

Machine Words in Isabelle/HOL

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

May 22, 2012

Abstract

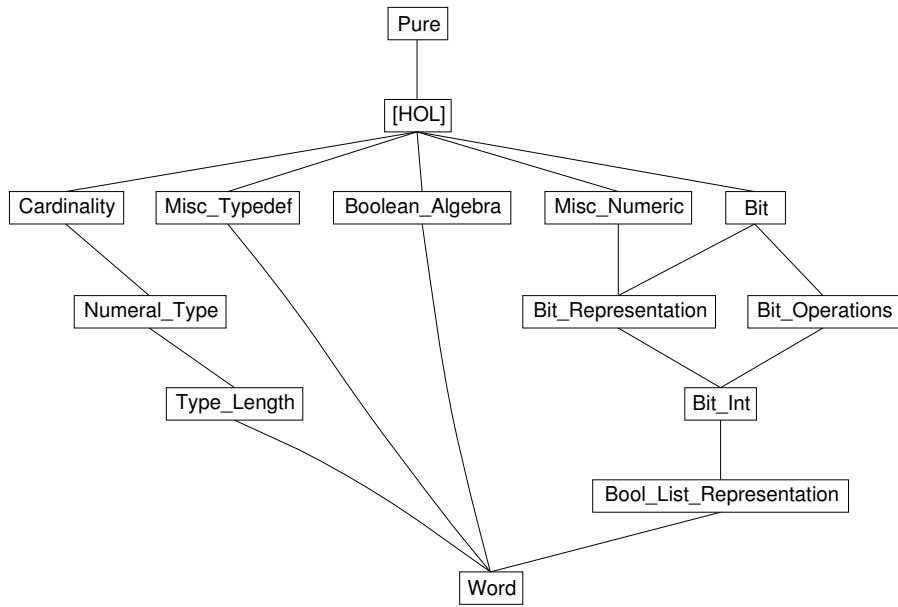
A formalisation of generic, fixed size machine words in Isabelle/HOL.
An earlier version of this formalisation is described in [1].

Contents

1	Cardinality: Cardinality of types	5
1.1	Preliminary lemmas	5
1.2	Cardinalities of types	5
1.3	Classes with at least 1 and 2	6
2	Numeral-Type: Numeral Syntax for Types	6
2.1	Numeral Types	6
2.2	Locales for modular arithmetic subtypes	8
2.3	Ring class instances	10
2.4	Syntax	12
2.5	Examples	13
3	Type-Length: Assigning lengths to types by typeclasses	13
4	Misc-Typedef: Type Definition Theorems	14
5	More lemmas about normal type definitions	14
5.1	Extended form of type definition predicate	16
6	Boolean-Algebra: Boolean Algebras	19
6.1	Complement	20
6.2	Conjunction	20
6.3	Disjunction	21
6.4	De Morgan's Laws	22
6.5	Symmetric Difference	22
7	Misc-Numeric: Useful Numerical Lemmas	24

8	Bit: The Field of Integers mod 2	31
8.1	Bits as a datatype	31
8.2	Type <i>bit</i> forms a field	32
8.3	Numerals at type <i>bit</i>	33
9	Bit-Representation: Basic Definitions for Binary Integers	33
9.1	Further properties of numerals	34
9.2	Destructors for binary integers	36
9.3	Truncating binary integers	38
9.4	Simplifications for (s)bintrunc	39
9.5	Splitting and concatenation	48
9.6	Miscellaneous lemmas	49
10	Bit-Operations: Syntactic classes for bitwise operations	50
10.1	Bitwise operations on <i>bit</i>	50
11	Bit-Int: Bitwise Operations on Binary Integers	52
11.1	Logical operations	52
11.1.1	Basic simplification rules	52
11.1.2	Binary destructors	54
11.1.3	Derived properties	54
11.1.4	Simplification with numerals	56
11.1.5	Interactions with arithmetic	59
11.1.6	Truncating results of bit-wise operations	60
11.2	Setting and clearing bits	61
11.3	Splitting and concatenation	63
11.4	Miscellaneous lemmas	65
12	Bool-List-Representation: Bool lists and integers	66
12.1	Operations on lists of booleans	66
12.2	Arithmetic in terms of bool lists	67
12.3	Repeated splitting or concatenation	85
13	Word: A type of finite bit strings	90
13.1	Type definition	90
13.2	Basic code generation setup	91
13.3	Type conversions and casting	92
13.4	Type-definition locale instantiations	93
13.5	Correspondence relation for theorem transfer	95
13.6	Arithmetic operations	95
13.7	Ordering	97
13.8	Bit-wise operations	97
13.9	Shift operations	99
13.10	Rotation	99

13.11	Split and cat operations	100
13.12	Theorems about typedefs	100
13.13	Testing bits	105
13.14	Word Arithmetic	113
13.15	Transferring goals from words to ints	115
13.16	Order on fixed-length words	116
13.17	Conditions for the addition (etc) of two words to overflow	118
13.18	Definition of uint_arith	118
13.19	More on overflows and monotonicity	120
13.20	Arithmetic type class instantiations	125
13.21	Word and nat	125
13.22	Definition of unat_arith tactic	129
13.23	Cardinality, finiteness of set of words	133
13.24	Bitwise Operations on Words	133
13.25	Shifting, Rotating, and Splitting Words	144
13.25.1	shift functions in terms of lists of bools	147
13.25.2	Mask	153
13.25.3	Revcast	156
13.25.4	Slices	158
13.26	Split and cat	161
13.26.1	Split and slice	164
13.27	Rotation	168
13.27.1	Rotation of list to right	169
13.27.2	map, map2, commuting with rotate(r)	170
13.27.3	Word rotation commutes with bit-wise operations	174
13.28	Maximum machine word	176
13.29	Recursion combinator for words	182



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
  show Abs ' A  $\subseteq$  UNIV by (rule subset-UNIV)
  show UNIV  $\subseteq$  Abs ' A
  proof
    fix x :: 'b
    have x = Abs (Rep x) by (rule Rep-inverse [symmetric])
    moreover have Rep x  $\in$  A by (rule Rep)
    ultimately show x  $\in$  Abs ' A by (rule image-eqI)
  qed
qed
```

```
lemma (in type-definition) card: card (UNIV :: 'b set) = card A
  by (simp add: univ card-image inj-on-def Abs-inject)
```

1.2 Cardinalities of types

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

translations CARD('t) => CONST card (CONST UNIV :: 't set)

typed-print-translation (advanced) <<
  let
    fun card-univ-tr' ctxt - [Const (@{const-syntax UNIV}, Type (-, [T, -]))] =
      Syntax.const @ {syntax-const -type-card} $ Syntax-Phases.term-of-typ ctxt T;
    in [(@{const-syntax card}, card-univ-tr')] end
  >>

lemma card-unit [simp]: CARD(unit) = 1
  unfolding UNIV-unit by simp

lemma card-prod [simp]: CARD('a  $\times$  'b) = CARD('a::finite) * CARD('b::finite)
  unfolding UNIV-Times-UNIV [symmetric] by (simp only: card-cartesian-product)

lemma card-sum [simp]: CARD('a + 'b) = CARD('a::finite) + CARD('b::finite)
  unfolding UNIV-Plus-UNIV [symmetric] by (simp only: finite card-Plus)

lemma card-option [simp]: CARD('a option) = Suc CARD('a::finite)
  unfolding UNIV-option-conv
  apply (subgoal-tac (None::'a option)  $\notin$  range Some)
  apply (simp add: card-image)
```

apply *fast*
done

lemma *card-set* [*simp*]: $CARD('a \text{ set}) = 2 \wedge CARD('a::\text{finite})$
unfolding *Pow-UNIV* [*symmetric*]
by (*simp only: card-Pow finite*)

lemma *card-nat* [*simp*]: $CARD(\text{nat}) = 0$
by (*simp add: card-eq-0-iff*)

1.3 Classes with at least 1 and 2

Class *finite* already captures ”at least 1”

lemma *zero-less-card-finite* [*simp*]: $0 < CARD('a::\text{finite})$
unfolding *neq0-conv* [*symmetric*] **by** *simp*

lemma *one-le-card-finite* [*simp*]: $Suc\ 0 \leq CARD('a::\text{finite})$
by (*simp add: less-Suc-eq-le* [*symmetric*])

Class for cardinality ”at least 2”

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

lemma *one-less-card*: $Suc\ 0 < CARD('a::\text{card2})$
using *two-le-card* [**where** $'a='a$] **by** *simp*

lemma *one-less-int-card*: $1 < \text{int } CARD('a::\text{card2})$
using *one-less-card* [**where** $'a='a$] **by** *simp*

end

2 Numeral-Type: Numeral Syntax for Types

theory *Numeral-Type*
imports *Cardinality*
begin

2.1 Numeral Types

typedef (**open**) *num0* = *UNIV* :: *nat set* ..
typedef (**open**) *num1* = *UNIV* :: *unit set* ..

typedef (**open**) $'a \text{ bit0} = \{0 \dots < 2 * \text{int } CARD('a::\text{finite})\}$
proof
show $0 \in \{0 \dots < 2 * \text{int } CARD('a)\}$
by *simp*
qed

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

```
proof
```

```
  show 0 ∈ {0 ..< 1 + 2 * int CARD('a)}
```

```
    by simp
```

```
qed
```

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

```
  unfolding type-definition.card [OF type-definition-num0]
```

```
  by simp
```

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

```
  unfolding type-definition.card [OF type-definition-num1]
```

```
  by (simp only: card-unit)
```

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

```
  unfolding type-definition.card [OF type-definition-bit0]
```

```
  by simp
```

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

```
  unfolding type-definition.card [OF type-definition-bit1]
```

```
  by simp
```

```
instance num1 :: finite
```

```
proof
```

```
  show finite (UNIV::num1 set)
```

```
    unfolding type-definition.univ [OF type-definition-num1]
```

```
    using finite by (rule finite-imageI)
```

```
qed
```

```
instance bit0 :: (finite) card2
```

```
proof
```

```
  show finite (UNIV::'a bit0 set)
```

```
    unfolding type-definition.univ [OF type-definition-bit0]
```

```
    by simp
```

```
  show 2 ≤ CARD('a bit0)
```

```
    by simp
```

```
qed
```

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

```
proof
```

```
  show finite (UNIV::'a bit1 set)
```

```
    unfolding type-definition.univ [OF type-definition-bit1]
```

```
    by simp
```

```
  show 2 ≤ CARD('a bit1)
```

```
    by simp
```

```
qed
```

2.2 Locales for for modular arithmetic subtypes

```

locale mod-type =
  fixes n :: int
  and Rep :: 'a::{zero,one,plus,times,uminus,minus} ⇒ int
  and Abs :: int ⇒ '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
using size1 by simp

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

lemma Rep-less-n: Rep x < n
by (rule type-definition.Rep [OF type, simplified, THEN conjunct2])

lemma Rep-le-n: Rep x ≤ n
by (rule Rep-less-n [THEN order-less-imp-le])

lemma Rep-inject-sym: x = y ⟷ Rep x = Rep y
by (rule type-definition.Rep-inject [OF type, symmetric])

lemma Rep-inverse: Abs (Rep x) = x
by (rule type-definition.Rep-inverse [OF type])

lemma Abs-inverse: m ∈ {0.. $n$ } ⟹ Rep (Abs m) = m
by (rule type-definition.Abs-inverse [OF type])

lemma Rep-Abs-mod: Rep (Abs (m mod n)) = m mod n
by (simp add: Abs-inverse pos-mod-conj [OF size0])

lemma Rep-Abs-0: Rep (Abs 0) = 0
by (simp add: Abs-inverse size0)

lemma Rep-0: Rep 0 = 0
by (simp add: zero-def Rep-Abs-0)

lemma Rep-Abs-1: Rep (Abs 1) = 1
by (simp add: Abs-inverse size1)

lemma Rep-1: Rep 1 = 1

```

by (*simp add: one-def Rep-Abs-1*)

lemma *Rep-mod*: $Rep\ x\ mod\ n = Rep\ x$
apply (*rule-tac x=x in type-definition.Abs-cases [OF type]*)
apply (*simp add: type-definition.Abs-inverse [OF type]*)
apply (*simp add: mod-pos-pos-trivial*)
done

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)*
apply (*intro-classes, unfold definitions*)
apply (*simp-all add: Rep-simps zmod-simps field-simps*)
done

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*: $of-nat\ k = Abs\ (int\ k\ mod\ n)$
apply (*induct k*)
apply (*simp add: zero-def*)
apply (*simp add: Rep-simps add-def one-def zmod-simps add-ac*)
done

lemma *of-int-eq*: $of-int\ z = Abs\ (z\ mod\ n)$
apply (*cases z rule: int-diff-cases*)
apply (*simp add: Rep-simps of-nat-eq diff-def zmod-simps*)
done

lemma *Rep-numeral*:
 $Rep\ (numeral\ w) = numeral\ w\ mod\ n$
using *of-int-eq [of numeral w]*
by (*simp add: Rep-inject-sym Rep-Abs-mod*)

lemma *iszero-numeral*:
 $iszero\ (numeral\ w :: 'a) \longleftrightarrow numeral\ w\ mod\ n = 0$
by (*simp add: Rep-inject-sym Rep-numeral Rep-0 iszero-def*)

lemma *cases*:
assumes $1: \bigwedge z. \llbracket (x :: 'a) = of-int\ z; 0 \leq z; z < n \rrbracket \Longrightarrow P$
shows P
apply (*cases x rule: type-definition.Abs-cases [OF type]*)
apply (*rule-tac z=y in 1*)

apply (*simp-all add: of-int-eq mod-pos-pos-trivial*)
done

lemma *induct*:

$(\bigwedge z. \llbracket 0 \leq z; z < n \rrbracket \implies P (\text{of-int } z)) \implies P (x::'a)$
by (*cases x rule: cases*) *simp*

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*: $(x::\text{num1}) = (y::\text{num1}) \longleftrightarrow \text{True}$
by (*induct x, induct y*) *simp*

instance proof

qed (*simp-all add: num1-eq-iff*)

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 $x + y = \text{Abs-bit0}' (\text{Rep-bit0 } x + \text{Rep-bit0 } y)$

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

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

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

definition *0 = Abs-bit1 0*

definition *1 = Abs-bit1 1*

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

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

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

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

instance ..

end

interpretation bit0:

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

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

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

apply (*rule mod-type.intro*)

apply (*simp add: int-mult type-definition-bit0*)

apply (*rule one-less-int-card*)

apply (*rule zero-bit0-def*)

apply (*rule one-bit0-def*)

apply (*rule plus-bit0-def [unfolded Abs-bit0'-def]*)

apply (*rule times-bit0-def [unfolded Abs-bit0'-def]*)

apply (*rule minus-bit0-def [unfolded Abs-bit0'-def]*)

apply (*rule uminus-bit0-def [unfolded Abs-bit0'-def]*)

done

interpretation bit1:

mod-type int CARD('a::finite bit1)

Rep-bit1 :: 'a::finite bit1 \Rightarrow int

Abs-bit1 :: int \Rightarrow 'a::finite bit1

apply (*rule mod-type.intro*)

apply (*simp add: int-mult type-definition-bit1*)

apply (*rule one-less-int-card*)

apply (*rule zero-bit1-def*)

apply (*rule one-bit1-def*)

apply (*rule plus-bit1-def [unfolded Abs-bit1'-def]*)

apply (*rule times-bit1-def [unfolded Abs-bit1'-def]*)

apply (*rule minus-bit1-def [unfolded Abs-bit1'-def]*)

apply (*rule uminus-bit1-def [unfolded Abs-bit1'-def]*)

done

instance bit0 :: (finite) comm-ring-1

by (*rule bit0.comm-ring-1*)

instance bit1 :: (finite) comm-ring-1

by (*rule bit1.comm-ring-1*)

interpretation bit0:

mod-ring int CARD('a::finite bit0)

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

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

..

interpretation bit1:

mod-ring int CARD('a::finite bit1)

Rep-bit1 :: 'a::finite bit1 \Rightarrow int

Abs-bit1 :: int \Rightarrow 'a::finite bit1

..

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***parse-translation** <<*let**fun mk-bintype n =**let**fun mk-bit 0 = Syntax.const @{type-syntax bit0}**| mk-bit 1 = Syntax.const @{type-syntax bit1};**fun bin-of n =**if n = 1 then Syntax.const @{type-syntax num1}**else if n = 0 then Syntax.const @{type-syntax num0}**else if n = ~1 then raise TERM (negative type numeral, [])**else**let val (q, r) = Integer.div-mod n 2;**in mk-bit r \$ bin-of q end;**in bin-of n end;**fun numeral-tr [Free (str, -)] = mk-bintype (the (Int.fromString str))**| numeral-tr ts = raise TERM (numeral-tr, ts);**in [(@{syntax-const -NumeralType}, numeral-tr)] end;*

>>

print-translation <<*let*

```

fun int-of [] = 0
  | int-of (b :: bs) = b + 2 * int-of bs;

fun bin-of (Const (@{type-syntax num0}, -)) = []
  | bin-of (Const (@{type-syntax num1}, -)) = [1]
  | bin-of (Const (@{type-syntax bit0}, -) $ bs) = 0 :: bin-of bs
  | bin-of (Const (@{type-syntax bit1}, -) $ bs) = 1 :: bin-of bs
  | bin-of t = raise TERM (bin-of, [t]);

fun bit-tr' b [t] =
  let
    val rev-digs = b :: bin-of t handle TERM - => raise Match
    val i = int-of rev-digs;
    val num = string-of-int (abs i);
  in
    Syntax.const @{{syntax-const -NumeralType}} $ Syntax.free num
  end
  | bit-tr' b - = raise Match;
in [(@{type-syntax bit0}, bit-tr' 0), (@{type-syntax bit1}, bit-tr' 1)] end;
>>

```

2.5 Examples

```

lemma CARD(0) = 0 by simp
lemma CARD(17) = 17 by simp
lemma 8 * 11 ^ 3 - 6 = (2::5) by simp

```

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 ⇒ nat

```

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

```

class len = len0 +
  assumes len-gt-0 [iff]: 0 < 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 ..

end

instantiation *bit0* and *bit1* :: (*len0*) *len0*

begin

definition

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

definition

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

instance ..

end

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

instance *num1* :: *len* **proof** **qed** *simp*

instance *bit0* :: (*len*) *len* **proof** **qed** *simp*

instance *bit1* :: (*len0*) *len* **proof** **qed** *simp*

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* $\implies \forall y. y \in A \longrightarrow \text{Rep } (\text{Abs } y) = y$

by (*auto simp: type-definition-def*)

```

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

context type-definition
begin

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

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

lemma Abs-inverse':
   $r : A \implies Abs\ r = a \implies Rep\ a = r$ 
  by (safe elim!: Abs-inverse)

lemma Rep-comp-inverse:
   $Rep\ o\ f = g \implies Abs\ o\ g = f$ 
  using Rep-inverse by auto

lemma Rep-eqD [elim!]:  $Rep\ x = Rep\ y \implies x = y$ 
  by simp

lemma Rep-inverse':  $Rep\ a = r \implies Abs\ r = a$ 
  by (safe intro!: Rep-inverse)

lemma comp-Abs-inverse:
   $f\ o\ Abs = g \implies g\ o\ Rep = f$ 
  using Rep-inverse by auto

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

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

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

```

lemma *image*: $Abs \text{ ‘ } A = UNIV$
by (*auto intro!*: *image-eqI*)

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

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

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

lemma *fns4*:
 $Rep \ o \ fa \ o \ Abs = fr \ ==>$
 $Rep \ o \ fa = fr \ o \ Rep \ \& \ fa \ o \ Abs = Abs \ o \ fr$
by *auto*

end

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

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

5.1 Extended form of type definition predicate

lemma *td-conds*:
 $norm \ o \ norm = norm \ ==> \ (fr \ o \ norm = norm \ o \ fr) =$
 $(norm \ o \ fr \ o \ norm = fr \ o \ norm \ \& \ norm \ o \ fr \ o \ norm = norm \ o \ fr)$
apply *safe*
apply (*simp-all add*: *o-assoc* [*symmetric*])
apply (*simp-all add*: *o-assoc*)
done

lemma *fn-comm-power*:
 $fa \ o \ tr = tr \ o \ fr \ ==> \ fa \ \wedge \wedge \ n \ o \ tr = tr \ o \ fr \ \wedge \wedge \ n$
apply (*rule ext*)
apply (*induct n*)
apply (*auto dest*: *fun-cong*)
done

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

```

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

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

lemma td-th:
   $g \circ \text{Abs} = f \implies f (\text{Rep } x) = g x$ 
  by (drule comp-Abs-inverse [symmetric]) simp

lemma eq-norm':  $\text{Rep } \circ \text{Abs} = \text{norm}$ 
  by (auto simp: eq-norm)

lemma norm-Rep [simp]:  $\text{norm } (\text{Rep } x) = \text{Rep } x$ 
  by (auto simp: eq-norm' intro: td-th)

lemmas td = td-thm

lemma set-iff-norm:  $w : A \longleftrightarrow w = \text{norm } w$ 
  by (auto simp: set-Rep-Abs eq-norm' eq-norm [symmetric])

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

lemma norm-eq-iff:
   $(\text{norm } x = \text{norm } y) = (\text{Abs } x = \text{Abs } y)$ 
  by (simp add: eq-norm' [symmetric])

lemma norm-comps:
   $\text{Abs } \circ \text{norm} = \text{Abs}$ 
   $\text{norm } \circ \text{Rep} = \text{Rep}$ 
   $\text{norm } \circ \text{norm} = \text{norm}$ 
  by (auto simp: eq-norm' [symmetric] o-def)

lemmas norm-norm [simp] = norm-comps

lemma fns5:
   $\text{Rep } \circ \text{fa } \circ \text{Abs} = \text{fr} \implies$ 
   $\text{fr } \circ \text{norm} = \text{fr} \ \& \ \text{norm } \circ \text{fr} = \text{fr}$ 
  by (fold eq-norm') auto

```

```

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

```

```

lemma fns3:
  Abs o fr o Rep = fa ==>
  (norm o fr o norm = norm o fr) = (fa o Abs = Abs o fr)
  apply (fold eq-norm ^)
  apply safe
  prefer 2
  apply (simp add: o-assoc [symmetric])
  apply (rule ext)
  apply (drule fun-cong)
  apply simp
  done

```

```

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

```

```

lemma range-norm:
  range (Rep o Abs) = A
  by (simp add: set-Rep-Abs)

```

end

```

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

```

end

6 Boolean-Algebra: Boolean Algebras

```
theory Boolean-Algebra
imports Main
begin
```

```
locale boolean =
  fixes conj :: 'a ⇒ 'a ⇒ 'a (infixr  $\sqcap$  70)
  fixes disj :: 'a ⇒ 'a ⇒ 'a (infixr  $\sqcup$  65)
  fixes compl :: 'a ⇒ 'a ( $\sim$  - [81] 80)
  fixes zero :: 'a ( $\mathbf{0}$ )
  fixes one :: 'a ( $\mathbf{1}$ )
  assumes conj-assoc:  $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ 
  assumes disj-assoc:  $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ 
  assumes conj-commute:  $x \sqcap y = y \sqcap x$ 
  assumes disj-commute:  $x \sqcup y = y \sqcup x$ 
  assumes conj-disj-distrib:  $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ 
  assumes disj-conj-distrib:  $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ 
  assumes conj-one-right [simp]:  $x \sqcap \mathbf{1} = x$ 
  assumes disj-zero-right [simp]:  $x \sqcup \mathbf{0} = x$ 
  assumes conj-cancel-right [simp]:  $x \sqcap \sim x = \mathbf{0}$ 
  assumes disj-cancel-right [simp]:  $x \sqcup \sim x = \mathbf{1}$ 
```

```
sublocale boolean < conj!: abel-semigroup conj proof
qed (fact conj-assoc conj-commute)+
```

```
sublocale boolean < disj!: abel-semigroup disj proof
qed (fact disj-assoc disj-commute)+
```

```
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
apply (rule boolean.intro)
apply (rule disj-assoc)
apply (rule conj-assoc)
apply (rule disj-commute)
apply (rule conj-commute)
apply (rule disj-conj-distrib)
```

```

apply (rule conj-disj-distrib)
apply (rule disj-zero-right)
apply (rule conj-one-right)
apply (rule disj-cancel-right)
apply (rule conj-cancel-right)
done

```

6.1 Complement

lemma *complement-unique*:

assumes 1: $a \sqcap x = \mathbf{0}$

assumes 2: $a \sqcup x = \mathbf{1}$

assumes 3: $a \sqcap y = \mathbf{0}$

assumes 4: $a \sqcup y = \mathbf{1}$

shows $x = y$

proof –

have $(a \sqcap x) \sqcup (x \sqcap y) = (a \sqcap y) \sqcup (x \sqcap y)$ **using** 1 3 **by** *simp*

hence $(x \sqcap a) \sqcup (x \sqcap y) = (y \sqcap a) \sqcup (y \sqcap x)$ **using** *conj-commute* **by** *simp*

hence $x \sqcap (a \sqcup y) = y \sqcap (a \sqcup x)$ **using** *conj-disj-distrib* **by** *simp*

hence $x \sqcap \mathbf{1} = y \sqcap \mathbf{1}$ **using** 2 4 **by** *simp*

thus $x = y$ **using** *conj-one-right* **by** *simp*

qed

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

by (rule *complement-unique* [*OF* *conj-cancel-right* *disj-cancel-right*])

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

proof (rule *compl-unique*)

from *conj-cancel-right* **show** $\sim x \sqcap x = \mathbf{0}$ **by** (*simp* only: *conj-commute*)

from *disj-cancel-right* **show** $\sim x \sqcup x = \mathbf{1}$ **by** (*simp* only: *disj-commute*)

qed

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

by (rule *inj-eq* [*OF* *inj-on-inverseI*], rule *double-compl*)

6.2 Conjunction

lemma *conj-absorb* [*simp*]: $x \sqcap x = x$

proof –

have $x \sqcap x = (x \sqcap x) \sqcup \mathbf{0}$ **using** *disj-zero-right* **by** *simp*

also have $\dots = (x \sqcap x) \sqcup (x \sqcap \sim x)$ **using** *conj-cancel-right* **by** *simp*

also have $\dots = x \sqcap (x \sqcup \sim x)$ **using** *conj-disj-distrib* **by** (*simp* only:)

also have $\dots = x \sqcap \mathbf{1}$ **using** *disj-cancel-right* **by** *simp*

also have $\dots = x$ **using** *conj-one-right* **by** *simp*

finally show *?thesis* .

qed

lemma *conj-zero-right* [*simp*]: $x \sqcap \mathbf{0} = \mathbf{0}$

proof –

have $x \sqcap \mathbf{0} = x \sqcap (x \sqcap \sim x)$ **using** *conj-cancel-right* **by** *simp*

also have $\dots = (x \sqcap x) \sqcap \sim x$ **using** *conj-assoc* **by** (*simp only*):

also have $\dots = x \sqcap \sim x$ **using** *conj-absorb* **by** *simp*

also have $\dots = \mathbf{0}$ **using** *conj-cancel-right* **by** *simp*

finally show *?thesis* .

qed

lemma *compl-one* [*simp*]: $\sim \mathbf{1} = \mathbf{0}$

by (*rule compl-unique* [*OF conj-zero-right disj-zero-right*])

lemma *conj-zero-left* [*simp*]: $\mathbf{0} \sqcap x = \mathbf{0}$

by (*subst conj-commute*) (*rule conj-zero-right*)

lemma *conj-one-left* [*simp*]: $\mathbf{1} \sqcap x = x$

by (*subst conj-commute*) (*rule conj-one-right*)

lemma *conj-cancel-left* [*simp*]: $\sim x \sqcap x = \mathbf{0}$

by (*subst conj-commute*) (*rule conj-cancel-right*)

lemma *conj-left-absorb* [*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$

by (*simp only: conj-assoc* [*symmetric*] *conj-absorb*)

lemma *conj-disj-distrib2*:

$(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$

by (*simp only: conj-commute conj-disj-distrib*)

lemmas *conj-disj-distrib* =

conj-disj-distrib conj-disj-distrib2

6.3 Disjunction

lemma *disj-absorb* [*simp*]: $x \sqcup x = x$

by (*rule boolean.conj-absorb* [*OF dual*])

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$

by (*rule boolean.conj-zero-right* [*OF dual*])

lemma *compl-zero* [*simp*]: $\sim \mathbf{0} = \mathbf{1}$

by (*rule boolean.compl-one* [*OF dual*])

lemma *disj-zero-left* [*simp*]: $\mathbf{0} \sqcup x = x$

by (*rule boolean.conj-one-left* [*OF dual*])

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$

by (*rule boolean.conj-zero-left* [*OF dual*])

lemma *disj-cancel-left* [*simp*]: $\sim x \sqcup x = \mathbf{1}$

by (*rule boolean.conj-cancel-left* [*OF dual*])

lemma *disj-left-absorb* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$

by (rule *boolean.conj-left-absorb* [*OF dual*])

lemma *disj-conj-distrib2*:

$$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$$

by (rule *boolean.conj-disj-distrib2* [*OF dual*])

lemmas *disj-conj-distrib* =

$$\text{disj-conj-distrib disj-conj-distrib2}$$

6.4 De Morgan’s Laws

lemma *de-Morgan-conj* [*simp*]: $\sim (x \sqcap y) = \sim x \sqcup \sim y$

proof (rule *compl-unique*)

$$\text{have } (x \sqcap y) \sqcap (\sim x \sqcup \sim y) = ((x \sqcap y) \sqcap \sim x) \sqcup ((x \sqcap y) \sqcap \sim y)$$

by (rule *conj-disj-distrib*)

$$\text{also have } \dots = (y \sqcap (x \sqcap \sim x)) \sqcup (x \sqcap (y \sqcap \sim y))$$

by (*simp only: conj-ac*)

$$\text{finally show } (x \sqcap y) \sqcap (\sim x \sqcup \sim y) = \mathbf{0}$$

by (*simp only: conj-cancel-right conj-zero-right disj-zero-right*)

next

$$\text{have } (x \sqcap y) \sqcup (\sim x \sqcup \sim y) = (x \sqcup (\sim x \sqcup \sim y)) \sqcap (y \sqcup (\sim x \sqcup \sim y))$$

by (rule *disj-conj-distrib2*)

$$\text{also have } \dots = (\sim y \sqcup (x \sqcup \sim x)) \sqcap (\sim x \sqcup (y \sqcup \sim y))$$

by (*simp only: disj-ac*)

$$\text{finally show } (x \sqcap y) \sqcup (\sim x \sqcup \sim y) = \mathbf{1}$$

by (*simp only: disj-cancel-right disj-one-right conj-one-right*)

qed

lemma *de-Morgan-disj* [*simp*]: $\sim (x \sqcup y) = \sim x \sqcap \sim y$

by (rule *boolean.de-Morgan-conj* [*OF dual*])

end

6.5 Symmetric Difference

locale *boolean-xor* = *boolean* +

fixes *xor* :: 'a => 'a => 'a (**infix** \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**

fix *x y z* :: 'a

$$\text{let } ?t = (x \sqcap y \sqcap z) \sqcup (x \sqcap \sim y \sqcap \sim z) \sqcup \\ (\sim x \sqcap y \sqcap \sim z) \sqcup (\sim x \sqcap \sim y \sqcap z)$$

$$\text{have } ?t \sqcup (z \sqcap x \sqcap \sim x) \sqcup (z \sqcap y \sqcap \sim y) =$$

$$?t \sqcup (x \sqcap y \sqcap \sim y) \sqcup (x \sqcap z \sqcap \sim z)$$

by (*simp only: conj-cancel-right conj-zero-right*)

$$\text{thus } (x \oplus y) \oplus z = x \oplus (y \oplus z)$$

apply (*simp only: xor-def de-Morgan-disj de-Morgan-conj double-compl*)

apply (*simp only: conj-disj-distrib conj-ac disj-ac*)

done

```

show  $x \oplus y = y \oplus x$ 
  by (simp only: xor-def conj-commute disj-commute)
qed

```

```

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:
   $x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$ 
by (simp only: xor-def conj-disj-distrib disj-ac conj-ac conj-cancel-right disj-zero-left)

```

```

lemma xor-zero-right [simp]:  $x \oplus \mathbf{0} = x$ 
by (simp only: xor-def compl-zero conj-one-right conj-zero-right disj-zero-right)

```

```

lemma xor-zero-left [simp]:  $\mathbf{0} \oplus x = x$ 
by (subst xor-commute) (rule xor-zero-right)

```

```

lemma xor-one-right [simp]:  $x \oplus \mathbf{1} = \sim x$ 
by (simp only: xor-def compl-one conj-zero-right conj-one-right disj-zero-left)

```

```

lemma xor-one-left [simp]:  $\mathbf{1} \oplus x = \sim x$ 
by (subst xor-commute) (rule xor-one-right)

```

```

lemma xor-self [simp]:  $x \oplus x = \mathbf{0}$ 
by (simp only: xor-def conj-cancel-right conj-cancel-left disj-zero-right)

```

```

lemma xor-left-self [simp]:  $x \oplus (x \oplus y) = y$ 
by (simp only: xor-assoc [symmetric] xor-self xor-zero-left)

```

```

lemma xor-compl-left [simp]:  $\sim x \oplus y = \sim (x \oplus y)$ 
apply (simp only: xor-def de-Morgan-disj de-Morgan-conj double-compl)
apply (simp only: conj-disj-distrib)
apply (simp only: conj-cancel-right conj-cancel-left)
apply (simp only: disj-zero-left disj-zero-right)
apply (simp only: disj-ac conj-ac)
done

```

```

lemma xor-compl-right [simp]:  $x \oplus \sim y = \sim (x \oplus y)$ 
apply (simp only: xor-def de-Morgan-disj de-Morgan-conj double-compl)
apply (simp only: conj-disj-distrib)
apply (simp only: conj-cancel-right conj-cancel-left)
apply (simp only: disj-zero-left disj-zero-right)

```

apply (*simp only: disj-ac conj-ac*)
done

lemma *xor-cancel-right*: $x \oplus \sim x = \mathbf{1}$
by (*simp only: xor-compl-right xor-self compl-zero*)

lemma *xor-cancel-left*: $\sim x \oplus x = \mathbf{1}$
by (*simp only: xor-compl-left xor-self compl-zero*)

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

proof –

have $(x \sqcap y \sqcap \sim z) \sqcup (x \sqcap \sim y \sqcap z) =$
 $(y \sqcap x \sqcap \sim x) \sqcup (z \sqcap x \sqcap \sim x) \sqcup (x \sqcap y \sqcap \sim z) \sqcup (x \sqcap \sim y \sqcap z)$

by (*simp only: conj-cancel-right conj-zero-right disj-zero-left*)

thus $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$

by (*simp (no-asm-use) only:*

xor-def de-Morgan-disj de-Morgan-conj double-compl
conj-disj-distrib conj-ac disj-ac)

qed

lemma *conj-xor-distrib2*:

$(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$

proof –

have $x \sqcap (y \oplus z) = (x \sqcap y) \oplus (x \sqcap z)$

by (*rule conj-xor-distrib*)

thus $(y \oplus z) \sqcap x = (y \sqcap x) \oplus (z \sqcap x)$

by (*simp only: conj-commute*)

qed

lemmas *conj-xor-distrib* =

conj-xor-distrib conj-xor-distrib2

end

end

7 Misc-Numeric: Useful Numerical Lemmas

theory *Misc-Numeric*

imports *~~/src/HOL/Main* *~~/src/HOL/Parity*

begin

lemma *the-elemI*: $y = \{x\} \implies \text{the-elem } y = x$

by *simp*

lemma *nonemptyE*: $S \neq \{\} \implies (!x. x : S \implies R) \implies R$ **by** *auto*

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

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*: $j = k + i \implies j - i = (k :: \text{nat})$ **by** *arith*

lemma *zless2*: $0 < (2 :: \text{int})$ **by** *arith*

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*: $n \bmod (2 :: \text{int}) = 0 \mid n \bmod 2 = 1$ **by** *arith*

lemma *emep1*:

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

apply (*simp* *add*: *add-commute*)

apply (*safe* *dest*!: *even-equiv-def* [*THEN* *iffD1*])

apply (*subst* *pos-zmod-mult-2*)

apply *arith*

apply (*simp* *add*: *mod-mult-mult1*)

done

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

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

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

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

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

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 $p1mod22k$:

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

by (simp add: p1mod22k' add-commute)

lemma $z1pmod2$:

$(2 * b + 1) \bmod 2 = (1::int)$ **by** arith

lemma $z1pdiv2$:

$(2 * b + 1) \bmod 2 = (b::int)$ **by** arith

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

lemma $axbby$:

$a + m + m = b + n + n ==> (a = 0 \mid a = 1) ==> (b = 0 \mid b = 1) ==>$
 $a = b \ \& \ m = (n :: int)$ **by** arith

lemma $axxmod2$:

$(1 + x + x) \bmod 2 = (1 :: int) \ \& \ (0 + x + x) \bmod 2 = (0 :: int)$ **by** arith

lemma $axxdiv2$:

$(1 + x + x) \bmod 2 = (x :: int) \ \& \ (0 + x + x) \bmod 2 = (x :: int)$ **by** arith

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

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

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

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

$((b :: int) - a) \bmod a = b \bmod a$

by (simp add: mod-diff-right-eq)

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 $mod-plus-right$:

$((a + x) \bmod m = (b + x) \bmod m) = (a \bmod m = b \bmod (m :: nat))$

apply (induct x)

apply (simp-all add: mod-Suc)

apply arith

done

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

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

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

lemmas *push-mods = push-mods' [THEN eq-reflection]*
lemmas *pull-mods = push-mods [symmetric] rdmodes [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*:
 $!!b. b < n ==> a \text{ mod } n = b \text{ mod } n ==> a \text{ mod } n = (b :: \text{nat})$
by (*induct a*) *auto*

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

lemma *nat-mod-lem*:
 $(0 :: \text{nat}) < n ==> b < n = (b \text{ mod } n = b)$
apply *safe*
apply (*erule nat-mod-eq'*)
apply (*erule subst*)
apply (*erule mod-less-divisor*)
done

lemma *mod-nat-add*:
 $(x :: \text{nat}) < z ==> y < z ==>$
 $(x + y) \text{ mod } z = (\text{if } x + y < z \text{ then } x + y \text{ else } x + y - z)$
apply (*rule nat-mod-eq*)
apply *auto*
apply (*rule trans*)
apply (*rule le-mod-geq*)
apply *simp*
apply (*rule nat-mod-eq'*)
apply *arith*
done

lemma *mod-nat-sub*:
 $(x :: \text{nat}) < z ==> (x - y) \text{ mod } z = x - y$
by (*rule nat-mod-eq'*) *arith*

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

```

apply safe
  apply (erule (1) mod-pos-pos-trivial)
  apply (erule-tac [!] subst)
  apply auto
done

```

```

lemma int-mod-eq:
   $(0 :: \text{int}) \leq b \implies b < n \implies a \bmod n = b \bmod n \implies a \bmod n = b$ 
  by clarsimp (rule mod-pos-pos-trivial)

```

```

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

```

```

lemma int-mod-le:  $(0 :: \text{int}) \leq a \implies a \bmod n \leq a$ 
  by (fact zmod-le-nonneg-dividend)

```

```

lemma int-mod-le':  $(0 :: \text{int}) \leq b - n \implies b \bmod n \leq b - n$ 
  using zmod-le-nonneg-dividend [of b - n] by simp

```

```

lemma int-mod-ge:  $a < n \implies 0 < (n :: \text{int}) \implies a \leq a \bmod n$ 
  apply (cases 0 ≤ a)
  apply (erule (1) mod-pos-pos-trivial)
  apply simp
  apply (rule order-trans [OF - pos-mod-sign])
  apply simp
  apply assumption
done

```

```

lemma int-mod-ge':  $b < 0 \implies 0 < (n :: \text{int}) \implies b + n \leq b \bmod n$ 
  by (rule int-mod-ge [where a = b + n and n = n, simplified])

```

```

lemma mod-add-if-z:
   $(x :: \text{int}) < z \implies y < z \implies 0 \leq y \implies 0 \leq x \implies 0 \leq z \implies$ 
   $(x + y) \bmod z = (\text{if } x + y < z \text{ then } x + y \text{ else } x + y - z)$ 
  by (auto intro: int-mod-eq)

```

```

lemma mod-sub-if-z:
   $(x :: \text{int}) < z \implies y < z \implies 0 \leq y \implies 0 \leq x \implies 0 \leq z \implies$ 
   $(x - y) \bmod z = (\text{if } y \leq x \text{ then } x - y \text{ else } x - y + z)$ 
  by (auto intro: int-mod-eq)

```

```

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:  $m \sim (0 :: \text{int}) \implies (a + m * n) \text{ div } m = a \text{ div } m + n$ 
  apply (rule mcl)
  prefer 2
  apply (erule asm-rl)
  apply (simp add: zmde ring-distrib)

```

done

lemma *mod-power-lem*:

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

apply *clarsimp*

apply *safe*

apply (*simp add: dvd-eq-mod-eq-0 [symmetric]*)

apply (*drule le-iff-add [THEN iffD1]*)

apply (*force simp: power-add*)

apply (*rule mod-pos-pos-trivial*)

apply (*simp*)

apply (*rule power-strict-increasing*)

apply *auto*

done

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

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

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

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

lemma *pl-pl-rels*:

$a + b = c + d \implies$

$a \geq c \ \& \ b \leq d \mid a \leq c \ \& \ b \geq d$ ($d :: \text{nat}$) **by** *arith*

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

lemma *minus-eq*: $(m - k = m) = (k = 0 \mid m = (0 :: \text{nat}))$ **by** *arith*

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

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

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

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*:

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

apply *safe*

apply (*erule (1) xtr4 [OF div-le-mono div-mult-self-is-m]*)

apply (*erule th2*)

done

```

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

lemma div-mult-le: (a :: nat) div b * b <= a
  by (fact dtle)

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

lemma given-quot: f > (0 :: nat) ==> (f * l + (f - 1)) div f = l
  by (rule sdl, assumption) (simp (no-asm))

lemma given-quot-alt: f > (0 :: nat) ==> (l * f + f - Suc 0) div f = l
  apply (frule given-quot)
  apply (rule trans)
  prefer 2
  apply (erule asm-rl)
  apply (rule-tac f=%n. n div f in arg-cong)
  apply (simp add : mult-ac)
  done

lemma diff-mod-le: (a::nat) < d ==> b dvd d ==> a - a mod b <= d - b
  apply (unfold dvd-def)
  apply (clarify)
  apply (case-tac k)
  apply (clarsimp)
  apply (clarify)
  apply (cases b > 0)
  apply (drule mult-commute [THEN xtr1])
  apply (frule (1) td-gal-lt [THEN iffD1])
  apply (clarsimp simp: le-simps)
  apply (rule mult-div-cancel [THEN [2] xtr4])
  apply (rule mult-mono)
  apply (auto)
  done

lemma less-le-mult':
  w * c < b * c ==> 0 ≤ c ==> (w + 1) * c ≤ b * (c::int)
  apply (rule mult-right-mono)
  apply (rule zless-imp-add1-zle)
  apply (erule (1) mult-right-less-imp-less)
  apply (assumption)
  done

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  $d: (i::nat) \leq j \vee m < j'$ 
assumes  $R1: i * k \leq j * k \implies R$ 
assumes  $R2: Suc\ m * k' \leq j' * k' \implies R$ 
shows  $R$  using  $d$ 
apply safe
apply (rule  $R1$ , erule mult-le-mono1)
apply (rule  $R2$ , erule Suc-le-eq [THEN iffD2 [THEN mult-le-mono1]])
done

```

```

lemma lrlem:  $(0::nat) < sc \implies$ 
   $(sc - n + (n + lb * n) \leq m * n) = (sc + lb * n \leq m * n)$ 
apply safe
apply arith
apply (case-tac  $sc \geq n$ )
apply arith
apply (insert linorder-le-less-linear [of  $m\ lb$ ])
apply (erule-tac  $k=n$  and  $k'=n$  in lrlem')
apply arith
apply simp
done

```

```

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

```

```

lemma mpl-lem:  $j \leq (i :: nat) \implies k < j \implies i - j + k < i$  by arith

```

```

lemma nonneg-mod-div:
   $0 \leq a \implies 0 \leq b \implies 0 \leq (a \bmod b :: int) \ \& \ 0 \leq a \operatorname{div} b$ 
apply (cases  $b = 0$ , clarsimp)
apply (auto intro: pos-imp-zdiv-nonneg-iff [THEN iffD2])
done

```

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 ..

```

```

instantiation bit :: {zero, one}
begin

```

```

definition zero-bit-def:

```

$0 = \text{Abs-bit False}$

definition *one-bit-def*:

$1 = \text{Abs-bit True}$

instance ..

end

rep-datatype $0::\text{bit } 1::\text{bit}$

proof –

fix P **and** $x :: \text{bit}$

assume $P (0::\text{bit})$ **and** $P (1::\text{bit})$

then have $\forall b. P (\text{Abs-bit } b)$

unfolding *zero-bit-def one-bit-def*

by (*simp add: all-bool-eq*)

then show $P x$

by (*induct x simp*)

next

show $(0::\text{bit}) \neq (1::\text{bit})$

unfolding *zero-bit-def one-bit-def*

by (*simp add: Abs-bit-inject*)

qed

lemma *bit-not-0-iff [iff]*: $(x::\text{bit}) \neq 0 \longleftrightarrow x = 1$

by (*induct x simp-all*)

lemma *bit-not-1-iff [iff]*: $(x::\text{bit}) \neq 1 \longleftrightarrow x = 0$

by (*induct x simp-all*)

8.2 Type *bit* forms a field

instantiation $\text{bit} :: \text{field-inverse-zero}$

begin

definition *plus-bit-def*:

$x + y = \text{bit-case } y (\text{bit-case } 1 \ 0 \ y) \ x$

definition *times-bit-def*:

$x * y = \text{bit-case } 0 \ y \ x$

definition *uminus-bit-def [simp]*:

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

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

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

definition *inverse-bit-def [simp]*:

$\text{inverse } x = (x :: \text{bit})$

definition *divide-bit-def* [*simp*]:

$$x / y = (x * y :: bit)$$

lemmas *field-bit-defs* =

plus-bit-def times-bit-def minus-bit-def uminus-bit-def
divide-bit-def inverse-bit-def

instance proof

qed (*unfold field-bit-defs, auto split: bit.split*)

end

lemma *bit-add-self*: $x + x = (0 :: bit)$

unfolding *plus-bit-def* **by** (*simp split: bit.split*)

lemma *bit-mult-eq-1-iff* [*simp*]: $x * y = (1 :: bit) \longleftrightarrow x = 1 \wedge y = 1$

unfolding *times-bit-def* **by** (*simp split: bit.split*)

Not sure whether the next two should be simp rules.

lemma *bit-add-eq-0-iff*: $x + y = (0 :: bit) \longleftrightarrow x = y$

unfolding *plus-bit-def* **by** (*simp split: bit.split*)

lemma *bit-add-eq-1-iff*: $x + y = (1 :: bit) \longleftrightarrow x \neq y$

unfolding *plus-bit-def* **by** (*simp split: bit.split*)

8.3 Numerals at type *bit*

All numerals reduce to either 0 or 1.

lemma *bit-minus1* [*simp*]: $-1 = (1 :: bit)$

by (*simp only: minus-one [symmetric] uminus-bit-def*)

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

by (*simp only: neg-numeral-def uminus-bit-def*)

lemma *bit-numeral-even* [*simp*]: $numeral (Num.Bit0\ w) = (0 :: bit)$

by (*simp only: numeral-Bit0 bit-add-self*)

lemma *bit-numeral-odd* [*simp*]: $numeral (Num.Bit1\ w) = (1 :: bit)$

by (*simp only: numeral-Bit1 bit-add-self add-0-left*)

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
by (*simp-all add: bitval-def*)

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*
unfolding *bin-rest-def bin-last-def Bit-def*
using *mod-div-equality* [of *w 2*]
by (*cases w mod 2 = 0, simp-all*)

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

lemma *bin-last-BIT* [*simp*]: *bin-last (x BIT b)* = *b*
unfolding *bin-last-def Bit-def*
by (*cases b, simp-all add: z1pmod2*)

lemma *BIT-eq-iff* [*iff*]: *u BIT b = v BIT c* \longleftrightarrow *u = v* \wedge *b = c*
by (*metis bin-rest-BIT bin-last-BIT*)

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)*
unfolding *neg-numeral-def numeral.simps numeral-BitM*
unfolding *Bit-def bitval-simps*
by (*simp-all del: arith-simps add-numeral-special diff-numeral-special*)

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

shows $0 \text{ BIT } 0 = 0$ and $0 \text{ BIT } 1 = 1$ and $1 \text{ BIT } 0 = 2$ and $1 \text{ BIT } 1 = 3$
unfolding *Bit-def* **by** *simp-all*

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

lemma *expand-BIT*:

$\text{numeral } (\text{Num.Bit0 } w) = \text{numeral } w \text{ BIT } 0$
 $\text{numeral } (\text{Num.Bit1 } w) = \text{numeral } w \text{ BIT } 1$
 $\text{neg-numeral } (\text{Num.Bit0 } w) = \text{neg-numeral } w \text{ BIT } 0$
 $\text{neg-numeral } (\text{Num.Bit1 } w) = \text{neg-numeral } (w + \text{Num.One}) \text{ BIT } 1$
unfolding *add-One* **by** (*simp-all add: BitM-inc*)

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

$\text{bin-last } 0 = 0$
 $\text{bin-last } 1 = 1$
 $\text{bin-last } -1 = 1$
 $\text{bin-last } \text{Numeral1} = 1$
 $\text{bin-last } (\text{numeral } (\text{Num.Bit0 } w)) = 0$
 $\text{bin-last } (\text{numeral } (\text{Num.Bit1 } w)) = 1$
 $\text{bin-last } (\text{neg-numeral } (\text{Num.Bit0 } w)) = 0$
 $\text{bin-last } (\text{neg-numeral } (\text{Num.Bit1 } w)) = 1$
unfolding *expand-BIT bin-last-BIT* **by** (*simp-all add: bin-last-def*)

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

$\text{bin-rest } 0 = 0$
 $\text{bin-rest } 1 = 0$
 $\text{bin-rest } -1 = -1$
 $\text{bin-rest } \text{Numeral1} = 0$
 $\text{bin-rest } (\text{numeral } (\text{Num.Bit0 } w)) = \text{numeral } w$
 $\text{bin-rest } (\text{numeral } (\text{Num.Bit1 } w)) = \text{numeral } w$
 $\text{bin-rest } (\text{neg-numeral } (\text{Num.Bit0 } w)) = \text{neg-numeral } w$
 $\text{bin-rest } (\text{neg-numeral } (\text{Num.Bit1 } w)) = \text{neg-numeral } (w + \text{Num.One})$
unfolding *expand-BIT bin-rest-BIT* **by** (*simp-all add: bin-rest-def*)

lemma *less-Bits*:

$(v \text{ BIT } b < w \text{ BIT } c) = (v < w \mid v \leq w \ \& \ b = (0::\text{bit}) \ \& \ c = (1::\text{bit}))$
unfolding *Bit-def* **by** (*auto simp add: bitval-def split: bit.split*)

lemma *le-Bits*:

$(v \text{ BIT } b \leq w \text{ BIT } c) = (v < w \mid v \leq w \ \& \ (b \sim = (1::\text{bit}) \mid c \sim = (0::\text{bit})))$
unfolding *Bit-def* **by** (*auto simp add: bitval-def split: bit.split*)

lemma *Bit-B0*:

$k \text{ BIT } (0::\text{bit}) = k + k$
by (*unfold Bit-def*) *simp*

lemma *Bit-B1*:

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

by (unfold Bit-def) simp

lemma Bit-B0-2t: $k \text{ BIT } (0::\text{bit}) = 2 * k$
 by (rule trans, rule Bit-B0) simp

lemma Bit-B1-2t: $k \text{ BIT } (1::\text{bit}) = 2 * k + 1$
 by (rule trans, rule Bit-B1) simp

lemma B-mod-2':
 $X = 2 \implies (w \text{ BIT } (1::\text{bit})) \text{ mod } X = 1 \ \& \ (w \text{ BIT } (0::\text{bit})) \text{ mod } X = 0$
 apply (simp (no-asm) only: Bit-B0 Bit-B1)
 apply (simp add: z1pmod2)
 done

lemma neB1E [elim!]:
 assumes ne: $y \neq (1::\text{bit})$
 assumes y: $y = (0::\text{bit}) \implies P$
 shows P
 apply (rule y)
 apply (cases y rule: bit.exhaust, simp)
 apply (simp add: ne)
 done

lemma bin-ex-rl: $EX \ w \ b. \ w \ \text{BIT} \ b = \text{bin}$
 by (metis bin-rl-simp)

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

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:
 $\text{bin} = (w \ \text{BIT} \ b) \implies \text{bin} \ \sim = \ -1 \ \dashrightarrow \ \text{bin} \ \sim = \ 0 \ \dashrightarrow$
 $\text{nat } (\text{abs } w) < \text{nat } (\text{abs } \text{bin})$
 apply clarsimp
 apply (unfold Bit-def)
 apply (cases b)
 apply (clarsimp, arith)
 apply (clarsimp, arith)

done

lemma *bin-induct*:

assumes *PPls*: $P\ 0$

and *PMin*: $P\ -1$

and *PBit*: $\forall \text{bin bit. } P\ \text{bin} \implies P\ (\text{bin BIT bit})$

shows $P\ \text{bin}$

apply (rule-tac $P=P$ and $a=\text{bin}$ and $f1=\text{nat o abs}$
in *wf-measure* [*THEN wf-induct*])

apply (simp add: *measure-def inv-image-def*)

apply (case-tac x rule: *bin-exhaust*)

apply (frule *bin-abs-lem*)

apply (auto simp add : *PPls PMin PBit*)

done

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

unfolding *bin-rest-def* [*symmetric*] by (rule *bin-rest-BIT*)

lemma *bin-nth-lem* [*rule-format*]:

ALL y. bin-nth x = bin-nth y \implies x = y

apply (induct x rule: *bin-induct*)

apply safe

apply (erule *rev-mp*)

apply (induct-tac y rule: *bin-induct*)

apply safe

apply (drule-tac $x=0$ in *fun-cong, force*)

apply (erule *notE, rule ext,*

drule-tac x=Suc x in fun-cong, force)

apply (drule-tac $x=0$ in *fun-cong, force*)

apply (erule *rev-mp*)

apply (induct-tac y rule: *bin-induct*)

apply safe

apply (drule-tac $x=0$ in *fun-cong, force*)

apply (erule *notE, rule ext,*

drule-tac x=Suc x in fun-cong, force)

apply (drule-tac $x=0$ in *fun-cong, force*)

apply (case-tac y rule: *bin-exhaust*)

apply clarify

apply (erule *allE*)

apply (erule *impE*)

prefer 2

apply (erule *conjI*)

apply (drule-tac $x=0$ in *fun-cong, force*)

apply (rule *ext*)

apply (drule-tac $x=\text{Suc } ?x$ in *fun-cong, force*)

done

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

by (auto elim: *bin-nth-lem*)

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

lemma *bin-eq-iff*: $x = y \longleftrightarrow (\forall n. \text{bin-nth } x \ n = \text{bin-nth } y \ n)$
 by (auto intro!: *bin-nth-lem* del: *equalityI*)

lemma *bin-nth-zero* [*simp*]: $\neg \text{bin-nth } 0 \ n$
 by (induct *n*) auto

lemma *bin-nth-1* [*simp*]: $\text{bin-nth } 1 \ n \longleftrightarrow n = 0$
 by (cases *n*) *simp-all*

lemma *bin-nth-minus1* [*simp*]: $\text{bin-nth } -1 \ n$
 by (induct *n*) auto

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

lemma *bin-nth-Suc-BIT*: $\text{bin-nth } (w \ \text{BIT } b) \ (\text{Suc } n) = \text{bin-nth } w \ n$
 by auto

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

lemma *bin-nth-numeral*:
 $\text{bin-rest } x = y \implies \text{bin-nth } x \ (\text{numeral } n) = \text{bin-nth } y \ (\text{pred-numeral } n)$
 by (*simp add: numeral-eq-Suc*)

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

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

9.3 Truncating binary integers

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

lemma *bin-sign-simps* [*simp*]:
 $\text{bin-sign } 0 = 0$
 $\text{bin-sign } 1 = 0$
 $\text{bin-sign } (\text{numeral } k) = 0$

```

bin-sign (neg-numeral k) = -1
bin-sign (w BIT b) = bin-sign w
unfolding bin-sign-def Bit-def bitval-def
by (simp-all split: bit.split)

```

```

lemma bin-sign-rest [simp]:
  bin-sign (bin-rest w) = bin-sign w
by (cases w rule: bin-exhaust) auto

```

```

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

```

```

primrec sbintrunc :: nat => int => int where
  Z : sbintrunc 0 bin = (case bin-last bin of (1::bit) ⇒ -1 | (0::bit) ⇒ 0)
| Suc : sbintrunc (Suc n) bin = sbintrunc n (bin-rest bin) BIT (bin-last bin)

```

```

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

```

```

lemma bintrunc-mod2p: bintrunc n w = (w mod 2 ^ n)
apply (induct n arbitrary: w, clarsimp)
apply (simp add: bin-last-def bin-rest-def Bit-def zmod-zmult2-eq)
done

```

```

lemma sbintrunc-mod2p: sbintrunc n w = (w + 2 ^ n) mod 2 ^ (Suc n) - 2 ^ n
apply (induct n arbitrary: w)
apply simp
apply (subst mod-add-left-eq)
apply (simp add: bin-last-def)
apply (simp add: bin-last-def bin-rest-def Bit-def)
apply (clarsimp simp: mod-mult-mult1 [symmetric]
  zmod-zdiv-equality [THEN diff-eq-eq [THEN iffD2 [THEN sym]]])
apply (rule trans [symmetric, OF - emep1])
apply auto
apply (auto simp: even-def)
done

```

9.4 Simplifications for (s)bintrunc

```

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

```

```

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

```

```

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

```

lemma *bintrunc-Suc-numeral*:

```

bintrunc (Suc n) 1 = 1
bintrunc (Suc n) -1 = bintrunc n -1 BIT 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-numeral (Num.Bit0 w)) =
  bintrunc n (neg-numeral w) BIT 0
bintrunc (Suc n) (neg-numeral (Num.Bit1 w)) =
  bintrunc n (neg-numeral (w + Num.One)) BIT 1
by simp-all

```

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

```

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

```

lemma *sbintrunc-Suc-numeral*:

```

sbintrunc (Suc n) 1 = 1
sbintrunc (Suc n) (numeral (Num.Bit0 w)) =
  sbintrunc n (numeral w) BIT 0
sbintrunc (Suc n) (numeral (Num.Bit1 w)) =
  sbintrunc n (numeral w) BIT 1
sbintrunc (Suc n) (neg-numeral (Num.Bit0 w)) =
  sbintrunc n (neg-numeral w) BIT 0
sbintrunc (Suc n) (neg-numeral (Num.Bit1 w)) =
  sbintrunc n (neg-numeral (w + Num.One)) BIT 1
by simp-all

```

lemma *bit-bool*:

```

(b = (b' = (1::bit))) = (b' = (if b then (1::bit) else (0::bit)))
by (cases b') auto

```

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

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

```

apply (induct n arbitrary: bin)
apply (case-tac bin rule: bin-exhaust, case-tac b, auto)
done

```

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

```

apply (induct n arbitrary: w m)
  apply (case-tac m, auto)[1]
  apply (case-tac m, auto)[1]
done

```

lemma *nth-sbintr*:

```

bin-nth (sbintrunc m w) n =
  (if n < m then bin-nth w n else bin-nth w m)
apply (induct n arbitrary: w m)
apply (case-tac m, simp-all split: bit.splits)[1]
apply (case-tac m, simp-all split: bit.splits)[1]
done

```

lemma *bin-nth-Bit*:

```

bin-nth (w BIT b) n = (n = 0 & b = (1::bit) | (EX m. n = Suc m & bin-nth w
m))
by (cases n) auto

```

lemma *bin-nth-Bit0*:

```

bin-nth (numeral (Num.Bit0 w)) n <=>
  (∃ m. n = Suc m ∧ bin-nth (numeral w) m)
using bin-nth-Bit [where w=numeral w and b=(0::bit)] by simp

```

lemma *bin-nth-Bit1*:

```

bin-nth (numeral (Num.Bit1 w)) n <=>
  n = 0 ∨ (∃ m. n = Suc m ∧ bin-nth (numeral w) m)
using bin-nth-Bit [where w=numeral w and b=(1::bit)] by simp

```

lemma *bintrunc-bintrunc-l*:

```

n <= m ==> (bintrunc m (bintrunc n w) = bintrunc n w)
by (rule bin-eqI) (auto simp add : nth-bintr)

```

lemma *sbintrunc-sbintrunc-l*:

```

n <= m ==> (sbintrunc m (sbintrunc n w) = sbintrunc n w)
by (rule bin-eqI) (auto simp: nth-sbintr)

```

lemma *bintrunc-bintrunc-ge*:

```

n <= m ==> (bintrunc n (bintrunc m w) = bintrunc n w)
by (rule bin-eqI) (auto simp: nth-bintr)

```

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

```

bintrunc m (bintrunc n w) = bintrunc (min m n) w
apply (rule bin-eqI)
apply (auto simp: nth-bintr)
done

```

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

```

sbintrunc m (sbintrunc n w) = sbintrunc (min m n) w
apply (rule bin-eqI)
apply (auto simp: nth-sbintr min-max.inf-absorb1 min-max.inf-absorb2)
done

```

lemmas *bintrunc-Pls* =

```

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

```

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

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

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

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

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

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

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

lemmas *sbintrunc-Pls* =
sbintrunc.Z [**where** *bin*=0,
simplified bin-last-numeral-simps bin-rest-numeral-simps 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*:
 $0 < n ==> \text{bintrunc } (\text{Suc } (n - 1)) \ w = \text{bintrunc } n \ w$
by *auto*

lemma *sbintrunc-minus*:

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

by *auto*

lemmas *bintrunc-minus-simps* =

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

lemmas *sbintrunc-minus-simps* =

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

lemmas *thobini1* = *arg-cong* [where $f = \%w. w \text{ BIT } b$] **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*:

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

by *auto*

lemmas *bintrunc-Suc-Ialts* =

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

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

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

lemmas *sbintrunc-Suc-Is* =

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

lemmas *sbintrunc-Suc-minus-Is* =

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

lemma *sbintrunc-Suc-lem*:

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

by *auto*

lemmas *sbintrunc-Suc-Ialts* =

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

lemma *sbintrunc-bintrunc-lt*:

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

by (rule *bin-eqI*) (auto simp: *nth-sbintr nth-bintr*)

lemma *bintrunc-sbintrunc-le*:

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

apply (rule *bin-eqI*)

```

apply (auto simp: nth-sbintr nth-bintr)
apply (subgoal-tac x=n, safe, arith+)[1]
apply (subgoal-tac x=n, safe, arith+)[1]
done

```

```

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

```

```

lemma bintrunc-sbintrunc' [simp]:
   $0 < n \implies \text{bintrunc } n (\text{sbintrunc } (n - 1) w) = \text{bintrunc } n w$ 
by (cases n) (auto simp del: bintrunc.Suc)

```

```

lemma sbintrunc-bintrunc' [simp]:
   $0 < n \implies \text{sbintrunc } (n - 1) (\text{bintrunc } n w) = \text{sbintrunc } (n - 1) w$ 
by (cases n) (auto simp del: bintrunc.Suc)

```

```

lemma bin-sbin-eq-iff:
   $\text{bintrunc } (\text{Suc } n) x = \text{bintrunc } (\text{Suc } n) y \iff$ 
   $\text{sbintrunc } n x = \text{sbintrunc } n y$ 
apply (rule iffI)
apply (rule box-equals [OF - sbintrunc-bintrunc sbintrunc-bintrunc])
apply simp
apply (rule box-equals [OF - bintrunc-sbintrunc bintrunc-sbintrunc])
apply simp
done

```

```

lemma bin-sbin-eq-iff':
   $0 < n \implies \text{bintrunc } n x = \text{bintrunc } n y \iff$ 
   $\text{sbintrunc } (n - 1) x = \text{sbintrunc } (n - 1) y$ 
by (cases n) (simp-all add: bin-sbin-eq-iff del: bintrunc.Suc)

```

```

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

```

```

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

```

```

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

```

```

lemma bintrunc-numeral:
   $\text{bintrunc } (\text{numeral } k) x =$ 
   $\text{bintrunc } (\text{pred-numeral } k) (\text{bin-rest } x) \text{ BIT } \text{bin-last } x$ 
by (simp add: numeral-eq-Suc)

```

lemma *sbintrunc-numeral*:

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

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

bintrunc (numeral *k*) (numeral (Num.Bit0 *w*)) =
bintrunc (pred-numeral *k*) (numeral *w*) BIT 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
 by (simp-all add: bintrunc-numeral)

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

sbintrunc (numeral *k*) (numeral (Num.Bit0 *w*)) =
sbintrunc (pred-numeral *k*) (numeral *w*) BIT 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
 by (simp-all add: sbintrunc-numeral)

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

by (rule ext) (rule bintrunc-mod2p)

lemma *range-bintrunc*: range (*bintrunc* *n*) = {*i*. 0 <= *i* & *i* < 2 ^ *n*}

apply (unfold no-bintr-alt1)

apply (auto simp add: image-iff)

apply (rule exI)

apply (auto intro: int-mod-lem [THEN iffD1, symmetric])

done

lemma *no-sbintr-alt2*:

sbintrunc *n* = (%*w*. (*w* + 2 ^ *n*) mod 2 ^ Suc *n* - 2 ^ *n* :: int)

by (rule ext) (simp add: sbintrunc-mod2p)

lemma *range-sbintrunc*:

range (*sbintrunc* *n*) = {*i*. - (2 ^ *n*) <= *i* & *i* < 2 ^ *n*}

apply (unfold no-sbintr-alt2)

apply (auto simp add: image-iff eq-diff-eq)

apply (rule exI)

apply (auto intro: int-mod-lem [THEN iffD1, symmetric])

done

lemma *sb-inc-lem*:

$(a::int) + 2^k < 0 \implies a + 2^k + 2^{(Suc\ k)} \leq (a + 2^k) \bmod 2^{(Suc\ k)}$
apply (*erule int-mod-ge'* [**where** $n = 2^{(Suc\ k)}$ **and** $b = a + 2^k$, *simplified*
zless2p])
apply (*rule TrueI*)
done

lemma *sb-inc-lem'*:

$(a::int) < -(2^k) \implies a + 2^k + 2^{(Suc\ k)} \leq (a + 2^k) \bmod 2^{(Suc\ k)}$
by (*rule sb-inc-lem*) *simp*

lemma *sbintrunc-inc*:

$x < -(2^n) \implies x + 2^{(Suc\ n)} \leq \text{sbintrunc } n\ x$
unfolding *no-sbintr-alt2* **by** (*drule sb-inc-lem'*) *simp*

lemma *sb-dec-lem*:

$(0::int) \leq -(2^k) + a \implies (a + 2^k) \bmod (2 * 2^k) \leq -(2^k) + a$
by (*rule int-mod-le'* [**where** $n = 2^{(Suc\ k)}$ **and** $b = a + 2^k$, *simplified*])

lemma *sb-dec-lem'*:

$(2::int)^k \leq a \implies (a + 2^k) \bmod (2 * 2^k) \leq -(2^k) + a$
by (*rule iffD1* [*OF diff-le-eq'*, *THEN sb-dec-lem*, *simplified*])

lemma *sbintrunc-dec*:

$x \geq (2^n) \implies x - 2^{(Suc\ n)} \geq \text{sbintrunc } n\ x$
unfolding *no-sbintr-alt2* **by** (*drule sb-dec-lem'*) *simp*

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*: $0 \leq \text{bintrunc } n \ w$
by (*simp add: bintrunc-mod2p*)

lemma *bintr-lt2p*: $\text{bintrunc } n \ w < 2^n$
by (*simp add: bintrunc-mod2p*)

lemma *bintr-Min*: $\text{bintrunc } n \ -1 = 2^n - 1$
by (*simp add: bintrunc-mod2p m1mod2k*)

lemma *sbintr-ge*: $-(2^n) \leq \text{sbintrunc } n \ w$
by (*simp add: sbintrunc-mod2p*)

lemma *sbintr-lt*: $\text{sbintrunc } n \ w < 2^n$
by (*simp add: sbintrunc-mod2p*)

lemma *sign-Pls-ge-0*:
 $(\text{bin-sign } \text{bin} = 0) = (\text{bin} \geq (0 :: int))$
unfolding *bin-sign-def* **by** *simp*

lemma *sign-Min-lt-0*:
 $(\text{bin-sign } \text{bin} = -1) = (\text{bin} < (0 :: int))$
unfolding *bin-sign-def* **by** *simp*

lemma *bin-rest-trunc*:
 $(\text{bin-rest } (\text{bintrunc } n \ \text{bin})) = \text{bintrunc } (n - 1) (\text{bin-rest } \text{bin})$
by (*induct n arbitrary: bin*) *auto*

lemma *bin-rest-power-trunc* [*rule-format*]:
 $(\text{bin-rest } ^k) (\text{bintrunc } n \ \text{bin}) =$
 $\text{bintrunc } (n - k) ((\text{bin-rest } ^k) \text{bin})$
by (*induct k*) (*auto simp: bin-rest-trunc*)

lemma *bin-rest-trunc-i*:
 $\text{bintrunc } n (\text{bin-rest } \text{bin}) = \text{bin-rest } (\text{bintrunc } (\text{Suc } n) \ \text{bin})$
by *auto*

lemma *bin-rest-strunc*:
 $\text{bin-rest } (\text{sbintrunc } (\text{Suc } n) \ \text{bin}) = \text{sbintrunc } n (\text{bin-rest } \text{bin})$
by (*induct n arbitrary: bin*) *auto*

lemma *bintrunc-rest* [*simp*]:
 $\text{bintrunc } n (\text{bin-rest } (\text{bintrunc } n \ \text{bin})) = \text{bin-rest } (\text{bintrunc } n \ \text{bin})$
apply (*induct n arbitrary: bin, simp*)
apply (*case-tac bin rule: bin-exhaust*)
apply (*auto simp: bintrunc-bintrunc-l*)

done

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

sbintrunc n (bin-rest (sbintrunc n bin)) = bin-rest (sbintrunc n bin)

apply (*induct n arbitrary: bin, simp*)

apply (*case-tac bin rule: bin-exhaust*)

apply (*auto simp: bintrunc-bintrunc-l split: bit.splits*)

done

lemma *bintrunc-rest'*:

bintrunc n o bin-rest o bintrunc n = bin-rest o bintrunc n

by (*rule ext*) *auto*

lemma *sbintrunc-rest'* :

sbintrunc n o bin-rest o sbintrunc n = bin-rest o sbintrunc n

by (*rule ext*) *auto*

lemma *rco-lem*:

f o g o f = g o f ==> f o (g o f) ^^ n = g ^^ n o f

apply (*rule ext*)

apply (*induct-tac n*)

apply (*simp-all (no-asm)*)

apply (*drule fun-cong*)

apply (*unfold o-def*)

apply (*erule trans*)

apply *simp*

done

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

apply (*rule ext*)

apply (*induct n*)

apply (*simp-all add: o-def*)

done

lemmas *rco-bintr = bintrunc-rest'*

[*THEN rco-lem [THEN fun-cong], unfolded o-def*]

lemmas *rco-sbintr = sbintrunc-rest'*

[*THEN rco-lem [THEN fun-cong], unfolded o-def*]

9.5 Splitting and concatenation

primrec *bin-split* :: *nat* \Rightarrow *int* \Rightarrow *int* \times *int* **where**

Z: *bin-split 0 w = (w, 0)*

| *Suc*: *bin-split (Suc n) w = (let (w1, w2) = bin-split n (bin-rest w)*
in (w1, w2 BIT bin-last w))

lemma [*code*]:

bin-split (Suc n) w = (let (w1, w2) = bin-split n (bin-rest w) in (w1, w2 BIT bin-last w))

bin-split 0 w = (w, 0)
by *simp-all*

primrec *bin-cat* :: *int* \Rightarrow *nat* \Rightarrow *int* \Rightarrow *int* **where**

Z: *bin-cat w 0 v = w*

| *Suc*: *bin-cat w (Suc n) v = bin-cat w n (bin-rest v) BIT bin-last v*

9.6 Miscellaneous lemmas

lemma *funpow-minus-simp*:

$0 < n \implies f^{^^} n = f \circ f^{^^} (n - 1)$

by (*cases n*) *simp-all*

lemma *funpow-numeral* [*simp*]:

$f^{^^} \text{numeral } k = f \circ f^{^^} (\text{pred-numeral } k)$

by (*simp add: numeral-eq-Suc*)

lemma *replicate-numeral* [*simp*]:

replicate (numeral k) x = x # replicate (pred-numeral k) x

by (*simp add: numeral-eq-Suc*)

lemmas *power-minus-simp =*

trans [OF gen-minus [where f = power f] power-Suc] for f

lemma *list-exhaust-size-gt0*:

assumes *y*: $\bigwedge a \text{ list. } y = a \# \text{list} \implies P$

shows $0 < \text{length } y \implies P$

apply (*cases y, simp*)

apply (*rule y*)

apply *fastforce*

done

lemma *list-exhaust-size-eq0*:

assumes *y*: $y = [] \implies P$

shows $\text{length } y = 0 \implies P$

apply (*cases y*)

apply (*rule y, simp*)

apply *simp*

done

lemma *size-Cons-lem-eq*:

$y = xa \# \text{list} \implies \text{size } y = \text{Suc } k \implies \text{size list} = k$

by *auto*

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

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

by (*cases y*) *auto*

```

lemma B1-ass-B0:
  assumes  $y: y = (0::bit) \implies y = (1::bit)$ 
  shows  $y = (1::bit)$ 
  apply (rule classical)
  apply (drule not-B1-is-B0)
  apply (erule y)
  done

```

— simplifications for specific word lengths

```

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

```

```

lemmas s2n-ths = n2s-ths [symmetric]

```

```

end

```

10 Bit-Operations: Syntactic classes for bitwise operations

```

theory Bit-Operations
imports ~/src/HOL/Library/Bit
begin

```

```

class bit =
  fixes bitNOT :: 'a  $\Rightarrow$  'a (NOT - [70] 71)
  and bitAND :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixr AND 64)
  and bitOR :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (infixr OR 59)
  and bitXOR :: 'a  $\Rightarrow$  'a  $\Rightarrow$  '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  $\Rightarrow$  nat  $\Rightarrow$  bool (infixl !! 100)
  and lsb :: 'a  $\Rightarrow$  bool
  and set-bit :: 'a  $\Rightarrow$  nat  $\Rightarrow$  bool  $\Rightarrow$  'a
  and set-bits :: (nat  $\Rightarrow$  bool)  $\Rightarrow$  'a (binder BITS 10)
  and shiffl :: 'a  $\Rightarrow$  nat  $\Rightarrow$  'a (infixl << 55)
  and shiftr :: 'a  $\Rightarrow$  nat  $\Rightarrow$  'a (infixl >> 55)

```

```

class bitss = bits +
  fixes msb :: 'a  $\Rightarrow$  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 ..

end

lemmas *bit-simps* =

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

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

$x\ AND\ 0 = (0::bit)$
 $x\ AND\ 1 = (x::bit)$
 $x\ OR\ 1 = (1::bit)$
 $x\ OR\ 0 = (x::bit)$
 $x\ XOR\ 1 = NOT\ (x::bit)$
 $x\ XOR\ 0 = (x::bit)$
by (*cases x, auto*)+

lemma *bit-ops-comm*:

$(x::bit)\ AND\ y = y\ AND\ x$
 $(x::bit)\ OR\ y = y\ OR\ x$
 $(x::bit)\ XOR\ y = y\ XOR\ x$
by (*cases y, auto*)+

lemma *bit-ops-same* [*simp*]:

$(x::bit)\ AND\ x = x$
 $(x::bit)\ OR\ x = x$
 $(x::bit)\ XOR\ x = 0$
by (*cases x, auto*)+

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

by (*cases x auto*)

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 = ( $\lambda x::int. - x - 1$ )
```

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

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

```
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 ..
```

```
end
```

11.1.1 Basic simplification rules

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

```
lemma int-not-simps [simp]:
  NOT (0::int) = -1
  NOT (1::int) = -2
  NOT (-1::int) = 0
```

$NOT (numeral\ w::int) = neg_numeral\ (w + Num.One)$
 $NOT (neg_numeral\ (Num.Bit0\ w)::int) = numeral\ (Num.BitM\ w)$
 $NOT (neg_numeral\ (Num.Bit1\ w)::int) = numeral\ (Num.Bit0\ w)$
unfolding *int-not-def* **by** *simp-all*

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

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

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

lemma *Bit-eq-0-iff*: $w\ BIT\ b = 0 \longleftrightarrow w = 0 \wedge b = 0$
by (*subst BIT-eq-iff [symmetric], simp*)

lemma *Bit-eq-m1-iff*: $w\ BIT\ b = -1 \longleftrightarrow w = -1 \wedge b = 1$
by (*subst BIT-eq-iff [symmetric], simp*)

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

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

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

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

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

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

lemma *bit-xor-def*: $(b::bit) XOR\ c = (b AND NOT\ c) OR (NOT\ b AND c)$
by (*induct b, simp-all*)

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

11.1.2 Binary destructors

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

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

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

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

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

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

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

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

lemma *bit-NOT-eq-1-iff* [simp]: $\text{NOT } (b::\text{bit}) = 1 \longleftrightarrow b = 0$
by (*induct b, simp-all*)

lemma *bit-AND-eq-1-iff* [simp]: $(a::\text{bit}) \text{ AND } b = 1 \longleftrightarrow a = 1 \wedge b = 1$
by (*induct a, simp-all*)

lemma *bin-nth-ops*:

!! $x y$. $\text{bin-nth } (x \text{ AND } y) \ n = (\text{bin-nth } x \ n \ \& \ \text{bin-nth } y \ n)$
!! $x y$. $\text{bin-nth } (x \text{ OR } y) \ n = (\text{bin-nth } x \ n \ | \ \text{bin-nth } y \ n)$
!! $x y$. $\text{bin-nth } (x \text{ XOR } y) \ n = (\text{bin-nth } x \ n \ \sim = \ \text{bin-nth } y \ n)$
!! x . $\text{bin-nth } (\text{NOT } x) \ n = (\sim \ \text{bin-nth } x \ n)$
by (*induct n auto*)

11.1.3 Derived properties

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

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

lemma *int-or-extra-simps* [simp]:

$w \text{ OR } (0::\text{int}) = w$
 $w \text{ OR } (-1::\text{int}) = -1$
by (auto simp add: bin-eq-iff bin-nth-ops)

lemma *int-and-extra-simps* [simp]:

$w \text{ AND } (0::\text{int}) = 0$
 $w \text{ AND } (-1::\text{int}) = w$
by (auto simp add: bin-eq-iff bin-nth-ops)

lemma *bin-ops-comm*:

shows

int-and-comm: $\forall y::\text{int}. x \text{ AND } y = y \text{ AND } x$ **and**
int-or-comm: $\forall y::\text{int}. x \text{ OR } y = y \text{ OR } x$ **and**
int-xor-comm: $\forall y::\text{int}. x \text{ XOR } y = y \text{ XOR } x$
by (auto simp add: bin-eq-iff bin-nth-ops)

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

$(x::\text{int}) \text{ AND } x = x$
 $(x::\text{int}) \text{ OR } x = x$
 $(x::\text{int}) \text{ XOR } x = 0$
by (auto simp add: bin-eq-iff bin-nth-ops)

lemmas *bin-log-esimps* =

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

lemma *bbw-ao-absorb*:

$\forall y::\text{int}. x \text{ AND } (y \text{ OR } x) = x \ \& \ x \text{ OR } (y \text{ AND } x) = x$
by (auto simp add: bin-eq-iff bin-nth-ops)

lemma *bbw-ao-absorbs-other*:

$x \text{ AND } (x \text{ OR } y) = x \ \wedge \ (y \text{ AND } x) \text{ OR } x = (x::\text{int})$
 $(y \text{ OR } x) \text{ AND } x = x \ \wedge \ x \text{ OR } (x \text{ AND } y) = (x::\text{int})$
 $(x \text{ OR } y) \text{ AND } x = x \ \wedge \ (x \text{ AND } y) \text{ OR } x = (x::\text{int})$
by (auto simp add: bin-eq-iff bin-nth-ops)

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

lemma *int-xor-not*:

$\forall y::\text{int}. (\text{NOT } x) \text{ XOR } y = \text{NOT } (x \text{ XOR } y) \ \&$
 $x \text{ XOR } (\text{NOT } y) = \text{NOT } (x \text{ XOR } y)$
by (auto simp add: bin-eq-iff bin-nth-ops)

lemma *int-and-assoc*:

$(x \text{ AND } y) \text{ AND } (z::\text{int}) = x \text{ AND } (y \text{ AND } z)$
by (auto simp add: bin-eq-iff bin-nth-ops)

lemma *int-or-assoc*:

$(x \text{ OR } y) \text{ OR } (z::\text{int}) = x \text{ OR } (y \text{ OR } z)$
by (*auto simp add: bin-eq-iff bin-nth-ops*)

lemma *int-xor-assoc*:

$(x \text{ XOR } y) \text{ XOR } (z::\text{int}) = x \text{ XOR } (y \text{ XOR } z)$
by (*auto simp add: bin-eq-iff bin-nth-ops*)

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

lemma *bbw-lcs* [*simp*]:

$(y::\text{int}) \text{ AND } (x \text{ AND } z) = x \text{ AND } (y \text{ AND } z)$
 $(y::\text{int}) \text{ OR } (x \text{ OR } z) = x \text{ OR } (y \text{ OR } z)$
 $(y::\text{int}) \text{ XOR } (x \text{ XOR } z) = x \text{ XOR } (y \text{ XOR } z)$
by (*auto simp add: bin-eq-iff bin-nth-ops*)

lemma *bbw-not-dist*:

$!!y::\text{int}. \text{NOT } (x \text{ OR } y) = (\text{NOT } x) \text{ AND } (\text{NOT } y)$
 $!!y::\text{int}. \text{NOT } (x \text{ AND } y) = (\text{NOT } x) \text{ OR } (\text{NOT } y)$
by (*auto simp add: bin-eq-iff bin-nth-ops*)

lemma *bbw-oa-dist*:

$!!y \ z::\text{int}. (x \text{ AND } y) \text{ OR } z =$
 $(x \text{ OR } z) \text{ AND } (y \text{ OR } z)$
by (*auto simp add: bin-eq-iff bin-nth-ops*)

lemma *bbw-ao-dist*:

$!!y \ z::\text{int}. (x \text{ OR } y) \text{ AND } z =$
 $(x \text{ AND } z) \text{ OR } (y \text{ AND } z)$
by (*auto simp add: bin-eq-iff bin-nth-ops*)

11.1.4 Simplification with numerals

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

lemma *bin-rl-eqI*: $[[\text{bin-rest } x = \text{bin-rest } y; \text{bin-last } x = \text{bin-last } y]] \implies x = y$
by (*metis bin-rl-simp*)

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

$\text{bin-rest } (\text{neg-numeral } (\text{Num.BitM } w)) = \text{neg-numeral } w$
by (*simp only: BIT-bin-simps [symmetric] bin-rest-BIT*)

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

$\text{bin-last } (\text{neg-numeral } (\text{Num.BitM } w)) = 1$
by (*simp only: BIT-bin-simps [symmetric] bin-last-BIT*)

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

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

$\text{numeral } (\text{Num.Bit0 } x) \text{ AND numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ AND numeral } y) \text{ BIT } 0$
 $\text{numeral } (\text{Num.Bit0 } x) \text{ AND numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ AND numeral } y) \text{ BIT } 0$
 $\text{numeral } (\text{Num.Bit1 } x) \text{ AND numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ AND numeral } y) \text{ BIT } 0$
 $\text{numeral } (\text{Num.Bit1 } x) \text{ AND numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ AND numeral } y) \text{ BIT } 1$
 $\text{numeral } (\text{Num.Bit0 } x) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ AND neg-numeral } y) \text{ BIT } 0$
 $\text{numeral } (\text{Num.Bit0 } x) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ AND neg-numeral } (y + \text{Num.One})) \text{ BIT } 0$
 $\text{numeral } (\text{Num.Bit1 } x) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = (\text{numeral } x \text{ AND neg-numeral } y) \text{ BIT } 0$
 $\text{numeral } (\text{Num.Bit1 } x) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = (\text{numeral } x \text{ AND neg-numeral } (y + \text{Num.One})) \text{ BIT } 1$
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } x \text{ AND numeral } y) \text{ BIT } 0$
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } x \text{ AND numeral } y) \text{ BIT } 0$
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ AND numeral } y) \text{ BIT } 0$
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ AND numeral } y) \text{ BIT } 1$
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } x \text{ AND neg-numeral } y) \text{ BIT } 0$
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } x \text{ AND neg-numeral } (y + \text{Num.One})) \text{ BIT } 0$
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ AND neg-numeral } y) \text{ BIT } 0$
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = (\text{neg-numeral } (x + \text{Num.One}) \text{ AND neg-numeral } (y + \text{Num.One})) \text{ BIT } 1$
 $(1::\text{int}) \text{ AND numeral } (\text{Num.Bit0 } y) = 0$
 $(1::\text{int}) \text{ AND numeral } (\text{Num.Bit1 } y) = 1$
 $(1::\text{int}) \text{ AND neg-numeral } (\text{Num.Bit0 } y) = 0$
 $(1::\text{int}) \text{ AND neg-numeral } (\text{Num.Bit1 } y) = 1$
 $\text{numeral } (\text{Num.Bit0 } x) \text{ AND } (1::\text{int}) = 0$
 $\text{numeral } (\text{Num.Bit1 } x) \text{ AND } (1::\text{int}) = 1$
 $\text{neg-numeral } (\text{Num.Bit0 } x) \text{ AND } (1::\text{int}) = 0$
 $\text{neg-numeral } (\text{Num.Bit1 } x) \text{ AND } (1::\text{int}) = 1$
by (rule *bin-rl-eqI*, *simp*, *simp*)+

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

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

BIT 1

numeral (Num.Bit1 x) OR numeral (Num.Bit1 y) = (numeral x OR numeral y)

BIT 1

numeral (Num.Bit0 x) OR neg-numeral (Num.Bit0 y) = (numeral x OR neg-numeral y) BIT 0

numeral (Num.Bit0 x) OR neg-numeral (Num.Bit1 y) = (numeral x OR neg-numeral (y + Num.One)) BIT 1

numeral (Num.Bit1 x) OR neg-numeral (Num.Bit0 y) = (numeral x OR neg-numeral y) BIT 1

numeral (Num.Bit1 x) OR neg-numeral (Num.Bit1 y) = (numeral x OR neg-numeral (y + Num.One)) BIT 1

neg-numeral (Num.Bit0 x) OR numeral (Num.Bit0 y) = (neg-numeral x OR numeral y) BIT 0

neg-numeral (Num.Bit0 x) OR numeral (Num.Bit1 y) = (neg-numeral x OR numeral y) BIT 1

neg-numeral (Num.Bit1 x) OR numeral (Num.Bit0 y) = (neg-numeral (x + Num.One) OR numeral y) BIT 1

neg-numeral (Num.Bit1 x) OR numeral (Num.Bit1 y) = (neg-numeral (x + Num.One) OR numeral y) BIT 1

neg-numeral (Num.Bit0 x) OR neg-numeral (Num.Bit0 y) = (neg-numeral x OR neg-numeral y) BIT 0

neg-numeral (Num.Bit0 x) OR neg-numeral (Num.Bit1 y) = (neg-numeral x OR neg-numeral (y + Num.One)) BIT 1

neg-numeral (Num.Bit1 x) OR neg-numeral (Num.Bit0 y) = (neg-numeral (x + Num.One) OR neg-numeral y) BIT 1

neg-numeral (Num.Bit1 x) OR neg-numeral (Num.Bit1 y) = (neg-numeral (x + Num.One) OR neg-numeral (y + Num.One)) BIT 1

(1::int) OR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)

(1::int) OR numeral (Num.Bit1 y) = numeral (Num.Bit1 y)

(1::int) OR neg-numeral (Num.Bit0 y) = neg-numeral (Num.BitM y)

(1::int) OR neg-numeral (Num.Bit1 y) = neg-numeral (Num.Bit1 y)

numeral (Num.Bit0 x) OR (1::int) = numeral (Num.Bit1 x)

numeral (Num.Bit1 x) OR (1::int) = numeral (Num.Bit1 x)

neg-numeral (Num.Bit0 x) OR (1::int) = neg-numeral (Num.BitM x)

neg-numeral (Num.Bit1 x) OR (1::int) = neg-numeral (Num.Bit1 x)

by (rule bin-rl-eqI, simp, simp)+

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

numeral (Num.Bit0 x) XOR numeral (Num.Bit0 y) = (numeral x XOR numeral y) BIT 0

numeral (Num.Bit0 x) XOR numeral (Num.Bit1 y) = (numeral x XOR numeral y) BIT 1

numeral (Num.Bit1 x) XOR numeral (Num.Bit0 y) = (numeral x XOR numeral y) BIT 1

numeral (Num.Bit1 x) XOR numeral (Num.Bit1 y) = (numeral x XOR numeral y) BIT 0

numeral (Num.Bit0 x) XOR neg-numeral (Num.Bit0 y) = (numeral x XOR neg-numeral y) BIT 0

numeral (Num.Bit0 x) XOR neg-numeral (Num.Bit1 y) = (numeral x XOR

```

neg-numeral (y + Num.One)) BIT 1
  numeral (Num.Bit1 x) XOR neg-numeral (Num.Bit0 y) = (numeral x XOR
neg-numeral y) BIT 1
  numeral (Num.Bit1 x) XOR neg-numeral (Num.Bit1 y) = (numeral x XOR
neg-numeral (y + Num.One)) BIT 0
  neg-numeral (Num.Bit0 x) XOR numeral (Num.Bit0 y) = (neg-numeral x XOR
numeral y) BIT 0
  neg-numeral (Num.Bit0 x) XOR numeral (Num.Bit1 y) = (neg-numeral x XOR
numeral y) BIT 1
  neg-numeral (Num.Bit1 x) XOR numeral (Num.Bit0 y) = (neg-numeral (x +
Num.One) XOR numeral y) BIT 1
  neg-numeral (Num.Bit1 x) XOR numeral (Num.Bit1 y) = (neg-numeral (x +
Num.One) XOR numeral y) BIT 0
  neg-numeral (Num.Bit0 x) XOR neg-numeral (Num.Bit0 y) = (neg-numeral x
XOR neg-numeral y) BIT 0
  neg-numeral (Num.Bit0 x) XOR neg-numeral (Num.Bit1 y) = (neg-numeral x
XOR neg-numeral (y + Num.One)) BIT 1
  neg-numeral (Num.Bit1 x) XOR neg-numeral (Num.Bit0 y) = (neg-numeral (x
+ Num.One) XOR neg-numeral y) BIT 1
  neg-numeral (Num.Bit1 x) XOR neg-numeral (Num.Bit1 y) = (neg-numeral (x
+ Num.One) XOR neg-numeral (y + Num.One)) BIT 0
  (1::int) XOR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)
  (1::int) XOR numeral (Num.Bit1 y) = numeral (Num.Bit0 y)
  (1::int) XOR neg-numeral (Num.Bit0 y) = neg-numeral (Num.BitM y)
  (1::int) XOR neg-numeral (Num.Bit1 y) = neg-numeral (Num.Bit0 (y + Num.One))
  numeral (Num.Bit0 x) XOR (1::int) = numeral (Num.Bit1 x)
  numeral (Num.Bit1 x) XOR (1::int) = numeral (Num.Bit0 x)
  neg-numeral (Num.Bit0 x) XOR (1::int) = neg-numeral (Num.BitM x)
  neg-numeral (Num.Bit1 x) XOR (1::int) = neg-numeral (Num.Bit0 (x + Num.One))
by (rule bin-rl-eqI, simp, simp)+

```

11.1.5 Interactions with arithmetic

```

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

```

```

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

```

```

apply (induct y arbitrary: x rule: bin-induct)
  apply clarsimp
  apply clarsimp
apply (case-tac x rule: bin-exhaust)
apply (case-tac b)
  apply (case-tac [!] bit)
  apply (auto simp: le-Bits)
done

```

```

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

```

```

lemma add-BIT-simps [simp]:
  x BIT 0 + y BIT 0 = (x + y) BIT 0
  x BIT 0 + y BIT 1 = (x + y) BIT 1
  x BIT 1 + y BIT 0 = (x + y) BIT 1
  x BIT 1 + y BIT 1 = (x + y + 1) BIT 0
by (simp-all add: Bit-B0-2t Bit-B1-2t)

```

```

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

```

11.1.6 Truncating results of bit-wise operations

```

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

```

```

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

```

```

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

```

```

lemma bintr-bintr-i:
  x = bintrunc n y ==> bintrunc n x = bintrunc n y
by auto

```

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

11.2 Setting and clearing bits

primrec

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

where

Z: *bin-sc* 0 *b w* = *bin-rest w BIT b*

| *Suc*: *bin-sc* (*Suc n*) *b w* = *bin-sc n b (bin-rest w) BIT bin-last w*

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

bin-nth (bin-sc n b w) n = (*b* = 1)

by (*induct n arbitrary: w*) *auto*

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

bin-sc n c (bin-sc n b w) = *bin-sc n c w*

by (*induct n arbitrary: w*) *auto*

lemma *bin-sc-sc-diff*:

m ~ = *n* ==>

bin-sc m c (bin-sc n b w) = *bin-sc n b (bin-sc m c w)*

apply (*induct n arbitrary: w m*)

apply (*case-tac* [!] *m*)

apply *auto*

done

lemma *bin-nth-sc-gen*:

bin-nth (bin-sc n b w) m = (*if m = n then b = 1 else bin-nth w m*)

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

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

(bin-sc n (If (bin-nth w n) 1 0) w) = *w*

by (*induct n arbitrary: w*) *auto*

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

bin-sign (bin-sc n b w) = *bin-sign w*

by (*induct n arbitrary: w*) *auto*

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

bintrunc m (bin-sc n x (bintrunc m (w))) = *bintrunc m (bin-sc n x w)*

apply (*induct n arbitrary: w m*)

apply (*case-tac* [!] *w rule: bin-exhaust*)

apply (*case-tac* [!] *m, auto*)

done

lemma *bin-clr-le*:

```

bin-sc n 0 w <= w
apply (induct n arbitrary: w)
apply (case-tac [] w rule: bin-exhaust)
apply (auto simp: le-Bits)
done

```

```

lemma bin-set-ge:
bin-sc n 1 w >= w
apply (induct n arbitrary: w)
apply (case-tac [] w rule: bin-exhaust)
apply (auto simp: le-Bits)
done

```

```

lemma bintr-bin-clr-le:
bintrunc n (bin-sc m 0 w) <= bintrunc n w
apply (induct n arbitrary: w m)
apply simp
apply (case-tac w rule: bin-exhaust)
apply (case-tac m)
apply (auto simp: le-Bits)
done

```

```

lemma bintr-bin-set-ge:
bintrunc n (bin-sc m 1 w) >= bintrunc n w
apply (induct n arbitrary: w m)
apply simp
apply (case-tac w rule: bin-exhaust)
apply (case-tac m)
apply (auto simp: le-Bits)
done

```

```

lemma bin-sc-FP [simp]: bin-sc n 0 0 = 0
by (induct n) auto

```

```

lemma bin-sc-TM [simp]: bin-sc n 1 -1 = -1
by (induct n) auto

```

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

```

lemma bin-sc-minus:
0 < n ==> bin-sc (Suc (n - 1)) b w = bin-sc n b w
by auto

```

```

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

```

```

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

```

by (simp add: numeral-eq-Suc)

11.3 Splitting and concatenation

definition *bin-rcat* :: *nat* \Rightarrow *int list* \Rightarrow *int* **where**

bin-rcat *n* = *foldl* ($\lambda u v. \text{bin-cat } u \ n \ v$) 0

fun *bin-rsplit-aux* :: *nat* \Rightarrow *nat* \Rightarrow *int* \Rightarrow *int list* \Rightarrow *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* \Rightarrow *nat* \times *int* \Rightarrow *int list* **where**

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

fun *bin-rsplitl-aux* :: *nat* \Rightarrow *nat* \Rightarrow *int* \Rightarrow *int list* \Rightarrow *int list* **where**

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

definition *bin-rsplitl* :: *nat* \Rightarrow *nat* \times *int* \Rightarrow *int list* **where**

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

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

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

lemma *bin-sign-cat*:

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

lemma *bin-cat-Suc-Bit*:

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

lemma *bin-nth-cat*:

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

lemma *bin-nth-split*:

bin-split *n c* = (*a*, *b*) \implies
 (*ALL* *k. bin-nth* *a k* = *bin-nth* *c* (*n* + *k*)) &
 (*ALL* *k. bin-nth* *b k* = (*k* < *n* & *bin-nth* *c k*))
apply (induct *n* arbitrary: *b c*)

```

apply clarsimp
apply (clarsimp simp: Let-def split: ls-splits)
apply (case-tac k)
apply auto
done

```

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

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

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

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

lemma *bintr-cat*: $\text{bintrunc } m \ (\text{bin-cat } a \ n \ b) =$
 $\text{bin-cat } (\text{bintrunc } (m - n) \ a) \ n \ (\text{bintrunc } (\text{min } m \ n) \ b)$
by (*rule bin-eqI*) (*auto simp: bin-nth-cat nth-bintr*)

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

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

lemma *split-bintrunc*:
 $\text{bin-split } n \ c = (a , b) \implies b = \text{bintrunc } n \ c$
by (*induct n arbitrary: b c*) (*auto simp: Let-def split: ls-splits*)

lemma *bin-cat-split*:
 $\text{bin-split } n \ w = (u , v) \implies w = \text{bin-cat } u \ n \ v$
by (*induct n arbitrary: v w*) (*auto simp: Let-def split: ls-splits*)

lemma *bin-split-cat*:
 $\text{bin-split } n \ (\text{bin-cat } v \ n \ w) = (v , \text{bintrunc } n \ w)$
by (*induct n arbitrary: w*) *auto*

lemma *bin-split-zero* [*simp*]: $\text{bin-split } n \ 0 = (0 , 0)$
by (*induct n*) *auto*

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

lemma *bin-split-trunc*:
 $bin-split\ (min\ m\ n)\ c = (a, b) ==>$
 $bin-split\ n\ (bintrunc\ m\ c) = (bintrunc\ (m - n)\ a, b)$
apply (induct n arbitrary: m b c, clarsimp)
apply (simp add: bin-rest-trunc Let-def split: ls-splits)
apply (case-tac m)
apply (auto simp: Let-def split: ls-splits)
done

lemma *bin-split-trunc1*:
 $bin-split\ n\ c = (a, b) ==>$
 $bin-split\ n\ (bintrunc\ m\ c) = (bintrunc\ (m - n)\ a, bintrunc\ m\ b)$
apply (induct n arbitrary: m b c, clarsimp)
apply (simp add: bin-rest-trunc Let-def split: ls-splits)
apply (case-tac m)
apply (auto simp: Let-def split: ls-splits)
done

lemma *bin-cat-num*:
 $bin-cat\ a\ n\ b = a * 2^{\wedge} n + bintrunc\ n\ b$
apply (induct n arbitrary: b, clarsimp)
apply (simp add: Bit-def)
done

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

11.4 Miscellaneous lemmas

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

lemma *ex-eq-or*:

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

by *auto*

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*]: $\text{map2 } f \ [] \ ys = []$
unfolding *map2-def* **by** *auto*

lemma *map2-Nil2* [*simp*]: $\text{map2 } f \ xs \ [] = []$
unfolding *map2-def* **by** *auto*

lemma *map2-Cons* [*simp*]:
 $\text{map2 } f \ (x \# \ xs) \ (y \# \ ys) = f \ x \ y \# \ \text{map2 } f \ xs \ ys$
unfolding *map2-def* **by** *auto*

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*
by (*induct n*) (*auto simp: butlast-take*)

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

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

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

$0 < n \implies \text{bin-to-bl-aux } n \ 1 \ \text{bl} =$
 $\text{bin-to-bl-aux } (n - 1) \ 0 \ (\text{True} \ \# \ \text{bl})$
by (cases n) auto

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

$0 < n \implies \text{bin-to-bl-aux } n \ (w \ \text{BIT } b) \ \text{bl} =$
 $\text{bin-to-bl-aux } (n - 1) \ w \ ((b = 1) \ \# \ \text{bl})$
by (cases n) auto

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

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

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

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

Link between bin and bool list.

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

$\text{bl-to-bin-aux } (bs \ @ \ cs) \ w = \text{bl-to-bin-aux } cs \ (\text{bl-to-bin-aux } bs \ w)$
by (induct bs arbitrary: w) auto

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

$\text{bin-to-bl-aux } n \ w \ bs \ @ \ cs = \text{bin-to-bl-aux } n \ w \ (bs \ @ \ cs)$
by (induct n arbitrary: w bs) auto

lemma *bl-to-bin-append*:

$\text{bl-to-bin } (bs \ @ \ cs) = \text{bl-to-bin-aux } cs \ (\text{bl-to-bin } bs)$
unfolding *bl-to-bin-def* **by** (rule *bl-to-bin-aux-append*)

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

$\text{bin-to-bl-aux } n \ w \ bs = \text{bin-to-bl } n \ w \ @ \ bs$
unfolding *bin-to-bl-def* **by** (simp add : *bin-to-bl-aux-append*)

lemma *bin-to-bl-0* [simp]: $\text{bin-to-bl } 0 \ bs = []$

unfolding *bin-to-bl-def* **by** auto

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

$\text{size } (\text{bin-to-bl-aux } n \ w \ bs) = n + \text{length } bs$
by (induct n arbitrary: w bs) auto

lemma *size-bin-to-bl* [simp]: $\text{size } (\text{bin-to-bl } n \ w) = n$

unfolding *bin-to-bl-def* **by** (simp add : *size-bin-to-bl-aux*)

lemma *bin-bl-bin'*:

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

```

    bl-to-bin-aux bs (bintrunc n w)
  by (induct n arbitrary: w bs) (auto simp add : bl-to-bin-def)

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

lemma bl-bin-bl':
  bin-to-bl (n + length bs) (bl-to-bin-aux bs w) =
    bin-to-bl-aux n w bs
  apply (induct bs arbitrary: w n)
  apply auto
  apply (simp-all only : add-Suc [symmetric])
  apply (auto simp add : bin-to-bl-def)
  done

lemma bl-bin-bl [simp]: bin-to-bl (length bs) (bl-to-bin bs) = bs
  unfolding bl-to-bin-def
  apply (rule box-equals)
  apply (rule bl-bin-bl')
  prefer 2
  apply (rule bin-to-bl-aux.Z)
  apply simp
  done

lemma bl-to-bin-inj:
  bl-to-bin bs = bl-to-bin cs ==> length bs = length cs ==> bs = cs
  apply (rule-tac box-equals)
  defer
  apply (rule bl-bin-bl)
  apply (rule bl-bin-bl)
  apply simp
  done

lemma bl-to-bin-False [simp]: bl-to-bin (False # bl) = bl-to-bin bl
  unfolding bl-to-bin-def by auto

lemma bl-to-bin-Nil [simp]: bl-to-bin [] = 0
  unfolding bl-to-bin-def by auto

lemma bin-to-bl-zero-aux:
  bin-to-bl-aux n 0 bl = replicate n False @ bl
  by (induct n arbitrary: bl) (auto simp: replicate-app-Cons-same)

lemma bin-to-bl-zero: bin-to-bl n 0 = replicate n False
  unfolding bin-to-bl-def by (simp add: bin-to-bl-zero-aux)

lemma bin-to-bl-minus1-aux:
  bin-to-bl-aux n -1 bl = replicate n True @ bl
  by (induct n arbitrary: bl) (auto simp: replicate-app-Cons-same)

```

lemma *bin-to-bl-minus1*: $\text{bin-to-bl } n - 1 = \text{replicate } n \text{ True}$
unfolding *bin-to-bl-def* **by** (*simp add: bin-to-bl-minus1-aux*)

lemma *bl-to-bin-rep-F*:
 $\text{bl-to-bin } (\text{replicate } n \text{ False } @ \text{bl}) = \text{bl-to-bin } \text{bl}$
apply (*simp add: bin-to-bl-zero-aux [symmetric] bin-bl-bin'*)
apply (*simp add: bl-to-bin-def*)
done

lemma *bin-to-bl-trunc* [*simp*]:
 $n \leq m \implies \text{bin-to-bl } n (\text{bintrunc } m \ w) = \text{bin-to-bl } n \ w$
by (*auto intro: bl-to-bin-inj*)

lemma *bin-to-bl-aux-bintr*:
 $\text{bin-to-bl-aux } n (\text{bintrunc } m \ \text{bin}) \ \text{bl} =$
 $\text{replicate } (n - m) \ \text{False } @ \text{bin-to-bl-aux } (\min \ n \ m) \ \text{bin} \ \text{bl}$
apply (*induct n arbitrary: m bin bl*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac m*)
apply (*clarsimp simp: bin-to-bl-zero-aux*)
apply (*erule thin-rl*)
apply (*induct-tac n*)
apply *auto*
done

lemma *bin-to-bl-bintr*:
 $\text{bin-to-bl } n (\text{bintrunc } m \ \text{bin}) =$
 $\text{replicate } (n - m) \ \text{False } @ \text{bin-to-bl } (\min \ n \ m) \ \text{bin}$
unfolding *bin-to-bl-def* **by** (*rule bin-to-bl-aux-bintr*)

lemma *bl-to-bin-rep-False*: $\text{bl-to-bin } (\text{replicate } n \ \text{False}) = 0$
by (*induct n*) *auto*

lemma *len-bin-to-bl-aux*:
 $\text{length } (\text{bin-to-bl-aux } n \ w \ \text{bs}) = n + \text{length } \text{bs}$
by (*induct n arbitrary: w bs*) *auto*

lemma *len-bin-to-bl* [*simp*]: $\text{length } (\text{bin-to-bl } n \ w) = n$
by (*fact size-bin-to-bl*)

lemma *sign-bl-bin'*:
 $\text{bin-sign } (\text{bl-to-bin-aux } \text{bs} \ w) = \text{bin-sign } w$
by (*induct bs arbitrary: w*) *auto*

lemma *sign-bl-bin*: $\text{bin-sign } (\text{bl-to-bin } \text{bs}) = 0$
unfolding *bl-to-bin-def* **by** (*simp add : sign-bl-bin'*)

lemma *bl-sbin-sign-aux*:

```

hd (bin-to-bl-aux (Suc n) w bs) =
  (bin-sign (sbintrunc n w) = -1)
apply (induct n arbitrary: w bs)
apply clarsimp
apply (cases w rule: bin-exhaust)
apply (simp split add : bit.split)
apply clarsimp
done

```

lemma *bl-sbin-sign*:

```

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

```

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

```

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

```

lemma *bin-nth-of-bl*: $\text{bin-nth } (bl\text{-to-bin } bl) n = (n < \text{length } bl \ \& \ \text{rev } bl \ ! \ n)$
unfolding *bl-to-bin-def* **by** (*simp add : bin-nth-of-bl-aux*)

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

```

apply (induct n arbitrary: m w)
apply clarsimp
apply (case-tac m, clarsimp)
apply (clarsimp simp: bin-to-bl-def)
apply (simp add: bin-to-bl-aux-alt)
apply clarsimp
apply (case-tac m, clarsimp)
apply (clarsimp simp: bin-to-bl-def)
apply (simp add: bin-to-bl-aux-alt)
done

```

lemma *nth-rev*:

```

n < length xs  $\implies$  rev xs ! n = xs ! (length xs - 1 - n)
apply (induct xs)
apply simp
apply (clarsimp simp add : nth-append nth.simps split add : nat.split)
apply (rule-tac f =  $\lambda n. xs \ ! \ n$  in arg-cong)
apply arith
done

```

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

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* $bl ! (n - m)$)
apply (*induct m arbitrary: w n bl*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac w rule: bin-exhaust*)
apply *simp*
done

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

lemma *bl-to-bin-lt2p-aux*:
 $\text{bl-to-bin-aux } bs \ w < (w + 1) * (2 ^ \text{length } bs)$
apply (*induct bs arbitrary: w*)
apply *clarsimp*
apply *clarsimp*
apply *safe*
apply (*drule meta-spec, erule xtr8 [rotated], simp add: Bit-def*)
done

lemma *bl-to-bin-lt2p*: $\text{bl-to-bin } bs < (2 ^ \text{length } bs)$
apply (*unfold bl-to-bin-def*)
apply (*rule xtr1*)
prefer 2
apply (*rule bl-to-bin-lt2p-aux*)
apply *simp*
done

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

lemma *bl-to-bin-ge0*: $\text{bl-to-bin } bs \geq 0$
apply (*unfold bl-to-bin-def*)
apply (*rule xtr4*)
apply (*rule bl-to-bin-ge2p-aux*)
apply *simp*
done

lemma *butlast-rest-bin*:

```

butlast (bin-to-bl n w) = bin-to-bl (n - 1) (bin-rest w)
apply (unfold bin-to-bl-def)
apply (cases w rule: bin-exhaust)
apply (cases n, clarsimp)
apply clarsimp
apply (auto simp add: bin-to-bl-aux-alt)
done

```

lemma *butlast-bin-rest*:

```

butlast bl = bin-to-bl (length bl - Suc 0) (bin-rest (bl-to-bin bl))
using butlast-rest-bin [where w=bl-to-bin bl and n=length bl] by simp

```

lemma *butlast-rest-bl2bin-aux*:

```

bl ~ = []  $\implies$ 
  bl-to-bin-aux (butlast bl) w = bin-rest (bl-to-bin-aux bl w)
by (induct bl arbitrary: w) auto

```

lemma *butlast-rest-bl2bin*:

```

bl-to-bin (butlast bl) = bin-rest (bl-to-bin bl)
apply (unfold bl-to-bin-def)
apply (cases bl)
apply (auto simp add: butlast-rest-bl2bin-aux)
done

```

lemma *trunc-bl2bin-aux*:

```

bintrunc m (bl-to-bin-aux bl w) =
  bl-to-bin-aux (drop (length bl - m) bl) (bintrunc (m - length bl) w)
apply (induct bl arbitrary: w)
apply clarsimp
apply clarsimp
apply safe
apply (case-tac m - size bl)
apply (simp add : diff-is-0-eq [THEN iffD1, THEN Suc-diff-le])
apply simp
apply (rule-tac f = %nat. bl-to-bin-aux bl (bintrunc nat w BIT 1)
in arg-cong)
apply simp
apply (case-tac m - size bl)
apply (simp add: diff-is-0-eq [THEN iffD1, THEN Suc-diff-le])
apply simp
apply (rule-tac f = %nat. bl-to-bin-aux bl (bintrunc nat w BIT 0)
in arg-cong)
apply simp
done

```

lemma *trunc-bl2bin*:

```

bintrunc m (bl-to-bin bl) = bl-to-bin (drop (length bl - m) bl)

```

unfolding *bl-to-bin-def* **by** (*simp add : trunc-bl2bin-aux*)

lemma *trunc-bl2bin-len* [*simp*]:
 $\text{bintrunc } (\text{length } \text{bl}) (\text{bl-to-bin } \text{bl}) = \text{bl-to-bin } \text{bl}$
by (*simp add: trunc-bl2bin*)

lemma *bl2bin-drop*:
 $\text{bl-to-bin } (\text{drop } k \text{ bl}) = \text{bintrunc } (\text{length } \text{bl} - k) (\text{bl-to-bin } \text{bl})$
apply (*rule trans*)
prefer 2
apply (*rule trunc-bl2bin [symmetric]*)
apply (*cases k <= length bl*)
apply *auto*
done

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

lemma *take-rest-power-bin*:
 $m \leq n \implies \text{take } m (\text{bin-to-bl } n w) = \text{bin-to-bl } m ((\text{bin-rest } \wedge^{(n - m)}) w)$
apply (*rule nth-equalityI*)
apply *simp*
apply (*clarsimp simp add: nth-bin-to-bl nth-rest-power-bin*)
done

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

lemma *last-bin-last'*:
 $\text{size } xs > 0 \implies \text{last } xs = (\text{bin-last } (\text{bl-to-bin-aux } xs w) = 1)$
by (*induct xs arbitrary: w*) *auto*

lemma *last-bin-last*:
 $\text{size } xs > 0 \implies \text{last } xs = (\text{bin-last } (\text{bl-to-bin } xs) = 1)$
unfolding *bl-to-bin-def* **by** (*erule last-bin-last'*)

lemma *bin-last-last*:
 $\text{bin-last } w = (\text{if } \text{last } (\text{bin-to-bl } (\text{Suc } n) w) \text{ then } 1 \text{ else } 0)$
apply (*unfold bin-to-bl-def*)
apply *simp*
apply (*auto simp add: bin-to-bl-aux-alt*)
done

lemma *bl-xor-aux-bin*:

```
map2 (%x y. x ~ = y) (bin-to-bl-aux n v bs) (bin-to-bl-aux n w cs) =
  bin-to-bl-aux n (v XOR w) (map2 (%x y. x ~ = y) bs cs)
apply (induct n arbitrary: v w bs cs)
apply simp
apply (case-tac v rule: bin-exhaust)
apply (case-tac w rule: bin-exhaust)
apply clarsimp
apply (case-tac b)
apply (case-tac ba, safe, simp-all)+
done
```

lemma *bl-or-aux-bin*:

```
map2 (op | ) (bin-to-bl-aux n v bs) (bin-to-bl-aux n w cs) =
  bin-to-bl-aux n (v OR w) (map2 (op | ) bs cs)
apply (induct n arbitrary: v w bs cs)
apply simp
apply (case-tac v rule: bin-exhaust)
apply (case-tac w rule: bin-exhaust)
apply clarsimp
apply (case-tac b)
apply (case-tac ba, safe, simp-all)+
done
```

lemma *bl-and-aux-bin*:

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

lemma *bl-not-aux-bin*:

```
map Not (bin-to-bl-aux n w cs) =
  bin-to-bl-aux n (NOT w) (map Not cs)
apply (induct n arbitrary: w cs)
apply clarsimp
apply clarsimp
done
```

lemma *bl-not-bin*: $\text{map Not (bin-to-bl } n \ w) = \text{bin-to-bl } n \ (\text{NOT } w)$
unfolding *bin-to-bl-def* **by** (*simp add: bl-not-aux-bin*)

lemma *bl-and-bin*:

```
map2 (op ∧) (bin-to-bl n v) (bin-to-bl n w) = bin-to-bl n (v AND w)
unfolding bin-to-bl-def by (simp add: bl-and-aux-bin)
```

lemma *bl-or-bin*:

$map2 (op \vee) (bin-to-bl\ n\ v) (bin-to-bl\ n\ w) = bin-to-bl\ n\ (v\ OR\ w)$

unfolding *bin-to-bl-def* **by** (*simp add: bl-or-aux-bin*)

lemma *bl-xor-bin*:

$map2 (\lambda x\ y. x \neq y) (bin-to-bl\ n\ v) (bin-to-bl\ n\ w) = bin-to-bl\ n\ (v\ XOR\ w)$

unfolding *bin-to-bl-def* **by** (*simp only: bl-xor-aux-bin map2-Nil*)

lemma *drop-bin2bl-aux*:

$drop\ m\ (bin-to-bl-aux\ n\ bin\ bs) =$

$bin-to-bl-aux\ (n - m)\ bin\ (drop\ (m - n)\ bs)$

apply (*induct n arbitrary: m bin bs, clarsimp*)

apply *clarsimp*

apply (*case-tac bin rule: bin-exhaust*)

apply (*case-tac m <= n, simp*)

apply (*case-tac m - n, simp*)

apply *simp*

apply (*rule-tac f = %nat. drop nat bs in arg-cong*)

apply *simp*

done

lemma *drop-bin2bl*: $drop\ m\ (bin-to-bl\ n\ bin) = bin-to-bl\ (n - m)\ bin$

unfolding *bin-to-bl-def* **by** (*simp add : drop-bin2bl-aux*)

lemma *take-bin2bl-lem1*:

$take\ m\ (bin-to-bl-aux\ m\ w\ bs) = bin-to-bl\ m\ w$

apply (*induct m arbitrary: w bs, clarsimp*)

apply *clarsimp*

apply (*simp add: bin-to-bl-aux-alt*)

apply (*simp add: bin-to-bl-def*)

apply (*simp add: bin-to-bl-aux-alt*)

done

lemma *take-bin2bl-lem*:

$take\ m\ (bin-to-bl-aux\ (m + n)\ w\ bs) =$

$take\ m\ (bin-to-bl\ (m + n)\ w)$

apply (*induct n arbitrary: w bs*)

apply (*simp-all (no-asm) add: bin-to-bl-def take-bin2bl-lem1*)

apply *simp*

done

lemma *bin-split-take*:

$bin-split\ n\ c = (a, b) \implies$

$bin-to-bl\ m\ a = take\ m\ (bin-to-bl\ (m + n)\ c)$

apply (*induct n arbitrary: b c*)

apply *clarsimp*

apply (*clarsimp simp: Let-def split: ls-splits*)

apply (*simp add: bin-to-bl-def*)

apply (*simp add: take-bin2bl-lem*)

done

lemma *bin-split-take1*:

$k = m + n \implies \text{bin-split } n \ c = (a, b) \implies$
 $\text{bin-to-bl } m \ a = \text{take } m \ (\text{bin-to-bl } k \ c)$
by (*auto elim: bin-split-take*)

lemma *nth-takefill*: $m < n \implies$

$\text{takefill fill } n \ l \ ! \ m = (\text{if } m < \text{length } l \ \text{then } l \ ! \ m \ \text{else } \text{fill})$
apply (*induct n arbitrary: m l, clarsimp*)
apply *clarsimp*
apply (*case-tac m*)
apply (*simp split: list.split*)
apply (*simp split: list.split*)
done

lemma *takefill-alt*:

$\text{takefill fill } n \ l = \text{take } n \ l \ @ \ \text{replicate } (n - \text{length } l) \ \text{fill}$
by (*induct n arbitrary: l*) (*auto split: list.split*)

lemma *takefill-replicate* [*simp*]:

$\text{takefill fill } n \ (\text{replicate } m \ \text{fill}) = \text{replicate } n \ \text{fill}$
by (*simp add : takefill-alt replicate-add [symmetric]*)

lemma *takefill-le'*:

$n = m + k \implies \text{takefill } x \ m \ (\text{takefill } x \ n \ l) = \text{takefill } x \ m \ l$
by (*induct m arbitrary: l n*) (*auto split: list.split*)

lemma *length-takefill* [*simp*]: $\text{length } (\text{takefill fill } n \ l) = n$

by (*simp add : takefill-alt*)

lemma *take-takefill'*:

$!!w \ n. \ n = k + m \implies \text{take } k \ (\text{takefill fill } n \ w) = \text{takefill fill } k \ w$
by (*induct k*) (*auto split add : list.split*)

lemma *drop-takefill*:

$!!w. \ \text{drop } k \ (\text{takefill fill } (m + k) \ w) = \text{takefill fill } m \ (\text{drop } k \ w)$
by (*induct k*) (*auto split add : list.split*)

lemma *takefill-le* [*simp*]:

$m \leq n \implies \text{takefill } x \ m \ (\text{takefill } x \ n \ l) = \text{takefill } x \ m \ l$
by (*auto simp: le-iff-add takefill-le'*)

lemma *take-takefill* [*simp*]:

$m \leq n \implies \text{take } m \ (\text{takefill fill } n \ w) = \text{takefill fill } m \ w$
by (*auto simp: le-iff-add take-takefill'*)

lemma *takefill-append*:

$\text{takefill fill } (m + \text{length } xs) \ (xs \ @ \ w) = xs \ @ \ (\text{takefill fill } m \ w)$

by (*induct xs*) *auto*

lemma *takefill-same'*:

$l = \text{length } xs \implies \text{takefill fill } l \ xs = xs$

by *clarify (induct xs, auto)*

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

lemma *takefill-bintrunc*:

$\text{takefill False } n \ bl = \text{rev } (\text{bin-to-bl } n \ (\text{bl-to-bin } (\text{rev } bl)))$

apply (*rule nth-equalityI*)

apply *simp*

apply (*clarsimp simp: nth-takefill nth-rev nth-bin-to-bl bin-nth-of-bl*)

done

lemma *bl-bin-bl-rtf*:

$\text{bin-to-bl } n \ (\text{bl-to-bin } bl) = \text{rev } (\text{takefill False } n \ (\text{rev } bl))$

by (*simp add : takefill-bintrunc*)

lemma *bl-bin-bl-rep-drop*:

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

$\text{replicate } (n - \text{length } bl) \ \text{False} \ @ \ \text{drop } (\text{length } bl - n) \ bl$

by (*simp add: bl-bin-bl-rtf takefill-alt rev-take*)

lemma *tf-rev*:

$n + k = m + \text{length } bl \implies \text{takefill } x \ m \ (\text{rev } (\text{takefill } y \ n \ bl)) =$

$\text{rev } (\text{takefill } y \ m \ (\text{rev } (\text{takefill } x \ k \ (\text{rev } bl))))$

apply (*rule nth-equalityI*)

apply (*auto simp add: nth-takefill nth-rev*)

apply (*rule-tac f = %n. bl ! n in arg-cong*)

apply *arith*

done

lemma *takefill-minus*:

$0 < n \implies \text{takefill fill } (\text{Suc } (n - 1)) \ w = \text{takefill fill } n \ w$

by *auto*

lemmas *takefill-Suc-cases =*

list.cases [THEN takefill.Suc [THEN trans]]

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

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

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

takefill-minus [symmetric, THEN trans]]

lemma *takefill-numeral-Nil [simp]*:

$\text{takefill fill } (\text{numeral } k) \ [] = \text{fill } \# \ \text{takefill fill } (\text{pred-numeral } k) \ []$

by (*simp add: numeral-eq-Suc*)

lemma *takefill-numeral-Cons* [simp]:
takefill fill (numeral k) (x # xs) = x # takefill fill (pred-numeral k) xs
by (*simp add: numeral-eq-Suc*)

lemma *bl-to-bin-aux-cat*:
 $!!nv v. \text{bl-to-bin-aux } bs (\text{bin-cat } w \text{ } nv \text{ } v) =$
 $\text{bin-cat } w (nv + \text{length } bs) (\text{bl-to-bin-aux } bs \text{ } v)$
apply (*induct bs*)
apply *simp*
apply (*simp add: bin-cat-Suc-Bit [symmetric] del: bin-cat.simps*)
done

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

lemma *bl-to-bin-aux-alt*:
 $\text{bl-to-bin-aux } bs \text{ } w = \text{bin-cat } w (\text{length } bs) (\text{bl-to-bin } bs)$
using *bl-to-bin-aux-cat* [**where** $nv = 0$ **and** $v = 0$]
unfolding *bl-to-bin-def* [*symmetric*] **by** *simp*

lemma *bin-to-bl-cat*:
 $\text{bin-to-bl } (nv + nw) (\text{bin-cat } v \text{ } nw \text{ } w) =$
 $\text{bin-to-bl-aux } nv \text{ } v (\text{bin-to-bl } nw \text{ } w)$
unfolding *bin-to-bl-def* **by** (*simp add: bin-to-bl-aux-cat*)

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

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

lemma *bl-to-bin-app-cat*:
 $\text{bl-to-bin } (bsa @ bs) = \text{bin-cat } (\text{bl-to-bin } bsa) (\text{length } bs) (\text{bl-to-bin } bs)$
by (*simp only: bl-to-bin-aux-app-cat bl-to-bin-def*)

lemma *bin-to-bl-cat-app*:
 $\text{bin-to-bl } (n + nw) (\text{bin-cat } w \text{ } nw \text{ } wa) = \text{bin-to-bl } n \text{ } w @ \text{bin-to-bl } nw \text{ } wa$
by (*simp only: bin-to-bl-def bin-to-bl-aux-cat-app*)

lemma *bl-to-bin-app-cat-alt*:
 $\text{bin-cat } (\text{bl-to-bin } cs) n \text{ } w = \text{bl-to-bin } (cs @ \text{bin-to-bl } n \text{ } w)$
by (*simp add : bl-to-bin-app-cat*)

```

lemma mask-lem: (bl-to-bin (True # replicate n False)) =
  (bl-to-bin (replicate n True)) + 1
apply (unfold bl-to-bin-def)
apply (induct n)
apply simp
apply (simp only: Suc-eq-plus1 replicate-add
  append-Cons [symmetric] bl-to-bin-aux-append)
apply (simp add: Bit-B0-2t Bit-B1-2t)
done

```

```

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

```

```

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

```

```

lemma bl-of-nth-inj:
  (!!k. k < n ==> f k = g k) ==> bl-of-nth n f = bl-of-nth n g
by (induct n) auto

```

```

lemma bl-of-nth-nth-le:
  n ≤ length xs ==> bl-of-nth n (nth (rev xs)) = drop (length xs - n) xs
apply (induct n arbitrary: xs, clarsimp)
apply clarsimp
apply (rule trans [OF - hd-Cons-tl])
apply (frule Suc-le-lessD)
apply (simp add: nth-rev trans [OF drop-Suc drop-tl, symmetric])
apply (subst hd-drop-conv-nth)
apply force
apply simp-all
apply (rule-tac f = %n. drop n xs in arg-cong)
apply simp
done

```

```

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

```

```

lemma size-rbl-pred: length (rbl-pred bl) = length bl
by (induct bl) auto

```

```

lemma size-rbl-succ: length (rbl-succ bl) = length bl
by (induct bl) auto

```

lemma *size-rbl-add*:

!!*cl*. $\text{length (rbl-add bl cl)} = \text{length bl}$
by (*induct bl*) (*auto simp: Let-def size-rbl-succ*)

lemma *size-rbl-mult*:

!!*cl*. $\text{length (rbl-mult bl cl)} = \text{length bl}$
by (*induct bl*) (*auto simp add : Let-def size-rbl-add*)

lemmas *rbl-sizes* [*simp*] =

size-rbl-pred size-rbl-succ size-rbl-add size-rbl-mult

lemmas *rbl-Nils* =

rbl-pred.Nil rbl-succ.Nil rbl-add.Nil rbl-mult.Nil

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

$x \text{ BIT } 0 - 1 = (x - 1) \text{ BIT } 1$
 $x \text{ BIT } 1 - 1 = x \text{ BIT } 0$
by (*simp-all add: Bit-B0-2t Bit-B1-2t*)

lemma *rbl-pred*:

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

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

$x \text{ BIT } 0 + 1 = x \text{ BIT } 1$
 $x \text{ BIT } 1 + 1 = (x + 1) \text{ BIT } 0$
by (*simp-all add: Bit-B0-2t Bit-B1-2t*)

lemma *rbl-succ*:

$\text{rbl-succ (rev (bin-to-bl n bin))} = \text{rev (bin-to-bl n (bin + 1))}$
apply (*induct n arbitrary: bin, simp*)
apply (*unfold bin-to-bl-def*)
apply *clarsimp*
apply (*case-tac bin rule: bin-exhaust*)
apply (*case-tac b*)
apply (*clarsimp simp: bin-to-bl-aux-alt*) +
done

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

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

$x \text{ BIT } 1 + y \text{ BIT } 1 = (x + y + 1) \text{ BIT } 0$
by (*simp-all add: Bit-B0-2t Bit-B1-2t*)

lemma *rbl-add*:

!!*bina binb*. *rbl-add* (*rev* (*bin-to-bl n bina*)) (*rev* (*bin-to-bl n binb*)) =
rev (*bin-to-bl n (bina + binb)*)
apply (*induct n, simp*)
apply (*unfold bin-to-bl-def*)
apply *clarsimp*
apply (*case-tac bina rule: bin-exhaust*)
apply (*case-tac binb rule: bin-exhaust*)
apply (*case-tac b*)
apply (*case-tac [!] ba*)
apply (*auto simp: rbl-succ bin-to-bl-aux-alt Let-def add-ac*)
done

lemma *rbl-add-app2*:

!!*blb*. *length blb* >= *length bla* ==>
rbl-add bla (blb @ blc) = rbl-add bla blb
apply (*induct bla, simp*)
apply *clarsimp*
apply (*case-tac blb, clarsimp*)
apply (*clarsimp simp: Let-def*)
done

lemma *rbl-add-take2*:

!!*blb*. *length blb* >= *length bla* ==>
rbl-add bla (take (length bla) blb) = rbl-add bla blb
apply (*induct bla, simp*)
apply *clarsimp*
apply (*case-tac blb, clarsimp*)
apply (*clarsimp simp: Let-def*)
done

lemma *rbl-add-long*:

$m \geq n \implies \text{rbl-add} (\text{rev} (\text{bin-to-bl } n \text{ bina})) (\text{rev} (\text{bin-to-bl } m \text{ binb})) =$
 $\text{rev} (\text{bin-to-bl } n \text{ (bina + binb)})$
apply (*rule box-equals [OF - rbl-add-take2 rbl-add]*)
apply (*rule-tac f = rbl-add (rev (bin-to-bl n bina)) in arg-cong*)
apply (*rule rev-swap [THEN iffD1]*)
apply (*simp add: rev-take drop-bin2bl*)
apply *simp*
done

lemma *rbl-mult-app2*:

!!*blb*. *length blb* >= *length bla* ==>
rbl-mult bla (blb @ blc) = rbl-mult bla blb
apply (*induct bla, simp*)
apply *clarsimp*

```

apply (case-tac blb, clarsimp)
apply (clarsimp simp: Let-def rbl-add-app2)
done

```

```

lemma rbl-mult-take2:
  length blb >= length bla ==>
    rbl-mult bla (take (length bla) blb) = rbl-mult bla blb
apply (rule trans)
apply (rule rbl-mult-app2 [symmetric])
apply simp
apply (rule-tac f = rbl-mult bla in arg-cong)
apply (rule append-take-drop-id)
done

```

```

lemma rbl-mult-gt1:
  m >= length bl ==> rbl-mult bl (rev (bin-to-bl m binb)) =
    rbl-mult bl (rev (bin-to-bl (length bl) binb))
apply (rule trans)
apply (rule rbl-mult-take2 [symmetric])
apply simp-all
apply (rule-tac f = rbl-mult bl in arg-cong)
apply (rule rev-swap [THEN iffD1])
apply (simp add: rev-take drop-bin2bl)
done

```

```

lemma rbl-mult-gt:
  m > n ==> rbl-mult (rev (bin-to-bl n bina)) (rev (bin-to-bl m binb)) =
    rbl-mult (rev (bin-to-bl n bina)) (rev (bin-to-bl n binb))
by (auto intro: trans [OF rbl-mult-gt1])

```

```

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

```

```

lemma rbb1-Cons:
  b # rev (bin-to-bl n x) = rev (bin-to-bl (Suc n) (x BIT If b 1 0))
apply (unfold bin-to-bl-def)
apply simp
apply (simp add: bin-to-bl-aux-alt)
done

```

```

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
by (simp-all add: Bit-B0-2t Bit-B1-2t algebra-simps)

```

```

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))
apply (induct n)

```

```

apply simp
apply (unfold bin-to-bl-def)
apply clarsimp
apply (case-tac bina rule: bin-exhaust)
apply (case-tac binb rule: bin-exhaust)
apply (case-tac b)
apply (case-tac [!] ba)
  apply (auto simp: bin-to-bl-aux-alt Let-def)
  apply (auto simp: rbbl-Cons rbl-mult-Suc rbl-add)
done

```

lemma *rbl-add-split*:

```

P (rbl-add (y # ys) (x # xs)) =
  (ALL ws. length ws = length ys --> ws = rbl-add ys xs -->
    (y --> ((x --> P (False # rbl-succ ws)) & (~ x --> P (True # ws)))) &
    (~ y --> P (x # ws)))
apply (auto simp add: Let-def)
apply (case-tac [!] y)
apply auto
done

```

lemma *rbl-mult-split*:

```

P (rbl-mult (y # ys) xs) =
  (ALL ws. length ws = Suc (length ys) --> ws = False # rbl-mult ys xs -->
    (y --> P (rbl-add ws xs)) & (~ y --> P ws))
by (clarsimp simp add : Let-def)

```

lemma *and-len*: $xs = ys \implies xs = ys \ \& \ \text{length } xs = \text{length } ys$

by *auto*

lemma *size-if*: $\text{size } (\text{if } p \text{ then } xs \text{ else } ys) = (\text{if } p \text{ then } \text{size } xs \text{ else } \text{size } ys)$

by *auto*

lemma *tl-if*: $\text{tl } (\text{if } p \text{ then } xs \text{ else } ys) = (\text{if } p \text{ then } \text{tl } xs \text{ else } \text{tl } ys)$

by *auto*

lemma *hd-if*: $\text{hd } (\text{if } p \text{ then } xs \text{ else } ys) = (\text{if } p \text{ then } \text{hd } xs \text{ else } \text{hd } ys)$

by *auto*

lemma *if-Not-x*: $(\text{if } p \text{ then } \sim x \text{ else } x) = (p = (\sim x))$

by *auto*

lemma *if-x-Not*: $(\text{if } p \text{ then } x \text{ else } \sim x) = (p = x)$

by *auto*

lemma *if-same-and*: $(\text{If } p \ x \ y \ \& \ \text{If } p \ u \ v) = (\text{if } p \ \text{then } x \ \& \ u \ \text{else } y \ \& \ v)$

by *auto*

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

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

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

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

lemma *if-bool-simps*:
 $\text{If } p \ \text{True } y = (p \mid y) \ \& \ \text{If } p \ \text{False } y = (\sim p \ \& \ y) \ \&$
 $\text{If } p \ y \ \text{True} = (p \ \dashrightarrow y) \ \& \ \text{If } p \ y \ \text{False} = (p \ \& \ y)$
by *auto*

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

lemmas *seqr = eq-reflection [where $x = \text{size } w$] for w*

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

12.3 Repeated splitting or concatenation

lemma *sclem*:
 $\text{size } (\text{concat } (\text{map } (\text{bin-to-bl } n) \ xs)) = \text{length } xs * n$
by $(\text{induct } xs) \ \text{auto}$

lemma *bin-cat-foldl-lem*:
 $\text{foldl } (\%u. \ \text{bin-cat } u \ n) \ x \ xs =$
 $\text{bin-cat } x \ (\text{size } xs * n) \ (\text{foldl } (\%u. \ \text{bin-cat } u \ n) \ y \ xs)$
apply $(\text{induct } xs \ \text{arbitrary: } x)$
apply *simp*
apply $(\text{simp } (\text{no-asm}))$
apply $(\text{frule } \text{asm-rl})$
apply $(\text{drule } \text{meta-spec})$
apply $(\text{erule } \text{trans})$
apply $(\text{drule-tac } x = \text{bin-cat } y \ n \ \mathbf{in} \ \text{meta-spec})$
apply $(\text{simp } \text{add} : \text{bin-cat-assoc-sym } \text{min-max.inf-absorb2})$
done

lemma *bin-rcat-bl*:
 $(\text{bin-rcat } n \ wl) = \text{bl-to-bin } (\text{concat } (\text{map } (\text{bin-to-bl } n) \ wl))$

```

apply (unfold bin-rcat-def)
apply (rule sym)
apply (induct wl)
  apply (auto simp add : bl-to-bin-append)
  apply (simp add : bl-to-bin-aux-alt sclem)
  apply (simp add : bin-cat-foldl-lem [symmetric])
done

```

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

apply (induct n m c bs rule: *bin-rsplit-aux.induct*)

apply (subst *bin-rsplit-aux.simps*)

apply (subst *bin-rsplit-aux.simps*)

apply (*clarsimp split: ls-splits*)

apply *auto*

done

lemma *bin-rsplittl-aux-append*:

bin-rsplittl-aux n m c (bs @ cs) = bin-rsplittl-aux n m c bs @ cs

apply (induct n m c bs rule: *bin-rsplittl-aux.induct*)

apply (subst *bin-rsplittl-aux.simps*)

apply (subst *bin-rsplittl-aux.simps*)

apply (*clarsimp split: ls-splits*)

apply *auto*

done

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

lemma *bin-split-minus*: $0 < n \implies \text{bin-split } (\text{Suc } (n - 1)) w = \text{bin-split } n w$
 by *auto*

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

lemma *bin-split-pred-simp* [*simp*]:
 $(0::\text{nat}) < \text{numeral } \text{bin} \implies$
 $\text{bin-split } (\text{numeral } \text{bin}) w =$
 $(\text{let } (w1, w2) = \text{bin-split } (\text{numeral } \text{bin} - 1) (\text{bin-rest } w)$
 $\text{in } (w1, w2 \text{ BIT } \text{bin-last } w))$
 by (*simp only: bin-split-minus-simp*)

lemma *bin-rsplit-aux-simp-alt*:
 $\text{bin-rsplit-aux } n m c \text{ bs} =$
 $(\text{if } m = 0 \vee n = 0$
 $\text{then } \text{bs}$
 $\text{else let } (a, b) = \text{bin-split } n c \text{ in } \text{bin-rsplit } n (m - n, a) @ b \# \text{bs})$
unfolding *bin-rsplit-aux.simps* [of *n m c bs*]
apply *simp*
apply (*subst rsplit-aux-alts*)
apply (*simp add: bin-rsplit-def*)
done

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

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

lemma *bin-rsplit-size-sign'* [*rule-format*] :
 $\llbracket n > 0; \text{rev } \text{sw} = \text{bin-rsplit } n (\text{nw}, w) \rrbracket \implies$
 $(\text{ALL } v: \text{set } \text{sw}. \text{bintrunc } n v = v)$
apply (*induct sw arbitrary: nw w v*)
apply *clarsimp*
apply *clarsimp*
apply (*drule bthrs*)
apply (*simp (no-asm-use) add: Let-def split: ls-splits*)
apply *clarify*
apply (*drule split-bintrunc*)
apply *simp*
done

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

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

```

n > 0 ==> m < n ==> (ALL w k nw. rev sw = bin-rsplit n (nw, w) -->
  k < size sw --> bin-nth (sw ! k) m = bin-nth w (k * n + m))
apply (induct sw)
apply clarsimp
apply clarsimp
apply (drule bthrs)
apply (simp (no-asm-use) add: Let-def split: ls-splits)
apply clarify
apply (erule allE, erule impE, erule exI)
apply (case-tac k)
apply clarsimp
prefer 2
apply clarsimp
apply (erule allE)
apply (erule (1) impE)
apply (drule bin-nth-split, erule conjE, erule allE,
  erule trans, simp add : add-ac)
done

```

lemma *bin-rsplit-all*:

```

0 < nw ==> nw <= n ==> bin-rsplit n (nw, w) = [bintrunc n w]
unfolding bin-rsplit-def
by (clarsimp dest!: split-bintrunc simp: rsplit-aux-simp2ls split: ls-splits)

```

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

```

ALL bin. bin-rsplittl n (m, bin) = bin-rsplit n (m, bintrunc m bin)
apply (rule-tac a = m in wf-less-than [THEN wf-induct])
apply (simp (no-asm) add : bin-rsplittl-def bin-rsplit-def)
apply (rule allI)
apply (subst bin-rsplittl-aux.simps)
apply (subst bin-rsplit-aux.simps)
apply (clarsimp simp: Let-def split: ls-splits)
apply (drule bin-split-trunc)
apply (drule sym [THEN trans], assumption)
apply (subst rsplit-aux-alts(1))
apply (subst rsplit-aux-alts(2))
apply clarsimp
unfolding bin-rsplit-def bin-rsplittl-def
apply simp
done

```

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

```

n > 0 --> bin-rsplit n (n * size ws, bin-rcat n ws) = map (bintrunc n) ws
apply (unfold bin-rsplit-def bin-rcat-def)
apply (rule-tac xs = ws in rev-induct)
apply clarsimp
apply clarsimp
apply (subst rsplit-aux-alts)
unfolding bin-split-cat

```

apply *simp*
done

lemma *bin-rsplit-aux-len-le* [rule-format] :
 $\forall ws m. n \neq 0 \longrightarrow ws = \text{bin-rsplit-aux } n \text{ } nw \text{ } w \text{ } bs \longrightarrow$
 $\text{length } ws \leq m \iff nw + \text{length } bs * n \leq m * n$
apply (*induct* $n \text{ } nw \text{ } w \text{ } bs$ *rule: bin-rsplit-aux.induct*)
apply (*subst* *bin-rsplit-aux.simps*)
apply (*simp* *add: lrlem Let-def split: ls-splits*)
done

lemma *bin-rsplit-len-le*:
 $n \neq 0 \dashrightarrow ws = \text{bin-rsplit } n \text{ } (nw, w) \dashrightarrow (\text{length } ws \leq m) = (nw \leq m * n)$
unfolding *bin-rsplit-def* **by** (*clarsimp simp add : bin-rsplit-aux-len-le*)

lemma *bin-rsplit-aux-len*:
 $n \neq 0 \implies \text{length } (\text{bin-rsplit-aux } n \text{ } nw \text{ } w \text{ } cs) =$
 $(nw + n - 1) \text{ div } n + \text{length } cs$
apply (*induct* $n \text{ } nw \text{ } w \text{ } cs$ *rule: bin-rsplit-aux.induct*)
apply (*subst* *bin-rsplit-aux.simps*)
apply (*clarsimp simp: Let-def split: ls-splits*)
apply (*erule thin-rl*)
apply (*case-tac* m)
apply *simp*
apply (*case-tac* $m \leq n$)
apply *auto*
done

lemma *bin-rsplit-len*:
 $n \neq 0 \implies \text{length } (\text{bin-rsplit } n \text{ } (nw, w)) = (nw + n - 1) \text{ div } n$
unfolding *bin-rsplit-def* **by** (*clarsimp simp add : bin-rsplit-aux-len*)

lemma *bin-rsplit-aux-len-indep*:
 $n \neq 0 \implies \text{length } bs = \text{length } cs \implies$
 $\text{length } (\text{bin-rsplit-aux } n \text{ } nw \text{ } v \text{ } bs) =$
 $\text{length } (\text{bin-rsplit-aux } n \text{ } nw \text{ } w \text{ } cs)$
proof (*induct* $n \text{ } nw \text{ } w \text{ } cs$ *arbitrary: v bs rule: bin-rsplit-aux.induct*)
case ($1 \text{ } n \text{ } m \text{ } w \text{ } cs \text{ } v \text{ } bs$) **show** ?*case*
proof (*cases* $m = 0$)
case *True* **then show** ?*thesis* **using** $\langle \text{length } bs = \text{length } cs \rangle$ **by** *simp*
next
case *False*
from $1.\text{hyps } \langle m \neq 0 \rangle \langle n \neq 0 \rangle$ **have** *hyp*: $\bigwedge v \text{ } bs. \text{length } bs = \text{Suc } (\text{length } cs)$
 \implies
 $\text{length } (\text{bin-rsplit-aux } n \text{ } (m - n) \text{ } v \text{ } bs) =$
 $\text{length } (\text{bin-rsplit-aux } n \text{ } (m - n) \text{ } (\text{fst } (\text{bin-split } n \text{ } w)) \text{ } (\text{snd } (\text{bin-split } n \text{ } w) \#$
 $cs))$
by *auto*

```

show ?thesis using ⟨length bs = length cs⟩ ⟨n ≠ 0⟩
by (auto simp add: bin-rsplit-aux-simp-alt Let-def bin-rsplit-len
      split: ls-splits)
qed
qed

```

```

lemma bin-rsplit-len-indep:
  n≠0 ==> length (bin-rsplit n (nw, v)) = length (bin-rsplit n (nw, w))
apply (unfold bin-rsplit-def)
apply (simp (no-asm))
apply (erule bin-rsplit-aux-len-indep)
apply (rule refl)
done

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 by auto

```

```

lemma uint-nonnegative:
  0 ≤ uint w
using word.uint [of w] by simp

```

```

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

```

```

lemma uint-idem:
fixes w :: 'a::len0 word
shows uint w mod 2 ^ len-of TYPE('a) = uint w

```

using *uint-nonnegative uint-bounded* **by** (*rule mod-pos-pos-trivial*)

definition *word-of-int* :: *int* \Rightarrow *'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)*

by (*auto simp add: word-of-int-def intro: Abs-word-inverse*)

lemma *word-of-int-uint*:

word-of-int (*uint* *w*) = *w*

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

lemma *word-uint-eq-iff*:

a = *b* \longleftrightarrow *uint* *a* = *uint* *b*

by (*simp add: uint-inject*)

lemma *word-uint-eqI*:

uint *a* = *uint* *b* \implies *a* = *b*

by (*simp add: word-uint-eq-iff*)

13.2 Basic code generation setup

definition *Word* :: *int* \Rightarrow *'a::len0 word*

where

[*code-post*]: *Word* = *word-of-int*

lemma [*code abstype*]:

Word (*uint* *w*) = *w*

by (*simp add: Word-def word-of-int-uint*)

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

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

begin

definition *equal-word* :: *'a word* \Rightarrow *'a word* \Rightarrow *bool* **where**

equal-word *k* *l* \longleftrightarrow *HOL.equal* (*uint* *k*) (*uint* *l*)

instance proof

qed (*simp add: equal-equal-word-def word-uint-eq-iff*)

end

notation *fcomp* (**infixl** $\circ >$ 60)

notation *scomp* (**infixl** $\circ \rightarrow$ 60)

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

definition

random-word *i* = *Random.range* *i* $\circ \rightarrow$ (λk . *Pair* (
let *j* = *word-of-int* (*Code-Numeral.int-of* *k*) :: 'a *word*
in (*j*, $\lambda :: \text{unit}$. *Code-Evaluation.term-of* *j*)))

instance ..

end

no-notation *fcomp* (**infixl** $\circ >$ 60)

no-notation *scomp* (**infixl** $\circ \rightarrow$ 60)

13.3 Type conversions and casting

definition *sint* :: 'a :: *len word* \Rightarrow *int* **where**

— treats the most-significant-bit as a sign bit

sint-uint: *sint* *w* = *sbintrunc* (*len-of TYPE* ('a) - 1) (*uint* *w*)

definition *unat* :: 'a :: *len0 word* \Rightarrow *nat* **where**

unat *w* = *nat* (*uint* *w*)

definition *uints* :: *nat* \Rightarrow *int set* **where**

— the sets of integers representing the words

uints *n* = *range* (*bintrunc* *n*)

definition *sints* :: *nat* \Rightarrow *int set* **where**

sints *n* = *range* (*sbintrunc* (*n* - 1))

definition *unats* :: *nat* \Rightarrow *nat set* **where**

unats *n* = {*i*. *i* < $2 \wedge n$ }

definition *norm-sint* :: *nat* \Rightarrow *int* \Rightarrow *int* **where**

norm-sint *n* *w* = (*w* + $2 \wedge (n - 1)$) *mod* $2 \wedge n - 2 \wedge (n - 1)$

definition *scast* :: 'a :: *len word* \Rightarrow 'b :: *len word* **where**

— cast a word to a different length

scast *w* = *word-of-int* (*sint* *w*)

definition *ucast* :: 'a :: *len0 word* \Rightarrow 'b :: *len0 word* **where**

ucast *w* = *word-of-int* (*uint* *w*)

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

begin

definition

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

instance ..

end

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 \longleftrightarrow *source-size* c <= *target-size* c

definition *is-down* :: ('a :: len0 word => 'b :: len0 word) => bool **where**
is-down c \longleftrightarrow *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)
 by (fact xtr1 [OF word-size len-gt-0])

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}
 by (simp add: uints-def range-bintrunc)

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

lemma

uint-0:0 \leq *uint* *x* **and**
uint-lt: *uint* (*x*::*'a*::*len0* *word*) $<$ $2 \wedge \text{len-of TYPE('a)}$
by (*auto simp*: *uint* [*unfolded atLeastLessThan-iff*])

lemma *uint-mod-same*:

uint *x* *mod* $2 \wedge \text{len-of TYPE('a)}$ = *uint* (*x*::*'a*::*len0* *word*)
by (*simp add*: *int-mod-eq uint-lt uint-0*)

lemma *td-ext-uint*:

td-ext (*uint* :: *'a* *word* \Rightarrow *int*) *word-of-int* (*uints* (*len-of TYPE('a*::*len0*)))
 (%*w*::*int*. *w* *mod* $2 \wedge \text{len-of TYPE('a)}$)
apply (*unfold td-ext-def'*)
apply (*simp add*: *uints-num word-of-int-def bintrunc-mod2p*)
apply (*simp add*: *uint-mod-same uint-0 uint-lt*
 word.uint-inverse word.Abs-word-inverse int-mod-lem)
done

interpretation *word-uint*:

td-ext *uint*::*'a*::*len0* *word* \Rightarrow *int*
 word-of-int
 uints (*len-of TYPE('a*::*len0*))
 $\lambda w. w \text{ mod } 2 \wedge \text{len-of TYPE('a::*len0}*$)
by (*rule td-ext-uint*)

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*))
by (*rule td-ext-ubin*)

lemma *split-word-all*:

$(\bigwedge x::'a::\text{len0 } \text{word}. \text{PROP } P \ x) \equiv (\bigwedge x. \text{PROP } P \ (\text{word-of-int } x))$

proof

fix *x* :: *'a* *word*
assume $\bigwedge x. \text{PROP } P \ (\text{word-of-int } x)$
hence *PROP* *P* (*word-of-int* (*uint* *x*)).
thus *PROP* *P* *x* **by** *simp*

qed

13.5 Correspondence relation for theorem transfer

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

lemma *Quotient-word*:

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

unfolding *Quotient-alt-def cr-word-def*

by $(simp\ add: word\text{-}ubin.\text{norm}\text{-}eq\text{-}iff)$

lemma *reflp-word*:

reflp $(\lambda x\ y. bintrunc\ (len\text{-}of\ TYPE('a::len0))\ x = bintrunc\ (len\text{-}of\ TYPE('a))\ y)$

by $(simp\ add: reflp\text{-}def)$

setup-lifting *Quotient-word reflp-word*

TODO: The next lemma could be generated automatically.

lemma *uint-transfer* [*transfer-rule*]:

$(fun\text{-}rel\ cr\text{-}word\ op =) (bintrunc\ (len\text{-}of\ TYPE('a)))$

$(uint :: 'a::len0\ word \Rightarrow int)$

unfolding *fun-rel-def cr-word-def* **by** $(simp\ add: word\text{-}ubin.\text{eq}\text{-}norm)$

13.6 Arithmetic operations

lift-definition $word\text{-}succ :: 'a::len0\ word \Rightarrow 'a\ word\ \mathbf{is}\ \lambda x. x + 1$
by $(metis\ bintr\text{-}ariths(6))$

lift-definition $word\text{-}pred :: 'a::len0\ word \Rightarrow 'a\ word\ \mathbf{is}\ \lambda x. x - 1$
by $(metis\ bintr\text{-}ariths(7))$

instantiation $word :: (len0)\ \{neg\text{-}numeral, Divides.\text{div}, comm\text{-}monoid\text{-}mult, comm\text{-}ring\}$
begin

lift-definition $zero\text{-}word :: 'a\ word\ \mathbf{is}\ 0 .$

lift-definition $one\text{-}word :: 'a\ word\ \mathbf{is}\ 1 .$

lift-definition $plus\text{-}word :: 'a\ word \Rightarrow 'a\ word \Rightarrow 'a\ word\ \mathbf{is}\ op +$
by $(metis\ bintr\text{-}ariths(2))$

lift-definition $minus\text{-}word :: 'a\ word \Rightarrow 'a\ word \Rightarrow 'a\ word\ \mathbf{is}\ op -$
by $(metis\ bintr\text{-}ariths(3))$

lift-definition $uminus\text{-}word :: 'a\ word \Rightarrow 'a\ word\ \mathbf{is}\ uminus$
by $(metis\ bintr\text{-}ariths(5))$

lift-definition $times\text{-}word :: 'a\ word \Rightarrow 'a\ word \Rightarrow 'a\ word\ \mathbf{is}\ op *$
by $(metis\ bintr\text{-}ariths(4))$

definition

word-div-def: $a \text{ div } b = \text{word-of-int } (\text{uint } a \text{ div } \text{uint } b)$

definition

word-mod-def: $a \text{ mod } b = \text{word-of-int } (\text{uint } a \text{ mod } \text{uint } b)$

instance

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

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: $1 = \text{word-of-int } 1$

unfolding *plus-word-def minus-word-def times-word-def uminus-word-def*

unfolding *word-succ-def word-pred-def zero-word-def one-word-def*

by *simp-all*

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: $\text{word-pred } (\text{word-of-int } a) = \text{word-of-int } (a - 1)$

by (*transfer*, *simp*)⁺

lemmas *wi-hom-syms* = *wi-homs* [*symmetric*]**lemmas** *word-of-int-homs* = *wi-homs word-0-wi word-1-wi***lemmas** *word-of-int-hom-syms* = *word-of-int-homs* [*symmetric*]**instance** *word* :: (*len*) *comm-ring-1***proof**

have $0 < \text{len-of } \text{TYPE}'a$ **by** (*rule len-gt-0*)

then show $(0::'a \text{ word}) \neq 1$

by $-$ (*transfer, auto simp add: gr0-conv-Suc*)
qed

lemma *word-of-nat*: $of\text{-}nat\ n = word\text{-}of\text{-}int\ (int\ n)$
by (*induct n*) (*auto simp add : word-of-int-hom-syms*)

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

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

13.7 Ordering

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

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

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

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

end

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

definition *word-sless* :: $'a :: len\ word \Rightarrow 'a\ word \Rightarrow bool$ ($(-/ <s -)$ [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* :: $'a\ word \Rightarrow 'a\ word\ is\ bitNOT$
by (*metis bin-trunc-not*)

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

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

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

definition

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

definition

word-set-bit-def: *set-bit a n x =*
word-of-int (bin-sc n (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 *shiffl1* :: 'a word \Rightarrow 'a word **where**

shiffl1 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

shiffl-def: *w << n = (shiffl1 ^^ n) w*

definition

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

instance ..

end

lemma [*code*]: **shows**

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

word-and-def: *(a::'a word) AND b = word-of-int (uint a AND uint b)* **and**

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

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

unfolding *bitNOT-word-def bitAND-word-def bitOR-word-def bitXOR-word-def*

by *simp-all*

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

begin

definition

word-msb-def:

$msb\ a \longleftrightarrow bin\text{-}sign\ (sint\ a) = -1$

instance ..

end

definition $setBit :: 'a :: len0\ word \Rightarrow nat \Rightarrow 'a\ word$ **where**
 $setBit\ w\ n = set\text{-}bit\ w\ n\ True$

definition $clearBit :: 'a :: len0\ word \Rightarrow nat \Rightarrow 'a\ word$ **where**
 $clearBit\ w\ n = set\text{-}bit\ w\ n\ False$

13.9 Shift operations

definition $sshiftr1 :: 'a :: len\ word \Rightarrow 'a\ word$ **where**
 $sshiftr1\ w = word\text{-}of\text{-}int\ (bin\text{-}rest\ (sint\ w))$

definition $bshiftr1 :: bool \Rightarrow 'a :: len\ word \Rightarrow 'a\ word$ **where**
 $bshiftr1\ b\ w = of\text{-}bl\ (b\ \# \text{butlast}\ (to\text{-}bl\ w))$

definition $sshiftr :: 'a :: len\ word \Rightarrow nat \Rightarrow 'a\ word$ (**infixl** $>>>$ 55) **where**
 $w\ >>>\ n = (sshiftr1\ \wedge\wedge\ n)\ w$

definition $mask :: nat \Rightarrow 'a::len\ word$ **where**
 $mask\ n = (1\ \ll\ n) - 1$

definition $revcast :: 'a :: len0\ word \Rightarrow 'b :: len0\ word$ **where**
 $revcast\ w = of\text{-}bl\ (takefill\ False\ (len\text{-}of\ TYPE('b))\ (to\text{-}bl\ w))$

definition $slice1 :: nat \Rightarrow 'a :: len0\ word \Rightarrow 'b :: len0\ word$ **where**
 $slice1\ n\ w = of\text{-}bl\ (takefill\ False\ n\ (to\text{-}bl\ w))$

definition $slice :: nat \Rightarrow 'a :: len0\ word \Rightarrow 'b :: len0\ word$ **where**
 $slice\ n\ w = slice1\ (size\ w - n)\ w$

13.10 Rotation

definition $rotater1 :: 'a\ list \Rightarrow 'a\ list$ **where**
 $rotater1\ ys =$
 $(case\ ys\ of\ [] \Rightarrow [] \mid x\ \#\ xs \Rightarrow last\ ys\ \# \text{butlast}\ ys)$

definition $rotater :: nat \Rightarrow 'a\ list \Rightarrow 'a\ list$ **where**
 $rotater\ n = rotater1\ \wedge\wedge\ n$

definition $word\text{-}rotr :: nat \Rightarrow 'a :: len0\ word \Rightarrow 'a :: len0\ word$ **where**
 $word\text{-}rotr\ n\ w = of\text{-}bl\ (rotater\ n\ (to\text{-}bl\ w))$

definition $word\text{-}rotl :: nat \Rightarrow 'a :: len0\ word \Rightarrow 'a :: len0\ word$ **where**
 $word\text{-}rotl\ n\ w = of\text{-}bl\ (rotate\ n\ (to\text{-}bl\ w))$

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*)
unfolding *sint-uint*
by (*auto simp: word-ubin.eq-norm sbintrunc-bintrunc-lt*)

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

lemma *bintr-uint*:
fixes *w* :: '*a*::*len0 word*
shows $\text{len-of TYPE}('a) \leq n \implies \text{bintrunc } n \text{ (uint } w) = \text{uint } w$
apply (*subst word-ubin.norm-Rep [symmetric]*)

```

apply (simp only: bintrunc-bintrunc-min word-size)
apply (simp add: min-max.inf-absorb2)
done

```

lemma *wi-bintr*:

```

  len-of TYPE('a::len0) ≤ n ⇒
    word-of-int (bintrunc n w) = (word-of-int w :: 'a word)
by (clarsimp simp add: word-ubin.norm-eq-iff [symmetric] min-max.inf-absorb1)

```

lemma *td-ext-sbin*:

```

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

```

lemmas *td-ext-sint = td-ext-sbin*

```

  [simplified len-gt-0 no-sbintr-alt2 Suc-pred' [symmetric]]

```

interpretation *word-sint*:

```

  td-ext sint :: 'a::len word => int
    word-of-int
    sints (len-of TYPE('a::len))
    %w. (w + 2^(len-of TYPE('a::len) - 1)) mod 2^len-of TYPE('a::len) -
      2^(len-of TYPE('a::len) - 1)
by (rule td-ext-sint)

```

interpretation *word-sbin*:

```

  td-ext sint :: 'a::len word => int
    word-of-int
    sints (len-of TYPE('a::len))
    sbintrunc (len-of TYPE('a::len) - 1)
by (rule td-ext-sbin)

```

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

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

lemma *to-bl-def'*:

```

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

```

by (*auto simp: to-bl-def*)

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

lemma *uints-mod*: *uints n = range (λw. w mod 2 ^ n)*

by (*fact uints-def [unfolded no-bintr-alt1]*)

lemma *word-numeral-alt*:

numeral b = word-of-int (numeral b)

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

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

lemma *word-neg-numeral-alt*:

neg-numeral b = word-of-int (neg-numeral b)

by (*simp only: neg-numeral-def word-numeral-alt wi-hom-neg*)

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*

unfolding *fun-rel-def cr-word-def word-numeral-alt word-neg-numeral-alt*

by *simp-all*

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

uint (numeral bin :: 'a word) =

bintrunc (len-of TYPE ('a :: len0)) (numeral bin)

unfolding *word-numeral-alt* **by** (*rule word-ubin.eq-norm*)

lemma *uint-bintrunc-neg* [*simp*]: *uint (neg-numeral bin :: 'a word) =*

bintrunc (len-of TYPE ('a :: len0)) (neg-numeral bin)

by (*simp only: word-neg-numeral-alt word-ubin.eq-norm*)

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

sint (numeral bin :: 'a word) =

sbintrunc (len-of TYPE ('a :: len) - 1) (numeral bin)

by (*simp only: word-numeral-alt word-sbin.eq-norm*)

lemma *sint-sbintrunc-neg* [*simp*]: *sint (neg-numeral bin :: 'a word) =*

sbintrunc (len-of TYPE ('a :: len) - 1) (neg-numeral bin)

by (*simp only: word-neg-numeral-alt word-sbin.eq-norm*)

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

unat (numeral bin :: 'a :: len0 word) =

nat (bintrunc (len-of TYPE ('a)) (numeral bin))

by (*simp only: unat-def uint-bintrunc*)

lemma *unat-bintrunc-neg* [*simp*]:

```

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

```

```

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

```

```

lemma uint-ge-0 [iff]: 0  $\leq$  uint (x::'a::len0 word)
  using word-uint.Rep [of x] by (simp add: uints-num)

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma uint-m2p-not-non-neg:
   $\neg$  0  $\leq$  uint (x::'a::len0 word) - 2 ^ len-of TYPE('a)
  by (simp only: not-le uint-m2p-neg)

```

```

lemma lt2p-lem:
  len-of TYPE('a)  $\leq$  n  $\implies$  uint (w :: 'a :: len0 word) < 2 ^ n
  by (rule xtr8 [OF - uint-lt2p]) simp

```

```

lemma uint-le-0-iff [simp]: uint x  $\leq$  0  $\iff$  uint x = 0
  by (fact uint-ge-0 [THEN leD, THEN linorder-antisym-conv1])

```

```

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

```

```

lemma uint-numeral:
  uint (numeral b :: 'a :: len0 word) = numeral b mod 2 ^ len-of TYPE('a)
  unfolding word-numeral-alt

```

by (*simp only: int-word-uint*)

lemma *uint-neg-numeral*:

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

unfolding *word-neg-numeral-alt*

by (*simp only: int-word-uint*)

lemma *unat-numeral*:

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

apply (*unfold unat-def*)

apply (*clarsimp simp only: uint-numeral*)

apply (*rule nat-mod-distrib [THEN trans]*)

apply (*rule zero-le-numeral*)

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

done

lemma *sint-numeral*: *sint* (*numeral* *b :: 'a :: len word*) = (*numeral* *b +*

$2 ^ (\text{len-of TYPE('a)} - 1) \text{ mod } 2 ^ \text{len-of TYPE('a)} -$

$2 ^ (\text{len-of TYPE('a)} - 1)$

unfolding *word-numeral-alt* **by** (*rule int-word-sint*)

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

unfolding *word-0-wi* ..

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

unfolding *word-1-wi* ..

lemma *word-of-int-numeral* [*simp*] :

(*word-of-int* (*numeral* *bin*) :: '*a* :: *len0 word*) = (*numeral* *bin*)

unfolding *word-numeral-alt* ..

lemma *word-of-int-neg-numeral* [*simp*]:

(*word-of-int* (*neg-numeral* *bin*) :: '*a* :: *len0 word*) = (*neg-numeral* *bin*)

unfolding *neg-numeral-def word-numeral-alt wi-hom-syms* ..

lemma *word-int-case-wi*:

word-int-case *f* (*word-of-int* *i :: 'b word*) =

f (*i mod 2 ^ len-of TYPE('b :: len0)*)

unfolding *word-int-case-def* **by** (*simp add: word-uint.eq-norm*)

lemma *word-int-split*:

P (*word-int-case* *f* *x*) =

(*ALL* *i. x = (word-of-int* *i :: 'b :: len0 word)* &

$0 \leq i \ \& \ i < 2 ^ \text{len-of TYPE('b)} \ \longrightarrow P (f \ i)$)

unfolding *word-int-case-def*

by (*auto simp: word-uint.eq-norm int-mod-eq*)

lemma *word-int-split-asm*:

P (*word-int-case* *f* *x*) =

(\sim (EX n . $x = (\text{word-of-int } n :: 'b::\text{len0 } \text{word}) \ \&$
 $0 \leq n \ \& \ n < 2 \wedge \text{len-of } \text{TYPE}('b::\text{len0}) \ \& \ \sim P (f \ n)$))

unfolding *word-int-case-def*

by (*auto simp: word-uint.eq-norm int-mod-eq'*)

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*: $0 \leq \text{uint } w \ \& \ \text{uint } w < 2 \wedge \text{size } w$

unfolding *word-size* **by** (*rule uint-range'*)

lemma *sint-range-size*:

$-(2 \wedge (\text{size } w - \text{Suc } 0)) \leq \text{sint } w \ \& \ \text{sint } w < 2 \wedge (\text{size } w - \text{Suc } 0)$

unfolding *word-size* **by** (*rule sint-range'*)

lemma *sint-above-size*: $2 \wedge (\text{size } (w::'a::\text{len } \text{word}) - 1) \leq x \implies \text{sint } w < x$

unfolding *word-size* **by** (*rule less-le-trans [OF sint-lt]*)

lemma *sint-below-size*:

$x \leq -(2 \wedge (\text{size } (w::'a::\text{len } \text{word}) - 1)) \implies x \leq \text{sint } w$

unfolding *word-size* **by** (*rule order-trans [OF -sint-ge]*)

13.13 Testing bits

lemma *test-bit-eq-iff*: $(\text{test-bit } (u::'a::\text{len0 } \text{word}) = \text{test-bit } v) = (u = v)$

unfolding *word-test-bit-def* **by** (*simp add: bin-nth-eq-iff*)

lemma *test-bit-size* [*rule-format*]: $(w::'a::\text{len0 } \text{word}) !! n \dashrightarrow n < \text{size } w$

apply (*unfold word-test-bit-def*)

apply (*subst word-ubin.norm-Rep [symmetric]*)

apply (*simp only: nth-bintr word-size*)

apply *fast*

done

lemma *word-eq-iff*:

fixes $x \ y :: 'a::\text{len0 } \text{word}$

shows $x = y \longleftrightarrow (\forall n < \text{len-of } \text{TYPE}('a). \ x !! n = y !! n)$

unfolding *uint-inject [symmetric] bin-eq-iff word-test-bit-def [symmetric]*

by (*metis test-bit-size [unfolded word-size]*)

lemma *word-eqI* [*rule-format*]:

fixes $u :: 'a::\text{len0 } \text{word}$

shows $(\text{ALL } n. \ n < \text{size } u \dashrightarrow u !! n = v !! n) \implies u = v$

by (*simp add: word-size word-eq-iff*)

lemma *word-eqD*: $(u::'a::\text{len0 } \text{word}) = v \implies u !! x = v !! x$

by *simp*

lemma *test-bit-bin'*: $w !! n = (n < \text{size } w \ \& \ \text{bin-nth } (\text{uint } w) \ n)$

unfolding *word-test-bit-def word-size*
by (*simp add: nth-bintr [symmetric]*)

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

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

lemma *bin-nth-sint:*
fixes *w :: 'a::len word*
shows *len-of TYPE('a) \leq n \implies*
bin-nth (sint w) n = bin-nth (sint w) (len-of TYPE('a) - 1)
apply (*subst word-sbin.norm-Rep [symmetric]*)
apply (*auto simp add: nth-sbintr*)
done

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

interpretation *word-bl:*
type-definition to-bl :: 'a::len0 word => bool list
of-bl
{bl. length bl = len-of TYPE('a::len0)}
by (*rule td-bl*)

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

lemma *word-size-bl: size w = size (to-bl w)*
unfolding *word-size* **by** *auto*

lemma *to-bl-use-of-bl:*
(to-bl w = bl) = (w = of-bl bl \wedge length bl = length (to-bl w))
by (*fastforce elim!: word-bl.Abs-inverse [unfolded mem-Collect-eq]*)

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

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

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

lemma *word-rev-gal'*: *u* = *word-reverse* *w* \implies *w* = *word-reverse* *u*
by *simp*

lemma *length-bl-gt-0* [*iff*]: $0 < \text{length} (\text{to-bl } (x::'a::\text{len } \text{word}))$
unfolding *word-bl-Rep'* **by** (*rule len-gt-0*)

lemma *bl-not-Nil* [*iff*]: $\text{to-bl } (x::'a::\text{len } \text{word}) \neq []$
by (*fact length-bl-gt-0* [*unfolded length-greater-0-conv*])

lemma *length-bl-neq-0* [*iff*]: $\text{length} (\text{to-bl } (x::'a::\text{len } \text{word})) \neq 0$
by (*fact length-bl-gt-0* [*THEN gr-implies-not0*])

lemma *hd-bl-sign-sint*: $\text{hd} (\text{to-bl } w) = (\text{bin-sign } (\text{sint } w) = -1)$
apply (*unfold to-bl-def sint-uint*)
apply (*rule trans* [*OF* - *bl-sbin-sign*])
apply *simp*
done

lemma *of-bl-drop'*:
 $\text{lend} = \text{length } \text{bl} - \text{len-of TYPE } ('a :: \text{len0}) \implies$
 $\text{of-bl } (\text{drop } \text{lend } \text{bl}) = (\text{of-bl } \text{bl} :: 'a \text{ word})$
apply (*unfold of-bl-def*)
apply (*clarsimp simp add* : *trunc-bl2bin* [*symmetric*])
done

lemma *test-bit-of-bl*:
 $(\text{of-bl } \text{bl}::'a::\text{len0 } \text{word}) !! n = (\text{rev } \text{bl} ! n \wedge n < \text{len-of TYPE } ('a) \wedge n < \text{length } \text{bl})$
apply (*unfold of-bl-def word-test-bit-def*)
apply (*auto simp add*: *word-size word-ubin.eq-norm nth-bintr bin-nth-of-bl*)
done

lemma *no-of-bl*:
 $(\text{numeral } \text{bin} :: 'a::\text{len0 } \text{word}) = \text{of-bl } (\text{bin-to-bl } (\text{len-of TYPE } ('a)) (\text{numeral } \text{bin}))$
unfolding *of-bl-def* **by** *simp*

lemma *uint-bl*: $\text{to-bl } w = \text{bin-to-bl } (\text{size } w) (\text{uint } w)$
unfolding *word-size to-bl-def* **by** *auto*

lemma *to-bl-bin*: $\text{bl-to-bin } (\text{to-bl } w) = \text{uint } w$
unfolding *uint-bl* **by** (*simp add* : *word-size*)

lemma *to-bl-of-bin*:

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

lemma *to-bl-numeral* [*simp*]:

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

lemma *to-bl-neg-numeral* [*simp*]:

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

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

unfolding *uint-bl* **by** (*simp add : word-size*)

lemma *uint-bl-bin*:

fixes *x* :: 'a::len0 *word*
shows *bl-to-bin* (*bin-to-bl* (*len-of TYPE*('a)) (*uint x*)) = *uint x*
by (*rule trans [OF bin-bl-bin word-ubin.norm-Rep]*)

lemma *uints-unats*: *uints n* = *int* ‘ *unats n*

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

lemma *unats-uints*: *unats n* = *nat* ‘ *uints n*

by (*auto simp add : uints-unats image-iff*)

lemmas *bintr-num* = *word-ubin.norm-eq-iff*

[*of numeral a numeral b, symmetric, folded word-numeral-alt*] **for** *a b*

lemmas *sbintr-num* = *word-sbin.norm-eq-iff*

[*of numeral a numeral b, symmetric, folded word-numeral-alt*] **for** *a b*

lemma *num-of-bintr'*:

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

lemma *num-of-sbintr'*:

sbintrunc (*len-of TYPE*('a :: len) - 1) (*numeral a*) = (*numeral b*) \implies

numeral a = (numeral b :: 'a word)
unfolding *sbintr-num* **by** (*erule subst, simp*)

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

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

lemma *ucast-id*: *ucast w = w*
unfolding *ucast-def* **by** *auto*

lemma *scast-id*: *scast w = w*
unfolding *scast-def* **by** *auto*

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

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

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

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

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

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

unfolding *target-size-def word-size Let-def ..*

lemma *is-down:*

fixes $c :: 'a::len0\ word \Rightarrow 'b::len0\ word$
shows $is_down\ c \longleftrightarrow len_of\ TYPE('b) \leq len_of\ TYPE('a)$
unfolding *is-down-def source-size target-size ..*

lemma *is-up:*

fixes $c :: 'a::len0\ word \Rightarrow 'b::len0\ word$
shows $is_up\ c \longleftrightarrow len_of\ TYPE('a) \leq len_of\ TYPE('b)$
unfolding *is-up-def source-size target-size ..*

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

lemma *down-cast-same [OF refl]: uc = ucast \implies is-down uc \implies uc = scast*

apply (*unfold is-down*)
apply *safe*
apply (*rule ext*)
apply (*unfold ucast-def scast-def uint-sint*)
apply (*rule word-ubin.norm-eq-iff [THEN iffD1]*)
apply *simp*
done

lemma *word-rev-tf:*

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

lemma *word-rep-drop:*

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

lemma *to-bl-ucast:*

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

lemma *ucast-up-app [OF refl]:*

$uc = ucast \implies source_size\ uc + n = target_size\ uc \implies$
 $to_bl\ (uc\ w) = replicate\ n\ False\ @\ (to_bl\ w)$
by (*auto simp add : source-size target-size to-bl-ucast*)

lemma *ucast-down-drop* [*OF refl*]:
 $uc = ucast \implies source\text{-}size\ uc = target\text{-}size\ uc + n \implies$
 $to\text{-}bl\ (uc\ w) = drop\ n\ (to\text{-}bl\ w)$
by (*auto simp add : source-size target-size to-bl-ucast*)

lemma *scast-down-drop* [*OF refl*]:
 $sc = scast \implies source\text{-}size\ sc = target\text{-}size\ sc + n \implies$
 $to\text{-}bl\ (sc\ w) = drop\ n\ (to\text{-}bl\ w)$
apply (*subgoal-tac sc = ucast*)
apply *safe*
apply *simp*
apply (*erule ucast-down-drop*)
apply (*rule down-cast-same [symmetric]*)
apply (*simp add : source-size target-size is-down*)
done

lemma *sint-up-scast* [*OF refl*]:
 $sc = scast \implies is\text{-}up\ sc \implies sint\ (sc\ w) = sint\ w$
apply (*unfold is-up*)
apply *safe*
apply (*simp add: scast-def word-sbin.eq-norm*)
apply (*rule box-equals*)
prefer 3
apply (*rule word-sbin.norm-Rep*)
apply (*rule sbintrunc-sbintrunc-l*)
defer
apply (*subst word-sbin.norm-Rep*)
apply (*rule refl*)
apply *simp*
done

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

lemma *ucast-up-ucast* [*OF refl*]:
 $uc = ucast \implies is\text{-}up\ uc \implies ucast\ (uc\ w) = ucast\ w$
apply (*simp (no-asm) add: ucast-def*)
apply (*clarsimp simp add: uint-up-ucast*)
done

lemma *scast-up-scast* [*OF refl*]:

```

sc = scast  $\implies$  is-up sc  $\implies$  scast (sc w) = scast w
apply (simp (no-asm) add: scast-def)
apply (clarsimp simp add: sint-up-scast)
done

```

lemma *ucast-of-bl-up* [OF refl]:

```

w = of-bl bl  $\implies$  size bl <= size w  $\implies$  ucast w = of-bl bl
by (auto simp add : nth-ucast word-size test-bit-of-bl intro!: word-eqI)

```

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

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

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

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

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

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

lemma *up-ucast-surj*:

```

is-up (ucast :: 'b::len0 word => 'a::len0 word)  $\implies$ 
  surj (ucast :: 'a word => 'b word)
by (rule surjI, erule ucast-up-ucast-id)

```

lemma *up-scast-surj*:

```

is-up (scast :: 'b::len word => 'a::len word)  $\implies$ 
  surj (scast :: 'a word => 'b word)
by (rule surjI, erule scast-up-scast-id)

```

lemma *down-scast-inj*:

```

is-down (scast :: 'b::len word => 'a::len word)  $\implies$ 
  inj-on (ucast :: 'a word => 'b word) A
by (rule inj-on-inverseI, erule scast-down-scast-id)

```

lemma *down-ucast-inj*:

```

is-down (ucast :: 'b::len0 word => 'a::len0 word)  $\implies$ 
  inj-on (ucast :: 'a word => 'b word) A
by (rule inj-on-inverseI, erule ucast-down-ucast-id)

```

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

by (rule word-bl.Rep-eqD) (simp add: word-rep-drop)

lemma *ucast-down-wi* [OF refl]:

```

uc = ucast  $\implies$  is-down uc  $\implies$  uc (word-of-int x) = word-of-int x
apply (unfold is-down)
apply (clarsimp simp add: ucast-def word-ubin.eq-norm)
apply (rule word-ubin.norm-eq-iff [THEN iffD1])
apply (erule bintrunc-bintrunc-ge)
done

```

lemma *ucast-down-no* [OF refl]:

$uc = ucast \implies is_down\ uc \implies uc\ (numeral\ bin) = numeral\ bin$
unfolding *word-numeral-alt* **by** *clarify* (rule *ucast-down-wi*)

lemma *ucast-down-bl* [*OF refl*]:

$uc = ucast \implies is_down\ uc \implies uc\ (of_bl\ bl) = of_bl\ bl$
unfolding *of-bl-def* **by** *clarify* (erule *ucast-down-wi*)

lemmas *slice-def' = slice-def* [*unfolded word-size*]

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

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

13.14 Word Arithmetic

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

unfolding *word-less-def word-le-def* **by** (*simp add: less-le*)

lemma *signed-linorder*: *class.linorder word-sle word-sless*

by *default* (*unfold word-sle-def word-sless-def, auto*)

interpretation *signed*: *linorder word-sle word-sless*

by (*rule signed-linorder*)

lemma *udvdI*:

$0 \leq n \implies uint\ b = n * uint\ a \implies a\ udvd\ b$

by (*auto simp: udvd-def*)

lemmas *word-div-no* [*simp*] = *word-div-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-mod-no* [*simp*] = *word-mod-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-less-no* [*simp*] = *word-less-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-le-no* [*simp*] = *word-le-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-sless-no* [*simp*] = *word-sless-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-sle-no* [*simp*] = *word-sle-def* [*of numeral a numeral b*] **for** *a b*

lemma *word-1-no*: $(1::'a::len0\ word) = Numeral1$

by (*simp add: word-numeral-alt*)

lemma *word-m1-wi*: $-1 = word-of-int\ -1$

by (*rule word-neg-numeral-alt*)

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

unfolding *of-bl-def* **by** *simp*

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

unfolding *of-bl-def* **by** (*simp add: bl-to-bin-def*)

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

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

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

lemma *wint-0-iff*: $(wint\ x = 0) = (x = 0)$
by (*auto intro!: word-wint.Rep-eqD*)

lemma *unat-0-iff*: $(unat\ x = 0) = (x = 0)$
unfolding *unat-def* **by** (*auto simp add : nat-eq-iff wint-0-iff*)

lemma *unat-0* [*simp*]: $unat\ 0 = 0$
unfolding *unat-def* **by** *auto*

lemma *size-0-same'*: $size\ w = 0 \implies w = (v :: 'a :: len0\ word)$
apply (*unfold word-size*)
apply (*rule box-equals*)
defer
apply (*rule word-wint.Rep-inverse*)+
apply (*rule word-ubin.norm-eq-iff [THEN iffD1]*)
apply *simp*
done

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

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

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

lemma *ucast-0* [*simp*]: $ucast\ 0 = 0$
unfolding *ucast-def* **by** *simp*

lemma *sint-0* [*simp*]: $sint\ 0 = 0$
unfolding *sint-wint* **by** *simp*

lemma *scast-0* [*simp*]: $scast\ 0 = 0$
unfolding *scast-def* **by** *simp*

lemma *sint-n1* [*simp*]: $sint\ -1 = -1$

unfolding *word-m1-wi* **by** (*simp add: word-sbin.eq-norm*)

lemma *scast-n1* [*simp*]: *scast -1 = -1*
unfolding *scast-def* **by** *simp*

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

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

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

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

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

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*: $\text{word-pred } 0 = \text{word-of-int } -1$
unfolding *word-pred-m1* **by** *simp*

lemma *succ-pred-no* [*simp*]:
 $\text{word-succ } (\text{numeral } w) = \text{numeral } w + 1$
 $\text{word-pred } (\text{numeral } w) = \text{numeral } w - 1$
 $\text{word-succ } (\text{neg-numeral } w) = \text{neg-numeral } w + 1$
 $\text{word-pred } (\text{neg-numeral } w) = \text{neg-numeral } w - 1$
unfolding *word-succ-p1* *word-pred-m1* **by** *simp-all*

lemma *word-sp-01* [*simp*] :
 $\text{word-succ } -1 = 0 \ \& \ \text{word-succ } 0 = 1 \ \& \ \text{word-pred } 0 = -1 \ \& \ \text{word-pred } 1 = 0$
unfolding *word-succ-p1* *word-pred-m1* **by** *simp-all*

lemma *word-of-int-Ex*:
 $\exists y. x = \text{word-of-int } y$
by (*rule-tac* $x = \text{uint } x$ **in** *exI*) *simp*

13.16 Order on fixed-length words

lemma *word-zero-le* [*simp*] :
 $0 \leq (y :: 'a :: \text{len0 word})$
unfolding *word-le-def* **by** *auto*

lemma *word-m1-ge* [*simp*] : $\text{word-pred } 0 \geq y$
unfolding *word-le-def*
by (*simp only* : *word-pred-0-n1* *word-uint.eq-norm* *m1mod2k*) *auto*

lemma *word-n1-ge* [*simp*]: $y \leq (-1 :: 'a :: \text{len0 word})$
unfolding *word-le-def*
by (*simp only*: *word-m1-wi* *word-uint.eq-norm* *m1mod2k*) *auto*

lemmas *word-not-simps* [*simp*] =
 $\text{word-zero-le } [THEN \text{leD}] \ \text{word-m1-ge } [THEN \text{leD}] \ \text{word-n1-ge } [THEN \text{leD}]$

lemma *word-gt-0*: $0 < y \iff 0 \neq (y :: 'a :: \text{len0 word})$
by (*simp add*: *less-le*)

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

lemma *word-sless-alt*: $(a < s \ b) = (\text{sint } a < \text{sint } b)$
unfolding *word-sle-def* *word-sless-def*
by (*auto simp add*: *less-le*)

lemma *word-le-nat-alt*: $(a \leq b) = (\text{unat } a \leq \text{unat } b)$
unfolding *unat-def* *word-le-def*
by (*rule nat-le-eq-zle* [*symmetric*]) *simp*

lemma *word-less-nat-alt*: $(a < b) = (\text{unat } a < \text{unat } b)$
unfolding *unat-def word-less-alt*
by (*rule nat-less-eq-zless [symmetric]*) *simp*

lemma *wi-less*:
 $(\text{word-of-int } n < (\text{word-of-int } m :: 'a :: \text{len0 word})) =$
 $(n \bmod 2 \wedge \text{len-of TYPE('a)} < m \bmod 2 \wedge \text{len-of TYPE('a)})$
unfolding *word-less-alt* **by** (*simp add: word-uint.eq-norm*)

lemma *wi-le*:
 $(\text{word-of-int } n \leq (\text{word-of-int } m :: 'a :: \text{len0 word})) =$
 $(n \bmod 2 \wedge \text{len-of TYPE('a)} \leq m \bmod 2 \wedge \text{len-of TYPE('a)})$
unfolding *word-le-def* **by** (*simp add: word-uint.eq-norm*)

lemma *udvd-nat-alt*: $a \text{ udvd } b = (\exists x n \geq 0. \text{unat } b = n * \text{unat } a)$
apply (*unfold udvd-def*)
apply *safe*
apply (*simp add: unat-def nat-mult-distrib*)
apply (*simp add: uint-nat int-mult*)
apply (*rule exI*)
apply *safe*
prefer 2
apply (*erule notE*)
apply (*rule refl*)
apply *force*
done

lemma *udvd-iff-dvd*: $x \text{ udvd } y \iff \text{unat } x \text{ dvd } \text{unat } y$
unfolding *dvd-def udvd-nat-alt* **by** *force*

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

lemma *unat-minus-one*: $x \sim 0 \implies \text{unat } (x - 1) = \text{unat } x - 1$
apply (*unfold unat-def*)
apply (*simp only: int-word-uint word-arith-alts rdmods*)
apply (*subgoal-tac uint x >= 1*)
prefer 2
apply (*drule contrapos-nn*)
apply (*erule word-uint.Rep-inverse' [symmetric]*)
apply (*insert uint-ge-0 [of x]*)[1]
apply *arith*
apply (*rule box-equals*)
apply (*rule nat-diff-distrib*)
prefer 2
apply *assumption*
apply *simp*
apply (*subst mod-pos-pos-trivial*)
apply *arith*
apply (*insert uint-lt2p [of x]*)[1]

```

apply arith
apply (rule refl)
apply simp
done

```

lemma *measure-unat*: $p \sim = 0 \implies \text{unat } (p - 1) < \text{unat } p$
by (*simp add: unat-minus-one*) (*simp add: unat-0-iff* [*symmetric*])

lemmas *uint-add-ge0* [*simp*] = *add-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]
lemmas *uint-mult-ge0* [*simp*] = *mult-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]

lemma *uint-sub-lt2p* [*simp*]:
 $\text{uint } (x :: 'a :: \text{len0 word}) - \text{uint } (y :: 'b :: \text{len0 word}) <$
 $2 \wedge \text{len-of TYPE('a)}$
using *uint-ge-0* [*of y*] *uint-lt2p* [*of x*] **by** *arith*

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

lemma *uint-add-lem*:
 $(\text{uint } x + \text{uint } y < 2 \wedge \text{len-of TYPE('a)}) =$
 $(\text{uint } (x + y :: 'a :: \text{len0 word}) = \text{uint } x + \text{uint } y)$
by (*unfold uint-word-ariths*) (*auto intro!*: *trans* [*OF - int-mod-lem*])

lemma *uint-mult-lem*:
 $(\text{uint } x * \text{uint } y < 2 \wedge \text{len-of TYPE('a)}) =$
 $(\text{uint } (x * y :: 'a :: \text{len0 word}) = \text{uint } x * \text{uint } y)$
by (*unfold uint-word-ariths*) (*auto intro!*: *trans* [*OF - int-mod-lem*])

lemma *uint-sub-lem*:
 $(\text{uint } x \geq \text{uint } y) = (\text{uint } (x - y) = \text{uint } x - \text{uint } y)$
by (*unfold uint-word-ariths*) (*auto intro!*: *trans* [*OF - int-mod-lem*])

lemma *uint-add-le*: $\text{uint } (x + y) \leq \text{uint } x + \text{uint } y$
unfolding *uint-word-ariths* **by** (*auto simp: mod-add-if-z*)

lemma *uint-sub-ge*: $\text{uint } (x - y) \geq \text{uint } x - \text{uint } y$
unfolding *uint-word-ariths* **by** (*auto simp: mod-sub-if-z*)

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*:
 $\text{word-of-int } r = a \implies 0 \leq r \implies r < 2 \wedge \text{len-of TYPE('a)} \implies$
 $\text{uint } (a :: 'a :: \text{len0 word}) = r$
apply (*erule word-uint.Abs-inverse'* [*rotated*])
apply (*simp add: uints-num*)

done

lemma *uint-split*:

fixes $x::'a::len0$ *word*

shows P (*uint* x) =

(*ALL* i . *word-of-int* $i = x$ & $0 \leq i$ & $i < 2^{\text{len-of TYPE('a)}}$ $\longrightarrow P$ i)

apply (*fold word-int-case-def*)

apply (*auto dest!*: *word-of-int-inverse simp: int-word-uint int-mod-eq'*
split: word-int-split)

done

lemma *uint-split-asm*:

fixes $x::'a::len0$ *word*

shows P (*uint* x) =

(\sim (*EX* i . *word-of-int* $i = x$ & $0 \leq i$ & $i < 2^{\text{len-of TYPE('a)}}$ & $\sim P$ i))

by (*auto dest!*: *word-of-int-inverse*

simp: int-word-uint int-mod-eq'

split: uint-split)

lemmas *uint-splits* = *uint-split uint-split-asm*

lemmas *uint-arith-simps* =

word-le-def word-less-alt

word-uint.Rep-inject [*symmetric*]

uint-sub-if' uint-plus-if'

lemma *power-False-cong*: $\text{False} \Longrightarrow a \wedge b = c \wedge d$

by *auto*

ML $\langle\langle$

fun *uint-arith-ss-of* $ss =$

ss *addsims* @{*thms uint-arith-simps*}

delsims @{*thms word-uint.Rep-inject*}

|> *fold Splitter.add-split* @{*thms split-if-asm*}

|> *fold Simplifier.add-cong* @{*thms power-False-cong*}

fun *uint-arith-tacs* $ctxt =$

let

fun *arith-tac'* n $t =$

Arith-Data.verbose-arith-tac $ctxt$ n t

handle Cooper.COOPER \Rightarrow *Seq.empty*;

in

[*clarify-tac* $ctxt$ 1 ,

full-simp-tac (*uint-arith-ss-of* (*simpset-of* $ctxt$)) 1 ,

ALLGOALS (*full-simp-tac* (*HOL-ss* |> *fold Splitter.add-split* @{*thms uint-splits*}

|> *fold Simplifier.add-cong* @{*thms power-False-cong*})),

rewrite-goals-tac @{*thms word-size*}

```

    ALLGOALS (fn n => REPEAT (resolve-tac [allI, impI] n) THEN
      REPEAT (etac conjE n) THEN
      REPEAT (dtac @{thm word-of-int-inverse} n
        THEN atac n
        THEN atac n)),
    TRYALL arith-tac' ]
end

```

```

fun uint-arith-tac ctxt = SELECT-GOAL (EVERY (uint-arith-tacs ctxt))
>>

```

```

method-setup uint-arith =
  << Scan.succeed (SIMPLE-METHOD' o uint-arith-tac) >>
  solving word arithmetic via integers and arith

```

13.19 More on overflows and monotonicity

lemma *no-plus-overflow-uint-size*:

```

((x :: 'a :: len0 word) <= x + y) = (uint x + uint y < 2 ^ size x)
  unfolding word-size by uint-arith

```

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

lemma *no-ulen-sub*: ((x :: 'a :: len0 word) >= x - y) = (uint y <= uint x)
by *uint-arith*

lemma *no-olen-add'*:

```

fixes x :: 'a::len0 word
shows (x ≤ y + x) = (uint y + uint x < 2 ^ len-of TYPE('a))
by (simp add: add-ac no-olen-add)

```

lemmas *olen-add-egv* = *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 :: len word) ~ = 0 ==> (1 < x) = (0 < x - 1)
  by uint-arith

```

lemma *word-le-sub1*:

```

(x :: 'a :: len word) ~ = 0 ==> (1 <= x) = (0 <= x - 1)
  by uint-arith

```

lemma *sub-wrap-lt*:

$((x :: 'a :: \text{len0 word}) < x - z) = (x < z)$
by *uint-arith*

lemma *sub-wrap*:

$((x :: 'a :: \text{len0 word}) <= x - z) = (z = 0 \mid x < z)$
by *uint-arith*

lemma *plus-minus-not-NULL-ab*:

$(x :: 'a :: \text{len0 word}) <= ab - c \implies c <= ab \implies c \sim= 0 \implies x + c \sim= 0$
by *uint-arith*

lemma *plus-minus-no-overflow-ab*:

$(x :: 'a :: \text{len0 word}) <= ab - c \implies c <= ab \implies x <= x + c$
by *uint-arith*

lemma *le-minus'*:

$(a :: 'a :: \text{len0 word}) + c <= b \implies a <= a + c \implies c <= b - a$
by *uint-arith*

lemma *le-plus'*:

$(a :: 'a :: \text{len0 word}) <= b \implies c <= b - a \implies a + c <= b$
by *uint-arith*

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

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

lemma *word-plus-mono-right*:

$(y :: 'a :: \text{len0 word}) <= z \implies x <= x + z \implies x + y <= x + z$
by *uint-arith*

lemma *word-less-minus-cancel*:

$y - x < z - x \implies x <= z \implies (y :: 'a :: \text{len0 word}) < z$
by *uint-arith*

lemma *word-less-minus-mono-left*:

$(y :: 'a :: \text{len0 word}) < z \implies x <= y \implies y - x < z - x$
by *uint-arith*

lemma *word-less-minus-mono*:

$a < c \implies d < b \implies a - b < a \implies c - d < c$
 $\implies a - b < c - (d :: 'a :: \text{len word})$
by *uint-arith*

lemma *word-le-minus-cancel*:

$y - x <= z - x \implies x <= z \implies (y :: 'a :: \text{len0 word}) <= z$
by *uint-arith*

lemma *word-le-minus-mono-left*:

$(y :: 'a :: \text{len0 word}) <= z \implies x <= y \implies y - x <= z - x$
by *uint-arith*

lemma *word-le-minus-mono*:

$a <= c \implies d <= b \implies a - b <= a \implies c - d <= c$
 $\implies a - b <= c - (d :: 'a :: \text{len word})$
by *uint-arith*

lemma *plus-le-left-cancel-wrap*:

$(x :: 'a :: \text{len0 word}) + y' < x \implies x + y < x \implies (x + y' < x + y) = (y' < y)$
by *uint-arith*

lemma *plus-le-left-cancel-nowrap*:

$(x :: 'a :: \text{len0 word}) <= x + y' \implies x <= x + y \implies$
 $(x + y' < x + y) = (y' < y)$
by *uint-arith*

lemma *word-plus-mono-right2*:

$(a :: 'a :: \text{len0 word}) <= a + b \implies c <= b \implies a <= a + c$
by *uint-arith*

lemma *word-less-add-right*:

$(x :: 'a :: \text{len0 word}) < y - z \implies z <= y \implies x + z < y$
by *uint-arith*

lemma *word-less-sub-right*:

$(x :: 'a :: \text{len0 word}) < y + z \implies y <= x \implies x - y < z$
by *uint-arith*

lemma *word-le-plus-either*:

$(x :: 'a :: \text{len0 word}) <= y \mid x <= z \implies y <= y + z \implies x <= y + z$
by *uint-arith*

lemma *word-less-nowrapI*:

$(x :: 'a :: \text{len0 word}) < z - k \implies k <= z \implies 0 < k \implies x < x + k$
by *uint-arith*

lemma *inc-le*: $(i :: 'a :: \text{len word}) < m \implies i + 1 <= m$

by *uint-arith*

lemma *inc-i*:

$(1 :: 'a :: \text{len word}) <= i \implies i < m \implies 1 <= (i + 1) \ \& \ i + 1 <= m$
by *uint-arith*

lemma *udvd-incr-lem*:

$up < uq \implies up = ua + n * \text{uint } K \implies$
 $uq = ua + n' * \text{uint } K \implies up + \text{uint } K <= uq$
apply *clarsimp*
apply (*drule less-le-mult*)

apply *safe*
done

lemma *udvd-incr'*:
 $p < q \implies \text{uint } p = ua + n * \text{uint } K \implies$
 $\text{uint } q = ua + n' * \text{uint } K \implies p + K \leq q$
apply (*unfold word-less-alt word-le-def*)
apply (*drule (2) udvd-incr-lem*)
apply (*erule uint-add-le [THEN order-trans]*)
done

lemma *udvd-decr'*:
 $p < q \implies \text{uint } p = ua + n * \text{uint } K \implies$
 $\text{uint } q = ua + n' * \text{uint } K \implies p \leq q - K$
apply (*unfold word-less-alt word-le-def*)
apply (*drule (2) udvd-incr-lem*)
apply (*drule le-diff-eq [THEN iffD2]*)
apply (*erule order-trans*)
apply (*rule uint-sub-ge*)
done

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

lemma *udvd-minus-le'*:
 $xy < k \implies z \text{ udvd } xy \implies z \text{ udvd } k \implies xy \leq k - z$
apply (*unfold udvd-def*)
apply *clarify*
apply (*erule (2) udvd-decr0*)
done

ML $\ll \text{Delsimprocs } [\@ \{ \text{simproc linordered-ring-less-cancel-factor} \}] \gg$

lemma *udvd-incr2-K*:
 $p < a + s \implies a \leq a + s \implies K \text{ udvd } s \implies K \text{ udvd } p - a \implies a \leq p \implies$
 $0 < K \implies p \leq p + K \ \& \ p + K \leq a + s$
apply (*unfold udvd-def*)
apply *clarify*
apply (*simp add: uint-arith-simps split: split-if-asm*)
prefer 2
apply (*insert uint-range' [of s][1]*)
apply *arith*
apply (*drule add-commute [THEN xtr1]*)
apply (*simp add: diff-less-eq [symmetric]*)
apply (*drule less-le-mult*)
apply *arith*
apply *simp*
done

ML \ll *Addsimprocs* [$\@$ {*simproc linordered-ring-less-cancel-factor*}] \gg

lemma *word-succ-rbl*:

$to-bl\ w = bl \implies to-bl\ (word-succ\ w) = (rev\ (rbl-succ\ (rev\ bl)))$
apply (*unfold word-succ-def*)
apply *clarify*
apply (*simp add: to-bl-of-bin*)
apply (*simp add: to-bl-def rbl-succ*)
done

lemma *word-pred-rbl*:

$to-bl\ w = bl \implies to-bl\ (word-pred\ w) = (rev\ (rbl-pred\ (rev\ bl)))$
apply (*unfold word-pred-def*)
apply *clarify*
apply (*simp add: to-bl-of-bin*)
apply (*simp add: to-bl-def rbl-pred*)
done

lemma *word-add-rbl*:

$to-bl\ v = vbl \implies to-bl\ w = wbl \implies$
 $to-bl\ (v + w) = (rev\ (rbl-add\ (rev\ vbl)\ (rev\ wbl)))$
apply (*unfold word-add-def*)
apply *clarify*
apply (*simp add: to-bl-of-bin*)
apply (*simp add: to-bl-def rbl-add*)
done

lemma *word-mult-rbl*:

$to-bl\ v = vbl \implies to-bl\ w = wbl \implies$
 $to-bl\ (v * w) = (rev\ (rbl-mult\ (rev\ vbl)\ (rev\ wbl)))$
apply (*unfold word-mult-def*)
apply *clarify*
apply (*simp add: to-bl-of-bin*)
apply (*simp add: to-bl-def rbl-mult*)
done

lemma *rtb-rbl-ariths*:

$rev\ (to-bl\ w) = ys \implies rev\ (to-bl\ (word-succ\ w)) = rbl-succ\ ys$
 $rev\ (to-bl\ w) = ys \implies rev\ (to-bl\ (word-pred\ w)) = rbl-pred\ ys$
 $rev\ (to-bl\ v) = ys \implies rev\ (to-bl\ w) = xs \implies rev\ (to-bl\ (v * w)) = rbl-mult\ ys$
 xs
 $rev\ (to-bl\ v) = ys \implies rev\ (to-bl\ w) = xs \implies rev\ (to-bl\ (v + w)) = rbl-add\ ys\ xs$
by (*auto simp: rev-swap [symmetric] word-succ-rbl*
word-pred-rbl word-mult-rbl word-add-rbl)

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*:
 $0 \leq x \implies \text{word-of-int } x = \text{of-nat } (\text{nat } x)$
by (*simp add: of-nat-nat word-of-int*)

lemma *iszero-word-no* [*simp*]:
 $\text{iszero } (\text{numeral bin} :: 'a :: \text{len word}) =$
 $\text{iszero } (\text{bintrunc } (\text{len-of TYPE } ('a)) (\text{numeral bin}))$
using *word-ubin.norm-eq-iff* [**where** $'a = 'a$, *of numeral bin 0*]
by (*simp add: iszero-def [symmetric]*)

Use *iszero* to simplify equalities between word numerals.

lemmas *word-eq-numeral-iff-iszero* [*simp*] =
eq-numeral-iff-iszero [**where** $'a = 'a :: \text{len word}$]

13.21 Word and nat

lemma *td-ext-unat* [*OF refl*]:
 $n = \text{len-of TYPE } ('a :: \text{len}) \implies$
 $\text{td-ext } (\text{unat} :: 'a \text{ word} \Rightarrow \text{nat}) \text{ of-nat}$
 $(\text{unats } n) (\%i. i \bmod 2 \wedge n)$
apply (*unfold td-ext-def' unat-def word-of-nat unats-uints*)
apply (*auto intro!: imageI simp add : word-of-int-hom-syms*)
apply (*erule word-uint.Abs-inverse [THEN arg-cong]*)
apply (*simp add: int-word-uint nat-mod-distrib nat-power-eq*)
done

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

interpretation *word-unat*:
 $\text{td-ext unat} :: 'a :: \text{len word} \Rightarrow \text{nat}$
of-nat
 $\text{unats } (\text{len-of TYPE } ('a :: \text{len}))$
 $\%i. i \bmod 2 \wedge \text{len-of TYPE } ('a :: \text{len})$
by (*rule td-ext-unat*)

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

lemmas *unat-lt2p* [*iff*] = *word-unat.Rep* [*unfolded unats-def mem-Collect-eq*]

lemma *unat-le*: $y \leq \text{unat } (z :: 'a :: \text{len word}) \implies y : \text{unats } (\text{len-of TYPE } ('a))$
apply (*unfold unats-def*)
apply *clarsimp*
apply (*rule xtrans, rule unat-lt2p, assumption*)
done

lemma *word-nchotomy*:

ALL w . *EX* n . $(w :: 'a :: \text{len word}) = \text{of-nat } n \ \& \ n < 2 \wedge \text{len-of TYPE } ('a)$
apply (*rule allI*)
apply (*rule word-unat.Abs-cases*)
apply (*unfold unats-def*)
apply *auto*
done

lemma *of-nat-eq*:

fixes $w :: 'a :: \text{len word}$
shows $(\text{of-nat } n = w) = (\exists q. n = \text{unat } w + q * 2 \wedge \text{len-of TYPE } ('a))$
apply (*rule trans*)
apply (*rule word-unat.inverse-norm*)
apply (*rule iffI*)
apply (*rule mod-eqD*)
apply *simp*
apply *clarsimp*
done

lemma *of-nat-eq-size*:

$(\text{of-nat } n = w) = (\text{EX } q. n = \text{unat } w + q * 2 \wedge \text{size } w)$
unfolding *word-size* **by** (*rule of-nat-eq*)

lemma *of-nat-0*:

$(\text{of-nat } m = (0 :: 'a :: \text{len word})) = (\exists q. m = q * 2 \wedge \text{len-of TYPE } ('a))$
by (*simp add: of-nat-eq*)

lemma *of-nat-2p* [*simp*]:

$\text{of-nat } (2 \wedge \text{len-of TYPE } ('a)) = (0 :: 'a :: \text{len word})$
by (*fact mult-1* [*symmetric*, *THEN iffD2* [*OF of-nat-0 exI*]])

lemma *of-nat-gt-0*: $\text{of-nat } k \sim = 0 \implies 0 < k$

by (*cases k*) *auto*

lemma *of-nat-neq-0*:

$0 < k \implies k < 2 \wedge \text{len-of TYPE } ('a :: \text{len}) \implies \text{of-nat } k \sim = (0 :: 'a \text{ word})$
by (*clarsimp simp add : of-nat-0*)

lemma *Abs-fnat-hom-add*:

$\text{of-nat } a + \text{of-nat } b = \text{of-nat } (a + b)$
by *simp*

lemma *Abs-fnat-hom-mult*:

$\text{of-nat } a * \text{of-nat } b = (\text{of-nat } (a * b) :: 'a :: \text{len word})$
by (*simp add: word-of-nat wi-hom-mult zmult-int*)

lemma *Abs-fnat-hom-Suc*:

$\text{word-succ } (\text{of-nat } a) = \text{of-nat } (\text{Suc } a)$

by (*simp add: word-of-nat wi-hom-succ add-ac*)

lemma *Abs-fnat-hom-0*: $(0::'a::\text{len } \text{word}) = \text{of-nat } 0$
by *simp*

lemma *Abs-fnat-hom-1*: $(1::'a::\text{len } \text{word}) = \text{of-nat } (\text{Suc } 0)$
by *simp*

lemmas *Abs-fnat-homs* =
Abs-fnat-hom-add Abs-fnat-hom-mult Abs-fnat-hom-Suc
Abs-fnat-hom-0 Abs-fnat-hom-1

lemma *word-arith-nat-add*:
 $a + b = \text{of-nat } (\text{unat } a + \text{unat } b)$
by *simp*

lemma *word-arith-nat-mult*:
 $a * b = \text{of-nat } (\text{unat } a * \text{unat } b)$
by (*simp add: of-nat-mult*)

lemma *word-arith-nat-Suc*:
 $\text{word-succ } a = \text{of-nat } (\text{Suc } (\text{unat } a))$
by (*subst Abs-fnat-hom-Suc [symmetric] simp*)

lemma *word-arith-nat-div*:
 $a \text{ div } b = \text{of-nat } (\text{unat } a \text{ div } \text{unat } b)$
by (*simp add: word-div-def word-of-nat zdiv-int uint-nat*)

lemma *word-arith-nat-mod*:
 $a \text{ mod } b = \text{of-nat } (\text{unat } a \text{ mod } \text{unat } b)$
by (*simp add: word-mod-def word-of-nat zmod-int uint-nat*)

lemmas *word-arith-nat-defs* =
word-arith-nat-add word-arith-nat-mult
word-arith-nat-Suc Abs-fnat-hom-0
Abs-fnat-hom-1 word-arith-nat-div
word-arith-nat-mod

lemma *unat-cong*: $x = y \implies \text{unat } x = \text{unat } y$
by *simp*

lemmas *unat-word-ariths* = *word-arith-nat-defs*
[*THEN trans [OF unat-cong unat-of-nat]*]

lemmas *word-sub-less-iff* = *word-sub-le-iff*
[*unfolded linorder-not-less [symmetric] Not-eq-iff*]

lemma *unat-add-lem*:
 $(\text{unat } x + \text{unat } y < 2 \wedge \text{len-of } \text{TYPE}('a)) =$

(*unat* ($x + y :: 'a :: \text{len word}$) = *unat* $x + \text{unat } y$)
unfolding *unat-word-ariths*
by (*auto intro!*: *trans* [*OF* - *nat-mod-lem*])

lemma *unat-mult-lem*:
(*unat* $x * \text{unat } y < 2 ^ \text{len-of TYPE}('a)$) =
(*unat* ($x * y :: 'a :: \text{len word}$) = *unat* $x * \text{unat } y$)
unfolding *unat-word-ariths*
by (*auto intro!*: *trans* [*OF* - *nat-mod-lem*])

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

lemma *le-no-overflow*:
 $x \leq b \implies a \leq a + b \implies x \leq a + (b :: 'a :: \text{len0 word})$
apply (*erule order-trans*)
apply (*erule olen-add-egu* [*THEN iffD1*])
done

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

lemma *unat-sub-if-size*:
unat ($x - y$) = (*if* *unat* $y \leq \text{unat } x$
then *unat* $x - \text{unat } y$
else *unat* $x + 2 ^ \text{size } x - \text{unat } y$)
apply (*unfold word-size*)
apply (*simp add: un-ui-le*)
apply (*auto simp add: unat-def uint-sub-if'*)
apply (*rule nat-diff-distrib*)
prefer 3
apply (*simp add: algebra-simps*)
apply (*rule nat-diff-distrib* [*THEN trans*])
prefer 3
apply (*subst nat-add-distrib*)
prefer 3
apply (*simp add: nat-power-eg*)
apply *auto*
apply *uint-arith*
done

lemmas *unat-sub-if'* = *unat-sub-if-size* [*unfolded word-size*]

lemma *unat-div*: *unat* ($(x :: 'a :: \text{len word}) \text{div } y$) = *unat* $x \text{div } \text{unat } y$
apply (*simp add : unat-word-ariths*)
apply (*rule unat-lt2p* [*THEN xtr7*, *THEN nat-mod-eg'*])
apply (*rule div-le-dividend*)
done

lemma *unat-mod*: *unat* ($(x :: 'a :: \text{len word}) \text{mod } y$) = *unat* $x \text{mod } \text{unat } y$
apply (*clarsimp simp add : unat-word-ariths*)

```

apply (cases unat y)
prefer 2
apply (rule unat-lt2p [THEN xtr7, THEN nat-mod-eq])
apply (rule mod-le-divisor)
apply auto
done

```

lemma *uint-div*: $\text{uint } ((x :: 'a :: \text{len word}) \text{ div } y) = \text{uint } x \text{ div uint } y$
unfolding *uint-nat* **by** (*simp add : unat-div zdiv-int*)

lemma *uint-mod*: $\text{uint } ((x :: 'a :: \text{len word}) \text{ mod } y) = \text{uint } x \text{ mod uint } y$
unfolding *uint-nat* **by** (*simp add : unat-mod zmod-int*)

13.22 Definition of unat_arith tactic

lemma *unat-split*:
fixes $x :: 'a :: \text{len word}$
shows $P (\text{unat } x) =$
 $(\text{ALL } n. \text{ of-nat } n = x \ \& \ n < 2^{\text{len-of TYPE('a)}} \longrightarrow P \ n)$
by (*auto simp: unat-of-nat*)

lemma *unat-split-asm*:
fixes $x :: 'a :: \text{len word}$
shows $P (\text{unat } x) =$
 $(\sim (\text{EX } n. \text{ of-nat } n = x \ \& \ n < 2^{\text{len-of TYPE('a)}} \ \& \ \sim P \ n))$
by (*auto simp: unat-of-nat*)

lemmas *of-nat-inverse* =
word-unat.Abs-inverse' [*rotated, unfolded unats-def, simplified*]

lemmas *unat-splits* = *unat-split unat-split-asm*

lemmas *unat-arith-simps* =
word-le-nat-alt word-less-nat-alt
word-unat.Rep-inject [*symmetric*]
unat-sub-if' unat-plus-if' unat-div unat-mod

ML $\langle\langle$
fun unat-arith-ss-of ss =
 $\text{ss addsimps } @\{\text{thms unat-arith-simps}\}$
 $\text{delsimps } @\{\text{thms word-unat.Rep-inject}\}$
 $|> \text{fold Splitter.add-split } @\{\text{thms split-if-asm}\}$
 $|> \text{fold Simplifier.add-cong } @\{\text{thms power-False-cong}\}$

fun unat-arith-tacs ctxt =
let
 $\text{fun arith-tac' } n \ t =$
 $\text{Arith-Data.verbose-arith-tac } \text{ctxt } n \ t$

```

      handle Cooper.COOPER - => Seq.empty;
in
  [ clarify-tac ctxt 1,
    full-simp-tac (unat-arith-ss-of (simpset-of ctxt)) 1,
    ALLGOALS (full-simp-tac (HOL-ss |> fold Splitter.add-split @{thms unat-splits}
      |> fold Simplifier.add-cong @{thms power-False-cong})),
    rewrite-goals-tac @{thms word-size},
    ALLGOALS (fn n => REPEAT (resolve-tac [allI, impI] n) THEN
      REPEAT (etac conjE n) THEN
      REPEAT (dtac @{thm of-nat-inverse} n THEN atac n)),
    TRYALL arith-tac' ]
end

```

```

fun unat-arith-tac ctxt = SELECT-GOAL (EVERY (unat-arith-tacs ctxt))
>>

```

```

method-setup unat-arith =
  << Scan.succeed (SIMPLE-METHOD' o unat-arith-tac) >>
  solving word arithmetic via natural numbers and arith

```

```

lemma no-plus-overflow-unat-size:
  ((x :: 'a :: len word) <= x + y) = (unat x + unat y < 2 ^ size x)
unfolding word-size by unat-arith

```

```

lemmas no-olen-add-nat = no-plus-overflow-unat-size [unfolded word-size]

```

```

lemmas unat-plus-simple = trans [OF no-olen-add-nat unat-add-lem]

```

```

lemma word-div-mult:
  (0 :: 'a :: len word) < y ==> unat x * unat y < 2 ^ len-of TYPE('a) ==>
    x * y div y = x
apply unat-arith
apply clarsimp
apply (subst unat-mult-lem [THEN iffD1])
apply auto
done

```

```

lemma div-lt': (i :: 'a :: len word) <= k div x ==>
  unat i * unat x < 2 ^ len-of TYPE('a)
apply unat-arith
apply clarsimp
apply (drule mult-le-mono1)
apply (erule order-le-less-trans)
apply (rule xtr7 [OF unat-lt2p div-mult-le])
done

```

```

lemmas div-lt'' = order-less-imp-le [THEN div-lt']

```

```

lemma div-lt-mult: (i :: 'a :: len word) < k div x ==> 0 < x ==> i * x < k

```

```

apply (frule div-lt'' [THEN unat-mult-lem [THEN iffD1]])
apply (simp add: unat-arith-simps)
apply (drule (1) mult-less-mono1)
apply (erule order-less-le-trans)
apply (rule div-mult-le)
done

```

```

lemma div-le-mult:
  (i :: 'a :: len word) <= k div x  $\implies$  0 < x  $\implies$  i * x <= k
apply (frule div-lt' [THEN unat-mult-lem [THEN iffD1]])
apply (simp add: unat-arith-simps)
apply (drule mult-le-mono1)
apply (erule order-trans)
apply (rule div-mult-le)
done

```

```

lemma div-lt-uint':
  (i :: 'a :: len word) <= k div x  $\implies$  uint i * uint x < 2 ^ len-of TYPE('a)
apply (unfold uint-nat)
apply (drule div-lt')
apply (simp add: zmult-int zless-nat-eq-int-zless [symmetric]
          nat-power-eq)
done

```

```

lemmas div-lt-uint'' = order-less-imp-le [THEN div-lt-uint']

```

```

lemma word-le-exists':
  (x :: 'a :: len0 word) <= y  $\implies$ 
    (EX z. y = x + z & uint x + uint z < 2 ^ len-of TYPE('a))
apply (rule exI)
apply (rule conjI)
apply (rule zadd-diff-inverse)
apply uint-arith
done

```

```

lemmas plus-minus-not-NULL = order-less-imp-le [THEN plus-minus-not-NULL-ab]

```

```

lemmas plus-minus-no-overflow =
  order-less-imp-le [THEN plus-minus-no-overflow-ab]

```

```

lemmas mcs = word-less-minus-cancel word-less-minus-mono-left
  word-le-minus-cancel word-le-minus-mono-left

```

```

lemmas word-l-diffs = mcs [where y = w + x, unfolded add-diff-cancel] for w x
lemmas word-diff-ls = mcs [where z = w + x, unfolded add-diff-cancel] for w x
lemmas word-plus-mcs = word-diff-ls [where y = v + x, unfolded add-diff-cancel]
for v x

```

```

lemmas le-unat-uoI = unat-le [THEN word-unat.Abs-inverse]

```

lemmas *thd* = *refl* [THEN [2] *split-div-lemma* [THEN *iffD2*], THEN *conjunct1*]

lemma *thd1*:
 $a \text{ div } b * b \leq (a :: \text{nat})$
using *gt-or-eq-0* [of *b*]
apply (*rule disjE*)
apply (*erule xtr4* [OF *thd mult-commute*])
apply *clarsimp*
done

lemmas *uno-simps* [THEN *le-unat-uoI*] = *mod-le-divisor div-le-dividend thd1*

lemma *word-mod-div-equality*:
 $(n \text{ div } b) * b + (n \text{ mod } b) = (n :: 'a :: \text{len word})$
apply (*unfold word-less-nat-alt word-arith-nat-defs*)
apply (*cut-tac y=unat b in gt-or-eq-0*)
apply (*erule disjE*)
apply (*simp add: mod-div-equality uno-simps*)
apply *simp*
done

lemma *word-div-mult-le: a div b * b <= (a::'a::len word)*
apply (*unfold word-le-nat-alt word-arith-nat-defs*)
apply (*cut-tac y=unat b in gt-or-eq-0*)
apply (*erule disjE*)
apply (*simp add: div-mult-le uno-simps*)
apply *simp*
done

lemma *word-mod-less-divisor: 0 < n ==> m mod n < (n :: 'a :: len word)*
apply (*simp only: word-less-nat-alt word-arith-nat-defs*)
apply (*clarsimp simp add : uno-simps*)
done

lemma *word-of-int-power-hom*:
 $\text{word-of-int } a \wedge n = (\text{word-of-int } (a \wedge n) :: 'a :: \text{len word})$
by (*induct n*) (*simp-all add: wi-hom-mult [symmetric]*)

lemma *word-arith-power-alt*:
 $a \wedge n = (\text{word-of-int } (\text{uint } a \wedge n) :: 'a :: \text{len word})$
by (*simp add : word-of-int-power-hom [symmetric]*)

lemma *of-bl-length-less*:
 $\text{length } x = k \implies k < \text{len-of TYPE('a)} \implies (\text{of-bl } x :: 'a :: \text{len word}) < 2 \wedge k$
apply (*unfold of-bl-def word-less-alt word-numeral-alt*)
apply *safe*
apply (*simp (no-asm) add: word-of-int-power-hom word-uint.eq-norm del: word-of-int-numeral*)

```

apply (simp add: mod-pos-pos-trivial)
apply (subst mod-pos-pos-trivial)
  apply (rule bl-to-bin-ge0)
  apply (rule order-less-trans)
  apply (rule bl-to-bin-lt2p)
apply simp
apply (rule bl-to-bin-lt2p)
done

```

13.23 Cardinality, finiteness of set of words

```

instance word :: (len0) finite
  by (default, simp add: type-definition.univ [OF type-definition-word])

lemma card-word: CARD('a::len0 word) = 2 ^ len-of TYPE('a)
  by (simp add: type-definition.card [OF type-definition-word] nat-power-eq)

lemma card-word-size:
  card (UNIV :: 'a :: len0 word set) = (2 ^ size (x :: 'a word))
unfolding word-size by (rule card-word)

```

13.24 Bitwise Operations on Words

lemmas bin-log-bintrs = bin-trunc-not bin-trunc-xor bin-trunc-and bin-trunc-or

lemmas wils1 = bin-log-bintrs [THEN word-ubin.norm-eq-iff [THEN iffD1],
folded word-ubin.eq-norm, THEN eq-reflection]

lemmas word-log-binary-defs =
word-and-def word-or-def word-xor-def

lemma word-wi-log-defs:
 NOT word-of-int a = word-of-int (NOT a)
 word-of-int a AND word-of-int b = word-of-int (a AND b)
 word-of-int a OR word-of-int b = word-of-int (a OR b)
 word-of-int a XOR word-of-int b = word-of-int (a XOR b)
by (transfer, rule refl)+

lemma word-no-log-defs [simp]:
 NOT (numeral a) = word-of-int (NOT (numeral a))
 NOT (neg-numeral a) = word-of-int (NOT (neg-numeral a))
 numeral a AND numeral b = word-of-int (numeral a AND numeral b)
 numeral a AND neg-numeral b = word-of-int (numeral a AND neg-numeral b)
 neg-numeral a AND numeral b = word-of-int (neg-numeral a AND numeral b)

$neg\text{-numeral } a \text{ AND } neg\text{-numeral } b = \text{word-of-int } (neg\text{-numeral } a \text{ AND } neg\text{-numeral } b)$
 $numeral a \text{ OR } numeral b = \text{word-of-int } (numeral a \text{ OR } numeral b)$
 $numeral a \text{ OR } neg\text{-numeral } b = \text{word-of-int } (numeral a \text{ OR } neg\text{-numeral } b)$
 $neg\text{-numeral } a \text{ OR } numeral b = \text{word-of-int } (neg\text{-numeral } a \text{ OR } numeral b)$
 $neg\text{-numeral } a \text{ OR } neg\text{-numeral } b = \text{word-of-int } (neg\text{-numeral } a \text{ OR } neg\text{-numeral } b)$
 $numeral a \text{ XOR } numeral b = \text{word-of-int } (numeral a \text{ XOR } numeral b)$
 $numeral a \text{ XOR } neg\text{-numeral } b = \text{word-of-int } (numeral a \text{ XOR } neg\text{-numeral } b)$
 $neg\text{-numeral } a \text{ XOR } numeral b = \text{word-of-int } (neg\text{-numeral } a \text{ XOR } numeral b)$
 $neg\text{-numeral } a \text{ XOR } neg\text{-numeral } b = \text{word-of-int } (neg\text{-numeral } a \text{ XOR } neg\text{-numeral } b)$
by (transfer, rule refl)+

Special cases for when one of the arguments equals 1.

lemma *word-bitwise-1-simps* [simp]:

$NOT (1::'a::len0 \text{ word}) = -2$
 $1 \text{ AND } numeral b = \text{word-of-int } (1 \text{ AND } numeral b)$
 $1 \text{ AND } neg\text{-numeral } b = \text{word-of-int } (1 \text{ AND } neg\text{-numeral } b)$
 $numeral a \text{ AND } 1 = \text{word-of-int } (numeral a \text{ AND } 1)$
 $neg\text{-numeral } a \text{ AND } 1 = \text{word-of-int } (neg\text{-numeral } a \text{ AND } 1)$
 $1 \text{ OR } numeral b = \text{word-of-int } (1 \text{ OR } numeral b)$
 $1 \text{ OR } neg\text{-numeral } b = \text{word-of-int } (1 \text{ OR } neg\text{-numeral } b)$
 $numeral a \text{ OR } 1 = \text{word-of-int } (numeral a \text{ OR } 1)$
 $neg\text{-numeral } a \text{ OR } 1 = \text{word-of-int } (neg\text{-numeral } a \text{ OR } 1)$
 $1 \text{ XOR } numeral b = \text{word-of-int } (1 \text{ XOR } numeral b)$
 $1 \text{ XOR } neg\text{-numeral } b = \text{word-of-int } (1 \text{ XOR } neg\text{-numeral } b)$
 $numeral a \text{ XOR } 1 = \text{word-of-int } (numeral a \text{ XOR } 1)$
 $neg\text{-numeral } a \text{ XOR } 1 = \text{word-of-int } (neg\text{-numeral } a \text{ XOR } 1)$
by (transfer, simp)+

lemma *uint-or*: $uint (x \text{ OR } y) = (uint x) \text{ OR } (uint y)$

by (transfer, simp add: bin-trunc-ao)

lemma *uint-and*: $uint (x \text{ AND } y) = (uint x) \text{ AND } (uint y)$

by (transfer, simp add: bin-trunc-ao)

lemma *test-bit-wi* [simp]:

$(\text{word-of-int } x::'a::len0 \text{ word}) !! n \longleftrightarrow n < \text{len-of } TYPE('a) \wedge \text{bin-nth } x n$

unfolding *word-test-bit-def*

by (simp add: word-ubin.eq-norm nth-bintr)

lemma *word-test-bit-transfer* [transfer-rule]:

$(\text{fun-rel } cr\text{-word } (\text{fun-rel } op = op =))$

$(\lambda x n. n < \text{len-of } TYPE('a) \wedge \text{bin-nth } x n) (\text{test-bit } :: 'a::len0 \text{ word} \Rightarrow -)$

unfolding *fun-rel-def cr-word-def* **by** *simp*

lemma *word-ops-nth-size*:

$n < \text{size } (x::'a::len0 \text{ word}) \implies$

$$(x \text{ OR } y) !! n = (x !! n | y !! n) \&$$

$$(x \text{ AND } y) !! n = (x !! n \& y !! n) \&$$

$$(x \text{ XOR } y) !! n = (x !! n \sim = y !! n) \&$$

$$(\text{NOT } x) !! n = (\sim x !! n)$$

unfolding *word-size* **by** *transfer* (*simp add: bin-nth-ops*)

lemma *word-ao-nth*:
fixes $x :: 'a::\text{len0 word}$
shows $(x \text{ OR } y) !! n = (x !! n | y !! n) \&$
 $(x \text{ AND } y) !! n = (x !! n \& y !! n) \&$
by *transfer* (*auto simp add: bin-nth-ops*)

lemma *test-bit-numeral* [*simp*]:
 $(\text{numeral } w :: 'a::\text{len0 word}) !! n \longleftrightarrow$
 $n < \text{len-of TYPE}('a) \wedge \text{bin-nth } (\text{numeral } w) n$
by *transfer* (*rule refl*)

lemma *test-bit-neg-numeral* [*simp*]:
 $(\text{neg-numeral } w :: 'a::\text{len0 word}) !! n \longleftrightarrow$
 $n < \text{len-of TYPE}('a) \wedge \text{bin-nth } (\text{neg-numeral } w) n$
by *transfer* (*rule refl*)

lemma *test-bit-1* [*simp*]: $(1::'a::\text{len word}) !! n \longleftrightarrow n = 0$
by *transfer auto*

lemma *nth-0* [*simp*]: $\sim (0::'a::\text{len0 word}) !! n$
by *transfer simp*

lemma *nth-minus1* [*simp*]: $(-1::'a::\text{len0 word}) !! n \longleftrightarrow n < \text{len-of TYPE}('a)$
by *transfer simp*

lemmas *bwsimps* =
wi-hom-add
word-wi-log-defs

lemma *word-bw-assocs*:
fixes $x :: 'a::\text{len0 word}$
shows
 $(x \text{ AND } y) \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $(x \text{ OR } y) \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $(x \text{ XOR } y) \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-bw-comms*:
fixes $x :: 'a::\text{len0 word}$
shows
 $x \text{ AND } y = y \text{ AND } x$

$x \text{ OR } y = y \text{ OR } x$
 $x \text{ XOR } y = y \text{ XOR } x$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-bw-lcs*:

fixes $x :: 'a::len0 \text{ word}$
shows
 $y \text{ AND } x \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $y \text{ OR } x \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $y \text{ XOR } x \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-log-esimps [simp]*:

fixes $x :: 'a::len0 \text{ word}$
shows
 $x \text{ AND } 0 = 0$
 $x \text{ AND } -1 = x$
 $x \text{ OR } 0 = x$
 $x \text{ OR } -1 = -1$
 $x \text{ XOR } 0 = x$
 $x \text{ XOR } -1 = \text{NOT } x$
 $0 \text{ AND } x = 0$
 $-1 \text{ AND } x = x$
 $0 \text{ OR } x = x$
 $-1 \text{ OR } x = -1$
 $0 \text{ XOR } x = x$
 $-1 \text{ XOR } x = \text{NOT } x$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-not-dist*:

fixes $x :: 'a::len0 \text{ word}$
shows
 $\text{NOT } (x \text{ OR } y) = \text{NOT } x \text{ AND } \text{NOT } y$
 $\text{NOT } (x \text{ AND } y) = \text{NOT } x \text{ OR } \text{NOT } y$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-bw-same*:

fixes $x :: 'a::len0 \text{ word}$
shows
 $x \text{ AND } x = x$
 $x \text{ OR } x = x$
 $x \text{ XOR } x = 0$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-ao-absorbs [simp]*:

fixes $x :: 'a::len0 \text{ word}$
shows
 $x \text{ AND } (y \text{ OR } x) = x$
 $x \text{ OR } y \text{ AND } x = x$

$x \text{ AND } (x \text{ OR } y) = x$
 $y \text{ AND } x \text{ OR } x = x$
 $(y \text{ OR } x) \text{ AND } x = x$
 $x \text{ OR } x \text{ AND } y = x$
 $(x \text{ OR } y) \text{ AND } x = x$
 $x \text{ AND } y \text{ OR } x = x$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-not-not* [*simp*]:
 $\text{NOT NOT } (x :: 'a :: \text{len0 word}) = x$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-ao-dist*:
fixes $x :: 'a :: \text{len0 word}$
shows $(x \text{ OR } y) \text{ AND } z = x \text{ AND } z \text{ OR } y \text{ AND } z$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-oa-dist*:
fixes $x :: 'a :: \text{len0 word}$
shows $x \text{ AND } y \text{ OR } z = (x \text{ OR } z) \text{ AND } (y \text{ OR } z)$
by (*auto simp: word-eq-iff word-ops-nth-size [unfolded word-size]*)

lemma *word-add-not* [*simp*]:
fixes $x :: 'a :: \text{len0 word}$
shows $x + \text{NOT } x = -1$
by *transfer (simp add: bin-add-not)*

lemma *word-plus-and-or* [*simp*]:
fixes $x :: 'a :: \text{len0 word}$
shows $(x \text{ AND } y) + (x \text{ OR } y) = x + y$
by *transfer (simp add: plus-and-or)*

lemma *leoa*:
fixes $x :: 'a :: \text{len0 word}$
shows $(w = (x \text{ OR } y)) \implies (y = (w \text{ AND } y))$ **by** *auto*

lemma *leao*:
fixes $x' :: 'a :: \text{len0 word}$
shows $(w' = (x' \text{ AND } y')) \implies (x' = (x' \text{ OR } w'))$ **by** *auto*

lemmas *word-ao-equiv = leao [COMP leoa [COMP iffI]]*

lemma *le-word-or2*: $x \leq x \text{ OR } (y :: 'a :: \text{len0 word})$
unfolding *word-le-def uint-or*
by (*auto intro: le-int-or*)

lemmas *le-word-or1 = xtr3 [OF word-bw-comms (2) le-word-or2]*

lemmas *word-and-le1 = xtr3 [OF word-ao-absorbs (4) [symmetric] le-word-or2]*

lemmas *word-and-le2 = xtr3 [OF word-ao-absorbs (8) [symmetric] le-word-or2]*

lemma *bl-word-not*: $to-bl (NOT w) = map\ Not (to-bl w)$
unfolding *to-bl-def word-log-defs bl-not-bin*
by (*simp add: word-ubin.eq-norm*)

lemma *bl-word-xor*: $to-bl (v XOR w) = map2\ op\ \sim = (to-bl v) (to-bl w)$
unfolding *to-bl-def word-log-defs bl-xor-bin*
by (*simp add: word-ubin.eq-norm*)

lemma *bl-word-or*: $to-bl (v OR w) = map2\ op\ | (to-bl v) (to-bl w)$
unfolding *to-bl-def word-log-defs bl-or-bin*
by (*simp add: word-ubin.eq-norm*)

lemma *bl-word-and*: $to-bl (v AND w) = map2\ op\ \& (to-bl v) (to-bl w)$
unfolding *to-bl-def word-log-defs bl-and-bin*
by (*simp add: word-ubin.eq-norm*)

lemma *word-lsb-alt*: $lsb (w::'a::len0\ word) = test-bit\ w\ 0$
by (*auto simp: word-test-bit-def word-lsb-def*)

lemma *word-lsb-1-0* [*simp*]: $lsb (1::'a::len\ word) \&\ \sim\ lsb (0::'b::len0\ word)$
unfolding *word-lsb-def uint-eq-0 uint-1* **by** *simp*

lemma *word-lsb-last*: $lsb (w::'a::len\ word) = last (to-bl w)$
apply (*unfold word-lsb-def uint-bl bin-to-bl-def*)
apply (*rule-tac bin=uint w in bin-exhaust*)
apply (*cases size w*)
apply *auto*
apply (*auto simp add: bin-to-bl-aux-alt*)
done

lemma *word-lsb-int*: $lsb\ w = (uint\ w\ mod\ 2 = 1)$
unfolding *word-lsb-def bin-last-def* **by** *auto*

lemma *word-msb-sint*: $msb\ w = (sint\ w < 0)$
unfolding *word-msb-def sign-Min-lt-0* **..**

lemma *msb-word-of-int*:
 $msb (word-of-int\ x::'a::len\ word) = bin-nth\ x (len-of\ TYPE('a) - 1)$
unfolding *word-msb-def* **by** (*simp add: word-sbin.eq-norm bin-sign-lem*)

lemma *word-msb-numeral* [*simp*]:
 $msb (numeral\ w::'a::len\ word) = bin-nth (numeral\ w) (len-of\ TYPE('a) - 1)$
unfolding *word-numeral-alt* **by** (*rule msb-word-of-int*)

lemma *word-msb-neg-numeral* [*simp*]:
 $msb (neg-numeral\ w::'a::len\ word) = bin-nth (neg-numeral\ w) (len-of\ TYPE('a) - 1)$
unfolding *word-neg-numeral-alt* **by** (*rule msb-word-of-int*)

lemma *word-msb-0* [*simp*]: $\neg \text{msb } (0::'a::\text{len } \text{word})$
unfolding *word-msb-def* **by** *simp*

lemma *word-msb-1* [*simp*]: $\text{msb } (1::'a::\text{len } \text{word}) \longleftrightarrow \text{len-of } \text{TYPE}('a) = 1$
unfolding *word-1-wi msb-word-of-int eq-iff* [**where** $'a = \text{nat}$]
by (*simp add: Suc-le-eq*)

lemma *word-msb-nth*:
 $\text{msb } (w::'a::\text{len } \text{word}) = \text{bin-nth } (\text{uint } w) (\text{len-of } \text{TYPE}('a) - 1)$
unfolding *word-msb-def sint-uint* **by** (*simp add: bin-sign-lem*)

lemma *word-msb-alt*: $\text{msb } (w::'a::\text{len } \text{word}) = \text{hd } (\text{to-bl } w)$
apply (*unfold word-msb-nth uint-bl*)
apply (*subst hd-conv-nth*)
apply (*rule length-greater-0-conv [THEN iffD1]*)
apply *simp*
apply (*simp add : nth-bin-to-bl word-size*)
done

lemma *word-set-nth* [*simp*]:
 $\text{set-bit } w \ n \ (\text{test-bit } w \ n) = (w::'a::\text{len0 } \text{word})$
unfolding *word-test-bit-def word-set-bit-def* **by** *auto*

lemma *bin-nth-uint'*:
 $\text{bin-nth } (\text{uint } w) \ n = (\text{rev } (\text{bin-to-bl } (\text{size } w) (\text{uint } w))) ! \ n \ \& \ n < \text{size } w$
apply (*unfold word-size*)
apply (*safe elim!: bin-nth-uint-imp*)
apply (*frule bin-nth-uint-imp*)
apply (*fast dest!: bin-nth-bl*)
done

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

lemma *test-bit-bl*: $w \ ! \ n = (\text{rev } (\text{to-bl } w)) ! \ n \ \& \ n < \text{size } w$
unfolding *to-bl-def word-test-bit-def word-size*
by (*rule bin-nth-uint*)

lemma *to-bl-nth*: $n < \text{size } w \implies \text{to-bl } w \ ! \ n = w \ ! \ (\text{size } w - \text{Suc } n)$
apply (*unfold test-bit-bl*)
apply *clarsimp*
apply (*rule trans*)
apply (*rule nth-rev-alt*)
apply (*auto simp add: word-size*)
done

lemma *test-bit-set*:
fixes $w :: 'a::\text{len0 } \text{word}$
shows $(\text{set-bit } w \ n \ x) \ ! \ n = (n < \text{size } w \ \& \ x)$
unfolding *word-size word-test-bit-def word-set-bit-def*

by (clarsimp simp add : word-ubin.eq-norm nth-bintr)

lemma test-bit-set-gen:

fixes w :: 'a::len0 word

shows test-bit (set-bit w n x) m =

(if m = n then n < size w & x else test-bit w m)

apply (unfold word-size word-test-bit-def word-set-bit-def)

apply (clarsimp simp add: word-ubin.eq-norm nth-bintr bin-nth-sc-gen)

apply (auto elim!: test-bit-size [unfolded word-size])

simp add: word-test-bit-def [symmetric])

done

lemma of-bl-rep-False: of-bl (replicate n False @ bs) = of-bl bs

unfolding of-bl-def bl-to-bin-rep-F by auto

lemma msb-nth:

fixes w :: 'a::len word

shows msb w = w !! (len-of TYPE('a) - 1)

unfolding word-msb-nth word-test-bit-def by simp

lemmas msb0 = len-gt-0 [THEN diff-Suc-less, THEN word-ops-nth-size [unfolded word-size]]

lemmas msb1 = msb0 [where i = 0]

lemmas word-ops-msb = msb1 [unfolded msb-nth [symmetric, unfolded One-nat-def]]

lemmas lsb0 = len-gt-0 [THEN word-ops-nth-size [unfolded word-size]]

lemmas word-ops-lsb = lsb0 [unfolded word-lsb-alt]

lemma td-ext-nth [OF refl refl refl, unfolded word-size]:

n = size (w::'a::len0 word) \implies ofn = set-bits \implies [w, ofn g] = l \implies

td-ext test-bit ofn {f. ALL i. f i \implies i < n} (%h i. h i & i < n)

apply (unfold word-size td-ext-def')

apply safe

apply (rule-tac [3] ext)

apply (rule-tac [4] ext)

apply (unfold word-size of-nth-def test-bit-bl)

apply safe

defer

apply (clarsimp simp: word-bl.Abs-inverse)+

apply (rule word-bl.Rep-inverse')

apply (rule sym [THEN trans])

apply (rule bl-of-nth-nth)

apply simp

apply (rule bl-of-nth-inj)

apply (clarsimp simp add : test-bit-bl word-size)

done

interpretation test-bit:

td-ext op !! :: 'a::len0 word => nat => bool

set-bits
 $\{f. \forall i. f\ i \longrightarrow i < \text{len-of TYPE}('a::\text{len0})\}$
 $(\lambda h\ i. h\ i \wedge i < \text{len-of TYPE}('a::\text{len0}))$
by (*rule td-ext-nth*)

lemmas *td-nth = test-bit.td-thm*

lemma *word-set-set-same* [*simp*]:
fixes $w :: 'a::\text{len0}$ *word*
shows *set-bit* (*set-bit* $w\ n\ x$) $n\ y = \text{set-bit } w\ n\ y$
by (*rule word-eqI*) (*simp add : test-bit-set-gen word-size*)

lemma *word-set-set-diff*:
fixes $w :: 'a::\text{len0}$ *word*
assumes $m \sim n$
shows *set-bit* (*set-bit* $w\ m\ x$) $n\ y = \text{set-bit } (\text{set-bit } w\ n\ y)\ m\ x$
by (*rule word-eqI*) (*clarsimp simp add: test-bit-set-gen word-size assms*)

lemma *nth-sint*:
fixes $w :: 'a::\text{len}$ *word*
defines $l \equiv \text{len-of TYPE } ('a)$
shows *bin-nth* (*sint* w) $n = (\text{if } n < l - 1 \text{ then } w\ !!\ n \text{ else } w\ !!\ (l - 1))$
unfolding *sint-uint l-def*
by (*clarsimp simp add: nth-sbintr word-test-bit-def [symmetric]*)

lemma *word-lsb-numeral* [*simp*]:
 $\text{lsb } (\text{numeral } \text{bin} :: 'a :: \text{len } \text{word}) = (\text{bin-last } (\text{numeral } \text{bin}) = 1)$
unfolding *word-lsb-alt test-bit-numeral* **by** *simp*

lemma *word-lsb-neg-numeral* [*simp*]:
 $\text{lsb } (\text{neg-numeral } \text{bin} :: 'a :: \text{len } \text{word}) = (\text{bin-last } (\text{neg-numeral } \text{bin}) = 1)$
unfolding *word-lsb-alt test-bit-neg-numeral* **by** *simp*

lemma *set-bit-word-of-int*:
 $\text{set-bit } (\text{word-of-int } x)\ n\ b = \text{word-of-int } (\text{bin-sc } n\ (\text{if } b \text{ then } 1 \text{ else } 0)\ x)$
unfolding *word-set-bit-def*
apply (*rule word-eqI*)
apply (*simp add: word-size bin-nth-sc-gen word-ubin.eq-norm nth-bintr*)
done

lemma *word-set-numeral* [*simp*]:
 $\text{set-bit } (\text{numeral } \text{bin} :: 'a :: \text{len0 } \text{word})\ n\ b =$
 $\text{word-of-int } (\text{bin-sc } n\ (\text{if } b \text{ then } 1 \text{ else } 0)\ (\text{numeral } \text{bin}))$
unfolding *word-numeral-alt* **by** (*rule set-bit-word-of-int*)

lemma *word-set-neg-numeral* [*simp*]:
 $\text{set-bit } (\text{neg-numeral } \text{bin} :: 'a :: \text{len0 } \text{word})\ n\ b =$
 $\text{word-of-int } (\text{bin-sc } n\ (\text{if } b \text{ then } 1 \text{ else } 0)\ (\text{neg-numeral } \text{bin}))$
unfolding *word-neg-numeral-alt* **by** (*rule set-bit-word-of-int*)

```

lemma word-set-bit-0 [simp]:
  set-bit 0 n b = word-of-int (bin-sc n (if b then 1 else 0) 0)
  unfolding word-0-wi by (rule set-bit-word-of-int)

lemma word-set-bit-1 [simp]:
  set-bit 1 n b = word-of-int (bin-sc n (if b then 1 else 0) 1)
  unfolding word-1-wi by (rule set-bit-word-of-int)

lemma setBit-no [simp]:
  setBit (numeral bin) n = word-of-int (bin-sc n 1 (numeral bin))
  by (simp add: setBit-def)

lemma clearBit-no [simp]:
  clearBit (numeral bin) n = word-of-int (bin-sc n 0 (numeral bin))
  by (simp add: clearBit-def)

lemma to-bl-n1:
  to-bl (-1::'a::len0 word) = replicate (len-of TYPE ('a)) True
  apply (rule word-bl.Abs-inverse')
  apply simp
  apply (rule word-eqI)
  apply (clarsimp simp add: word-size)
  apply (auto simp add: word-bl.Abs-inverse test-bit-bl word-size)
  done

lemma word-msb-n1 [simp]: msb (-1::'a::len word)
  unfolding word-msb-alt to-bl-n1 by simp

lemma word-set-nth-iff:
  (set-bit w n b = w) = (w !! n = b | n >= size (w::'a::len0 word))
  apply (rule iffI)
  apply (rule disjCI)
  apply (drule word-eqD)
  apply (erule sym [THEN trans])
  apply (simp add: test-bit-set)
  apply (erule disjE)
  applyclarsimp
  apply (rule word-eqI)
  apply (clarsimp simp add: test-bit-set-gen)
  apply (drule test-bit-size)
  applyforce
  done

lemma test-bit-2p:
  (word-of-int (2 ^ n)::'a::len word) !! m  $\longleftrightarrow$  m = n  $\wedge$  m < len-of TYPE ('a)
  unfolding word-test-bit-def
  by (auto simp add: word-ubin.eq-norm nth-bintr nth-2p-bin)

```

```

lemma nth-w2p:
  ((2::a::len word) ^ n) !! m  $\longleftrightarrow$  m = n  $\wedge$  m < len-of TYPE('a::len)
  unfolding test-bit-2p [symmetric] word-of-int [symmetric]
  by (simp add: of-int-power)

lemma uint-2p:
  (0::a::len word) < 2 ^ n  $\implies$  uint (2 ^ n::a::len word) = 2 ^ n
  apply (unfold word-arith-power-alt)
  apply (case-tac len-of TYPE ('a))
  apply clarsimp
  apply (case-tac nat)
  apply clarsimp
  apply (case-tac n)
  apply clarsimp
  apply clarsimp
  apply (drule word-gt-0 [THEN iffD1])
  apply (safe intro!: word-eqI bin-nth-lem)
  apply (auto simp add: test-bit-2p nth-2p-bin word-test-bit-def [symmetric])
  done

lemma word-of-int-2p: (word-of-int (2 ^ n) :: 'a :: len word) = 2 ^ n
  apply (unfold word-arith-power-alt)
  apply (case-tac len-of TYPE ('a))
  apply clarsimp
  apply (case-tac nat)
  apply (rule word-ubin.norm-eq-iff [THEN iffD1])
  apply (rule box-equals)
  apply (rule-tac [2] bintr-ariths (1)+)
  apply simp
  apply simp
  done

lemma bang-is-le: x !! m  $\implies$  2 ^ m <= (x :: 'a :: len word)
  apply (rule xtr3)
  apply (rule-tac [2] y = x in le-word-or2)
  apply (rule word-eqI)
  apply (auto simp add: word-ao-nth nth-w2p word-size)
  done

lemma word-clr-le:
  fixes w :: 'a::len0 word
  shows w >= set-bit w n False
  apply (unfold word-set-bit-def word-le-def word-ubin.eq-norm)
  apply simp
  apply (rule order-trans)
  apply (rule bintr-bin-clr-le)
  apply simp
  done

```

```

lemma word-set-ge:
  fixes  $w :: 'a::len\ word$ 
  shows  $w \leq set\_bit\ w\ n\ True$ 
  apply (unfold word-set-bit-def word-le-def word-ubin.eq-norm)
  apply simp
  apply (rule order-trans [OF - bintr-bin-set-ge])
  apply simp
  done

```

13.25 Shifting, Rotating, and Splitting Words

```

lemma shiffl1-wi [simp]:  $shiffl1\ (word\_of\_int\ w) = word\_of\_int\ (w\ BIT\ 0)$ 
  unfolding shiffl1-def
  apply (simp add: word-ubin.norm-eq-iff [symmetric] word-ubin.eq-norm)
  apply (subst refl [THEN bintrunc-BIT-I, symmetric])
  apply (subst bintrunc-bintrunc-min)
  apply simp
  done

```

```

lemma shiffl1-numeral [simp]:
   $shiffl1\ (numeral\ w) = numeral\ (Num.Bit0\ w)$ 
  unfolding word-numeral-alt shiffl1-wi by simp

```

```

lemma shiffl1-neg-numeral [simp]:
   $shiffl1\ (neg\_numeral\ w) = neg\_numeral\ (Num.Bit0\ w)$ 
  unfolding word-neg-numeral-alt shiffl1-wi by simp

```

```

lemma shiffl1-0 [simp]:  $shiffl1\ 0 = 0$ 
  unfolding shiffl1-def by simp

```

```

lemma shiffl1-def-u:  $shiffl1\ w = word\_of\_int\ (uint\ w\ BIT\ 0)$ 
  by (simp only: shiffl1-def)

```

```

lemma shiffl1-def-s:  $shiffl1\ w = word\_of\_int\ (sint\ w\ BIT\ 0)$ 
  unfolding shiffl1-def Bit-B0 wi-hom-syms by simp

```

```

lemma shiftr1-0 [simp]:  $shiftr1\ 0 = 0$ 
  unfolding shiftr1-def by simp

```

```

lemma sshiftr1-0 [simp]:  $sshiftr1\ 0 = 0$ 
  unfolding sshiftr1-def by simp

```

```

lemma sshiftr1-n1 [simp]:  $sshiftr1\ -1 = -1$ 
  unfolding sshiftr1-def by simp

```

```

lemma shiffl-0 [simp]:  $(0::'a::len0\ word) \ll n = 0$ 
  unfolding shiffl-def by (induct n) auto

```

```

lemma shiftr-0 [simp]:  $(0::'a::len0\ word) \gg n = 0$ 

```

```

unfolding shiftr-def by (induct n) auto

lemma sshiftr-0 [simp] :  $0 \gg \gg n = 0$ 
unfolding sshiftr-def by (induct n) auto

lemma sshiftr-n1 [simp] :  $-1 \gg \gg n = -1$ 
unfolding sshiftr-def by (induct n) auto

lemma nth-shiffl1:  $\text{shiffl1 } w \text{ !! } n = (n < \text{size } w \ \& \ n > 0 \ \& \ w \text{ !! } (n - 1))$ 
apply (unfold shiffl1-def word-test-bit-def)
apply (simp add: nth-bintr word-ubin.eq-norm word-size)
apply (cases n)
apply auto
done

lemma nth-shiffl' [rule-format]:
  ALL n. ((w::'a::len0 word) << m) !! n = (n < size w & n >= m & w !! (n - m))
apply (unfold shiffl-def)
apply (induct m)
apply (force elim!: test-bit-size)
apply (clarsimp simp add : nth-shiffl1 word-size)
apply arith
done

lemmas nth-shiffl = nth-shiffl' [unfolded word-size]

lemma nth-shiftr1:  $\text{shiftr1 } w \text{ !! } n = w \text{ !! } \text{Suc } n$ 
apply (unfold shiftr1-def word-test-bit-def)
apply (simp add: nth-bintr word-ubin.eq-norm)
apply safe
apply (drule bin-nth.Suc [THEN iffD2, THEN bin-nth-uint-imp])
apply simp
done

lemma nth-shiftr:
   $\bigwedge n. ((w::'a::len0 word) \gg m) !! n = w \text{ !! } (n + m)$ 
apply (unfold shiftr-def)
apply (induct m)
apply (auto simp add : nth-shiftr1)
done

lemma uint-shiftr1:  $\text{uint } (\text{shiftr1 } w) = \text{bin-rest } (\text{uint } w)$ 
apply (unfold shiftr1-def word-ubin.eq-norm bin-rest-trunc-i)
apply (subst bintr-uint [symmetric, OF order-refl])
apply (simp only : bintrunc-bintrunc-l)
apply simp

```

done

lemma *nth-sshiftr1* :

sshiftr1 w !! n = (if n = size w - 1 then w !! n else w !! Suc n)

apply (*unfold sshiftr1-def word-test-bit-def*)

apply (*simp add: nth-bintr word-ubin.eq-norm*
bin-nth.Suc [symmetric] word-size
del: bin-nth.simps)

apply (*simp add: nth-bintr uint-sint del : bin-nth.simps*)

apply (*auto simp add: bin-nth-sint*)

done

lemma *nth-sshiftr [rule-format]* :

ALL n. sshiftr w m !! n = (n < size w &

(if n + m >= size w then w !! (size w - 1) else w !! (n + m)))

apply (*unfold sshiftr-def*)

apply (*induct-tac m*)

apply (*simp add: test-bit-bl*)

apply (*clarsimp simp add: nth-sshiftr1 word-size*)

apply *safe*

apply *arith*

apply *arith*

apply (*erule thin-rl*)

apply (*case-tac n*)

apply *safe*

apply *simp*

apply *simp*

apply (*erule thin-rl*)

apply (*case-tac n*)

apply *safe*

apply *simp*

apply *simp*

apply *arith+*

done

lemma *shiftr1-div-2*: *uint (shiftr1 w) = uint w div 2*

apply (*unfold shiftr1-def bin-rest-def*)

apply (*rule word-uint.Abs-inverse*)

apply (*simp add: uints-num pos-imp-zdiv-nonneg-iff*)

apply (*rule xtr7*)

prefer 2

apply (*rule zdiv-le-dividend*)

apply *auto*

done

lemma *sshiftr1-div-2*: *sint (sshiftr1 w) = sint w div 2*

apply (*unfold sshiftr1-def bin-rest-def [symmetric]*)

apply (*simp add: word-sbin.eq-norm*)

apply (*rule trans*)

```

defer
apply (subst word-sbin.norm-Rep [symmetric])
apply (rule refl)
apply (subst word-sbin.norm-Rep [symmetric])
apply (unfold One-nat-def)
apply (rule sbintrunc-rest)
done

```

```

lemma shiftr-div-2n:  $uint (shiftr w n) = uint w \text{ div } 2 \wedge n$ 
apply (unfold shiftr-def)
apply (induct n)
apply simp
apply (simp add: shiftr1-div-2 mult-commute
        zdiv-zmult2-eq [symmetric])
done

```

```

lemma sshiftr-div-2n:  $sint (sshiftr w n) = sint w \text{ div } 2 \wedge n$ 
apply (unfold sshiftr-def)
apply (induct n)
apply simp
apply (simp add: sshiftr1-div-2 mult-commute
        zdiv-zmult2-eq [symmetric])
done

```

13.25.1 shift functions in terms of lists of bools

```

lemmas bshiftr1-numeral [simp] =
bshiftr1-def [where w=numeral w, unfolded to-bl-numeral] for w

```

```

lemma bshiftr1-bl:  $to-bl (bshiftr1 b w) = b \# butlast (to-bl w)$ 
unfolding bshiftr1-def by (rule word-bl.Abs-inverse) simp

```

```

lemma shiftl1-of-bl:  $shiftl1 (of-bl bl) = of-bl (bl @ [False])$ 
by (simp add: of-bl-def bl-to-bin-append)

```

```

lemma shiftl1-bl:  $shiftl1 (w :: 'a :: len0 \text{ word}) = of-bl (to-bl w @ [False])$ 

```

proof –

```

have shiftl1 w = shiftl1 (of-bl (to-bl w)) by simp
also have  $\dots = of-bl (to-bl w @ [False])$  by (rule shiftl1-of-bl)
finally show ?thesis .

```

qed

```

lemma bl-shiftl1:
 $to-bl (shiftl1 (w :: 'a :: len \text{ word})) = tl (to-bl w) @ [False]$ 
apply (simp add: shiftl1-bl word-rep-drop drop-Suc drop-Cons)
apply (fast intro!: Suc-leI)
done

```

```

lemma bl-shiftl1':
  to-bl (shiftl1 w) = tl (to-bl w @ [False])
  unfolding shiftl1-bl
  by (simp add: word-rep-drop drop-Suc del: drop-append)

lemma shiftr1-bl: shiftr1 w = of-bl (butlast (to-bl w))
  apply (unfold shiftr1-def wint-bl of-bl-def)
  apply (simp add: butlast-rest-bin word-size)
  apply (simp add: bin-rest-trunc [symmetric, unfolded One-nat-def])
  done

lemma bl-shiftr1:
  to-bl (shiftr1 (w :: 'a :: len word)) = False # butlast (to-bl w)
  unfolding shiftr1-bl
  by (simp add : word-rep-drop len-gt-0 [THEN Suc-leI])

lemma bl-shiftr1':
  to-bl (shiftr1 w) = butlast (False # to-bl w)
  apply (rule word-bl.Abs-inverse')
  apply (simp del: butlast.simps)
  apply (simp add: shiftr1-bl of-bl-def)
  done

lemma shiftl1-rev:
  shiftl1 w = word-reverse (shiftr1 (word-reverse w))
  apply (unfold word-reverse-def)
  apply (rule word-bl.Rep-inverse' [symmetric])
  apply (simp add: bl-shiftl1' bl-shiftr1' word-bl.Abs-inverse)
  apply (cases to-bl w)
  apply auto
  done

lemma shiftr1-rev:
  shiftr1 w n = word-reverse (shiftr (word-reverse w) n)
  apply (unfold shiftr1-def shiftr-def)
  apply (induct n)
  apply (auto simp add : shiftr1-rev)
  done

lemma rev-shiftl: word-reverse w << n = word-reverse (w >> n)
  by (simp add: shiftl-rev)

lemma shiftr-rev: w >> n = word-reverse (word-reverse w << n)
  by (simp add: rev-shiftr)

lemma rev-shiftr: word-reverse w >> n = word-reverse (w << n)
  by (simp add: shiftr-rev)

```

```

lemma bl-sshiftr1:
  to-bl (sshiftr1 (w :: 'a :: len word)) = hd (to-bl w) # butlast (to-bl w)
  apply (unfold sshiftr1-def uint-bl word-size)
  apply (simp add: butlast-rest-bin word-ubin.eq-norm)
  apply (simp add: sint-uint)
  apply (rule nth-equalityI)
  apply clarsimp
  apply clarsimp
  apply (case-tac i)
  apply (simp-all add: hd-conv-nth length-0-conv [symmetric]
          nth-bin-to-bl bin-nth.Suc [symmetric]
          nth-sbintr
          del: bin-nth.Suc)
  apply force
  apply (rule impI)
  apply (rule-tac f = bin-nth (uint w) in arg-cong)
  apply simp
  done

lemma drop-shiftr:
  drop n (to-bl ((w :: 'a :: len word) >> n)) = take (size w - n) (to-bl w)
  apply (unfold shiftr-def)
  apply (induct n)
  prefer 2
  apply (simp add: drop-Suc bl-shiftr1 butlast-drop [symmetric])
  apply (rule butlast-take [THEN trans])
  apply (auto simp: word-size)
  done

lemma drop-sshiftr:
  drop n (to-bl ((w :: 'a :: len word) >>> n)) = take (size w - n) (to-bl w)
  apply (unfold sshiftr-def)
  apply (induct n)
  prefer 2
  apply (simp add: drop-Suc bl-sshiftr1 butlast-drop [symmetric])
  apply (rule butlast-take [THEN trans])
  apply (auto simp: word-size)
  done

lemma take-shiftr:
  n ≤ size w ⇒ take n (to-bl (w >> n)) = replicate n False
  apply (unfold shiftr-def)
  apply (induct n)
  prefer 2
  apply (simp add: bl-shiftr1' length-0-conv [symmetric] word-size)
  apply (rule take-butlast [THEN trans])
  apply (auto simp: word-size)
  done

```

```

lemma take-sshiftr' [rule-format] :
  n <= size (w :: 'a :: len word) --> hd (to-bl (w >>> n)) = hd (to-bl w) &
  take n (to-bl (w >>> n)) = replicate n (hd (to-bl w))
apply (unfold sshiftr-def)
apply (induct n)
prefer 2
apply (simp add: bl-sshiftr1)
apply (rule impI)
apply (rule take-butlast [THEN trans])
apply (auto simp: word-size)
done

```

```

lemmas hd-sshiftr = take-sshiftr' [THEN conjunct1]
lemmas take-sshiftr = take-sshiftr' [THEN conjunct2]

```

```

lemma atd-lem: take n xs = t ==> drop n xs = d ==> xs = t @ d
  by (auto intro: append-take-drop-id [symmetric])

```

```

lemmas bl-shiftr = atd-lem [OF take-shiftr drop-shiftr]
lemmas bl-sshiftr = atd-lem [OF take-sshiftr drop-sshiftr]

```

```

lemma shiftl-of-bl: of-bl bl << n = of-bl (bl @ replicate n False)
  unfolding shiftl-def
  by (induct n) (auto simp: shiftl1-of-bl replicate-app-Cons-same)

```

```

lemma shiftl-bl:
  (w::'a::len0 word) << (n::nat) = of-bl (to-bl w @ replicate n False)
proof -
  have w << n = of-bl (to-bl w) << n by simp
  also have ... = of-bl (to-bl w @ replicate n False) by (rule shiftl-of-bl)
  finally show ?thesis .
qed

```

```

lemmas shiftl-numeral [simp] = shiftl-def [where w=numeral w] for w

```

```

lemma bl-shiftl:
  to-bl (w << n) = drop n (to-bl w) @ replicate (min (size w) n) False
  by (simp add: shiftl-bl word-rep-drop word-size)

```

```

lemma shiftl-zero-size:
  fixes x :: 'a::len0 word
  shows size x <= n ==> x << n = 0
  apply (unfold word-size)
  apply (rule word-eqI)
  apply (clarsimp simp add: shiftl-bl word-size test-bit-of-bl nth-append)
done

```

```

lemma shiffl1-2t: shiffl1 (w :: 'a :: len word) = 2 * w
  by (simp add: shiffl1-def Bit-def wi-hom-mult [symmetric])

lemma shiffl1-p: shiffl1 (w :: 'a :: len word) = w + w
  by (simp add: shiffl1-2t)

lemma shiffl-t2n: shiffl (w :: 'a :: len word) n = 2 ^ n * w
  unfolding shiffl-def
  by (induct n) (auto simp: shiffl1-2t)

lemma shiftr1-bintr [simp]:
  (shiftr1 (numeral w) :: 'a :: len0 word) =
    word-of-int (bin-rest (bintrunc (len-of TYPE ('a)) (numeral w)))
  unfolding shiftr1-def word-numeral-alt
  by (simp add: word-ubin.eq-norm)

lemma sshiftr1-sbintr [simp]:
  (sshiftr1 (numeral w) :: 'a :: len word) =
    word-of-int (bin-rest (sbintrunc (len-of TYPE ('a) - 1) (numeral w)))
  unfolding sshiftr1-def word-numeral-alt
  by (simp add: word-sbin.eq-norm)

lemma shiftr-no [simp]:

  (numeral w::'a::len0 word) >> n = word-of-int
    ((bin-rest ^ n) (bintrunc (len-of TYPE('a)) (numeral w)))
  apply (rule word-eqI)
  apply (auto simp: nth-shiftr nth-rest-power-bin nth-bintr word-size)
  done

lemma sshiftr-no [simp]:

  (numeral w::'a::len word) >>> n = word-of-int
    ((bin-rest ^ n) (sbintrunc (len-of TYPE('a) - 1) (numeral w)))
  apply (rule word-eqI)
  apply (auto simp: nth-sshiftr nth-rest-power-bin nth-sbintr word-size)
  apply (subgoal-tac na + n = len-of TYPE('a) - Suc 0, simp, simp)
  done

lemma shiftr1-bl-of:
  length bl ≤ len-of TYPE('a) ⇒
    shiftr1 (of-bl bl::'a::len0 word) = of-bl (butlast bl)
  by (clarsimp simp: shiftr1-def of-bl-def butlast-rest-bl2bin
    word-ubin.eq-norm trunc-bl2bin)

lemma shiftr-bl-of:
  length bl ≤ len-of TYPE('a) ⇒
    (of-bl bl::'a::len0 word) >> n = of-bl (take (length bl - n) bl)
  apply (unfold shiftr-def)

```

```

apply (induct n)
apply clarsimp
apply clarsimp
apply (subst shiftr1-bl-of)
apply simp
apply (simp add: butlast-take)
done

```

```

lemma shiftr-bl:
  (x::'a::len0 word) >> n  $\equiv$  of-bl (take (len-of TYPE('a) - n) (to-bl x))
  using shiftr-bl-of [where 'a='a, of to-bl x] by simp

```

```

lemma msb-shift:
  msb (w::'a::len word)  $\longleftrightarrow$  (w >> (len-of TYPE('a) - 1))  $\neq$  0
apply (unfold shiftr-bl word-msb-alt)
apply (simp add: word-size Suc-le-eq take-Suc)
apply (cases hd (to-bl w))
apply (auto simp: word-1-bl
              of-bl-rep-False [where n=1 and bs=[], simplified])
done

```

```

lemma align-lem-or [rule-format] :
  ALL x m. length x = n + m  $\longrightarrow$  length y = n + m  $\longrightarrow$ 
  drop m x = replicate n False  $\longrightarrow$  take m y = replicate m False  $\longrightarrow$ 
  map2 op | x y = take m x @ drop m y
apply (induct-tac y)
apply force
apply clarsimp
apply (case-tac x, force)
apply (case-tac m, auto)
apply (drule sym)
apply auto
apply (induct-tac list, auto)
done

```

```

lemma align-lem-and [rule-format] :
  ALL x m. length x = n + m  $\longrightarrow$  length y = n + m  $\longrightarrow$ 
  drop m x = replicate n False  $\longrightarrow$  take m y = replicate m False  $\longrightarrow$ 
  map2 op & x y = replicate (n + m) False
apply (induct-tac y)
apply force
apply clarsimp
apply (case-tac x, force)
apply (case-tac m, auto)
apply (drule sym)
apply auto
apply (induct-tac list, auto)
done

```

lemma *aligned-bl-add-size* [*OF refl*]:
 $size\ x - n = m \implies n \leq size\ x \implies drop\ m\ (to-bl\ x) = replicate\ n\ False \implies$
 $take\ m\ (to-bl\ y) = replicate\ m\ False \implies$
 $to-bl\ (x + y) = take\ m\ (to-bl\ x) @ drop\ m\ (to-bl\ y)$
apply (*subgoal-tac* $x\ AND\ y = 0$)
prefer 2
apply (*rule* *word-bl.Rep-eqD*)
apply (*simp* *add: bl-word-and*)
apply (*rule* *align-lem-and* [*THEN* *trans*])
apply (*simp-all* *add: word-size*)[5]
apply *simp*
apply (*subst* *word-plus-and-or* [*symmetric*])
apply (*simp* *add: bl-word-or*)
apply (*rule* *align-lem-or*)
apply (*simp-all* *add: word-size*)
done

13.25.2 Mask

lemma *nth-mask* [*OF refl, simp*]:
 $m = mask\ n \implies test-bit\ m\ i = (i < n \ \&\ i < size\ m)$
apply (*unfold* *mask-def* *test-bit-bl*)
apply (*simp* *only: word-1-bl* [*symmetric*] *shiffl-of-bl*)
apply (*clarsimp* *simp* *add: word-size*)
apply (*simp* *only: of-bl-def* *mask-lem* *word-of-int-hom-syms* *add-diff-cancel2*)
apply (*fold* *of-bl-def*)
apply (*simp* *add: word-1-bl*)
apply (*rule* *test-bit-of-bl* [*THEN* *trans, unfolded* *test-bit-bl* *word-size*])
apply *auto*
done

lemma *mask-bl*: $mask\ n = of-bl\ (replicate\ n\ True)$
by (*auto* *simp* *add: test-bit-of-bl* *word-size* *intro: word-eqI*)

lemma *mask-bin*: $mask\ n = word-of-int\ (bintrunc\ n - 1)$
by (*auto* *simp* *add: nth-bintr* *word-size* *intro: word-eqI*)

lemma *and-mask-bintr*: $w\ AND\ mask\ n = word-of-int\ (bintrunc\ n\ (uint\ w))$
apply (*rule* *word-eqI*)
apply (*simp* *add: nth-bintr* *word-size* *word-ops-nth-size*)
apply (*auto* *simp* *add: test-bit-bin*)
done

lemma *and-mask-wi*: $word-of-int\ i\ AND\ mask\ n = word-of-int\ (bintrunc\ n\ i)$
by (*auto* *simp* *add: nth-bintr* *word-size* *word-ops-nth-size* *word-eq-iff*)

lemma *and-mask-no*: $numeral\ i\ AND\ mask\ n = word-of-int\ (bintrunc\ n\ (numeral\ i))$
unfolding *word-numeral-alt* **by** (*rule* *and-mask-wi*)

lemma *bl-and-mask'*:

```

to-bl (w AND mask n :: 'a :: len word) =
  replicate (len-of TYPE('a) - n) False @
  drop (len-of TYPE('a) - n) (to-bl w)
apply (rule nth-equalityI)
apply simp
apply (clarsimp simp add: to-bl-nth word-size)
apply (simp add: word-size word-ops-nth-size)
apply (auto simp add: word-size test-bit-bl nth-append nth-rev)
done

```

lemma *and-mask-mod-2p*: $w \text{ AND } \text{mask } n = \text{word-of-int } (\text{uint } w \text{ mod } 2 \wedge n)$
by (*simp only*: *and-mask-bintr bintrunc-mod2p*)

lemma *and-mask-lt-2p*: $\text{uint } (w \text{ AND } \text{mask } n) < 2 \wedge n$
apply (*simp add*: *and-mask-bintr word-ubin.eq-norm*)
apply (*simp add*: *bintrunc-mod2p*)
apply (rule *xtr8*)
prefer 2
apply (rule *pos-mod-bound*)
apply *auto*
done

lemma *eq-mod-iff*: $0 < (n::\text{int}) \implies b = b \text{ mod } n \iff 0 \leq b \wedge b < n$
by (*simp add*: *int-mod-lem eq-sym-conv*)

lemma *mask-eq-iff*: $(w \text{ AND } \text{mask } n) = w \iff \text{uint } w < 2 \wedge n$
apply (*simp add*: *and-mask-bintr*)
apply (*simp add*: *word-ubin.inverse-norm*)
apply (*simp add*: *eq-mod-iff bintrunc-mod2p min-def*)
apply (*fast intro!*: *lt2p-lem*)
done

lemma *and-mask-dvd*: $2 \wedge n \text{ dvd } \text{uint } w = (w \text{ AND } \text{mask } n = 0)$
apply (*simp add*: *dvd-eq-mod-eq-0 and-mask-mod-2p*)
apply (*simp add*: *word-uint.norm-eq-iff [symmetric] word-of-int-homs*
del: *word-of-int-0*)
apply (*subst word-uint.norm-Rep [symmetric]*)
apply (*simp only*: *bintrunc-bintrunc-min bintrunc-mod2p [symmetric] min-def*)
apply *auto*
done

lemma *and-mask-dvd-nat*: $2 \wedge n \text{ dvd } \text{unat } w = (w \text{ AND } \text{mask } n = 0)$
apply (*unfold unat-def*)
apply (rule *trans [OF - and-mask-dvd]*)
apply (*unfold dvd-def*)
apply *auto*
apply (*drule uint-ge-0 [THEN nat-int.Abs-inverse' [simplified], symmetric]*)

```

apply (simp add : int-mult int-power)
apply (simp add : nat-mult-distrib nat-power-eq)
done

```

lemma *word-2p-lem*:

```

 $n < \text{size } w \implies w < 2 \wedge n = (\text{uint } (w :: 'a :: \text{len } \text{word}) < 2 \wedge n)$ 
apply (unfold word-size word-less-alt word-numeral-alt)
apply (clarsimp simp add: word-of-int-power-hom word-uint.eq-norm
        int-mod-eq'
        simp del: word-of-int-numeral)
done

```

lemma *less-mask-eq*: $x < 2 \wedge n \implies x \text{ AND } \text{mask } n = (x :: 'a :: \text{len } \text{word})$

```

apply (unfold word-less-alt word-numeral-alt)
apply (clarsimp simp add: and-mask-mod-2p word-of-int-power-hom
        word-uint.eq-norm
        simp del: word-of-int-numeral)
apply (drule xtr8 [rotated])
apply (rule int-mod-le)
apply (auto simp add : mod-pos-pos-trivial)
done

```

lemmas *mask-eq-iff-w2p* = trans [OF *mask-eq-iff word-2p-lem* [symmetric]]

lemmas *and-mask-less'* = iffD2 [OF *word-2p-lem and-mask-lt-2p*, simplified *word-size*]

lemma *and-mask-less-size*: $n < \text{size } x \implies x \text{ AND } \text{mask } n < 2 \wedge n$
unfolding *word-size* **by** (erule *and-mask-less'*)

lemma *word-mod-2p-is-mask* [OF *refl*]:

```

 $c = 2 \wedge n \implies c > 0 \implies x \text{ mod } c = (x :: 'a :: \text{len } \text{word}) \text{ AND } \text{mask } n$ 
by (clarsimp simp add: word-mod-def uint-2p and-mask-mod-2p)

```

lemma *mask-egs*:

```

 $(a \text{ AND } \text{mask } n) + b \text{ AND } \text{mask } n = a + b \text{ AND } \text{mask } n$ 
 $a + (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a + b \text{ AND } \text{mask } n$ 
 $(a \text{ AND } \text{mask } n) - b \text{ AND } \text{mask } n = a - b \text{ AND } \text{mask } n$ 
 $a - (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a - b \text{ AND } \text{mask } n$ 
 $a * (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a * b \text{ AND } \text{mask } n$ 
 $(b \text{ AND } \text{mask } n) * a \text{ AND } \text{mask } n = b * a \text{ AND } \text{mask } n$ 
 $(a \text{ AND } \text{mask } n) + (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a + b \text{ AND } \text{mask } n$ 
 $(a \text{ AND } \text{mask } n) - (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a - b \text{ AND } \text{mask } n$ 
 $(a \text{ AND } \text{mask } n) * (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a * b \text{ AND } \text{mask } n$ 
 $-(a \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = -a \text{ AND } \text{mask } n$ 
 $\text{word-succ } (a \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = \text{word-succ } a \text{ AND } \text{mask } n$ 
 $\text{word-pred } (a \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = \text{word-pred } a \text{ AND } \text{mask } n$ 
using word-of-int-Ex [where  $x=a$ ] word-of-int-Ex [where  $x=b$ ]
by (auto simp: and-mask-wi bintr-ariths bintr-arith1s word-of-int-homs)

```

lemma *mask-power-eq*:
 $(x \text{ AND } \text{mask } n) \wedge k \text{ AND } \text{mask } n = x \wedge k \text{ AND } \text{mask } n$
using *word-of-int-Ex* [**where** $x=x$]
by (*clarsimp simp: and-mask-wi word-of-int-power-hom bintr-ariths*)

13.25.3 Recast

lemmas *recast-def' = recast-def* [*simplified*]
lemmas *recast-def'' = recast-def'* [*simplified word-size*]
lemmas *recast-no-def* [*simp*] = *recast-def'* [**where** $w=\text{numeral } w, \text{ unfolded word-size}$]
for w

lemma *to-bl-recast*:
 $\text{to-bl } (\text{recast } w :: 'a :: \text{len0 word}) =$
 $\text{takefill False } (\text{len-of TYPE } ('a)) (\text{to-bl } w)$
apply (*unfold recast-def' word-size*)
apply (*rule word-bl.Abs-inverse*)
apply *simp*
done

lemma *recast-rev-ucast* [*OF refl refl refl*]:
 $cs = [rc, uc] \implies rc = \text{recast } (\text{word-reverse } w) \implies uc = \text{ucast } w \implies$
 $rc = \text{word-reverse } uc$
apply (*unfold ucast-def recast-def' Let-def word-reverse-def*)
apply (*clarsimp simp add : to-bl-of-bin takefill-bintrunc*)
apply (*simp add : word-bl.Abs-inverse word-size*)
done

lemma *recast-ucast: recast w = word-reverse (ucast (word-reverse w))*
using *recast-rev-ucast* [*of word-reverse w*] **by** *simp*

lemma *ucast-recast: ucast w = word-reverse (recast (word-reverse w))*
by (*fact recast-rev-ucast* [*THEN word-rev-gal'*])

lemma *ucast-rec-recast: ucast (word-reverse w) = word-reverse (recast w)*
by (*fact recast-ucast* [*THEN word-rev-gal'*])

— linking *recast* and *cast* via *shift*

lemmas *wsst-TYs = source-size target-size word-size*

lemma *recast-down-uu* [*OF refl*]:
 $rc = \text{recast} \implies \text{source-size } rc = \text{target-size } rc + n \implies$
 $rc (w :: 'a :: \text{len word}) = \text{ucast } (w \gg n)$
apply (*simp add: recast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans, rule ucast-down-drop*)
prefer 2

```

apply (rule trans, rule drop-shiftr)
apply (auto simp: takefill-alt wsst-TYs)
done

```

```

lemma revcast-down-us [OF refl]:
  rc = revcast  $\implies$  source-size rc = target-size rc + n  $\implies$ 
    rc (w :: 'a :: len word) = ucast (w >>> n)
apply (simp add: revcast-def')
apply (rule word-bl.Rep-inverse')
apply (rule trans, rule ucast-down-drop)
prefer 2
apply (rule trans, rule drop-sshiftr)
apply (auto simp: takefill-alt wsst-TYs)
done

```

```

lemma revcast-down-su [OF refl]:
  rc = revcast  $\implies$  source-size rc = target-size rc + n  $\implies$ 
    rc (w :: 'a :: len word) = scast (w >> n)
apply (simp add: revcast-def')
apply (rule word-bl.Rep-inverse')
apply (rule trans, rule scast-down-drop)
prefer 2
apply (rule trans, rule drop-shiftr)
apply (auto simp: takefill-alt wsst-TYs)
done

```

```

lemma revcast-down-ss [OF refl]:
  rc = revcast  $\implies$  source-size rc = target-size rc + n  $\implies$ 
    rc (w :: 'a :: len word) = scast (w >>> n)
apply (simp add: revcast-def')
apply (rule word-bl.Rep-inverse')
apply (rule trans, rule scast-down-drop)
prefer 2
apply (rule trans, rule drop-sshiftr)
apply (auto simp: takefill-alt wsst-TYs)
done

```

```

lemma cast-down-rev:
  uc = ucast  $\implies$  source-size uc = target-size uc + n  $\implies$ 
    uc w = revcast ((w :: 'a :: len word) <<< n)
apply (unfold shiftl-rev)
apply clarify
apply (simp add: revcast-rev-ucast)
apply (rule word-rev-gal')
apply (rule trans [OF - revcast-rev-ucast])
apply (rule revcast-down-uu [symmetric])
apply (auto simp add: wsst-TYs)
done

```

lemma *revcast-up* [*OF refl*]:

```
rc = revcast  $\implies$  source-size rc + n = target-size rc  $\implies$ 
  rc w = (ucast w :: 'a :: len word) << n
apply (simp add: revcast-def')
apply (rule word-bl.Rep-inverse')
apply (simp add: takefill-alt)
apply (rule bl-shiftr [THEN trans])
apply (subst ucast-up-app)
apply (auto simp add: wsst-TYs)
done
```

lemmas *rc1 = revcast-up* [THEN

revcast-rev-ucast [symmetric, THEN trans, THEN word-rev-gal, symmetric]]

lemmas *rc2 = revcast-down-uu* [THEN

revcast-rev-ucast [symmetric, THEN trans, THEN word-rev-gal, symmetric]]

lemmas *ucast-up =*

rc1 [simplified rev-shiftr [symmetric] revcast-ucast [symmetric]]

lemmas *ucast-down =*

rc2 [simplified rev-shiftr revcast-ucast [symmetric]]

13.25.4 Slices

lemma *slice1-no-bin* [simp]:

```
slice1 n (numeral w :: 'b word) = of-bl (takefill False n (bin-to-bl (len-of TYPE('b
:: len0)) (numeral w)))
by (simp add: slice1-def)
```

lemma *slice-no-bin* [simp]:

```
slice n (numeral w :: 'b word) = of-bl (takefill False (len-of TYPE('b :: len0) -
n)
  (bin-to-bl (len-of TYPE('b :: len0)) (numeral w)))
by (simp add: slice-def word-size)
```

lemma *slice1-0* [simp] : *slice1 n 0 = 0*

unfolding *slice1-def* **by** *simp*

lemma *slice-0* [simp] : *slice n 0 = 0*

unfolding *slice-def* **by** *auto*

lemma *slice-take'*: *slice n w = of-bl (take (size w - n) (to-bl w))*

unfolding *slice-def' slice1-def*

by (*simp add : takefill-alt word-size*)

lemmas *slice-take = slice-take'* [unfolded word-size]

— shiftr to a word of the same size is just slice, slice is just shiftr then ucast

lemmas *shiftr-slice = trans* [*OF shiftr-bl* [THEN meta-eq-to-obj-eq] *slice-take* [symmetric]]

lemma *slice-shiftr*: $\text{slice } n \ w = \text{ucast } (w \gg n)$
apply (*unfold slice-take shiftr-bl*)
apply (*rule ucast-of-bl-up [symmetric]*)
apply (*simp add: word-size*)
done

lemma *nth-slice*:
 $(\text{slice } n \ w :: 'a :: \text{len0 word}) !! m =$
 $(w !! (m + n) \ \& \ m < \text{len-of TYPE } ('a))$
unfolding *slice-shiftr*
by (*simp add : nth-ucast nth-shiftr*)

lemma *slice1-down-alt'*:
 $sl = \text{slice1 } n \ w \implies fs = \text{size } sl \implies fs + k = n \implies$
 $\text{to-bl } sl = \text{takefill False } fs \ (\text{drop } k \ (\text{to-bl } w))$
unfolding *slice1-def word-size of-bl-def uint-bl*
by (*clarsimp simp: word-ubin.eq-norm bl-bin-bl-rep-drop drop-takefill*)

lemma *slice1-up-alt'*:
 $sl = \text{slice1 } n \ w \implies fs = \text{size } sl \implies fs = n + k \implies$
 $\text{to-bl } sl = \text{takefill False } fs \ (\text{replicate } k \ \text{False } @ \ (\text{to-bl } w))$
apply (*unfold slice1-def word-size of-bl-def uint-bl*)
apply (*clarsimp simp: word-ubin.eq-norm bl-bin-bl-rep-drop*
takefill-append [symmetric])
apply (*rule-tac f = %k. takefill False (len-of TYPE('a))*
(replicate k False @ bin-to-bl (len-of TYPE('b)) (uint w)) in arg-cong)
apply *arith*
done

lemmas *sd1* = *slice1-down-alt'* [*OF refl refl, unfolded word-size*]
lemmas *su1* = *slice1-up-alt'* [*OF refl refl, unfolded word-size*]
lemmas *slice1-down-alt* = *le-add-diff-inverse* [*THEN sd1*]
lemmas *slice1-up-alt* =
le-add-diff-inverse [*symmetric, THEN su1*]
le-add-diff-inverse2 [*symmetric, THEN su1*]

lemma *ucast-slice1*: $\text{ucast } w = \text{slice1 } (\text{size } w) \ w$
unfolding *slice1-def ucast-bl*
by (*simp add : takefill-same' word-size*)

lemma *ucast-slice*: $\text{ucast } w = \text{slice } 0 \ w$
unfolding *slice-def* **by** (*simp add : ucast-slice1*)

lemma *slice-id*: $\text{slice } 0 \ t = t$
by (*simp only: ucast-slice [symmetric] ucast-id*)

lemma *revcast-slice1* [*OF refl*]:
 $rc = \text{revcast } w \implies \text{slice1 } (\text{size } rc) \ w = rc$

unfolding *slice1-def revcast-def'* **by** (*simp add : word-size*)

lemma *slice1-tf-tf'*:

to-bl (slice1 n w :: 'a :: len0 word) =
rev (takefill False (len-of TYPE('a)) (rev (takefill False n (to-bl w))))
unfolding *slice1-def* **by** (*rule word-rev-tf*)

lemmas *slice1-tf-tf = slice1-tf-tf'* [*THEN word-bl.Rep-inverse', symmetric*]

lemma *rev-slice1*:

n + k = len-of TYPE('a) + len-of TYPE('b) \implies
slice1 n (word-reverse w :: 'b :: len0 word) =
word-reverse (slice1 k w :: 'a :: len0 word)
apply (*unfold word-reverse-def slice1-tf-tf*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule rev-swap [THEN iffD1]*)
apply (*rule trans [symmetric]*)
apply (*rule tf-rev*)
apply (*simp add: word-bl.Abs-inverse*)
apply (*simp add: word-bl.Abs-inverse*)
done

lemma *rev-slice*:

n + k + len-of TYPE('a::len0) = len-of TYPE('b::len0) \implies
slice n (word-reverse (w::'b word)) = word-reverse (slice k w::'a word)
apply (*unfold slice-def word-size*)
apply (*rule rev-slice1*)
apply *arith*
done

lemmas *sym-notr =*

not-iff [THEN iffD2, THEN not-sym, THEN not-iff [THEN iffD1]]

— problem posed by TPHOLs referee: criterion for overflow of addition of signed integers

lemma *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)
apply (*unfold word-size*)
apply (*cases len-of TYPE('a), simp*)
apply (*subst msb-shift [THEN sym-notr]*)
apply (*simp add: word-ops-msb*)
apply (*simp add: word-msb-sint*)
apply *safe*
apply *simp-all*
apply (*unfold sint-word-ariths*)
apply (*unfold word-sbin.set-iff-norm [symmetric] sints-num*)
apply *safe*

```

apply (insert sint-range' [where  $x=x$ ])
apply (insert sint-range' [where  $x=y$ ])
defer
apply (simp (no-asm), arith)
apply (simp (no-asm), arith)
defer
defer
apply (simp (no-asm), arith)
apply (simp (no-asm), arith)
apply (rule notI [THEN notnotD],
        drule leI not-leE,
        drule sbintrunc-inc sbintrunc-dec,
        simp)+
done

```

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)))
unfolding word-rsplit-def by (simp add: word-ubin.eq-norm)

```

```

lemmas word-rsplit-no-cl [simp] = word-rsplit-no
[unfolded bin-rsplitl-def bin-rsplit-l [symmetric]]

```

lemma *test-bit-cat*:

```

 $wc = \text{word-cat } a \ b \implies wc \ !! \ n = (n < \text{size } wc \ \& \\
\text{if } n < \text{size } b \ \text{then } b \ !! \ n \ \text{else } a \ !! \ (n - \text{size } b))$ 
apply (unfold word-cat-bin' test-bit-bin)
apply (auto simp add : word-ubin.eq-norm nth-bintr bin-nth-cat word-size)
apply (erule bin-nth-uint-imp)
done

```

lemma *word-cat-bl*: $\text{word-cat } a \ b = \text{of-bl } (\text{to-bl } a \ @ \ \text{to-bl } b)$

```

apply (unfold of-bl-def to-bl-def word-cat-bin')
apply (simp add: bl-to-bin-app-cat)
done

```

lemma *of-bl-append*:

```

(of-bl (xs @ ys) :: 'a :: len word) = of-bl xs *  $2^{(\text{length } ys)}$  + of-bl ys
apply (unfold of-bl-def)
apply (simp add: bl-to-bin-app-cat bin-cat-num)
apply (simp add: word-of-int-power-hom [symmetric] word-of-int-hom-syms)
done

```

```

lemma of-bl-False [simp]:
  of-bl (False#xs) = of-bl xs
by (rule word-eqI)
  (auto simp add: test-bit-of-bl nth-append)

lemma of-bl-True [simp]:
  (of-bl (True#xs)::'a::len word) = 2length xs + of-bl xs
by (subst of-bl-append [where xs=[True], simplified])
  (simp add: word-1-bl)

lemma of-bl-Cons:
  of-bl (x#xs) = of-bool x * 2length xs + of-bl xs
by (cases x) simp-all

lemma split-wint-lem: bin-split n (uint (w :: 'a :: len0 word)) = (a, b)  $\implies$ 
  a = bintrunc (len-of TYPE('a) - n) a & b = bintrunc (len-of TYPE('a)) b
apply (frule word-ubin.norm-Rep [THEN ssubst])
apply (drule bin-split-trunc1)
apply (drule sym [THEN trans])
apply assumption
apply safe
done

lemma word-split-bl':
  std = size c - size b  $\implies$  (word-split c = (a, b))  $\implies$ 
  (a = of-bl (take std (to-bl c)) & b = of-bl (drop std (to-bl c)))
apply (unfold word-split-bin')
apply safe
defer
apply (clarsimp split: prod.splits)
apply (drule word-ubin.norm-Rep [THEN ssubst])
apply (drule split-bintrunc)
apply (simp add : of-bl-def bl2bin-drop word-size
  word-ubin.norm-eq-iff [symmetric] min-def del : word-ubin.norm-Rep)
apply (clarsimp split: prod.splits)
apply (frule split-wint-lem [THEN conjunct1])
apply (unfold word-size)
apply (cases len-of TYPE('a) >= len-of TYPE('b))
defer
apply simp
apply (simp add : of-bl-def to-bl-def)
apply (subst bin-split-take1 [symmetric])
prefer 2
apply assumption
apply simp
apply (erule thin-rl)
apply (erule arg-cong [THEN trans])
apply (simp add : word-ubin.norm-eq-iff [symmetric])
done

```

lemma *word-split-bl*: $std = size\ c - size\ b \implies$
 $(a = of-bl\ (take\ std\ (to-bl\ c)) \ \&\ b = of-bl\ (drop\ std\ (to-bl\ c))) \iff$
 $word-split\ c = (a, b)$
apply (*rule iffI*)
defer
apply (*erule (1) word-split-bl'*)
apply (*case-tac word-split c*)
apply (*auto simp add : word-size*)
apply (*frule word-split-bl' [rotated]*)
apply (*auto simp add : word-size*)
done

lemma *word-split-bl-eq*:
 $(word-split\ (c::'a::len\ word) :: ('c :: len0\ word * 'd :: len0\ word)) =$
 $(of-bl\ (take\ (len-of\ TYPE('a::len) - len-of\ TYPE('d::len0))\ (to-bl\ c)),$
 $of-bl\ (drop\ (len-of\ TYPE('a) - len-of\ TYPE('d))\ (to-bl\ c)))$
apply (*rule word-split-bl [THEN iffD1]*)
apply (*unfold word-size*)
apply (*rule refl conjI*)
done

— keep quantifiers for use in simplification

lemma *test-bit-split'*:
 $word-split\ c = (a, b) \iff (ALL\ n\ m.\ b\ !!\ n = (n < size\ b \ \&\ c\ !!\ n) \ \&$
 $a\ !!\ m = (m < size\ a \ \&\ c\ !!\ (m + size\ b)))$
apply (*unfold word-split-bin' test-bit-bin*)
apply (*clarify*)
apply (*clarsimp simp: word-ubin.eq-norm nth-bintr word-size split: prod.splits*)
apply (*drule bin-nth-split*)
apply *safe*
apply (*simp-all add: add-commute*)
apply (*erule bin-nth-uint-imp*)
done

lemma *test-bit-split*:
 $word-split\ c = (a, b) \implies$
 $(\forall n::nat.\ b\ !!\ n \iff n < size\ b \ \wedge\ c\ !!\ n) \ \wedge\ (\forall m::nat.\ a\ !!\ m \iff m < size\ a$
 $\wedge\ c\ !!\ (m + size\ b))$
by (*simp add: test-bit-split'*)

lemma *test-bit-split-eq*: $word-split\ c = (a, b) \iff$
 $((ALL\ n::nat.\ b\ !!\ n = (n < size\ b \ \&\ c\ !!\ n)) \ \&$
 $(ALL\ m::nat.\ a\ !!\ m = (m < size\ a \ \&\ c\ !!\ (m + size\ b))))$
apply (*rule-tac iffI*)
apply (*rule-tac conjI*)
apply (*erule test-bit-split [THEN conjunct1]*)
apply (*erule test-bit-split [THEN conjunct2]*)
apply (*case-tac word-split c*)

```

apply (frule test-bit-split)
apply (erule trans)
apply (fastforce intro ! : word-eqI simp add : word-size)
done

```

— this odd result is analogous to *ucast-id*, result to the length given by the result type

```

lemma word-cat-id: word-cat a b = b
  unfolding word-cat-bin' by (simp add: word-ubin.inverse-norm)

```

— limited hom result

```

lemma word-cat-hom:
  len-of TYPE('a::len0) <= len-of TYPE('b::len0) + len-of TYPE('c::len0)
   $\implies$ 
  (word-cat (word-of-int w :: 'b word) (b :: 'c word) :: 'a word) =
  word-of-int (bin-cat w (size b) (uint b))
apply (unfold word-cat-def word-size)
apply (clarsimp simp add: word-ubin.norm-eq-iff [symmetric]
  word-ubin.eq-norm bintr-cat min-max.inf-absorb1)
done

```

```

lemma word-cat-split-alt:
  size w <= size u + size v  $\implies$  word-split w = (u, v)  $\implies$  word-cat u v = w
apply (rule word-eqI)
apply (drule test-bit-split)
apply (clarsimp simp add : test-bit-cat word-size)
apply safe
apply arith
done

```

```

lemmas word-cat-split-size = sym [THEN [2] word-cat-split-alt [symmetric]]

```

13.26.1 Split and slice

```

lemma split-slices:
  word-split w = (u, v)  $\implies$  u = slice (size v) w & v = slice 0 w
apply (drule test-bit-split)
apply (rule conjI)
  apply (rule word-eqI,clarsimp simp: nth-slice word-size)+
done

```

```

lemma slice-cat1 [OF refl]:
  wc = word-cat a b  $\implies$  size wc >= size a + size b  $\implies$  slice (size b) wc = a
apply safe
apply (rule word-eqI)
apply (simp add: nth-slice test-bit-cat word-size)
done

```

lemmas *slice-cat2* = *trans* [*OF slice-id word-cat-id*]

lemma *cat-slices*:

$a = \text{slice } n \ c \implies b = \text{slice } 0 \ c \implies n = \text{size } b \implies$
 $\text{size } a + \text{size } b \geq \text{size } c \implies \text{word-cat } a \ b = c$

apply *safe*

apply (*rule word-eqI*)

apply (*simp add: nth-slice test-bit-cat word-size*)

apply *safe*

apply *arith*

done

lemma *word-split-cat-alt*:

$w = \text{word-cat } u \ v \implies \text{size } u + \text{size } v \leq \text{size } w \implies \text{word-split } w = (u, v)$

apply (*case-tac word-split ?w*)

apply (*rule trans, assumption*)

apply (*drule test-bit-split*)

apply *safe*

apply (*rule word-eqI, clarsimp simp: test-bit-cat word-size*)+

done

lemmas *word-cat-bl-no-bin* [*simp*] =

word-cat-bl [**where** *a=numeral a and b=numeral b,*
unfolded to-bl-numeral]

for *a b*

lemmas *word-split-bl-no-bin* [*simp*] =

word-split-bl-eq [**where** *c=numeral c, unfolded to-bl-numeral*] **for** *c*

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

lemma *word-rsplit-same*: $\text{word-rsplit } w = [w]$

unfolding *word-rsplit-def* **by** (*simp add : bin-rsplit-all*)

lemma *word-rsplit-empty-iff-size*:

$(\text{word-rsplit } w = []) = (\text{size } w = 0)$

unfolding *word-rsplit-def bin-rsplit-def word-size*

by (*simp add: bin-rsplit-aux-simp-alt Let-def split: Product-Type.split-split*)

lemma *test-bit-rsplit*:

$sw = \text{word-rsplit } w \implies m < \text{size } (\text{hd } sw :: 'a :: \text{len } \text{word}) \implies$

$k < \text{length } sw \implies (\text{rev } sw ! k) !! m = (w !! (k * \text{size } (\text{hd } sw) + m))$

apply (*unfold word-rsplit-def word-test-bit-def*)

apply (*rule trans*)

apply (*rule-tac f = %x. bin-nth x m in arg-cong*)

apply (*rule nth-map [symmetric]*)

apply *simp*

apply (*rule bin-nth-rsplit*)

```

    apply simp-all
  apply (simp add : word-size rev-map)
  apply (rule trans)
  defer
  apply (rule map-ident [THEN fun-cong])
  apply (rule refl [THEN map-cong])
  apply (simp add : word-ubin.eq-norm)
  apply (erule bin-rsplit-size-sign [OF len-gt-0 refl])
done

lemma word-rcat-bl: word-rcat wl = of-bl (concat (map to-bl wl))
  unfolding word-rcat-def to-bl-def' of-bl-def
  by (clarsimp simp add : bin-rcat-bl)

lemma size-rcat-lem':
  size (concat (map to-bl wl)) = length wl * size (hd wl)
  unfolding word-size by (induct wl) auto

lemmas size-rcat-lem = size-rcat-lem' [unfolded word-size]

lemmas td-gal-lt-len = len-gt-0 [THEN td-gal-lt]

lemma nth-rcat-lem:
  n < length (wl :: 'a word list) * len-of TYPE('a::len)  $\implies$ 
  rev (concat (map to-bl wl)) ! n =
  rev (to-bl (rev wl ! (n div len-of TYPE('a)))) ! (n mod len-of TYPE('a))
  apply (induct wl)
  apply clarsimp
  apply (clarsimp simp add : nth-append size-rcat-lem)
  apply (simp (no-asm-use) only: mult-Suc [symmetric]
    td-gal-lt-len less-Suc-eq-le mod-div-equality')
  apply clarsimp
done

lemma test-bit-rcat:
  sw = size (hd wl :: 'a :: len word)  $\implies$  rc = word-rcat wl  $\implies$  rc !! n =
  (n < size rc & n div sw < size wl & (rev wl) ! (n div sw) !! (n mod sw))
  apply (unfold word-rcat-bl word-size)
  apply (clarsimp simp add :
    test-bit-of-bl size-rcat-lem word-size td-gal-lt-len)
  apply safe
  apply (auto simp add :
    test-bit-bl word-size td-gal-lt-len [THEN iffD2, THEN nth-rcat-lem])
done

lemma foldl-eq-foldr:
  foldl op + x xs = foldr op + (x # xs) (0 :: 'a :: comm-monoid-add)
  by (induct xs arbitrary: x) (auto simp add : add-assoc)

```

lemmas *test-bit-cong* = *arg-cong* [**where** *f* = *test-bit*, *THEN fun-cong*]

lemmas *test-bit-rsplit-alt* =
trans [*OF nth-rev-alt* [*THEN test-bit-cong*]
test-bit-rsplit [*OF refl asm-rl diff-Suc-less*]]

— lazy way of expressing that *u* and *v*, and *su* and *sv*, have same types

lemma *word-rsplit-len-indep* [*OF refl refl refl refl*]:
 $[u,v] = p \implies [su,sv] = q \implies \text{word-rsplit } u = su \implies$
 $\text{word-rsplit } v = sv \implies \text{length } su = \text{length } sv$
apply (*unfold word-rsplit-def*)
apply (*auto simp add : bin-rsplit-len-indep*)
done

lemma *length-word-rsplit-size*:
 $n = \text{len-of } TYPE ('a :: \text{len}) \implies$
 $(\text{length } (\text{word-rsplit } w :: 'a \text{ word list}) \leq m) = (\text{size } w \leq m * n)$
apply (*unfold word-rsplit-def word-size*)
apply (*clarsimp simp add : bin-rsplit-len-le*)
done

lemmas *length-word-rsplit-lt-size* =
length-word-rsplit-size [*unfolded Not-eq-iff linorder-not-less* [*symmetric*]]

lemma *length-word-rsplit-exp-size*:
 $n = \text{len-of } TYPE ('a :: \text{len}) \implies$
 $\text{length } (\text{word-rsplit } w :: 'a \text{ word list}) = (\text{size } w + n - 1) \text{ div } n$
unfolding *word-rsplit-def* **by** (*clarsimp simp add : word-size bin-rsplit-len*)

lemma *length-word-rsplit-even-size*:
 $n = \text{len-of } TYPE ('a :: \text{len}) \implies \text{size } w = m * n \implies$
 $\text{length } (\text{word-rsplit } w :: 'a \text{ word list}) = m$
by (*clarsimp simp add : length-word-rsplit-exp-size given-quot-alt*)

lemmas *length-word-rsplit-exp-size'* = *refl* [*THEN length-word-rsplit-exp-size*]

lemmas *tdle* = *iffD2* [*OF split-div-lemma refl*, *THEN conjunct1*]

lemmas *dtle* = *xtr4* [*OF tdle mult-commute*]

lemma *word-rcat-rsplit*: *word-rcat* (*word-rsplit w*) = *w*
apply (*rule word-eqI*)
apply (*clarsimp simp add : test-bit-rcat word-size*)
apply (*subst refl* [*THEN test-bit-rsplit*])
apply (*simp-all add: word-size*)
refl [*THEN length-word-rsplit-size* [*simplified not-less* [*symmetric*], *simplified*]]
apply *safe*
apply (*erule xtr7*, *rule len-gt-0* [*THEN dtle*])
done

lemma *size-word-rsplit-rcat-size*:
 $\llbracket \text{word-rcat } (ws::'a::\text{len } \text{word list}) = (\text{frcw}::'b::\text{len0 } \text{word});$
 $\text{size frcw} = \text{length } ws * \text{len-of TYPE}('a)\rrbracket$
 $\implies \text{length } (\text{word-rsplit } \text{frcw}::'a \text{ word list}) = \text{length } ws$
apply (*clarsimp simp add : word-size length-word-rsplit-exp-size'*)
apply (*fast intro: given-quot-alt*)
done

lemma *msrevs*:
fixes $n::\text{nat}$
shows $0 < n \implies (k * n + m) \text{ div } n = m \text{ div } n + k$
and $(k * n + m) \text{ mod } n = m \text{ mod } n$
by (*auto simp: add-commute*)

lemma *word-rsplit-rcat-size [OF refl]*:
 $\text{word-rcat } (ws :: 'a :: \text{len } \text{word list}) = \text{frcw} \implies$
 $\text{size frcw} = \text{length } ws * \text{len-of TYPE } ('a) \implies \text{word-rsplit frcw} = ws$
apply (*frule size-word-rsplit-rcat-size, assumption*)
apply (*clarsimp simp add : word-size*)
apply (*rule nth-equalityI, assumption*)
apply *clarsimp*
apply (*rule word-eqI [rule-format]*)
apply (*rule trans*)
apply (*rule test-bit-rsplit-alt*)
apply (*clarsimp simp: word-size*)+
apply (*rule trans*)
apply (*rule test-bit-rcat [OF refl refl]*)
apply (*simp add: word-size msrevs*)
apply (*subst nth-rev*)
apply *arith*
apply (*simp add: le0 [THEN [2] xtr7, THEN diff-Suc-less]*)
apply *safe*
apply (*simp add: diff-mult-distrib*)
apply (*rule mpl-lem*)
apply (*cases size ws*)
apply *simp-all*
done

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*:
 $m \text{ mod } \text{length } xs = n \text{ mod } \text{length } xs \implies \text{rotate } m \text{ } xs = \text{rotate } n \text{ } xs$
apply (*rule box-equals*)
defer

```

  apply (rule rotate-conv-mod [symmetric])+
  apply simp
done

```

```

lemmas rotate-eqs =
  trans [OF rotate0 [THEN fun-cong] id-apply]
  rotate-rotate [symmetric]
  rotate-id
  rotate-conv-mod
  rotate-eq-mod

```

13.27.1 Rotation of list to right

```

lemma rotate1-rl': rotater1 (l @ [a]) = a # l
  unfolding rotater1-def by (cases l) auto

```

```

lemma rotate1-rl [simp] : rotater1 (rotate1 l) = l
  apply (unfold rotater1-def)
  apply (cases l)
  apply (case-tac [2] list)
  apply auto
done

```

```

lemma rotate1-lr [simp] : rotate1 (rotater1 l) = l
  unfolding rotater1-def by (cases l) auto

```

```

lemma rotater1-rev': rotater1 (rev xs) = rev (rotate1 xs)
  apply (cases xs)
  apply (simp add : rotater1-def)
  apply (simp add : rotate1-rl')
done

```

```

lemma rotater-rev': rotater n (rev xs) = rev (rotate n xs)
  unfolding rotater-def by (induct n) (auto intro: rotater1-rev')

```

```

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

```

```

lemma rotater-drop-take:
  rotater n xs =
    drop (length xs - n mod length xs) xs @
    take (length xs - n mod length xs) xs
  by (clarsimp simp add : rotater-rev rotate-drop-take rev-take rev-drop)

```

```

lemma rotater-Suc [simp] :
  rotater (Suc n) xs = rotater1 (rotater n xs)
  unfolding rotater-def by auto

```

```

lemma rotate-inv-plus [rule-format] :

```

ALL k . $k = m + n \rightarrow \text{rotater } k (\text{rotate } n \text{ } xs) = \text{rotater } m \text{ } xs \ \&$
 $\text{rotate } k (\text{rotater } n \text{ } xs) = \text{rotate } m \text{ } xs \ \&$
 $\text{rotater } n (\text{rotate } k \text{ } xs) = \text{rotate } m \text{ } xs \ \&$
 $\text{rotate } n (\text{rotater } k \text{ } xs) = \text{rotater } m \text{ } xs$
unfolding *rotater-def rotate-def*
by (*induct n*) (*auto intro: funpow-swap1 [THEN trans]*)

lemmas *rotate-inv-rel = le-add-diff-inverse2 [symmetric, THEN rotate-inv-plus]*

lemmas *rotate-inv-eq = order-refl [THEN rotate-inv-rel, simplified]*

lemmas *rotate-lr [simp] = rotate-inv-eq [THEN conjunct1]*

lemmas *rotate-rl [simp] = rotate-inv-eq [THEN conjunct2, THEN conjunct1]*

lemma *rotate-gal: (rotater n xs = ys) = (rotate n ys = xs)*
by *auto*

lemma *rotate-gal': (ys = rotater n xs) = (xs = rotate n ys)*
by *auto*

lemma *length-rotater [simp]:*
 $\text{length } (\text{rotater } n \text{ } xs) = \text{length } xs$
by (*simp add : rotater-rev*)

lemma *restrict-to-left:*
assumes $x = y$
shows $(x = z) = (y = z)$
using *assms by simp*

lemmas *rrs0 = rotate-eqs [THEN restrict-to-left,*
simplified rotate-gal [symmetric] rotate-gal' [symmetric]]
lemmas *rrs1 = rrs0 [THEN refl [THEN rev-iffD1]]*
lemmas *rotater-eqs = rrs1 [simplified length-rotater]*
lemmas *rotater-0 = rotater-eqs (1)*
lemmas *rotater-add = rotater-eqs (2)*

13.27.2 map, map2, commuting with rotate(r)

lemma *last-map: xs $\sim = [] \implies \text{last } (\text{map } f \text{ } xs) = f (\text{last } xs)$*
by (*induct xs*) *auto*

lemma *butlast-map:*
 $xs \sim = [] \implies \text{butlast } (\text{map } f \text{ } xs) = \text{map } f (\text{butlast } xs)$
by (*induct xs*) *auto*

lemma *rotater1-map: rotater1 (map f xs) = map f (rotater1 xs)*
unfolding *rotater1-def*
by (*cases xs*) (*auto simp add: last-map butlast-map*)

lemma *rotater-map*:

$rotater\ n\ (map\ f\ xs) = map\ f\ (rotater\ n\ xs)$
unfolding *rotater-def*
by (*induct n*) (*auto simp add : rotater1-map*)

lemma *but-last-zip* [*rule-format*] :

ALL ys. length xs = length ys --> xs ~ = [] -->
 $last\ (zip\ xs\ ys) = (last\ xs,\ last\ ys)\ \&$
 $butlast\ (zip\ xs\ ys) = zip\ (butlast\ xs)\ (butlast\ ys)$
apply (*induct xs*)
apply *auto*
apply ((*case-tac ys, auto simp: neq-Nil-conv*)[1])+
done

lemma *but-last-map2* [*rule-format*] :

ALL ys. length xs = length ys --> xs ~ = [] -->
 $last\ (map2\ f\ xs\ ys) = f\ (last\ xs)\ (last\ ys)\ \&$
 $butlast\ (map2\ f\ xs\ ys) = map2\ f\ (butlast\ xs)\ (butlast\ ys)$
apply (*induct xs*)
apply *auto*
apply (*unfold map2-def*)
apply ((*case-tac ys, auto simp: neq-Nil-conv*)[1])+
done

lemma *rotater1-zip*:

$length\ xs = length\ ys \implies$
 $rotater1\ (zip\ xs\ ys) = zip\ (rotater1\ xs)\ (rotater1\ ys)$
apply (*unfold rotater1-def*)
apply (*cases xs*)
apply *auto*
apply ((*case-tac ys, auto simp: neq-Nil-conv but-last-zip*)[1])+
done

lemma *rotater1-map2*:

$length\ xs = length\ ys \implies$
 $rotater1\ (map2\ f\ xs\ ys) = map2\ f\ (rotater1\ xs)\ (rotater1\ ys)$
unfolding *map2-def* **by** (*simp add: rotater1-map rotater1-zip*)

lemmas *lrth* =

box-equals [*OF asm-rl length-rotater [symmetric]*
 $length-rotater\ [symmetric]$,
THEN rotater1-map2]

lemma *rotater-map2*:

$length\ xs = length\ ys \implies$
 $rotater\ n\ (map2\ f\ xs\ ys) = map2\ f\ (rotater\ n\ xs)\ (rotater\ n\ ys)$
by (*induct n*) (*auto intro!: lrth*)

lemma *rotate1-map2*:

```

length xs = length ys =>
  rotate1 (map2 f xs ys) = map2 f (rotate1 xs) (rotate1 ys)
apply (unfold map2-def)
apply (cases xs)
  apply (cases ys, auto)+
done

```

lemmas *lth = box-equals* [*OF asm-rl length-rotate* [*symmetric*]
length-rotate [*symmetric*], *THEN rotate1-map2*]

lemma *rotate-map2*:
length xs = length ys =>
rotate n (map2 f xs ys) = map2 f (rotate n xs) (rotate n ys)
by (*induct n*) (*auto intro! lth*)

— corresponding equalities for word rotation

lemma *to-bl-rotl*:
to-bl (word-rotl n w) = rotate n (to-bl w)
by (*simp add: word-bl.Abs-inverse' word-rotl-def*)

lemmas *blrs0 = rotate-egs* [*THEN to-bl-rotl* [*THEN trans*]]

lemmas *word-rotl-egs =*
blrs0 [*simplified word-bl-Rep' word-bl.Rep-inject to-bl-rotl* [*symmetric*]]

lemma *to-bl-rotr*:
to-bl (word-rotr n w) = rotater n (to-bl w)
by (*simp add: word-bl.Abs-inverse' word-rotr-def*)

lemmas *brrs0 = rotater-egs* [*THEN to-bl-rotr* [*THEN trans*]]

lemmas *word-rotr-egs =*
brrs0 [*simplified word-bl-Rep' word-bl.Rep-inject to-bl-rotr* [*symmetric*]]

declare *word-rotr-egs* (1) [*simp*]
declare *word-rotl-egs* (1) [*simp*]

lemma
word-rot-rl [*simp*]:
word-rotl k (word-rotr k v) = v **and**
word-rot-lr [*simp*]:
word-rotr k (word-rotl k v) = v
by (*auto simp add: to-bl-rotr to-bl-rotl word-bl.Rep-inject* [*symmetric*])

lemma
word-rot-gal:
(word-rotr n v = w) = (word-rotl n w = v) **and**

```

word-rot-gal':
(w = word-rotr n v) = (v = word-rotl n w)
by (auto simp: to-bl-rotr to-bl-rotl word-bl.Rep-inject [symmetric]
    dest: sym)

lemma word-rotr-rev:
word-rotr n w = word-reverse (word-rotl n (word-reverse w))
by (simp only: word-bl.Rep-inject [symmetric] to-bl-word-rev
    to-bl-rotr to-bl-rotl rotater-rev)

lemma word-roti-0 [simp]: word-roti 0 w = w
by (unfold word-rot-defs) auto

lemmas abl-cong = arg-cong [where f = of-bl]

lemma word-roti-add:
word-roti (m + n) w = word-roti m (word-roti n w)
proof -
have rotater-eq-lem:
 $\bigwedge m n xs. m = n \implies \text{rotater } m \text{ } xs = \text{rotater } n \text{ } xs$ 
by auto

have rotate-eq-lem:
 $\bigwedge m n xs. m = n \implies \text{rotate } m \text{ } xs = \text{rotate } n \text{ } xs$ 
by auto

note rpts [symmetric] =
rotate-inv-plus [THEN conjunct1]
rotate-inv-plus [THEN conjunct2, THEN conjunct1]
rotate-inv-plus [THEN conjunct2, THEN conjunct2, THEN conjunct1]
rotate-inv-plus [THEN conjunct2, THEN conjunct2, THEN conjunct2]

note rrp = trans [symmetric, OF rotate-rotate rotate-eq-lem]
note rrrp = trans [symmetric, OF rotater-add [symmetric] rotater-eq-lem]

show ?thesis
apply (unfold word-rot-defs)
apply (simp only: split: split-if)
apply (safe intro!: abl-cong)
apply (simp-all only: to-bl-rotl [THEN word-bl.Rep-inverse']
    to-bl-rotl
    to-bl-rotr [THEN word-bl.Rep-inverse']
    to-bl-rotr)
apply (rule rrp rrrp rpts,
    simp add: nat-add-distrib [symmetric]
    nat-diff-distrib [symmetric])+

done
qed

```

```

lemma word-roti-conv-mod': word-roti n w = word-roti (n mod int (size w)) w
  apply (unfold word-rot-defs)
  apply (cut-tac y=size w in gt-or-eq-0)
  apply (erule disjE)
  apply simp-all
  apply (safe intro!: abl-cong)
  apply (rule rotater-eqs)
  apply (simp add: word-size nat-mod-distrib)
  apply (simp add: rotater-add [symmetric] rotate-gal [symmetric])
  apply (rule rotater-eqs)
  apply (simp add: word-size nat-mod-distrib)
  apply (rule int-eq-0-conv [THEN iffD1])
  apply (simp only: zmod-int of-nat-add)
  apply (simp add: rmods)
done

```

```

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

```

```

by (rule word-bl.Rep-eqD,

```

```

rule word-rot-defs' [THEN trans],

```

```

simp only: blwl-syms [symmetric],

```

```

rule th1s [THEN trans],

```

```

rule refl)+

```

end

lemmas *word-rot-logs* = *word-rotate.word-rot-logs*

lemmas *bl-word-rotl-dt* = *trans* [*OF to-bl-rotl rotate-drop-take*,
simplified word-bl-Rep']

lemmas *bl-word-rotr-dt* = *trans* [*OF to-bl-rotr rotater-drop-take*,
simplified word-bl-Rep']

lemma *bl-word-roti-dt'*:

$n = \text{nat } ((- i) \text{ mod int } (\text{size } (w :: 'a :: \text{len word}))) \implies$

$\text{to-bl } (\text{word-roti } i w) = \text{drop } n (\text{to-bl } w) @ \text{take } n (\text{to-bl } w)$

apply (*unfold word-roti-def*)

apply (*simp add: bl-word-rotl-dt bl-word-rotr-dt word-size*)

apply *safe*

apply (*simp add: zmod-zminus1-eq-if*)

apply *safe*

apply (*simp add: nat-mult-distrib*)

apply (*simp add: nat-diff-distrib* [*OF pos-mod-sign pos-mod-conj*

[*THEN conjunct2, THEN order-less-imp-le*]

nat-mod-distrib)

apply (*simp add: nat-mod-distrib*)

done

lemmas *bl-word-roti-dt* = *bl-word-roti-dt'* [*unfolded word-size*]

lemmas *word-rotl-dt* = *bl-word-rotl-dt* [*THEN word-bl.Rep-inverse'* [*symmetric*]]

lemmas *word-rotr-dt* = *bl-word-rotr-dt* [*THEN word-bl.Rep-inverse'* [*symmetric*]]

lemmas *word-roti-dt* = *bl-word-roti-dt* [*THEN word-bl.Rep-inverse'* [*symmetric*]]

lemma *word-rotx-0* [*simp*] : *word-rotr i 0 = 0 & word-rotl i 0 = 0*

by (*simp add : word-rotr-dt word-rotl-dt replicate-add* [*symmetric*])

lemma *word-roti-0'* [*simp*] : *word-roti n 0 = 0*

unfolding *word-roti-def* **by** *auto*

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** $(x :: 'a::len0 \text{ word}) = \text{word-of-int } n$ **and** $0 \leq n$ **and** $n < 2^{\text{len-of TYPE('a)}}$

by (*cases x rule: word-uint.Abs-cases*) (*simp add: uints-num*)

lemma *word-nat-cases* [*cases type: word*]:

obtains n **where** $(x :: 'a::len \text{ word}) = \text{of-nat } n$ **and** $n < 2^{\text{len-of TYPE('a)}}$

by (*cases x rule: word-unat.Abs-cases*) (*simp add: unats-def*)

lemma *max-word-eq*: $(\text{max-word} :: 'a::len \text{ word}) = 2^{\text{len-of TYPE('a)}} - 1$

by (*simp add: max-word-def word-of-int-hom-syms word-of-int-2p*)

lemma *max-word-max* [*simp,intro!*]: $n \leq \text{max-word}$

by (*cases n rule: word-int-cases*)

(*simp add: max-word-def word-le-def int-word-uint int-mod-eq'*)

lemma *word-of-int-2p-len*: $\text{word-of-int } (2^{\text{len-of TYPE('a)}}) = (0 :: 'a::len0 \text{ word})$

by (*subst word-uint.Abs-norm [symmetric]*) *simp*

lemma *word-pow-0*:

$(2 :: 'a::len \text{ word})^{\text{len-of TYPE('a)}} = 0$

proof –

have $\text{word-of-int } (2^{\text{len-of TYPE('a)}}) = (0 :: 'a \text{ word})$

by (*rule word-of-int-2p-len*)

thus *?thesis* **by** (*simp add: word-of-int-2p*)

qed

lemma *max-word-wrap*: $x + 1 = 0 \implies x = \text{max-word}$

apply (*simp add: max-word-eq*)

apply *uint-arith*

apply *auto*

apply (*simp add: word-pow-0*)

done

lemma *max-word-minus*:

$\text{max-word} = (-1 :: 'a::len \text{ word})$

proof –

have $-1 + 1 = (0 :: 'a \text{ word})$ **by** *simp*

thus *?thesis* **by** (*rule max-word-wrap [symmetric]*)

qed

lemma *max-word-bl* [*simp*]:

to-bl $(\text{max-word} :: 'a::len \text{ word}) = \text{replicate } (\text{len-of TYPE('a)}) \text{ True}$

by (*subst max-word-minus to-bl-n1*) *simp*

lemma *max-test-bit* [*simp*]:

$(\text{max-word} :: 'a::len \text{ word}) !! n = (n < \text{len-of TYPE('a)})$

by (*auto simp add: test-bit-bl word-size*)

lemma *word-and-max* [*simp*]:
 $x \text{ AND } \text{max-word} = x$
by (*rule word-eqI*) (*simp add: word-ops-nth-size word-size*)

lemma *word-or-max* [*simp*]:
 $x \text{ OR } \text{max-word} = \text{max-word}$
by (*rule word-eqI*) (*simp add: word-ops-nth-size word-size*)

lemma *word-ao-dist2*:
 $x \text{ AND } (y \text{ OR } z) = x \text{ AND } y \text{ OR } x \text{ AND } (z::'a::\text{len0 word})$
by (*rule word-eqI*) (*auto simp add: word-ops-nth-size word-size*)

lemma *word-oa-dist2*:
 $x \text{ OR } y \text{ AND } z = (x \text{ OR } y) \text{ AND } (x \text{ OR } (z::'a::\text{len0 word}))$
by (*rule word-eqI*) (*auto simp add: word-ops-nth-size word-size*)

lemma *word-and-not* [*simp*]:
 $x \text{ AND } \text{NOT } x = (0::'a::\text{len0 word})$
by (*rule word-eqI*) (*auto simp add: word-ops-nth-size word-size*)

lemma *word-or-not* [*simp*]:
 $x \text{ OR } \text{NOT } x = \text{max-word}$
by (*rule word-eqI*) (*auto simp add: word-ops-nth-size word-size*)

lemma *word-boolean*:
 $\text{boolean } (op \text{ AND}) (op \text{ OR}) \text{ bitNOT } 0 \text{ max-word}$
apply (*rule boolean.intro*)
apply (*rule word-bw-assocs*)
apply (*rule word-bw-assocs*)
apply (*rule word-bw-comms*)
apply (*rule word-bw-comms*)
apply (*rule word-ao-dist2*)
apply (*rule word-oa-dist2*)
apply (*rule word-and-max*)
apply (*rule word-log-esimps*)
apply (*rule word-and-not*)
apply (*rule word-or-not*)
done

interpretation *word-bool-alg*:
 $\text{boolean } op \text{ AND } op \text{ OR } \text{bitNOT } 0 \text{ max-word}$
by (*rule word-boolean*)

lemma *word-xor-and-or*:
 $x \text{ XOR } y = x \text{ AND } \text{NOT } y \text{ OR } \text{NOT } x \text{ AND } (y::'a::\text{len0 word})$
by (*rule word-eqI*) (*auto simp add: word-ops-nth-size word-size*)

interpretation *word-bool-alg*:

boolean-xor op AND op OR bitNOT 0 max-word op XOR
apply (rule *boolean-xor.intro*)
apply (rule *word-boolean*)
apply (rule *boolean-xor-axioms.intro*)
apply (rule *word-xor-and-or*)
done

lemma *shiftr-x-0* [*iff*]:
 $(x :: 'a :: len\ 0\ word) \gg\ 0 = x$
by (*simp add: shiftr-bl*)

lemma *shiftr-x-0* [*simp*]:
 $(x :: 'a :: len\ word) \ll\ 0 = x$
by (*simp add: shiftr-t2n*)

lemma *shiftr-1* [*simp*]:
 $(1 :: 'a :: len\ word) \ll\ n = 2^n$
by (*simp add: shiftr-t2n*)

lemma *uint-lt-0* [*simp*]:
 $uint\ x < 0 = False$
by (*simp add: linorder-not-less*)

lemma *shiftr1-1* [*simp*]:
 $shiftr1\ (1 :: 'a :: len\ word) = 0$
unfolding *shiftr1-def* **by** *simp*

lemma *shiftr-1* [*simp*]:
 $(1 :: 'a :: len\ word) \gg\ n = (if\ n = 0\ then\ 1\ else\ 0)$
by (*induct n*) (*auto simp: shiftr-def*)

lemma *word-less-1* [*simp*]:
 $((x :: 'a :: len\ word) < 1) = (x = 0)$
by (*simp add: word-less-nat-alt unat-0-iff*)

lemma *to-bl-mask*:
 $to-bl\ (mask\ n :: 'a :: len\ word) =$
 $replicate\ (len-of\ TYPE('a) - n)\ False\ @$
 $replicate\ (min\ (len-of\ TYPE('a))\ n)\ True$
by (*simp add: mask-bl word-rep-drop min-def*)

lemma *map-replicate-True*:
 $n = length\ xs \implies$
 $map\ (\lambda(x,y). x \& y)\ (zip\ xs\ (replicate\ n\ True)) = xs$
by (*induct xs arbitrary: n*) *auto*

lemma *map-replicate-False*:
 $n = length\ xs \implies map\ (\lambda(x,y). x \& y)$
 $(zip\ xs\ (replicate\ n\ False)) = replicate\ n\ False$

by (induct xs arbitrary: n) auto

lemma *bl-and-mask*:

fixes $w :: 'a::len\ word$

fixes n

defines $n' \equiv len\ of\ TYPE('a) - n$

shows $to\text{-}bl\ (w\ AND\ mask\ n) = replicate\ n'\ False\ @\ drop\ n'\ (to\text{-}bl\ w)$

proof –

note [*simp*] = *map-replicate-True map-replicate-False*

have $to\text{-}bl\ (w\ AND\ mask\ n) =$

$map2\ op\ \&\ (to\text{-}bl\ w)\ (to\text{-}bl\ (mask\ n::'a::len\ word))$

by (*simp add: bl-word-and*)

also

have $to\text{-}bl\ w = take\ n'\ (to\text{-}bl\ w)\ @\ drop\ n'\ (to\text{-}bl\ w)$ **by** *simp*

also

have $map2\ op\ \&\ \dots\ (to\text{-}bl\ (mask\ n::'a::len\ word)) =$

$replicate\ n'\ False\ @\ drop\ n'\ (to\text{-}bl\ w)$

unfolding *to-bl-mask n'-def map2-def*

by (*subst zip-append*) auto

finally

show *?thesis* .

qed

lemma *drop-rev-takefill*:

$length\ xs \leq n \implies$

$drop\ (n - length\ xs)\ (rev\ (takefill\ False\ n\ (rev\ xs))) = xs$

by (*simp add: takefill-alt rev-take*)

lemma *map-nth-0* [*simp*]:

$map\ (op\ !!\ (0::'a::len0\ word))\ xs = replicate\ (length\ xs)\ False$

by (*induct xs*) auto

lemma *uint-plus-if-size*:

$uint\ (x + y) =$

(if $uint\ x + uint\ y < 2^{size\ x}$ then

$uint\ x + uint\ y$

else

$uint\ x + uint\ y - 2^{size\ x}$)

by (*simp add: word-arith-alts int-word-uint mod-add-if-z*

word-size)

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

$unat\ x + unat\ y - 2^{size\ x}$)

apply (*subst word-arith-nat-defs*)

apply (*subst unat-of-nat*)

apply (*simp add: mod-nat-add word-size*)
done

lemma *word-neq-0-conv*:
fixes $w :: 'a :: \text{len word}$
shows $(w \neq 0) = (0 < w)$
unfolding *word-gt-0* **by** *simp*

lemma *max-lt*:
 $\text{unat } (\max a b \text{ div } c) = \text{unat } (\max a b) \text{ div } \text{unat } (c :: 'a :: \text{len word})$
apply (*subst word-arith-nat-defs*)
apply (*subst word-unat.eq-norm*)
apply (*subst mod-if*)
apply *clarsimp*
apply (*erule notE*)
apply (*insert div-le-dividend [of unat (max a b) unat c]*)
apply (*erule order-le-less-trans*)
apply (*insert unat-lt2p [of max a b]*)
apply *simp*
done

lemma *uint-sub-if-size*:
 $\text{uint } (x - y) =$
(if uint y ≤ uint x then
 $\text{uint } x - \text{uint } y$
else
 $\text{uint } x - \text{uint } y + 2^{\text{size } x}$)
by (*simp add: word-arith-alts int-word-uint mod-sub-if-z*
word-size)

lemma *unat-sub*:
 $b \leq a \implies \text{unat } (a - b) = \text{unat } a - \text{unat } b$
by (*simp add: unat-def uint-sub-if-size word-le-def nat-diff-distrib*)

lemmas *word-less-sub1-numberof [simp] = word-less-sub1 [of numeral w] for w*
lemmas *word-le-sub1-numberof [simp] = word-le-sub1 [of numeral w] for w*

lemma *word-of-int-minus*:
 $\text{word-of-int } (2^{\text{len-of TYPE('a)}} - i) = (\text{word-of-int } (-i) :: 'a :: \text{len word})$
proof –
have $x: 2^{\text{len-of TYPE('a)}} - i = -i + 2^{\text{len-of TYPE('a)}}$ **by** *simp*
show *?thesis*
apply (*subst x*)
apply (*subst word-uint.Abs-norm [symmetric], subst mod-add-self2*)
apply *simp*
done

qed

lemmas *word-of-int-inj =*

word-uint.Abs-inject [unfolded uints-num, simplified]

lemma *word-le-less-eq*:

$(x :: 'z :: \text{len word}) \leq y = (x = y \vee x < y)$
by (*auto simp add: order-class.le-less*)

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 –

from 1 2[*symmetric*] 3[*symmetric*] **have** $(x + y) \bmod b = (x' \bmod b' + y' \bmod b') \bmod b'$
by (*simp add: mod-add-eq[symmetric]*)
also have $\dots = (x' + y') \bmod b'$
by (*simp add: mod-add-eq[symmetric]*)
finally show ?thesis **by** (*simp add: 4*)
qed

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 $(x - y) \bmod b = z' \bmod b'$
using *assms*
apply (*subst mod-diff-left-eq*)
apply (*subst mod-diff-right-eq*)
apply (*simp add: mod-diff-left-eq [symmetric] mod-diff-right-eq [symmetric]*)
done

lemma *word-induct-less*:

$\llbracket P (0 :: 'a :: \text{len word}); \bigwedge n. \llbracket n < m; P n \rrbracket \implies P (1 + n) \rrbracket \implies P m$
apply (*cases m*)
apply *atomize*
apply (*erule rev-mp*)
apply (*rule-tac x=m in spec*)
apply (*induct-tac n*)
apply *simp*
apply *clarsimp*
apply (*erule impE*)
apply *clarsimp*
apply (*erule-tac x=n in allE*)
apply (*erule impE*)
apply (*simp add: unat-arith-simps*)
apply (*clarsimp simp: unat-of-nat*)
apply *simp*

```

apply (erule-tac x=of-nat na in allE)
apply (erule impE)
apply (simp add: unat-arith-simps)
apply (clarsimp simp: unat-of-nat)
apply simp
done

```

lemma *word-induct*:
 $\llbracket P (0::'a::\text{len word}); \bigwedge n. P n \implies P (1 + n) \rrbracket \implies P m$
by (erule word-induct-less, simp)

lemma *word-induct2* [*induct type*]:
 $\llbracket P 0; \bigwedge n. \llbracket 1 + n \neq 0; P n \rrbracket \implies P (1 + n) \rrbracket \implies P (n::'b::\text{len word})$
apply (rule word-induct, simp)
apply (case-tac 1+n = 0, auto)
done

13.29 Recursion combinator for words

definition *word-rec* :: 'a \Rightarrow ('b::len word \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b word \Rightarrow 'a **where**
word-rec forZero forSuc n = nat-rec forZero (forSuc \circ of-nat) (unat n)

lemma *word-rec-0*: *word-rec* z s 0 = z
by (simp add: word-rec-def)

lemma *word-rec-Suc*:
 $1 + n \neq (0::'a::\text{len word}) \implies \text{word-rec } z \ s \ (1 + n) = s \ n \ (\text{word-rec } z \ s \ n)$
apply (simp add: word-rec-def unat-word-ariths)
apply (subst nat-mod-eq')
apply (cut-tac x=n **in** unat-lt2p)
apply (drule Suc-mono)
apply (simp add: less-Suc-eq-le)
apply (simp only: order-less-le, simp)
apply (erule contrapos-pn, simp)
apply (drule arg-cong[**where** f=of-nat])
apply simp
apply (subst (asm) word-unat.Rep-inverse[of n])
apply simp
apply simp
done

lemma *word-rec-Pred*:
 $n \neq 0 \implies \text{word-rec } z \ s \ n = s \ (n - 1) \ (\text{word-rec } z \ s \ (n - 1))$
apply (rule subst[**where** t=n **and** s=1 + (n - 1)])
apply simp
apply (subst word-rec-Suc)
apply simp
apply simp
done

lemma *word-rec-in*:

f (*word-rec* z ($\lambda\cdot$. f) n) = *word-rec* (f z) ($\lambda\cdot$. f) n
by (*induct* n) (*simp-all* *add*: *word-rec-0* *word-rec-Suc*)

lemma *word-rec-in2*:

f n (*word-rec* z f n) = *word-rec* (f 0 z) (f \circ *op* + 1) n
by (*induct* n) (*simp-all* *add*: *word-rec-0* *word-rec-Suc*)

lemma *word-rec-twice*:

$m \leq n \implies$ *word-rec* z f n = *word-rec* (*word-rec* z f ($n - m$)) (f \circ *op* + ($n - m$)) m
apply (*erule* *rev-mp*)
apply (*rule-tac* $x=z$ **in** *spec*)
apply (*rule-tac* $x=f$ **in** *spec*)
apply (*induct* n)
apply (*simp* *add*: *word-rec-0*)
apply *clarsimp*
apply (*rule-tac* $t=1 + n - m$ **and** $s=1 + (n - m)$ **in** *subst*)
apply *simp*
apply (*case-tac* $1 + (n - m) = 0$)
apply (*simp* *add*: *word-rec-0*)
apply (*rule-tac* $f =$ *word-rec* $?a$ $?b$ **in** *arg-cong*)
apply (*rule-tac* $t=m$ **and** $s=m + (1 + (n - m))$ **in** *subst*)
apply *simp*
apply (*simp* (*no-asm-use*))
apply (*simp* *add*: *word-rec-Suc* *word-rec-in2*)
apply (*erule* *impE*)
apply *uint-arith*
apply (*erule-tac* $x=x \circ$ *op* + 1 **in** *spec*)
apply (*erule-tac* $x=x$ 0 xa **in** *spec*)
apply *simp*
apply (*rule-tac* $t=\lambda a. x (1 + (n - m + a))$ **and** $s=\lambda a. x (1 + (n - m) + a)$ **in** *subst*)
apply (*clarsimp* *simp* *add*: *fun-eq-iff*)
apply (*rule-tac* $t=(1 + (n - m + xb))$ **and** $s=1 + (n - m) + xb$ **in** *subst*)
apply *simp*
apply (*rule* *refl*)
apply (*rule* *refl*)
done

lemma *word-rec-id*: *word-rec* z ($\lambda\cdot$. *id*) n = z

by (*induct* n) (*auto* *simp* *add*: *word-rec-0* *word-rec-Suc*)

lemma *word-rec-id-eq*: $\forall m < n. f$ m = *id* \implies *word-rec* z f n = z

apply (*erule* *rev-mp*)
apply (*induct* n)
apply (*auto* *simp* *add*: *word-rec-0* *word-rec-Suc*)
apply (*erule* *spec*, *erule* *mp*)

```

apply uint-arith
apply (drule-tac  $x=n$  in spec, erule impE)
apply uint-arith
apply simp
done

lemma word-rec-max:
   $\forall m \geq n. m \neq -1 \longrightarrow f\ m = id \implies word-rec\ z\ f\ -1 = word-rec\ z\ f\ n$ 
apply (subst word-rec-twice[where  $n=-1$  and  $m=-1 - n$ ])
apply simp
apply simp
apply (rule word-rec-id-eq)
apply clarsimp
apply (drule spec, rule mp, erule mp)
apply (rule word-plus-mono-right2[OF - order-less-imp-le])
prefer 2
apply assumption
apply simp
apply (erule contrapos-pn)
apply simp
apply (drule arg-cong[where  $f=\lambda x. x - n$ ])
apply simp
done

lemma unatSuc:
   $1 + n \neq (0::'a::len\ word) \implies unat\ (1 + n) = Suc\ (unat\ n)$ 
by unat-arith

lemma word-no-1 [simp]: (Numeral1::'a::len0 word) = 1
by (fact word-1-no [symmetric])

declare bin-to-bl-def [simp]

use ~/src/HOL/Word/Tools/word-lib.ML
use ~/src/HOL/Word/Tools/smt-word.ML
setup « SMT-Word.setup »

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.