2 rotater-map2*

**lemmas** *ths-map* [**where**  $xs$  = *to-bl*  $v$ ] = *rotate-map rotater-map* **for**  $v$

**lemmas** *th1s* [*simplified word-rot-defs'* [*symmetric*]] = *ths-map2 ths-map*

**lemma** *word-rot-logs*:

*word-rotl*  $n$  (*NOT*  $v$ ) = *NOT* *word-rotl*  $n$   $v$

*word-rotr*  $n$  (*NOT*  $v$ ) = *NOT* *word-rotr*  $n$   $v$

*word-rotl*  $n$  ( $x$  *AND*  $y$ ) = *word-rotl*  $n$   $x$  *AND* *word-rotl*  $n$   $y$

```

word-rotr n (x AND y) = word-rotr n x AND word-rotr n y
word-rotl n (x OR y) = word-rotl n x OR word-rotl n y
word-rotr n (x OR y) = word-rotr n x OR word-rotr n y
word-rotl n (x XOR y) = word-rotl n x XOR word-rotl n y
word-rotr n (x XOR y) = word-rotr n x XOR word-rotr n y
⟨proof⟩

```

**end**

**lemmas** *word-rot-logs* = *word-rotate.word-rot-logs*

**lemmas** *bl-word-rotl-dt* = *trans* [*OF to-bl-rotl rotate-drop-take*,  
*simplified word-bl-Rep*']

**lemmas** *bl-word-rotr-dt* = *trans* [*OF to-bl-rotr rotater-drop-take*,  
*simplified word-bl-Rep*']

**lemma** *bl-word-roti-dt'*:

```

n = nat ((- i) mod int (size (w :: 'a :: len word))) ==>
  to-bl (word-roti i w) = drop n (to-bl w) @ take n (to-bl w)
⟨proof⟩

```

**lemmas** *bl-word-roti-dt* = *bl-word-roti-dt'* [*unfolded word-size*]

**lemmas** *word-rotl-dt* = *bl-word-rotl-dt* [*THEN word-bl.Rep-inverse'* [*symmetric*]]

**lemmas** *word-rotr-dt* = *bl-word-rotr-dt* [*THEN word-bl.Rep-inverse'* [*symmetric*]]

**lemmas** *word-roti-dt* = *bl-word-roti-dt* [*THEN word-bl.Rep-inverse'* [*symmetric*]]

**lemma** *word-rotx-0* [*simp*] : *word-rotr i 0 = 0* & *word-rotl i 0 = 0*  
⟨*proof*⟩

**lemma** *word-roti-0'* [*simp*] : *word-roti n 0 = 0*  
⟨*proof*⟩

**lemmas** *word-rotr-dt-no-bin'* [*simp*] =  
*word-rotr-dt* [**where** *w=numeral w*, *unfolded to-bl-numeral*] **for** *w*

**lemmas** *word-rotl-dt-no-bin'* [*simp*] =  
*word-rotl-dt* [**where** *w=numeral w*, *unfolded to-bl-numeral*] **for** *w*

**declare** *word-roti-def* [*simp*]

### 13.28 Maximum machine word

**lemma** *word-int-cases*:

```

obtains n where (x :: 'a :: len0 word) = word-of-int n and 0 ≤ n and n <
2^len-of TYPE('a)
⟨proof⟩

```

**lemma** *word-nat-cases* [*cases type: word*]:

**obtains**  $n$  **where**  $(x :: 'a::len\ word) = of\_nat\ n$  **and**  $n < 2^{len-of\ TYPE('a)}$   
 ⟨*proof*⟩

**lemma** *max-word-eq*:  $(max\_word :: 'a::len\ word) = 2^{len-of\ TYPE('a)} - 1$

⟨*proof*⟩

**lemma** *max-word-max* [*simp,intro!*]:  $n \leq max\_word$

⟨*proof*⟩

**lemma** *word-of-int-2p-len*:  $word-of-int\ (2^{len-of\ TYPE('a)}) = (0 :: 'a::len0\ word)$

⟨*proof*⟩

**lemma** *word-pow-0*:

$(2 :: 'a::len\ word)^{len-of\ TYPE('a)} = 0$

⟨*proof*⟩

**lemma** *max-word-wrap*:  $x + 1 = 0 \implies x = max\_word$

⟨*proof*⟩

**lemma** *max-word-minus*:

$max\_word = (-1 :: 'a::len\ word)$

⟨*proof*⟩

**lemma** *max-word-bl* [*simp*]:

$to\_bl\ (max\_word :: 'a::len\ word) = replicate\ (len-of\ TYPE('a))\ True$

⟨*proof*⟩

**lemma** *max-test-bit* [*simp*]:

$(max\_word :: 'a::len\ word) !! n = (n < len-of\ TYPE('a))$

⟨*proof*⟩

**lemma** *word-and-max* [*simp*]:

$x\ AND\ max\_word = x$

⟨*proof*⟩

**lemma** *word-or-max* [*simp*]:

$x\ OR\ max\_word = max\_word$

⟨*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*]:

$x \text{ AND } \text{NOT } x = (0::'a::\text{len } 0 \text{ word})$

$\langle \text{proof} \rangle$

**lemma** *word-or-not* [*simp*]:

$x \text{ OR } \text{NOT } x = \text{max-word}$

$\langle \text{proof} \rangle$

**lemma** *word-boolean*:

$\text{boolean } (\text{op } \text{AND}) (\text{op } \text{OR}) \text{ bitNOT } 0 \text{ max-word}$

$\langle \text{proof} \rangle$

**interpretation** *word-bool-alg*:

$\text{boolean } \text{op } \text{AND } \text{op } \text{OR } \text{bitNOT } 0 \text{ max-word}$

$\langle \text{proof} \rangle$

**lemma** *word-xor-and-or*:

$x \text{ XOR } y = x \text{ AND } \text{NOT } y \text{ OR } \text{NOT } x \text{ AND } (y::'a::\text{len } 0 \text{ word})$

$\langle \text{proof} \rangle$

**interpretation** *word-bool-alg*:

$\text{boolean-xor } \text{op } \text{AND } \text{op } \text{OR } \text{bitNOT } 0 \text{ max-word } \text{op } \text{XOR}$

$\langle \text{proof} \rangle$

**lemma** *shiftr-x-0* [*iff*]:

$(x::'a::\text{len } 0 \text{ word}) \gg 0 = x$

$\langle \text{proof} \rangle$

**lemma** *shiftr-x-0* [*simp*]:

$(x :: 'a :: \text{len } \text{word}) \ll 0 = x$

$\langle \text{proof} \rangle$

**lemma** *shiftr-1* [*simp*]:

$(1::'a::\text{len } \text{word}) \ll n = 2^n$

$\langle \text{proof} \rangle$

**lemma** *uint-lt-0* [*simp*]:

$\text{uint } x < 0 = \text{False}$

$\langle \text{proof} \rangle$

**lemma** *shiftr1-1* [*simp*]:

$\text{shiftr1 } (1::'a::\text{len } \text{word}) = 0$

$\langle \text{proof} \rangle$

**lemma** *shiftr-1* [*simp*]:

$(1::'a::\text{len } \text{word}) \gg n = (\text{if } n = 0 \text{ then } 1 \text{ else } 0)$

$\langle \text{proof} \rangle$

**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*:

$n = length\ xs \implies$   
 $map\ (\lambda(x,y).\ x\ \&\ y)\ (zip\ xs\ (replicate\ n\ True)) = xs$   
 ⟨proof⟩

**lemma** *map-replicate-False*:

$n = length\ xs \implies map\ (\lambda(x,y).\ x\ \&\ y)$   
 $(zip\ xs\ (replicate\ n\ False)) = replicate\ n\ False$   
 ⟨proof⟩

**lemma** *bl-and-mask*:

**fixes**  $w :: 'a::len\ word$   
**fixes**  $n$   
**defines**  $n' \equiv len-of\ TYPE('a) - n$   
**shows**  $to-bl\ (w\ AND\ mask\ n) = replicate\ n'\ False\ @\ drop\ n'\ (to-bl\ w)$   
 ⟨proof⟩

**lemma** *drop-rev-takefill*:

$length\ xs \leq n \implies$   
 $drop\ (n - length\ xs)\ (rev\ (takefill\ False\ n\ (rev\ xs))) = xs$   
 ⟨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  
 $unat\ x + unat\ y - 2^{size\ x})$

*<proof>*

**lemma** *word-neq-0-conv*:

**fixes**  $w :: 'a :: \text{len word}$

**shows**  $(w \neq 0) = (0 < w)$

*<proof>*

**lemma** *max-lt*:

$\text{unat } (\max a b \text{ div } c) = \text{unat } (\max a b) \text{ div } \text{unat } (c :: 'a :: \text{len word})$

*<proof>*

**lemma** *uint-sub-if-size*:

$\text{uint } (x - y) =$

*(if*  $\text{uint } y \leq \text{uint } x$  *then*

$\text{uint } x - \text{uint } y$

*else*

$\text{uint } x - \text{uint } y + 2^{\text{size } x}$ )

*<proof>*

**lemma** *unat-sub*:

$b \leq a \implies \text{unat } (a - b) = \text{unat } a - \text{unat } b$

*<proof>*

**lemmas** *word-less-sub1-numberof* [simp] = *word-less-sub1* [of numeral  $w$ ] **for**  $w$

**lemmas** *word-le-sub1-numberof* [simp] = *word-le-sub1* [of numeral  $w$ ] **for**  $w$

**lemma** *word-of-int-minus*:

$\text{word-of-int } (2^{\text{len-of TYPE('a)}} - i) = (\text{word-of-int } (-i) :: 'a :: \text{len word})$

*<proof>*

**lemmas** *word-of-int-inj* =

*word-uint.Abs-inject* [unfolded *uints-num*, *simplified*]

**lemma** *word-le-less-eq*:

$(x :: 'z :: \text{len word}) \leq y = (x = y \vee x < y)$

*<proof>*

**lemma** *mod-plus-cong*:

**assumes** 1:  $(b :: \text{int}) = b'$

**and** 2:  $x \bmod b' = x' \bmod b'$

**and** 3:  $y \bmod b' = y' \bmod b'$

**and** 4:  $x' + y' = z'$

**shows**  $(x + y) \bmod b = z' \bmod b'$

*<proof>*

**lemma** *mod-minus-cong*:

**assumes** 1:  $(b :: \text{int}) = b'$

**and** 2:  $x \bmod b' = x' \bmod b'$

**and** 3:  $y \bmod b' = y' \bmod b'$

**and**  $4: x' - y' = z'$   
**shows**  $(x - y) \bmod b = z' \bmod b'$   
 $\langle \text{proof} \rangle$

**lemma** *word-induct-less*:

$\llbracket P (0::'a::\text{len word}); \bigwedge n. \llbracket n < m; P n \rrbracket \implies P (1 + n) \rrbracket \implies P m$   
 $\langle \text{proof} \rangle$

**lemma** *word-induct*:

$\llbracket P (0::'a::\text{len word}); \bigwedge n. P n \implies P (1 + n) \rrbracket \implies P m$   
 $\langle \text{proof} \rangle$

**lemma** *word-induct2* [*induct type*]:

$\llbracket P 0; \bigwedge n. \llbracket 1 + n \neq 0; P n \rrbracket \implies P (1 + n) \rrbracket \implies P (n::'b::\text{len word})$   
 $\langle \text{proof} \rangle$

### 13.29 Recursion combinator for words

**definition** *word-rec* ::  $'a \Rightarrow ('b::\text{len word} \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b \text{ word} \Rightarrow 'a$  **where**  
*word-rec* *forZero* *forSuc*  $n = \text{nat-rec } \text{forZero} (\text{forSuc} \circ \text{of-nat}) (\text{unat } n)$

**lemma** *word-rec-0*:  $\text{word-rec } z \ s \ 0 = z$

$\langle \text{proof} \rangle$

**lemma** *word-rec-Suc*:

$1 + n \neq (0::'a::\text{len word}) \implies \text{word-rec } z \ s \ (1 + n) = s \ n \ (\text{word-rec } z \ s \ n)$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-Pred*:

$n \neq 0 \implies \text{word-rec } z \ s \ n = s \ (n - 1) \ (\text{word-rec } z \ s \ (n - 1))$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-in*:

$f \ (\text{word-rec } z \ (\lambda-. f) \ n) = \text{word-rec } (f \ z) \ (\lambda-. f) \ n$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-in2*:

$f \ n \ (\text{word-rec } z \ f \ n) = \text{word-rec } (f \ 0 \ z) \ (f \circ \text{op} + 1) \ n$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-twice*:

$m \leq n \implies \text{word-rec } z \ f \ n = \text{word-rec } (\text{word-rec } z \ f \ (n - m)) \ (f \circ \text{op} + (n - m)) \ m$   
 $\langle \text{proof} \rangle$

**lemma** *word-rec-id*:  $\text{word-rec } z \ (\lambda-. \text{id}) \ n = z$

$\langle \text{proof} \rangle$

**lemma** *word-rec-id-eq*:  $\forall m < n. f \ m = \text{id} \implies \text{word-rec } z \ f \ n = z$

*<proof>*

**lemma** *word-rec-max*:

$\forall m \geq n. m \neq -1 \rightarrow f m = id \implies \text{word-rec } z f -1 = \text{word-rec } z f n$   
*<proof>*

**lemma** *unatSuc*:

$1 + n \neq (0::'a::len \text{ word}) \implies \text{unat } (1 + n) = \text{Suc } (\text{unat } n)$   
*<proof>*

**lemma** *word-no-1* [simp]:  $(\text{Numeral1}::'a::len 0 \text{ word}) = 1$

*<proof>*

**declare** *bin-to-bl-def* [simp]

*<ML>*

**hide-const** (open) *Word*

**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.