

Machine Words in Isabelle/HOL

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

October 10, 2011

Abstract

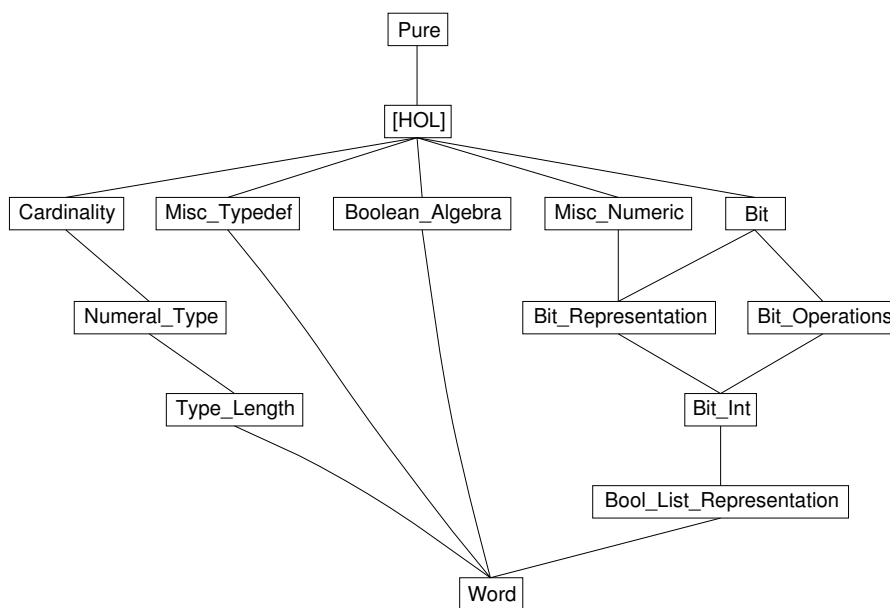
A formalisation of generic, fixed size machine words in Isabelle/HOL.
An earlier version of this formalisation is described in [1].

Contents

1	Cardinality: Cardinality of types	4
1.1	Preliminary lemmas	4
1.2	Cardinalities of types	4
1.3	Classes with at least 1 and 2	5
2	Numeral-Type: Numeral Syntax for Types	5
2.1	Numeral Types	5
2.2	Locales for modular arithmetic subtypes	7
2.3	Number ring instances	9
2.4	Syntax	11
2.5	Examples	12
3	Type-Length: Assigning lengths to types by typeclasses	12
4	Misc-Typedef: Type Definition Theorems	13
5	More lemmas about normal type definitions	14
5.1	Extended form of type definition predicate	15
6	Boolean-Algebra: Boolean Algebras	18
6.1	Complement	19
6.2	Conjunction	19
6.3	Disjunction	20
6.4	De Morgan's Laws	21
6.5	Symmetric Difference	21
7	Misc-Numeric: Useful Numerical Lemmas	24

8 Bit: The Field of Integers mod 2	33
8.1 Bits as a datatype	33
8.2 Type <i>bit</i> forms a field	34
8.3 Numerals at type <i>bit</i>	34
9 Bit-Representation: Basic Definitions for Binary Integers	35
9.1 Further properties of numerals	35
9.2 Destructors for binary integers	37
9.3 Truncating binary integers	41
9.4 Simplifications for (s)bintrunc	43
9.5 Splitting and concatenation	52
9.6 Miscellaneous lemmas	52
10 Bit-Operations: Syntactic classes for bitwise operations	53
10.1 Bitwise operations on <i>bit</i>	54
11 Bit-Int: Bitwise Operations on Binary Integers	55
11.1 Recursion combinators for bitstrings	55
11.2 Logical operations	56
11.3 Setting and clearing bits	64
11.4 Splitting and concatenation	66
11.5 Miscellaneous lemmas	69
12 Bool-List-Representation: Bool lists and integers	69
12.1 Operations on lists of booleans	69
12.2 Arithmetic in terms of bool lists	70
12.3 Repeated splitting or concatenation	88
13 Word: A type of finite bit strings	93
13.1 Type definition	93
13.2 Type conversions and casting	94
13.3 Arithmetic operations	95
13.4 Bit-wise operations	97
13.5 Shift operations	98
13.6 Rotation	98
13.7 Split and cat operations	99
13.8 Word Arithmetic	113
13.9 Transferring goals from words to ints	117
13.10 Order on fixed-length words	118
13.11 Conditions for the addition (etc) of two words to overflow . .	121
13.12 Definition of uint_arith	122
13.13 More on overflows and monotonicity	123
13.14 Arithmetic type class instantiations	128
13.15 Word and nat	129

13.16	Definition of <code>unat_arith</code> tactic	133
13.17	Cardinality, finiteness of set of words	137
13.18	Bitwise Operations on Words	138
13.19	Shifting, Rotating, and Splitting Words	148
13.19.1	shift functions in terms of lists of bools	151
13.19.2	Mask	157
13.19.3	Revcast	160
13.19.4	Slices	162
13.20	Split and cat	165
13.20.1	Split and slice	169
13.21	Rotation	173
13.21.1	Rotation of list to right	173
13.21.2	map, map2, commuting with rotate(r)	175
13.21.3	Word rotation commutes with bit-wise operations	178
13.22	Miscellaneous	180



1 Cardinality: Cardinality of types

```
theory Cardinality
imports 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  $\Rightarrow$  nat ((1CARD/(1'(-))))
```

```
translations CARD('t)  $\Rightarrow$  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 numeral-2-eq-2*)

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

lemma size0: 0 < n
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  $\leq$  n
by (rule Rep-less-n [THEN order-less-imp-le])

lemma Rep-inject-sym: x = y  $\longleftrightarrow$  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  $\in$  {0.. $n$ }  $\implies$  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* +
constrains *n :: int*
and *Rep :: 'a::{number-ring} \Rightarrow int*
and *Abs :: int \Rightarrow 'a::{number-ring}*
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-number-of*:
 $Rep\ (number\ of\ w) = number\ of\ w\ mod\ n$
by (*simp add: number-of-eq of-int-eq Rep-Abs-mod*)

lemma *iszero-number-of*:
 $iszero\ (number\ of\ w::'a) \iff number\ of\ w\ mod\ n = 0$
by (*simp add: Rep-simps number-of-eq of-int-eq iszero-def zero-def*)

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

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

end

2.3 Number ring instances

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

instantiation *num1* :: {*comm-ring, comm-monoid-mult, number*}
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*)**+**

instantiation bit0 and bit1 :: (finite) number-ring

begin

definition (*number-of w :: - bit0*) = *of-int w*

definition (*number-of w :: - bit1*) = *of-int w*

instance proof

qed (*rule number-of-bit0-def number-of-bit1-def*)**+**

end

interpretation *bit0*:

```

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

```

..

interpretation *bit1*:

```

mod-ring int CARD('a::finite bit1)
  Rep-bit1 :: 'a::finite bit1 ⇒ int
  Abs-bit1 :: int ⇒ '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-number-of* [simp] = *bit0.iszero-number-of*

lemmas *bit1-iszero-number-of* [simp] = *bit1.iszero-number-of*

2.4 Syntax

syntax

```

-NumeralType :: num-const => 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 (*-NumeralType*) [Const (str, -)] =
  mk-bintype (the (Int.fromString str))
  | numeral-tr (*-NumeralType*) 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: len-of (x::num0 itself) = 0
```

```
definition
  len-num1: len-of (x::num1 itself) = 1
```

```
instance ..
```

```
end
```

```
instantiation bit0 and bit1 :: (len0) len0
begin
```

```
definition
  len-bit0: len-of (x::'a::len0 bit0 itself) = 2 * len-of TYPE ('a)
```

```
definition
  len-bit1: len-of (x::'a::len0 bit1 itself) = 2 * len-of 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

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

declare *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 $\implies \text{Abs } r = a \implies \text{Rep } a = r$
by (*safe elim!: Abs-inverse*)

lemma *Rep-comp-inverse:*

Rep o f = g $\implies \text{Abs } o g = f$
using *Rep-inverse* **by** (*auto intro: ext*)

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

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

lemma *comp-Abs-inverse:*

f o Abs = g $\implies g o \text{Rep} = f$
using *Rep-inverse* **by** (*auto intro: ext*)

lemma *set-Rep:*

A = range Rep

proof (*rule set-eqI*)

fix *x*

show (*x ∈ A*) = (*x ∈ range Rep*)

by (*auto dest: Abs-inverse [of x, symmetric]*)

qed

lemma *set-Rep-Abs*: $A = \text{range } (\text{Rep } o \text{ Abs})$

proof (*rule set-eqI*)

fix x

show $(x \in A) = (x \in \text{range } (\text{Rep } o \text{ Abs}))$

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*: $\text{Abs } ` A = \text{UNIV}$

by (*auto intro!: image-eqI*)

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

lemma *fns1*:

$\text{Rep } o \text{ fa} = \text{fr } o \text{ Rep} \mid \text{fa } o \text{ Abs} = \text{Abs } o \text{ fr} \implies \text{Abs } o \text{ fr } o \text{ Rep} = \text{fa}$

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*:

$\text{Rep } o \text{ fa } o \text{ Abs} = \text{fr} \implies$

$\text{Rep } o \text{ fa} = \text{fr } o \text{ Rep} \ \& \ \text{fa } o \text{ Abs} = \text{Abs } o \text{ fr}$

by (*auto intro!: ext*)

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*:

$\text{norm } o \text{ norm} = \text{norm} \implies (\text{fr } o \text{ norm} = \text{norm } o \text{ fr}) =$

$(\text{norm } o \text{ fr } o \text{ norm} = \text{fr } o \text{ norm} \ \& \ \text{norm } o \text{ fr } o \text{ norm} = \text{norm } o \text{ fr})$

apply *safe*

apply (*simp-all add: o-assoc [symmetric]*)

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

done

lemma *fn-comm-power*:

$fa \circ tr = tr \circ fr \implies fa \wedge \wedge n \circ tr = tr \circ fr \wedge \wedge n$
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, standard*]

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

fixes *norm*

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

begin

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

$Abs (norm x) = Abs x$

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

lemma *td-th*:

$g \circ Abs = f \implies f (Rep x) = g x$

by (*drule comp-Abs-inverse* [*symmetric*]) *simp*

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

by (*auto simp: eq-norm intro!: ext*)

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

by (*auto simp: eq-norm' intro: td-th*)

lemmas *td* = *td-thm*

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

by (*auto simp: set-Rep-Abs eq-norm' eq-norm* [*symmetric*])

lemma *inverse-norm*:

$(Abs n = w) = (Rep w = norm n)$

apply (*rule iffI*)

apply (*clarsimp simp add: eq-norm*)

apply (*simp add: eq-norm'* [*symmetric*])

done

lemma *norm-eq-iff*:

$(norm x = norm y) = (Abs x = Abs y)$

by (*simp add: eq-norm'* [*symmetric*])

lemma *norm-comps*:

$Abs \circ norm = Abs$

```

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

```

lemmas norm-norm [simp] = norm-comps

lemma fns5:

```

Rep o fa o Abs = fr ==>
fr o norm = fr & norm o fr = fr
by (fold eq-norm') (auto intro!: ext)

```

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 (0)
  fixes one :: 'a (1)
  assumes conj-assoc:  $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$ 
  assumes disj-assoc:  $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$ 
  assumes conj-commute:  $x \sqcap y = y \sqcap x$ 
  assumes disj-commute:  $x \sqcup y = y \sqcup x$ 
  assumes conj-disj-distrib:  $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$ 
  assumes disj-conj-distrib:  $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$ 
  assumes conj-one-right [simp]:  $x \sqcap \mathbf{1} = x$ 
  assumes disj-zero-right [simp]:  $x \sqcup \mathbf{0} = x$ 
  assumes conj-cancel-right [simp]:  $x \sqcap \sim x = \mathbf{0}$ 
  assumes disj-cancel-right [simp]:  $x \sqcup \sim x = \mathbf{1}$ 

  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 = 0*

assumes *2: a \sqcup x = 1*

assumes *3: a \sqcap y = 0*

assumes *4: a \sqcup y = 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 1 = y \sqcap 1* **using** *2 4* **by** *simp*

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

qed

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

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** *$\sim x \sqcap x = 0$* **by** (*simp only: conj-commute*)

from *disj-cancel-right* **show** *$\sim x \sqcup x = 1$* **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* =
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*)

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*)

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

by (*simp only: conj-ac*)

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

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*)

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

by (*simp only: disj-ac*)

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 (**infixr** \oplus 65)

assumes *xor-def*: $x \oplus y = (x \sqcap \sim y) \sqcup (\sim x \sqcap y)$

```

sublocale boolean-xor < xor!: abel-semigroup xor proof
  fix x y z :: 'a
  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)
  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)
  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  $\oplus$  0 = x
by (simp only: xor-def compl-zero conj-one-right conj-zero-right disj-zero-right)

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

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

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

lemma xor-self [simp]: x  $\oplus$  x = 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 Main Parity
begin
```

```
lemma the-elimI:  $y = \{x\} \implies \text{the-elim } y = x$ 
  by simp
```

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

```
lemma gt-or-eq-0:  $0 < y \vee 0 = (y :: \text{nat})$  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 nobm1:
   $0 < (\text{number-of } w :: \text{nat}) \implies$ 
   $\text{number-of } w - (1 :: \text{nat}) = \text{number-of } (\text{Int.pred } w)$ 
  apply (unfold nat-number-of-def One-nat-def nat-1 [symmetric] pred-def)
  apply (simp add: number-of-eq nat-diff-distrib [symmetric])
  done
```

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

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

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

— the inverse(s) of *number-of*

```
lemma nmod2:  $n \bmod (2 :: \text{int}) = 0 \mid n \bmod 2 = 1$  by arith
```

```
lemma emep1:
   $\text{even } n \implies \text{even } d \implies 0 \leq d \implies (n + 1) \bmod (d :: \text{int}) = (n \bmod d) +$ 
```

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':  $(a \leq c - b) = (b + a \leq (c::int))$  by arith

lemma less-diff-eq':  $(a < c - b) = (b + a < (c::int))$  by arith

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

lemma diff-less-eq':  $(a - b < c) = (a < b + (c::int))$  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:
   $(2 * b + 1) \operatorname{div} 2 = (b::int)$  by arith

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

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

lemma axrmod2:
   $(1 + x + x) \bmod 2 = (1 :: int) \ \& \ (0 + x + x) \bmod 2 = (0 :: int)$  by arith

lemma axrdiv2:
   $(1 + x + x) \operatorname{div} 2 = (x :: int) \ \& \ (0 + x + x) \operatorname{div} 2 = (x :: int)$  by arith

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

```

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

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

lemma *zmod-uminus*: $-(a :: \text{int}) \bmod b \bmod b = -a \bmod b$
by (*simp add* : *zmod-zminus1-eq-if*)

lemma *zmod-zsub-distrib*: $((a :: \text{int}) - b) \bmod c = (a \bmod c - b \bmod c) \bmod c$
apply (*unfold diff-int-def*)
apply (*rule trans* [*OF - mod-add-eq* [*symmetric*]])
apply (*simp add*: *zmod-uminus mod-add-eq* [*symmetric*])
done

lemma *zmod-zsub-right-eq*: $((a :: \text{int}) - b) \bmod c = (a - b \bmod c) \bmod c$
apply (*unfold diff-int-def*)
apply (*rule trans* [*OF - mod-add-right-eq* [*symmetric*]])
apply (*simp add* : *zmod-uminus mod-add-right-eq* [*symmetric*])
done

lemma *zmod-zsub-left-eq*: $((a :: \text{int}) - b) \bmod c = (a \bmod c - b) \bmod c$
by (*rule mod-add-left-eq* [**where** $b = -b$, *simplified diff-int-def* [*symmetric*]])

lemma *zmod-zsub-self* [*simp*]:
 $((b :: \text{int}) - a) \bmod a = b \bmod a$
by (*simp add*: *zmod-zsub-right-eq*)

lemma *zmod-zmult1-eq-rev*:
 $b * a \bmod c = b \bmod c * a \bmod (c :: \text{int})$
apply (*simp add*: *mult-commute*)
apply (*subst zmod-zmult1-eq*)
apply *simp*
done

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

lemma *mod-plus-right*:
 $((a + x) \bmod m = (b + x) \bmod m) = (a \bmod m = b \bmod (m :: \text{nat}))$
apply (*induct x*)
apply (*simp-all add*: *mod-Suc*)
apply *arith*
done

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

lemmas *push-mods'* = *mod-add-eq* [*standard*]
mod-mult-eq [*standard*] *zmod-zsub-distrib* [*standard*]
zmod-uminus [*symmetric*, *standard*]

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

lemmas *mod-simps* =
mod-mult-self2-is-0 [*THEN eq-reflection*]
mod-mult-self1-is-0 [*THEN eq-reflection*]
mod-mod-trivial [*THEN eq-reflection*]

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

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

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

lemma *mod-nat-add*:
 $(x :: \text{nat}) < z \implies y < z \implies$
 $(x + y) \bmod z = (\text{if } x + y < z \text{ then } x + y \text{ else } x + y - z)$
 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*:
 $(x :: \text{nat}) < z \implies (x - y) \bmod z = x - y$
 by (*rule nat-mod-eq'*) *arith*

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

```

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

```

lemma *int-mod-eq*:

```

(0 :: int) <= b ==> b < n ==> a mod n = b mod n ==> a mod n = b
by clarsimp (rule mod-pos-pos-trivial)

```

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

lemma *int-mod-le*: $0 \leq a \implies 0 < (n :: int) \implies a \bmod n \leq a$

```

  apply (cases a < n)
  apply (auto dest: mod-pos-pos-trivial pos-mod-bound [where a=a])
done

```

lemma *int-mod-le'*: $0 \leq b - n \implies 0 < (n :: int) \implies b \bmod n \leq b - n$

```

  by (rule int-mod-le [where a = b - n and n = n, simplified])

```

lemma *int-mod-ge*: $a < n \implies 0 < (n :: int) \implies a \leq a \bmod n$

```

  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'*: $b < 0 \implies 0 < (n :: int) \implies b + n \leq b \bmod n$

```

  by (rule int-mod-ge [where a = b + n and n = n, simplified])

```

lemma *mod-add-if-z*:

```

(x :: int) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>
(x + y) mod z = (if x + y < z then x + y else x + y - z)
by (auto intro: int-mod-eq)

```

lemma *mod-sub-if-z*:

```

(x :: int) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>
(x - y) mod z = (if y <= x then x - y else x - y + z)
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*: $m \sim = (0 :: int) \implies (a + m * n) \bmod m = a \bmod m + n$

```

  apply (rule mcl)
  prefer 2
  apply (erule asm-rl)

```

apply (*simp add: zmde ring-distrib*)
done

lemma *eqne*: $\text{equiv } A \ r \implies X : A \ // \ r \implies X \sim = \{\}$
unfolding *equiv-def refl-on-def quotient-def Image-def* **by** *auto*

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

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

lemma *RI-int*: $(a, 0) : \text{Rep-Integ } (\text{int } a)$
unfolding *int-def* **by** *auto*

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

lemma *RI-minus*: $(a, b) : \text{Rep-Integ } x \implies (b, a) : \text{Rep-Integ } (- x)$
apply (*rule-tac z=x in eq-Abs-Integ*)
apply (*clarsimp simp: minus*)
done

lemma *RI-add*:
 $(a, b) : \text{Rep-Integ } x \implies (c, d) : \text{Rep-Integ } y \implies$
 $(a + c, b + d) : \text{Rep-Integ } (x + y)$
apply (*rule-tac z=x in eq-Abs-Integ*)
apply (*rule-tac z=y in eq-Abs-Integ*)
apply (*clarsimp simp: add*)
done

lemma *mem-same*: $a : S \implies a = b \implies b : S$
by *fast*

lemma *RI-eq-diff'*: $(a, b) : \text{Rep-Integ } (\text{int } a - \text{int } b)$
apply (*unfold diff-minus*)
apply (*rule mem-same*)
apply (*rule RI-minus RI-add RI-int*)
apply *simp*
done

lemma *RI-eq-diff*: $((a, b) : \text{Rep-Integ } x) = (\text{int } a - \text{int } b = x)$
apply *safe*

```

apply (rule Rep-Integ-same)
prefer 2
apply (erule asm-rl)
apply (rule RI-eq-diff')+
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]

```

```

lemma nat-no-eq-iff:
   $(\text{number-of } b :: \text{int}) \geq 0 \implies (\text{number-of } c :: \text{int}) \geq 0 \implies$ 
   $(\text{number-of } b = (\text{number-of } c :: \text{nat})) = (b = c)$ 
apply (unfold nat-number-of-def)
apply safe
apply (drule (2) eq-nat-nat-iff [THEN iffD1])

```

```

apply (simp add: number-of-eq)
done

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:  $(a :: \text{nat}) \text{ div } b * b \leq a$ 
apply (cases b)
prefer 2
apply (rule order-refl [THEN th2])
apply auto
done

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

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

lemma given-quot-alt:  $f > (0 :: \text{nat}) \implies (l * f + f - \text{Suc } 0) \text{ div } f = l$ 
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:  $(a::\text{nat}) < d \implies b \text{ dvd } d \implies a - a \text{ mod } b \leq d - b$ 
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'*:

$w * c < b * c \implies 0 \leq c \implies (w + 1) * c \leq b * (c :: int)$
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, standard*]

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*: $(0 :: nat) < sc \implies$

$(sc - n + (n + lb * n) <= m * n) = (sc + lb * n <= m * n)$
apply *safe*
apply *arith*
apply (*case-tac sc >= n*)
apply *arith*
apply (*insert linorder-le-less-linear [of m lb]*)
apply (*erule-tac k=n and k'=n in lrlem'*)
apply *arith*
apply *simp*
done

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

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

lemma *nonneg-mod-div*:

$0 <= a \implies 0 <= b \implies 0 <= (a \bmod b :: int) \ \& \ 0 <= a \operatorname{div} b$
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 = Abs-bit False
```

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

```
instance ..
```

end

```
rep-datatype (bit) 0::bit 1::bit
proof -
  fix P and x :: bit
  assume P (0::bit) and P (1::bit)
  then have  $\forall b. P (Abs-bit b)$ 
    unfolding zero-bit-def one-bit-def
    by (simp add: all-bool-eq)
  then show P x
    by (induct x) simp
next
  show (0::bit)  $\neq$  (1::bit)
    unfolding zero-bit-def one-bit-def
    by (simp add: Abs-bit-inject)
qed
```

```
lemma bit-not-0-iff [iff]: (x::bit)  $\neq$  0  $\longleftrightarrow$  x = 1
  by (induct x) simp-all
```

```
lemma bit-not-1-iff [iff]: (x::bit)  $\neq$  1  $\longleftrightarrow$  x = 0
  by (induct x) simp-all
```

8.2 Type *bit* forms a field

instantiation *bit* :: *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*]:
 $x / y = (x * y :: \text{bit})$

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

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

end

lemma *bit-add-self*: $x + x = (0 :: \text{bit})$
unfolding *plus-bit-def* **by** (*simp split: bit.split*)

lemma *bit-mult-eq-1-iff* [*simp*]: $x * y = (1 :: \text{bit}) \longleftrightarrow x = 1 \wedge y = 1$
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*: $x + y = (0 :: \text{bit}) \longleftrightarrow x = y$
unfolding *plus-bit-def* **by** (*simp split: bit.split*)

lemma *bit-add-eq-1-iff*: $x + y = (1 :: \text{bit}) \longleftrightarrow x \neq y$
unfolding *plus-bit-def* **by** (*simp split: bit.split*)

8.3 Numerals at type *bit*

instantiation *bit* :: *number-ring*
begin

definition *number-of-bit-def*:
(number-of w :: bit) = of-int w

instance proof
qed (*rule number-of-bit-def*)

end

All numerals reduce to either 0 or 1.

lemma *bit-minus1 [simp]: -1 = (1 :: bit)*
by (*simp only: number-of-Min uminus-bit-def*)

lemma *bit-number-of-even [simp]: number-of (Int.Bit0 w) = (0 :: bit)*
by (*simp only: number-of-Bit0 add-0-left bit-add-self*)

lemma *bit-number-of-odd [simp]: number-of (Int.Bit1 w) = (1 :: bit)*
by (*simp only: number-of-Bit1 add-assoc bit-add-self
monoid-add-class.add-0-right*)

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

lemma *BIT-B0-eq-Bit0 [simp]: w BIT 0 = Int.Bit0 w*
unfolding *Bit-def Bit0-def* **by** *simp*

lemma *BIT-B1-eq-Bit1 [simp]: w BIT 1 = Int.Bit1 w*
unfolding *Bit-def Bit1-def* **by** *simp*

lemmas *BIT-simps* = *BIT-B0-eq-Bit0 BIT-B1-eq-Bit1*

lemma *Min-ne-Pls* [iff]:
Int.Min \sim = *Int.Pl*
unfolding *Min-def Pls-def* **by** *auto*

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

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

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

lemma *number-of-False-cong*:
 $False \implies \text{number-of } x = \text{number-of } y$
by (*rule FalseE*)

lemma *BIT-eq*: $u \text{ BIT } b = v \text{ BIT } c \implies u = v \ \& \ b = c$
apply (*cases b*) **apply** (*simp-all*)
apply (*cases c*) **apply** (*simp-all*)
apply (*cases c*) **apply** (*simp-all*)
done

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

lemma *BIT-eq-iff* [simp]:
 $(u \text{ BIT } b = v \text{ BIT } c) = (u = v \ \wedge \ b = c)$
by (*rule iffI*) *auto*

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

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

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

lemma *no-no* [simp]: $\text{number-of } (\text{number-of } i) = i$
unfolding *number-of-eq* **by** *simp*

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 *B1-mod-2 [simp]*: $(\text{Int.Bit1 } w) \text{ mod } 2 = 1$
unfolding *numeral-simps number-of-is-id* **by** (*simp add: z1pmod2*)

lemma *B0-mod-2 [simp]*: $(\text{Int.Bit0 } w) \text{ mod } 2 = 0$
unfolding *numeral-simps number-of-is-id* **by** *simp*

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*: $EX \ w \ b. \ w \ \text{BIT} \ b = \text{bin}$
apply (*unfold Bit-def*)
apply (*cases even bin*)
apply (*clarsimp simp: even-equiv-def*)
apply (*auto simp: odd-equiv-def bitval-def split: bit.split*)
done

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

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**

$bin_rest\ w = w\ div\ 2$

definition $bin_rl :: int \Rightarrow int \times bit$ **where**

$bin_rl\ w = (bin_rest\ w, bin_last\ w)$

lemma $bin_rl_char: bin_rl\ w = (r, l) \longleftrightarrow r\ BIT\ l = w$

apply $(cases\ l)$

apply $(auto\ simp\ add: bin_rl-def\ bin_last-def\ bin_rest-def)$

unfolding $Pls-def\ Min-def\ Bit0-def\ Bit1-def\ number-of-is-id$

apply $arith+$

done

primrec bin_nth **where**

$Z: bin_nth\ w\ 0 = (bin_last\ w = (1::bit))$

$| Suc: bin_nth\ w\ (Suc\ n) = bin_nth\ (bin_rest\ w)\ n$

lemma $bin_rl-simps [simp]:$

$bin_rl\ Int.Pls = (Int.Pls, (0::bit))$

$bin_rl\ Int.Min = (Int.Min, (1::bit))$

$bin_rl\ (Int.Bit0\ r) = (r, (0::bit))$

$bin_rl\ (Int.Bit1\ r) = (r, (1::bit))$

$bin_rl\ (r\ BIT\ b) = (r, b)$

unfolding $bin_rl-char$ **by** $simp-all$

lemma $bin_rl-simp [simp]:$

$bin_rest\ w\ BIT\ bin_last\ w = w$

by $(simp\ add: iffD1 [OF\ bin_rl-char\ bin_rl-def])$

lemma $bin-abs-lem:$

$bin = (w\ BIT\ b) ==> \sim\ bin = Int.Min\ ---> \sim\ bin = Int.Pls\ --->$

$nat\ (abs\ w) < nat\ (abs\ bin)$

apply $(clarsimp\ simp\ add: bin_rl-char)$

apply $(unfold\ Pls-def\ Min-def\ Bit-def)$

apply $(cases\ b)$

apply $(clarsimp, arith)$

apply $(clarsimp, arith)$

done

lemma $bin-induct:$

assumes $PPls: P\ Int.Pls$

and $PMin: P\ Int.Min$

and $PBit: !!bin\ bit. P\ bin ==> P\ (bin\ BIT\ bit)$

shows $P\ bin$

apply $(rule-tac\ P=P\ and\ a=bin\ and\ f1=nat\ o\ abs$

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 numeral-induct:
assumes Pls: P Int.Pls
assumes Min: P Int.Min
assumes Bit0:  $\bigwedge w. \llbracket P w; w \neq \text{Int.Pls} \rrbracket \implies P (\text{Int.Bit0 } w)$ 
assumes Bit1:  $\bigwedge w. \llbracket P w; w \neq \text{Int.Min} \rrbracket \implies P (\text{Int.Bit1 } w)$ 
shows P x
apply (induct x rule: bin-induct)
  apply (rule Pls)
  apply (rule Min)
apply (case-tac bit)
  apply (case-tac bin = Int.Pls)
    apply simp
  apply (simp add: Bit0)
apply (case-tac bin = Int.Min)
  apply simp
apply (simp add: Bit1)
done

```

```

lemma bin-rest-simps [simp]:
bin-rest Int.Pls = Int.Pls
bin-rest Int.Min = Int.Min
bin-rest (Int.Bit0 w) = w
bin-rest (Int.Bit1 w) = w
bin-rest (w BIT b) = w
using bin-rl-simps bin-rl-def by auto

```

```

lemma bin-last-simps [simp]:
bin-last Int.Pls = (0::bit)
bin-last Int.Min = (1::bit)
bin-last (Int.Bit0 w) = (0::bit)
bin-last (Int.Bit1 w) = (1::bit)
bin-last (w BIT b) = b
using bin-rl-simps bin-rl-def by auto

```

```

lemma bin-r-l-extras [simp]:
bin-last 0 = (0::bit)
bin-last (- 1) = (1::bit)
bin-last -1 = (1::bit)
bin-last 1 = (1::bit)
bin-rest 1 = 0
bin-rest 0 = 0
bin-rest (- 1) = - 1
bin-rest -1 = -1
by (simp-all add: bin-last-def bin-rest-def)

```

lemma *bin-last-mod*:

```

bin-last w = (if w mod 2 = 0 then (0::bit) else (1::bit))
apply (case-tac w rule: bin-exhaust)
apply (case-tac b)
apply auto
done

```

lemma *bin-rest-div*:

```

bin-rest w = w div 2
apply (case-tac w rule: bin-exhaust)
apply (rule trans)
apply clarsimp
apply (rule refl)
apply (drule trans)
apply (rule Bit-def)
apply (simp add: bitval-def z1pdiv2 split: bit.split)
done

```

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

lemma *Bit0-div2* [simp]: (Int.Bit0 w) div 2 = w
using *Bit-div2* [where b=(0::bit)] **by** simp

lemma *Bit1-div2* [simp]: (Int.Bit1 w) div 2 = w
using *Bit-div2* [where b=(1::bit)] **by** simp

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

```

ALL y. bin-nth x = bin-nth y --> x = y
apply (induct x rule: bin-induct)
apply safe
apply (erule rev-mp)
apply (induct-tac y rule: bin-induct)
apply (safe del: subset-antisym)
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 del: subset-antisym)
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 BIT-eqI)
apply (drule-tac x=0 in fun-cong, force)
apply (rule ext)
apply (drule-tac x=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], standard]

lemma *bin-nth-Pls* [simp]: $\sim \text{bin-nth Int.Pl } n$
by (induct n) auto

lemma *bin-nth-Min* [simp]: $\text{bin-nth Int.Min } n$
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-minus-Bit0* [simp]:
 $0 < n \implies \text{bin-nth } (\text{Int.Bit0 } w) n = \text{bin-nth } w (n - 1)$
using bin-nth-minus [where b=(0::bit)] **by** simp

lemma *bin-nth-minus-Bit1* [simp]:
 $0 < n \implies \text{bin-nth } (\text{Int.Bit1 } w) n = \text{bin-nth } w (n - 1)$
using bin-nth-minus [where b=(1::bit)] **by** simp

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

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

lemmas *bin-nth-simps* =
bin-nth-0 bin-nth-Suc bin-nth-Pls bin-nth-Min bin-nth-minus
bin-nth-minus-Bit0 bin-nth-minus-Bit1

9.3 Truncating binary integers

definition

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

lemma *bin-sign-simps* [simp]:
bin-sign Int.Pls = Int.Pls
bin-sign Int.Min = Int.Min
bin-sign (Int.Bit0 w) = bin-sign w
bin-sign (Int.Bit1 w) = bin-sign w
bin-sign (w BIT b) = bin-sign w
by (*unfold bin-sign-def numeral-simps Bit-def bitval-def*) (*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* \Rightarrow *int* \Rightarrow *int* **where**
Z : *bintrunc 0 bin = Int.Pls*
| *Suc* : *bintrunc (Suc n) bin = bintrunc n (bin-rest bin) BIT (bin-last bin)*

primrec *sbintrunc* :: *nat* \Rightarrow *int* \Rightarrow *int* **where**
Z : *sbintrunc 0 bin =*
(case bin-last bin of (1::bit) \Rightarrow Int.Min | (0::bit) \Rightarrow Int.Pls)
| *Suc* : *sbintrunc (Suc n) bin = sbintrunc n (bin-rest bin) BIT (bin-last bin)*

lemma [code]:
sbintrunc 0 bin =
(case bin-last bin of (1::bit) \Rightarrow - 1 | (0::bit) \Rightarrow 0)
sbintrunc (Suc n) bin = sbintrunc n (bin-rest bin) BIT (bin-last bin)
apply *simp-all* **apply** (*cases bin-last bin*) **apply** *simp* **apply** (*unfold Min-def number-of-is-id*) **apply** *simp* **done**

lemma *sign-bintr*:
!!*w*. *bin-sign (bintrunc n w) = Int.Pls*
by (*induct n*) *auto*

lemma *bintrunc-mod2p*:
!!*w*. *bintrunc n w = (w mod 2 ^ n :: int)*
apply (*induct n, clarsimp*)
apply (*simp add: bin-last-mod bin-rest-div Bit-def zmod-zmult2-eq*
cong: number-of-False-cong)
done

lemma *sbintrunc-mod2p*:
!!*w*. *sbintrunc n w = ((w + 2 ^ n) mod 2 ^ (Suc n) - 2 ^ n :: int)*
apply (*induct n*)
apply *clarsimp*
apply (*subst mod-add-left-eq*)
apply (*simp add: bin-last-mod*)
apply (*simp add: number-of-eq*)
apply *clarsimp*
apply (*simp add: bin-last-mod bin-rest-div Bit-def*
cong: number-of-False-cong)

```

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 *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*:

```

!!bin. (bin-sign (sbintrunc n bin) = Int.Min) = bin-nth bin n
apply (induct n)
apply (case-tac bin rule: bin-exhaust, case-tac b, auto)+
done

```

lemma *nth-bintr*:

```

!!w m. bin-nth (bintrunc m w) n = (n < m & bin-nth w n)
apply (induct n)
apply (case-tac m, auto)[1]
apply (case-tac m, auto)[1]
done

```

lemma *nth-sbintr*:

```

!!w m. bin-nth (sbintrunc m w) n =
  (if n < m then bin-nth w n else bin-nth w m)
apply (induct n)
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 (Int.Bit0 w) n = (EX m. n = Suc m & bin-nth w m)
using bin-nth-Bit [where b=(0::bit)] by simp

```

lemma *bin-nth-Bit1*:

```

bin-nth (Int.Bit1 w) n = (n = 0 | (EX m. n = Suc m & bin-nth w m))
using bin-nth-Bit [where b=(1::bit)] by simp

```

lemma *bintrunc-bintrunc-l*:

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

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

lemma *sbintrunc-sbintrunc-l*:

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

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

lemma *bintrunc-bintrunc-ge*:

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

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

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

$\text{bintrunc } m (\text{bintrunc } n w) = \text{bintrunc } (\text{min } m n) w$

apply (*rule bin-eqI*)

apply (*auto simp: nth-bintr*)

done

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

$\text{sbintrunc } m (\text{sbintrunc } n w) = \text{sbintrunc } (\text{min } m n) w$

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=Int.Pls*, *simplified bin-last-simps bin-rest-simps, standard*]

lemmas *bintrunc-Min* [*simp*] =

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

lemmas *bintrunc-BIT* [*simp*] =

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

lemma *bintrunc-Bit0* [*simp*]:

$\text{bintrunc } (\text{Suc } n) (\text{Int.Bit0 } w) = \text{Int.Bit0 } (\text{bintrunc } n w)$

using *bintrunc-BIT* [**where** *b=(0::bit)*] **by** *simp*

lemma *bintrunc-Bit1* [*simp*]:

$\text{bintrunc } (\text{Suc } n) (\text{Int.Bit1 } w) = \text{Int.Bit1 } (\text{bintrunc } n w)$

using *bintrunc-BIT* [**where** *b=(1::bit)*] **by** *simp*

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

bintrunc-Bit0 bintrunc-Bit1

lemmas *sbintrunc-Suc-Pls* =

sbintrunc.Suc [**where** *bin=Int.Pls*, *simplified bin-last-simps bin-rest-simps, stan-*

dard]

lemmas *sbintrunc-Suc-Min* =

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

lemmas *sbintrunc-Suc-BIT* [*simp*] =

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

lemma *sbintrunc-Suc-Bit0* [*simp*]:

sbintrunc (Suc n) (Int.Bit0 w) = Int.Bit0 (sbintrunc n w)
using *sbintrunc-Suc-BIT* [**where** *b=(0::bit)*] **by** *simp*

lemma *sbintrunc-Suc-Bit1* [*simp*]:

sbintrunc (Suc n) (Int.Bit1 w) = Int.Bit1 (sbintrunc n w)
using *sbintrunc-Suc-BIT* [**where** *b=(1::bit)*] **by** *simp*

lemmas *sbintrunc-Sucs = sbintrunc-Suc-Pls sbintrunc-Suc-Min sbintrunc-Suc-BIT*
sbintrunc-Suc-Bit0 sbintrunc-Suc-Bit1

lemmas *sbintrunc-Pls* =

sbintrunc.Z [**where** *bin=Int.Pl*,
simplified bin-last-simps bin-rest-simps bit.simps, *standard*]

lemmas *sbintrunc-Min* =

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

lemmas *sbintrunc-0-BIT-B0* [*simp*] =

sbintrunc.Z [**where** *bin=w BIT (0::bit)*,
simplified bin-last-simps bin-rest-simps bit.simps, *standard*]

lemmas *sbintrunc-0-BIT-B1* [*simp*] =

sbintrunc.Z [**where** *bin=w BIT (1::bit)*,
simplified bin-last-simps bin-rest-simps bit.simps, *standard*]

lemma *sbintrunc-0-Bit0* [*simp*]: *sbintrunc 0 (Int.Bit0 w) = Int.Pl*

using *sbintrunc-0-BIT-B0* **by** *simp*

lemma *sbintrunc-0-Bit1* [*simp*]: *sbintrunc 0 (Int.Bit1 w) = Int.Min*

using *sbintrunc-0-BIT-B1* **by** *simp*

lemmas *sbintrunc-0-simps* =

sbintrunc-Pls sbintrunc-Min sbintrunc-0-BIT-B0 sbintrunc-0-BIT-B1
sbintrunc-0-Bit0 sbintrunc-0-Bit1

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

lemmas *sbintrunc-simps* = *sbintrunc-0-simps sbintrunc-Sucs*

lemma *bintrunc-minus*:

$0 < n \implies \text{bintrunc } (\text{Suc } (n - 1)) w = \text{bintrunc } n w$
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*], *standard*]

lemmas *sbintrunc-minus-simps* =

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

lemma *bintrunc-n-Pls* [*simp*]:

$\text{bintrunc } n \text{ Int.Pl} = \text{Int.Pl}$
by (*induct n*) *auto*

lemma *sbintrunc-n-PM* [*simp*]:

$\text{sbintrunc } n \text{ Int.Pl} = \text{Int.Pl}$
 $\text{sbintrunc } n \text{ Int.Min} = \text{Int.Min}$
by (*induct n*) *auto*

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

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

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

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

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

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

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

lemma *bintrunc-0-Min*: $\text{bintrunc } 0 \text{ Int.Min} = \text{Int.Pl}$

by *auto*

lemma *bintrunc-0-BIT*: $\text{bintrunc } 0 (w \text{ BIT } b) = \text{Int.Pl}$

by *auto*

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*, *standard*]

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

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

lemmas *sbintrunc-Suc-Is* =
sbintrunc-Sucs(1-3) [THEN *thobini1* [THEN [2] *trans*], *standard*]

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

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

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

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

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

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

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

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

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

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

$0 < n \implies \text{bintrunc } n \ x = \text{bintrunc } n \ y \iff$
 $\text{sbintrunc } (n - 1) \ x = \text{sbintrunc } (n - 1) \ y$
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, standard]

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

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

lemma no-bintr-alt:
number-of (bintrunc n w) = w mod 2^n
by (simp add: number-of-eq bintrunc-mod2p)

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. $0 \leq i \ \& \ i < 2^n$ }
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-bintr:
number-of (bintrunc n w) = (number-of w mod 2^n :: int)
by (simp add : bintrunc-mod2p number-of-eq)

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 no-sbintr:
number-of (sbintrunc n w) =
((number-of w + 2^n) mod $2^{Suc\ n} - 2^n$:: int)
by (simp add : no-sbintr-alt2 number-of-eq)

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 ==> a + 2^k + 2^(Suc k) <= (a + 2^k) mod 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) ==> a + 2^k + 2^(Suc k) <= (a + 2^k) mod 2^(Suc k)
by (rule sb-inc-lem) simp

```

lemma *sbintrunc-inc*:

```

x < - (2^n) ==> x + 2^(Suc n) <= sbintrunc n x
unfolding no-sbintr-alt2 by (drule sb-inc-lem') simp

```

lemma *sb-dec-lem*:

```

(0::int) <= - (2^k) + a ==> (a + 2^k) mod (2 * 2^k) <= - (2^k) + a
by (rule int-mod-le' [where n = 2 ^ (Suc k) and b = a + 2 ^ k,
simplified zless2p, OF - TrueI, simplified])

```

lemma *sb-dec-lem'*:

```

(2::int) ^ k <= a ==> (a + 2^k) mod (2 * 2^k) <= - (2^k) + a
by (rule iffD1 [OF diff-le-eq', THEN sb-dec-lem, simplified])

```

lemma *sbintrunc-dec*:

```

x >= (2^n) ==> x - 2^(Suc n) >= sbintrunc n x
unfolding no-sbintr-alt2 by (drule sb-dec-lem') simp

```

lemmas *zmod-uminus'* = *zmod-uminus* [where b=c, standard]

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

lemmas *brdmod1s'* [symmetric] =

```

mod-add-left-eq mod-add-right-eq
zmod-zsub-left-eq zmod-zsub-right-eq
zmod-zmult1-eq zmod-zmult1-eq-rev

```

lemmas *brdmods'* [symmetric] =

```

zpower-zmod' [symmetric]
trans [OF mod-add-left-eq mod-add-right-eq]
trans [OF zmod-zsub-left-eq zmod-zsub-right-eq]
trans [OF zmod-zmult1-eq zmod-zmult1-eq-rev]
zmod-uminus' [symmetric]

```

mod-add-left-eq [where $b = 1::int$]
zmod-zsub-left-eq [where $b = 1$]

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

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

lemma *bintr-ge0*: $(0 :: int) \leq \text{number-of } (\text{bintrunc } n \ w)$
by (*simp add : no-bintr m2pths*)

lemma *bintr-lt2p*: $\text{number-of } (\text{bintrunc } n \ w) < (2^n :: int)$
by (*simp add : no-bintr m2pths*)

lemma *bintr-Min*:
 $\text{number-of } (\text{bintrunc } n \ \text{Int.Min}) = (2^n :: int) - 1$
by (*simp add : no-bintr m1mod2k*)

lemma *sbintr-ge*: $(-(2^n :: int) \leq \text{number-of } (\text{sbintrunc } n \ w))$
by (*simp add : no-sbintr m2pths*)

lemma *sbintr-lt*: $\text{number-of } (\text{sbintrunc } n \ w) < (2^n :: int)$
by (*simp add : no-sbintr m2pths*)

lemma *bintrunc-Suc*:
 $\text{bintrunc } (\text{Suc } n) \ \text{bin} = \text{bintrunc } n \ (\text{bin-rest } \text{bin}) \ \text{BIT } \text{bin-last } \text{bin}$
by (*case-tac bin rule: bin-exhaust*) *auto*

lemma *sign-Pls-ge-0*:
 $(\text{bin-sign } \text{bin} = \text{Int.Pl}) = (\text{number-of } \text{bin} \geq (0 :: int))$
by (*induct bin rule: numeral-induct*) *auto*

lemma *sign-Min-lt-0*:
 $(\text{bin-sign } \text{bin} = \text{Int.Min}) = (\text{number-of } \text{bin} < (0 :: int))$
by (*induct bin rule: numeral-induct*) *auto*

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

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

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

by *auto*

lemma *bin-rest-strunc*:

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

by (*induct n*) *auto*

lemma *bintrunc-rest [simp]*:

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

apply (*induct n, simp*)

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

apply (*auto simp: bintrunc-bintrunc-l*)

done

lemma *sbintrunc-rest [simp]*:

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

apply (*induct n, simp*)

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

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

done

lemma *bintrunc-rest'*:

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

by (*rule ext*) *auto*

lemma *sbintrunc-rest'*:

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

by (*rule ext*) *auto*

lemma *rco-lem*:

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

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*: $(f \circ g) \wedge n \circ f = f \circ (g \circ f) \wedge n$

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*, *Int.Pls*)
 | *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 add: Pls-def*)

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* \Longrightarrow *f* ^^ *n* = *f* \circ *f* ^^ (*n* - 1)
by (*cases n*) *simp-all*

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

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

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

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

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

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

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

lemma *list-exhaust-size-gt0*:
assumes *y*: $\bigwedge a$ *list*. *y* = *a* # *list* \Longrightarrow *P*
shows 0 < *length y* \Longrightarrow *P*
apply (*cases y*, *simp*)

```

apply (rule y)
apply fastforce
done

```

```

lemma list-exhaust-size-eq0:
  assumes y: y = []  $\implies$  P
  shows length y = 0  $\implies$  P
  apply (cases y)
    apply (rule y, simp)
  apply simp
done

```

```

lemma size-Cons-lem-eq:
  y = xa # list  $\implies$  size y = Suc k  $\implies$  size list = k
  by auto

```

```

lemma size-Cons-lem-eq-bin:
  y = xa # list  $\implies$  size y = number-of (Int.succ k)  $\implies$ 
    size list = number-of k
  by (auto simp: pred-def succ-def split add : split-if-asm)

```

```

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

```

```

lemma not-B1-is-B0: y  $\neq$  (1::bit)  $\implies$  y = (0::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 ⇒ 'a      (NOT - [70] 71)
  and bitAND  :: 'a ⇒ 'a ⇒ 'a (infixr AND 64)
  and bitOR   :: 'a ⇒ 'a ⇒ 'a (infixr OR  59)
  and bitXOR  :: 'a ⇒ 'a ⇒ 'a (infixr XOR 59)

```

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

Testing and shifting operations.

```

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

```

```

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

```

10.1 Bitwise operations on *bit*

```

instantiation bit :: bit
begin

```

```

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

```

```

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

```

```

primrec bitOR-bit where
  0 OR y = (y::bit)
  | 1 OR y = (1::bit)

```

```

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

```

```

instance ..

```

```

end

```

```

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

```

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

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

lemma *bit-ops-comm*:

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

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

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

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

by (*cases x auto*)

end

11 Bit-Int: Bitwise Operations on Binary Integers

theory *Bit-Int*

imports *Bit-Representation Bit-Operations*

begin

11.1 Recursion combinators for bitstrings

function *bin-rec* :: $'a \Rightarrow 'a \Rightarrow (\text{int} \Rightarrow \text{bit} \Rightarrow 'a \Rightarrow 'a) \Rightarrow \text{int} \Rightarrow 'a$ **where**

$\text{bin-rec } f1 \ f2 \ f3 \ \text{bin} = (\text{if } \text{bin} = 0 \text{ then } f1$
 $\text{else if } \text{bin} = -1 \text{ then } f2$
 $\text{else } f3 \ (\text{bin-rest } \text{bin}) \ (\text{bin-last } \text{bin}) \ (\text{bin-rec } f1 \ f2 \ f3 \ (\text{bin-rest } \text{bin})))$
by *pat-completeness auto*

termination by (*relation measure (nat o abs o snd o snd o snd)*)

(*simp-all add: bin-last-def bin-rest-def*)

declare *bin-rec.simps* [*simp del*]

lemma *bin-rec-PM*:

$f = \text{bin-rec } f1 \ f2 \ f3 \ ==> f \ \text{Int.Pls} = f1 \ \& \ f \ \text{Int.Min} = f2$

by (*unfold Pls-def Min-def*) (*simp add: bin-rec.simps*)

lemma *bin-rec-Pls*: $\text{bin-rec } f1 \ f2 \ f3 \ \text{Int.Pl}s = f1$
by (*unfold Pls-def Min-def*) (*simp add: bin-rec.simps*)

lemma *bin-rec-Min*: $\text{bin-rec } f1 \ f2 \ f3 \ \text{Int.Min} = f2$
by (*unfold Pls-def Min-def*) (*simp add: bin-rec.simps*)

lemma *bin-rec-Bit0*:
 $f3 \ \text{Int.Pl}s \ (0::\text{bit}) \ f1 = f1 \implies$
 $\text{bin-rec } f1 \ f2 \ f3 \ (\text{Int.Bit0 } w) = f3 \ w \ (0::\text{bit}) \ (\text{bin-rec } f1 \ f2 \ f3 \ w)$
by (*unfold Pls-def Min-def Bit0-def Bit1-def*) (*simp add: bin-rec.simps bin-last-def bin-rest-def*)

lemma *bin-rec-Bit1*:
 $f3 \ \text{Int.Min} \ (1::\text{bit}) \ f2 = f2 \implies$
 $\text{bin-rec } f1 \ f2 \ f3 \ (\text{Int.Bit1 } w) = f3 \ w \ (1::\text{bit}) \ (\text{bin-rec } f1 \ f2 \ f3 \ w)$
apply (*cases w = Int.Min*)
apply (*simp add: bin-rec-Min*)
apply (*cases w = Int.Pl}s*)
apply (*simp add: bin-rec-Pls number-of-is-id Pls-def [symmetric] bin-rec.simps*)
apply (*subst bin-rec.simps*)
apply *auto* **unfolding** *Pls-def Min-def Bit0-def Bit1-def number-of-is-id* **apply** *auto*
done

lemma *bin-rec-Bit*:
 $f = \text{bin-rec } f1 \ f2 \ f3 \implies f3 \ \text{Int.Pl}s \ (0::\text{bit}) \ f1 = f1 \implies$
 $f3 \ \text{Int.Min} \ (1::\text{bit}) \ f2 = f2 \implies f \ (w \ \text{BIT } b) = f3 \ w \ b \ (f \ w)$
by (*cases b, simp add: bin-rec-Bit0, simp add: bin-rec-Bit1*)

lemmas *bin-rec-simps = refl [THEN bin-rec-Bit] bin-rec-Pls bin-rec-Min bin-rec-Bit0 bin-rec-Bit1*

11.2 Logical operations

bit-wise logical operations on the int type

instantiation *int :: bit*
begin

definition
int-not-def: $\text{bitNOT} = \text{bin-rec } (- \ 1) \ 0$
 $(\lambda w \ b \ s. \ s \ \text{BIT} \ (\text{NOT } b))$

definition
int-and-def: $\text{bitAND} = \text{bin-rec } (\lambda x. \ 0) \ (\lambda y. \ y)$
 $(\lambda w \ b \ s \ y. \ s \ (\text{bin-rest } y) \ \text{BIT} \ (b \ \text{AND} \ \text{bin-last } y))$

definition

int-or-def: $\text{bitOR} = \text{bin-rec } (\lambda x. x) (\lambda y. - 1)$
 $(\lambda w b s y. s (\text{bin-rest } y) \text{BIT } (b \text{ OR } \text{bin-last } y))$

definition

int-xor-def: $\text{bitXOR} = \text{bin-rec } (\lambda x. x) \text{bitNOT}$
 $(\lambda w b s y. s (\text{bin-rest } y) \text{BIT } (b \text{ XOR } \text{bin-last } y))$

instance ..**end**

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

$\text{NOT } \text{Int.Pls} = \text{Int.Min}$
 $\text{NOT } \text{Int.Min} = \text{Int.Pls}$
 $\text{NOT } (\text{Int.Bit0 } w) = \text{Int.Bit1 } (\text{NOT } w)$
 $\text{NOT } (\text{Int.Bit1 } w) = \text{Int.Bit0 } (\text{NOT } w)$
 $\text{NOT } (w \text{ BIT } b) = (\text{NOT } w) \text{ BIT } (\text{NOT } b)$

unfolding *int-not-def Pls-def* [*symmetric*] *Min-def* [*symmetric*] **by** (*simp-all add: bin-rec-simps*)

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

$\text{Int.Pls XOR } x = x$

unfolding *int-xor-def Pls-def* [*symmetric*] *Min-def* [*symmetric*] **by** (*simp add: bin-rec-PM*)

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

$\text{Int.Min XOR } x = \text{NOT } x$

unfolding *int-xor-def Pls-def* [*symmetric*] *Min-def* [*symmetric*] **by** (*simp add: bin-rec-PM*)

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

$(x \text{ BIT } b) \text{ XOR } (y \text{ BIT } c) = (x \text{ XOR } y) \text{ BIT } (b \text{ XOR } c)$

apply (*unfold int-xor-def Pls-def* [*symmetric*] *Min-def* [*symmetric*])

apply (*rule bin-rec-simps* (1) [*THEN fun-cong, THEN trans*])

apply (*rule ext, simp*)

prefer 2

apply *simp*

apply (*rule ext*)

apply (*simp add: int-not-simps* [*symmetric*])

done

lemma *int-xor-Bits2* [*simp*]:

$(\text{Int.Bit0 } x) \text{ XOR } (\text{Int.Bit0 } y) = \text{Int.Bit0 } (x \text{ XOR } y)$

$(\text{Int.Bit0 } x) \text{ XOR } (\text{Int.Bit1 } y) = \text{Int.Bit1 } (x \text{ XOR } y)$

$(\text{Int.Bit1 } x) \text{ XOR } (\text{Int.Bit0 } y) = \text{Int.Bit1 } (x \text{ XOR } y)$

$(\text{Int.Bit1 } x) \text{ XOR } (\text{Int.Bit1 } y) = \text{Int.Bit0 } (x \text{ XOR } y)$

unfolding *BIT-simps* [*symmetric*] *int-xor-Bits* **by** *simp-all*

lemma *int-xor-x-simps'*:

```

w XOR (Int.Pls BIT 0) = w
w XOR (Int.Min BIT 1) = NOT w
apply (induct w rule: bin-induct)
  apply simp-all[4]
  apply (unfold int-xor-Bits)
  apply clarsimp+
done

```

```

lemma int-xor-extra-simps [simp]:
w XOR Int.Pls = w
w XOR Int.Min = NOT w
using int-xor-x-simps' by simp-all

```

```

lemma int-or-Pls [simp]:
Int.Pls OR x = x
by (unfold int-or-def) (simp add: bin-rec-PM)

```

```

lemma int-or-Min [simp]:
Int.Min OR x = Int.Min
by (unfold int-or-def Pls-def [symmetric] Min-def [symmetric]) (simp add: bin-rec-PM)

```

```

lemma int-or-Bits [simp]:
(x BIT b) OR (y BIT c) = (x OR y) BIT (b OR c)
unfolding int-or-def Pls-def [symmetric] Min-def [symmetric] by (simp add:
bin-rec-simps)

```

```

lemma int-or-Bits2 [simp]:
(Int.Bit0 x) OR (Int.Bit0 y) = Int.Bit0 (x OR y)
(Int.Bit0 x) OR (Int.Bit1 y) = Int.Bit1 (x OR y)
(Int.Bit1 x) OR (Int.Bit0 y) = Int.Bit1 (x OR y)
(Int.Bit1 x) OR (Int.Bit1 y) = Int.Bit1 (x OR y)
unfolding BIT-simps [symmetric] int-or-Bits by simp-all

```

```

lemma int-or-x-simps':
w OR (Int.Pls BIT 0) = w
w OR (Int.Min BIT 1) = Int.Min
apply (induct w rule: bin-induct)
  apply simp-all[4]
  apply (unfold int-or-Bits)
  apply clarsimp+
done

```

```

lemma int-or-extra-simps [simp]:
w OR Int.Pls = w
w OR Int.Min = Int.Min
using int-or-x-simps' by simp-all

```

```

lemma int-and-Pls [simp]:
Int.Pls AND x = Int.Pls

```

unfolding *int-and-def* **by** (*simp add: bin-rec-PM*)

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

Int.Min AND x = x

unfolding *int-and-def* **by** (*simp add: bin-rec-PM*)

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

$(x \text{ BIT } b) \text{ AND } (y \text{ BIT } c) = (x \text{ AND } y) \text{ BIT } (b \text{ AND } c)$

unfolding *int-and-def Pls-def* [*symmetric*] *Min-def* [*symmetric*] **by** (*simp add: bin-rec-simps*)

lemma *int-and-Bits2* [*simp*]:

$(\text{Int.Bit0 } x) \text{ AND } (\text{Int.Bit0 } y) = \text{Int.Bit0 } (x \text{ AND } y)$

$(\text{Int.Bit0 } x) \text{ AND } (\text{Int.Bit1 } y) = \text{Int.Bit0 } (x \text{ AND } y)$

$(\text{Int.Bit1 } x) \text{ AND } (\text{Int.Bit0 } y) = \text{Int.Bit0 } (x \text{ AND } y)$

$(\text{Int.Bit1 } x) \text{ AND } (\text{Int.Bit1 } y) = \text{Int.Bit1 } (x \text{ AND } y)$

unfolding *BIT-simps* [*symmetric*] *int-and-Bits* **by** *simp-all*

lemma *int-and-x-simps'*:

$w \text{ AND } (\text{Int.PlS BIT } 0) = \text{Int.PlS}$

$w \text{ AND } (\text{Int.Min BIT } 1) = w$

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

apply *simp-all*[4]

apply (*unfold int-and-Bits*)

apply *clarsimp+*

done

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

$w \text{ AND } \text{Int.PlS} = \text{Int.PlS}$

$w \text{ AND } \text{Int.Min} = w$

using *int-and-x-simps'* **by** *simp-all*

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: $\forall y::\text{int. } x \text{ XOR } y = y \text{ XOR } x$

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

apply *simp-all*[6]

apply (*case-tac y rule: bin-exhaust, simp add: bit-ops-comm*)**+**

done

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

$(x::\text{int}) \text{ AND } x = x$

$(x::\text{int}) \text{ OR } x = x$

$(x::\text{int}) \text{ XOR } x = \text{Int.PlS}$

by (*induct x rule: bin-induct*) *auto*

lemma *int-not-not* [*simp*]: $NOT (NOT (x::int)) = x$
by (*induct x rule: bin-induct*) *auto*

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

lemma *bbw-ao-absorb*:
 $!!y::int. x AND (y OR x) = x \& x OR (y AND x) = x$
apply (*induct x rule: bin-induct*)
apply *auto*
apply (*case-tac* [!] *y rule: bin-exhaust*)
apply *auto*
apply (*case-tac* [!] *bit*)
apply *auto*
done

lemma *bbw-ao-absorbs-other*:
 $x AND (x OR y) = x \wedge (y AND x) OR x = (x::int)$
 $(y OR x) AND x = x \wedge x OR (x AND y) = (x::int)$
 $(x OR y) AND x = x \wedge (x AND y) OR x = (x::int)$
apply (*auto simp: bbw-ao-absorb int-or-comm*)
apply (*subst int-or-comm*)
apply (*simp add: bbw-ao-absorb*)
apply (*subst int-and-comm*)
apply (*subst int-or-comm*)
apply (*simp add: bbw-ao-absorb*)
apply (*subst int-and-comm*)
apply (*simp add: bbw-ao-absorb*)
done

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

lemma *int-xor-not*:
 $!!y::int. (NOT x) XOR y = NOT (x XOR y) \&$
 $x XOR (NOT y) = NOT (x XOR y)$
apply (*induct x rule: bin-induct*)
apply *auto*
apply (*case-tac y rule: bin-exhaust, auto,*
case-tac b, auto)
done

lemma *bbw-assocs'*:
 $!!y z::int. (x AND y) AND z = x AND (y AND z) \&$
 $(x OR y) OR z = x OR (y OR z) \&$
 $(x XOR y) XOR z = x XOR (y XOR z)$
apply (*induct x rule: bin-induct*)

```

apply (auto simp: int-xor-not)
apply (case-tac [!] y rule: bin-exhaust)
apply (case-tac [!] z rule: bin-exhaust)
apply (case-tac [!] bit)
      apply (case-tac [!] b)
      apply (auto simp del: BIT-simps)
done

lemma int-and-assoc:
  (x AND y) AND (z::int) = x AND (y AND z)
by (simp add: bbw-assocs')

lemma int-or-assoc:
  (x OR y) OR (z::int) = x OR (y OR z)
by (simp add: bbw-assocs')

lemma int-xor-assoc:
  (x XOR y) XOR (z::int) = x XOR (y XOR z)
by (simp add: bbw-assocs')

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

lemma bbw-lcs [simp]:
  (y::int) AND (x AND z) = x AND (y AND z)
  (y::int) OR (x OR z) = x OR (y OR z)
  (y::int) XOR (x XOR z) = x XOR (y XOR z)
apply (auto simp: bbw-assocs [symmetric])
apply (auto simp: bin-ops-comm)
done

lemma bbw-not-dist:
  !!y::int. NOT (x OR y) = (NOT x) AND (NOT y)
  !!y::int. NOT (x AND y) = (NOT x) OR (NOT y)
apply (induct x rule: bin-induct)
      apply auto
      apply (case-tac [!] y rule: bin-exhaust)
      apply (case-tac [!] bit, auto simp del: BIT-simps)
done

lemma bbw-oa-dist:
  !!y z::int. (x AND y) OR z =
    (x OR z) AND (y OR z)
apply (induct x rule: bin-induct)
      apply auto
      apply (case-tac y rule: bin-exhaust)
      apply (case-tac z rule: bin-exhaust)
      apply (case-tac ba, auto simp del: BIT-simps)
done

```

```

lemma bbw-ao-dist:
  !!y z::int. (x OR y) AND z =
    (x AND z) OR (y AND z)
  apply (induct x rule: bin-induct)
  apply auto
  apply (case-tac y rule: bin-exhaust)
  apply (case-tac z rule: bin-exhaust)
  apply (case-tac ba, auto simp del: BIT-simps)
done

```

```

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) = Int.Pls ==> 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: less-eq-int-code)
done

```

```

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

```

```

lemma bin-nth-ops:
  !!x y. bin-nth (x AND y) n = (bin-nth x n & bin-nth y n)
  !!x y. bin-nth (x OR y) n = (bin-nth x n | bin-nth y n)
  !!x y. bin-nth (x XOR y) n = (bin-nth x n ~ bin-nth y n)
  !!x. bin-nth (NOT x) n = (~ bin-nth x n)
  apply (induct n)
  apply safe
  apply (case-tac [!] x rule: bin-exhaust)
  apply (simp-all del: BIT-simps)

```

```

      apply (case-tac [!] y rule: bin-exhaust)
      apply (simp-all del: BIT-simps)
    apply (auto dest: not-B1-is-B0 intro: B1-ass-B0)
  done

```

```

lemma bin-add-not:  $x + NOT\ x = Int.Min$ 
  apply (induct x rule: bin-induct)
    apply clarsimp
    apply clarsimp
  apply (case-tac bit, auto)
  done

```

```

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)
  apply (induct n)
    apply auto
    apply (case-tac [!] x rule: bin-exhaust)
    apply (case-tac [!] y rule: bin-exhaust)
    apply auto
  done

```

```

lemma bin-trunc-xor:
  !!x y. bintrunc n (bintrunc n x XOR bintrunc n y) =
    bintrunc n (x XOR y)
  apply (induct n)
    apply auto
    apply (case-tac [!] x rule: bin-exhaust)
    apply (case-tac [!] y rule: bin-exhaust)
    apply auto
  done

```

```

lemma bin-trunc-not:
  !!x. bintrunc n (NOT (bintrunc n x)) = bintrunc n (NOT x)
  apply (induct n)
    apply auto
    apply (case-tac [!] x rule: bin-exhaust)
    apply auto
  done

```

```

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*]

11.3 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]:

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

by (*induct n*) *auto*

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

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

by (*induct n*) *auto*

lemma *bin-sc-sc-diff*:

!!*w m*. *m* ~ = *n* ==>

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

apply (*induct n*)

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

apply *auto*

done

lemma *bin-nth-sc-gen*:

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

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

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

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

by (*induct n*) *auto*

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

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

by (*induct n*) *auto*

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

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

apply (*induct n*)

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

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

done

lemma *bin-clr-le*:

!!*w*. *bin-sc n 0 w* <= *w*

```

apply (induct n)
apply (case-tac [!] w rule: bin-exhaust)
apply (auto simp del: BIT-simps)
apply (unfold Bit-def)
apply (simp-all add: bitval-def split: bit.split)
done

```

```

lemma bin-set-ge:
  !!w. bin-sc n 1 w >= w
apply (induct n)
apply (case-tac [!] w rule: bin-exhaust)
apply (auto simp del: BIT-simps)
apply (unfold Bit-def)
apply (simp-all add: bitval-def split: bit.split)
done

```

```

lemma bintr-bin-clr-le:
  !!w m. bintrunc n (bin-sc m 0 w) <= bintrunc n w
apply (induct n)
apply simp
apply (case-tac w rule: bin-exhaust)
apply (case-tac m)
apply (auto simp del: BIT-simps)
apply (unfold Bit-def)
apply (simp-all add: bitval-def split: bit.split)
done

```

```

lemma bintr-bin-set-ge:
  !!w m. bintrunc n (bin-sc m 1 w) >= bintrunc n w
apply (induct n)
apply simp
apply (case-tac w rule: bin-exhaust)
apply (case-tac m)
apply (auto simp del: BIT-simps)
apply (unfold Bit-def)
apply (simp-all add: bitval-def split: bit.split)
done

```

```

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

```

```

lemma bin-sc-TM [simp]: bin-sc n 1 Int.Min = Int.Min
  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, standard]

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

11.4 Splitting and concatenation

definition *bin-rcat* :: *nat* \Rightarrow *int list* \Rightarrow *int* **where**
bin-rcat n = foldl (%u v. bin-cat u n v) Int.Pls

lemma [*code*]:
bin-rcat n = foldl ($\lambda u v. bin-cat u n v$) 0
by (*simp add: bin-rcat-def Pls-def*)

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*:
 $!!y. bin-sign (bin-cat x n y) = bin-sign x$
by (*induct n 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*:
 $!!n y. bin-nth (bin-cat x k y) n =$
(if n < k then bin-nth y n else bin-nth x (n - k))
apply (*induct k*)

```

apply clarsimp
apply (case-tac n, auto)
done

```

```

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

```

```

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

```

```

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

```

```

lemma bin-cat-Pls [simp]:
  !!w. bin-cat Int.Pls n w = bintrunc n w
by (induct n auto)

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma split-bintrunc:
  !!b c. bin-split n c = (a, b) ==> b = bintrunc n c
by (induct n auto simp: Let-def split: ls-splits)

```

lemma *bin-cat-split*:

!!*v w. bin-split n w = (u, v) ==> w = bin-cat u n v*
by (*induct n*) (*auto simp: Let-def split: ls-splits*)

lemma *bin-split-cat*:

!!*w. bin-split n (bin-cat v n w) = (v, bintrunc n w)*
by (*induct n*) *auto*

lemma *bin-split-Pls* [*simp*]:

bin-split n Int.Pls = (Int.Pls, Int.Pls)
by (*induct n*) (*auto simp: Let-def split: ls-splits*)

lemma *bin-split-Min* [*simp*]:

bin-split n Int.Min = (Int.Min, bintrunc n Int.Min)
by (*induct n*) (*auto simp: Let-def split: ls-splits*)

lemma *bin-split-trunc*:

!!*m b c. bin-split (min m n) c = (a, b) ==>*
bin-split n (bintrunc m c) = (bintrunc (m - n) a, b)
apply (*induct n, 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*:

!!*m b c. bin-split n c = (a, b) ==>*
bin-split n (bintrunc m c) = (bintrunc (m - n) a, bintrunc m b)
apply (*induct n, 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*:

!!*b. bin-cat a n b = a * 2 ^ n + bintrunc n b*
apply (*induct n, clarsimp*)
apply (*simp add: Bit-def cong: number-of-False-cong*)
done

lemma *bin-split-num*:

!!*b. bin-split n b = (b div 2 ^ n, b mod 2 ^ n)*
apply (*induct n, clarsimp*)
apply (*simp add: bin-rest-div 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
cong: number-of-False-cong)

done

11.5 Miscellaneous lemmas

lemma *nth-2p-bin*:

!!*m*. *bin-nth* ($2 \wedge n$) *m* = (*m* = *n*)

apply (*induct n*)

apply *clarsimp*

apply *safe*

apply (*case-tac m*)

apply (*auto simp: trans [OF numeral-1-eq-1 [symmetric] number-of-eq]*)

apply (*case-tac m*)

apply (*auto simp: Bit-B0-2t [symmetric]*)

done

lemma *ex-eq-or*:

(*EX m*. *n* = *Suc m* & (*m* = *k* | *P m*)) = (*n* = *Suc k* | (*EX m*. *n* = *Suc m* & *P m*))

by *auto*

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* *Int.Pls*

lemma [*code*]:

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

by (*simp add: bl-to-bin-def Pls-def*)

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

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

primrec *takefill* :: '*a* ⇒ *nat* ⇒ '*a* list ⇒ '*a* list **where**
 Z: *takefill* *fill* 0 *xs* = []
 | *Suc*: *takefill* *fill* (*Suc* *n*) *xs* = (
 case *xs* of [] => *fill* # *takefill* *fill* *n* *xs*
 | *y* # *ys* => *y* # *takefill* *fill* *n* *ys*)

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

lemma *map2-Nil* [*simp*]: *map2* *f* [] *ys* = []
 unfolding *map2-def* **by** *auto*

lemma *map2-Nil2* [*simp*]: *map2* *f* *xs* [] = []
 unfolding *map2-def* **by** *auto*

lemma *map2-Cons* [*simp*]:
 map2 *f* (*x* # *xs*) (*y* # *ys*) = *f* *x* *y* # *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*:
 $(butlast\ \hat{\ }^n)\ bl = take\ (length\ bl - n)\ bl$
by (*induct n*) (*auto simp: butlast-take*)

lemma *bin-to-bl-aux-Pls-minus-simp* [*simp*]:
 $0 < n ==> bin_to_bl_aux\ n\ Int.Pls\ bl =$
 $bin_to_bl_aux\ (n - 1)\ Int.Pls\ (False\ \# \ bl)$
by (*cases n*) *auto*

lemma *bin-to-bl-aux-Min-minus-simp* [*simp*]:
 $0 < n ==> bin_to_bl_aux\ n\ Int.Min\ bl =$
 $bin_to_bl_aux\ (n - 1)\ Int.Min\ (True\ \# \ bl)$
by (*cases n*) *auto*

lemma *bin-to-bl-aux-Bit-minus-simp* [*simp*]:
 $0 < n ==> bin_to_bl_aux\ n\ (w\ BIT\ b)\ bl =$
 $bin_to_bl_aux\ (n - 1)\ w\ ((b = 1)\ \# \ bl)$
by (*cases n*) *auto*

lemma *bin-to-bl-aux-Bit0-minus-simp* [*simp*]:
 $0 < n ==> bin_to_bl_aux\ n\ (Int.Bit0\ w)\ bl =$
 $bin_to_bl_aux\ (n - 1)\ w\ (False\ \# \ bl)$
by (*cases n*) *auto*

lemma *bin-to-bl-aux-Bit1-minus-simp* [*simp*]:
 $0 < n ==> bin_to_bl_aux\ n\ (Int.Bit1\ w)\ bl =$
 $bin_to_bl_aux\ (n - 1)\ w\ (True\ \# \ bl)$
by (*cases n*) *auto*

Link between bin and bool list.

lemma *bl-to-bin-aux-append*:
 $bl_to_bin_aux\ (bs\ @ \ cs)\ w = bl_to_bin_aux\ cs\ (bl_to_bin_aux\ bs\ w)$
by (*induct bs arbitrary: w*) *auto*

lemma *bin-to-bl-aux-append*:
 $bin_to_bl_aux\ n\ w\ bs\ @ \ cs = bin_to_bl_aux\ n\ w\ (bs\ @ \ cs)$
by (*induct n arbitrary: w bs*) *auto*

lemma *bl-to-bin-append*:
 $bl_to_bin\ (bs\ @ \ cs) = bl_to_bin_aux\ cs\ (bl_to_bin\ bs)$
unfolding *bl-to-bin-def* **by** (*rule bl-to-bin-aux-append*)

lemma *bin-to-bl-aux-alt*:
 $bin_to_bl_aux\ n\ w\ bs = bin_to_bl\ n\ w\ @ \ bs$
unfolding *bin-to-bl-def* **by** (*simp add : bin-to-bl-aux-append*)

```

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

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

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

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

lemma bin-bl-bin: 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: 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

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

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

```

apply *simp*
done

lemma *bl-to-bin-False*: $bl\text{-to}\text{-bin} (False \# bl) = bl\text{-to}\text{-bin} bl$
unfolding *bl-to-bin-def* **by** *auto*

lemma *bl-to-bin-Nil*: $bl\text{-to}\text{-bin} [] = Int.Pl$
unfolding *bl-to-bin-def* **by** *auto*

lemma *bin-to-bl-Pls-aux*:
 $bin\text{-to}\text{-bl}\text{-aux} n Int.Pl$ $bl = replicate\ n\ False @ bl$
by (*induct n arbitrary: bl*) (*auto simp: replicate-app-Cons-same*)

lemma *bin-to-bl-Pls*: $bin\text{-to}\text{-bl} n Int.Pl = replicate\ n\ False$
unfolding *bin-to-bl-def* **by** (*simp add: bin-to-bl-Pls-aux*)

lemma *bin-to-bl-Min-aux* [*rule-format*] :
 $ALL\ bl.\ bin\text{-to}\text{-bl}\text{-aux} n Int.Min\ bl = replicate\ n\ True @ bl$
by (*induct n*) (*auto simp: replicate-app-Cons-same*)

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

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

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

declare
 $bin\text{-to}\text{-bl}\text{-trunc}$ [*simp*]
 $bl\text{-to}\text{-bin}\text{-False}$ [*simp*]
 $bl\text{-to}\text{-bin}\text{-Nil}$ [*simp*]

lemma *bin-to-bl-aux-bintr* [*rule-format*] :
 $ALL\ m\ bin\ bl.\ bin\text{-to}\text{-bl}\text{-aux} n (bintrunc\ m\ bin)\ bl =$
 $replicate\ (n - m)\ False @ bin\text{-to}\text{-bl}\text{-aux} (min\ n\ m)\ bin\ bl$
apply (*induct n*)
apply *clarsimp*
apply *clarsimp*
apply (*case-tac m*)
apply (*clarsimp simp: bin-to-bl-Pls-aux*)
apply (*erule thin-rl*)
apply (*induct-tac n*)
apply *auto*

done

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

lemma *bl-to-bin-rep-False*: *bl-to-bin* (*replicate n False*) = *Int.Pl*
by (*induct n*) *auto*

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

lemma *len-bin-to-bl* [*simp*]: *length* (*bin-to-bl n w*) = *n*
unfolding *bin-to-bl-def len-bin-to-bl-aux* **by** *auto*

lemma *sign-bl-bin'*:
bin-sign (*bl-to-bin-aux bs w*) = *bin-sign w*
by (*induct bs arbitrary: w*) *auto*

lemma *sign-bl-bin*: *bin-sign* (*bl-to-bin bs*) = *Int.Pl*
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) = Int.Min)
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) = Int.Min*)
unfolding *bin-to-bl-def* **by** (*rule bl-sbin-sign-aux*)

lemma *bin-nth-of-bl-aux* [*rule-format*]:
 $\forall w. \text{bin-nth } (\text{bl-to-bin-aux } \text{bl } w) \text{ } n =$
 $(n < \text{size bl} \ \& \ \text{rev bl ! } n \mid n \geq \text{length bl} \ \& \ \text{bin-nth } w \ (n - \text{size bl}))$
apply (*induct-tac bl*)
apply *clarsimp*
apply *clarsimp*
apply (*cut-tac x=n and y=size list in linorder-less-linear*)
apply (*erule disjE, simp add: nth-append*)
apply *auto*
done

lemma *bin-nth-of-bl*: *bin-nth* (*bl-to-bin bl*) *n* = (*n < length bl & rev bl ! n*)
unfolding *bl-to-bin-def* **by** (*simp add : bin-nth-of-bl-aux*)

lemma *bin-nth-bl* [rule-format] : ALL m w. n < m -->
 $bin_nth\ w\ n = nth\ (rev\ (bin_to_bl\ m\ w))\ n$
apply (induct n)
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* [rule-format] :
 $n < length\ xs \rightarrow rev\ xs\ !\ n = xs\ !\ (length\ xs - 1 - n)$
apply (induct-tac xs)
apply simp
apply (clarsimp simp add : nth-append nth.simps split add : nat.split)
apply (rule-tac f = %n. list ! n in arg-cong)
apply arith
done

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

lemma *nth-bin-to-bl-aux* [rule-format] :
 $ALL\ w\ n\ bl.\ n < m + length\ bl \rightarrow (bin_to_bl_aux\ m\ w\ bl)\ !\ n =$
 $(if\ n < m\ then\ bin_nth\ w\ (m - 1 - n)\ else\ bl\ !\ (n - m))$
apply (induct m)
apply clarsimp
apply clarsimp
apply (case-tac w rule: bin-exhaust)
apply simp
done

lemma *nth-bin-to-bl*: $n < m \implies (bin_to_bl\ m\ w)\ !\ n = bin_nth\ w\ (m - Suc\ n)$
unfolding bin-to-bl-def **by** (simp add : nth-bin-to-bl-aux)

lemma *bl-to-bin-lt2p-aux* [rule-format]:
 $\forall w.\ bl_to_bin_aux\ bs\ w < (w + 1) * (2 ^ length\ bs)$
apply (induct bs)
apply clarsimp
apply clarsimp
apply safe
apply (erule allE, erule xtr8 [rotated],
simp add: numeral-simps algebra-simps cong add : number-of-False-cong)+
done

lemma *bl-to-bin-lt2p*: $bl_to_bin\ bs < (2 ^ length\ bs)$

```

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 [rule-format] :
   $\forall w. \text{bl-to-bin-aux } bs \ w \geq w * (2 \wedge \text{length } bs)$ 
apply (induct bs)
apply clarsimp
apply clarsimp
apply safe
apply (erule alle, erule preorder-class.order-trans [rotated],
  simp add: numeral-simps algebra-simps cong add : number-of-False-cong)+
done

```

```

lemma bl-to-bin-ge0: 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:
   $\text{butlast } (\text{bin-to-bl } n \ w) = \text{bin-to-bl } (n - 1) (\text{bin-rest } w)$ 
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

```

```

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

```

```

lemma butlast-rest-bl2bin-aux:
   $bl \sim [] \implies$ 
   $\text{bl-to-bin-aux } (\text{butlast } bl) \ w = \text{bin-rest } (\text{bl-to-bin-aux } bl \ w)$ 
by (induct bl arbitrary: w) auto

```

```

lemma butlast-rest-bl2bin:
   $\text{bl-to-bin } (\text{butlast } bl) = \text{bin-rest } (\text{bl-to-bin } bl)$ 
apply (unfold bl-to-bin-def)
apply (cases bl)
apply (auto simp add: butlast-rest-bl2bin-aux)
done

```

```

lemma trunc-bl2bin-aux [rule-format]:

```

```

ALL w. bintrunc m (bl-to-bin-aux bl w) =
  bl-to-bin-aux (drop (length bl - m) bl) (bintrunc (m - length bl) w)
apply (induct-tac bl)
apply clarsimp
apply clarsimp
apply safe
apply (case-tac m - size list)
  apply (simp add : diff-is-0-eq [THEN iffD1, THEN Suc-diff-le])
apply simp
apply (rule-tac f = %nat. bl-to-bin-aux list (Int.Bit1 (bintrunc nat w))
        in arg-cong)
apply simp
apply (case-tac m - size list)
  apply (simp add: diff-is-0-eq [THEN iffD1, THEN Suc-diff-le])
apply simp
apply (rule-tac f = %nat. bl-to-bin-aux list (Int.Bit0 (bintrunc nat w))
        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)

```

lemmas *trunc-bl2bin-len* [simp] =

```

trunc-bl2bin [of length bl bl, simplified, standard]

```

lemma *bl2bin-drop*:

```

bl-to-bin (drop k bl) = bintrunc (length bl - k) (bl-to-bin bl)
apply (rule trans)
prefer 2
apply (rule trunc-bl2bin [symmetric])
apply (cases k <= length bl)
apply auto
done

```

lemma *nth-rest-power-bin* [rule-format] :

```

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

```

lemma *take-rest-power-bin*:

```

m <= n ==> take m (bin-to-bl n w) = bin-to-bl m ((bin-rest ^^ (n - m)) w)
apply (rule nth-equalityI)
apply simp
apply (clarsimp simp add: nth-bin-to-bl nth-rest-power-bin)
done

```

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

lemma *last-bin-last'*:
 $size\ xs > 0 \implies last\ xs = (bin-last\ (bl-to-bin-aux\ xs\ w) = 1)$
by (*induct xs arbitrary: w*) *auto*

lemma *last-bin-last*:
 $size\ xs > 0 \implies last\ xs = (bin-last\ (bl-to-bin\ xs) = 1)$
unfolding *bl-to-bin-def* **by** (*erule last-bin-last'*)

lemma *bin-last-last*:
 $bin-last\ w = (if\ last\ (bin-to-bl\ (Suc\ n)\ w)\ then\ 1\ else\ 0)$
apply (*unfold bin-to-bl-def*)
apply *simp*
apply (*auto simp add: bin-to-bl-aux-alt*)
done

lemma *bl-xor-aux-bin* [*rule-format*] : *ALL v w bs cs.*
 $map2\ (\%x\ y.\ x \sim = y)\ (bin-to-bl-aux\ n\ v\ bs)\ (bin-to-bl-aux\ n\ w\ cs) =$
 $bin-to-bl-aux\ n\ (v\ XOR\ w)\ (map2\ (\%x\ y.\ x \sim = y)\ bs\ cs)$
apply (*induct-tac n*)
apply *safe*
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* [*rule-format*] : *ALL v w bs cs.*
 $map2\ (op\ |)\ (bin-to-bl-aux\ n\ v\ bs)\ (bin-to-bl-aux\ n\ w\ cs) =$
 $bin-to-bl-aux\ n\ (v\ OR\ w)\ (map2\ (op\ |)\ bs\ cs)$
apply (*induct-tac n*)
apply *safe*
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* [*rule-format*] : *ALL v w bs cs.*
 $map2\ (op\ \&)\ (bin-to-bl-aux\ n\ v\ bs)\ (bin-to-bl-aux\ n\ w\ cs) =$

```

  bin-to-bl-aux n (v AND w) (map2 (op & ) bs cs)
apply (induct-tac n)
apply safe
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-not-aux-bin [rule-format] :
  ALL w cs. map Not (bin-to-bl-aux n w cs) =
    bin-to-bl-aux n (NOT w) (map Not cs)
apply (induct n)
apply clarsimp
apply clarsimp
apply (case-tac w rule: bin-exhaust)
apply (case-tac b)
apply auto
done

```

```

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

```

```

lemmas bl-and-bin = bl-and-aux-bin [where bs=[] and cs=[],
  unfolded map2-Nil, folded bin-to-bl-def]

```

```

lemmas bl-or-bin = bl-or-aux-bin [where bs=[] and cs=[],
  unfolded map2-Nil, folded bin-to-bl-def]

```

```

lemmas bl-xor-bin = bl-xor-aux-bin [where bs=[] and cs=[],
  unfolded map2-Nil, folded bin-to-bl-def]

```

```

lemma drop-bin2bl-aux [rule-format] :
  ALL m bin bs. drop m (bin-to-bl-aux n bin bs) =
    bin-to-bl-aux (n - m) bin (drop (m - n) bs)
apply (induct n, 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 [rule-format] :
  ALL w bs. take m (bin-to-bl-aux m w bs) = bin-to-bl m w
apply (induct m, 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 [rule-format] :
  ALL w bs. take m (bin-to-bl-aux (m + n) w bs) =
    take m (bin-to-bl (m + n) w)
apply (induct n)
apply clarify
apply (simp-all (no-asm) add: bin-to-bl-def take-bin2bl-lem1)
apply simp
done

lemma bin-split-take [rule-format] :
  ALL b c. bin-split n c = (a, b) -->
    bin-to-bl m a = take m (bin-to-bl (m + n) c)
apply (induct n)
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 ==> bin-split n c = (a, b) ==>
    bin-to-bl m a = take m (bin-to-bl k c)
by (auto elim: bin-split-take)

lemma nth-takefill [rule-format] : ALL m l. m < n -->
  takefill fill n l ! m = (if m < length l then l ! m else fill)
apply (induct n, clarsimp)
apply clarsimp
apply (case-tac m)
apply (simp split: list.split)
apply clarsimp
apply (erule allE)+
apply (erule (1) impE)
apply (simp split: list.split)
done

lemma takefill-alt [rule-format] :
  ALL l. takefill fill n l = take n l @ replicate (n - length l) fill
by (induct n) (auto split: list.split)

```

lemma *takefill-replicate* [*simp*]:
 $takefill\ fill\ n\ (replicate\ m\ fill) = replicate\ n\ fill$
by (*simp add : takefill-alt replicate-add [symmetric]*)

lemma *takefill-le'* [*rule-format*] :
 $ALL\ l\ n.\ n = m + k \longrightarrow takefill\ x\ m\ (takefill\ x\ n\ l) = takefill\ x\ m\ l$
by (*induct m*) (*auto split: list.split*)

lemma *length-takefill* [*simp*]: $length\ (takefill\ fill\ n\ l) = n$
by (*simp add : takefill-alt*)

lemma *take-takefill'*:
 $!!w\ n.\ n = k + m \implies take\ k\ (takefill\ fill\ n\ w) = takefill\ fill\ k\ w$
by (*induct k*) (*auto split add : list.split*)

lemma *drop-takefill*:
 $!!w.\ drop\ k\ (takefill\ fill\ (m + k)\ w) = takefill\ fill\ m\ (drop\ k\ w)$
by (*induct k*) (*auto split add : list.split*)

lemma *takefill-le* [*simp*]:
 $m \leq n \implies takefill\ x\ m\ (takefill\ x\ n\ l) = takefill\ x\ m\ l$
by (*auto simp: le-iff-add takefill-le'*)

lemma *take-takefill* [*simp*]:
 $m \leq n \implies take\ m\ (takefill\ fill\ n\ w) = takefill\ fill\ m\ w$
by (*auto simp: le-iff-add take-takefill'*)

lemma *takefill-append*:
 $takefill\ fill\ (m + length\ xs)\ (xs\ @\ w) = xs\ @\ (takefill\ fill\ m\ w)$
by (*induct xs*) *auto*

lemma *takefill-same'*:
 $l = length\ xs \implies takefill\ fill\ l\ xs = xs$
by *clarify* (*induct xs, auto*)

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

lemma *takefill-bintrunc*:
 $takefill\ False\ n\ bl = rev\ (bin-to-bl\ n\ (bl-to-bin\ (rev\ bl)))$
apply (*rule nth-equalityI*)
apply *simp*
apply (*clarsimp simp: nth-takefill nth-rev nth-bin-to-bl bin-nth-of-bl*)
done

lemma *bl-bin-bl-rtf*:
 $bin-to-bl\ n\ (bl-to-bin\ bl) = rev\ (takefill\ False\ n\ (rev\ bl))$
by (*simp add : takefill-bintrunc*)

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

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*:
 $0 < n \implies \text{takefill } \text{fill } (\text{Suc } (n - 1)) \ w = \text{takefill } \text{fill } n \ w$
by *auto*

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

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

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

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

lemma *bl-to-bin-aux-cat*:
 $!!nv \ v. \text{bl-to-bin-aux } bs \ (\text{bin-cat } w \ nv \ v) =$
 $\text{bin-cat } w \ (nv + \text{length } bs) \ (\text{bl-to-bin-aux } bs \ v)$
apply (*induct bs*)
apply *simp*
apply (*simp add: bin-cat-Suc-Bit* [*symmetric*] *del: bin-cat.simps*)
done

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

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

lemmas *bin-to-bl-cat* =

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

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

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

lemmas *bl-to-bin-app-cat* = *bl-to-bin-aux-app-cat*
[where $w = Int.Pl$ s, folded *bl-to-bin-def*]

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

lemma *bl-to-bin-app-cat-alt*:
bin-cat (*bl-to-bin* cs) n w = *bl-to-bin* (cs @ *bin-to-bl* n w)
by (*simp add* : *bl-to-bin-app-cat*)

lemma *mask-lem*: (*bl-to-bin* ($True$ # *replicate* n $False$)) =
Int.succ (*bl-to-bin* (*replicate* n $True$))
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*
done

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

lemma *nth-bl-of-nth* [*simp*]:
 $m < n \implies \text{rev } (\text{bl-of-nth } n \ f) ! m = f \ m$
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*:
($!!k. k < n \implies f \ k = g \ k$) $\implies \text{bl-of-nth } n \ f = \text{bl-of-nth } n \ g$
by (*induct n*) *auto*

lemma *bl-of-nth-nth-le* [*rule-format*] : *ALL* xs .
 $\text{length } xs \geq n \implies \text{bl-of-nth } n \ (\text{nth } (\text{rev } xs)) = \text{drop } (\text{length } xs - n) \ xs$
apply (*induct n, 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

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

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. length (rbl-add bl cl) = length bl
  by (induct bl) (auto simp: Let-def size-rbl-succ)

lemma size-rbl-mult:
  !!cl. length (rbl-mult bl cl) = 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 rbl-pred:
  !!bin. rbl-pred (rev (bin-to-bl n bin)) = rev (bin-to-bl n (Int.pred bin))
  apply (induct n, 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 rbl-succ:
  !!bin. rbl-succ (rev (bin-to-bl n bin)) = rev (bin-to-bl n (Int.succ bin))
  apply (induct n, 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 *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 succ-def 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 >= n ==> rbl-add (rev (bin-to-bl n bina)) (rev (bin-to-bl m binb)) =
  rev (bin-to-bl n (bina + binb))
apply (rule box-equals [OF - rbl-add-take2 rbl-add])
apply (rule-tac f = rbl-add (rev (bin-to-bl n bina)) in arg-cong)
apply (rule rev-swap [THEN iffD1])
apply (simp add: rev-take drop-bin2bl)
apply simp
done

```

lemma *rbl-mult-app2*:

```

!!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 rbbl-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 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*: $xs = ys ==> xs = ys \ \& \ \text{length } xs = \text{length } ys$
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*:

```

(If p x y = (~ If p u v)) = (if p then x = (~u) else y = (~v))

```

by *auto*

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

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

lemma *if-bool-simps*:
 $If\ p\ True\ y = (p\ |\ y) \ \&\ If\ p\ False\ y = (\sim p \ \&\ y) \ \&$
 $If\ p\ y\ True = (p\ \dashrightarrow y) \ \&\ If\ p\ y\ False = (p \ \&\ y)$
by *auto*

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

lemmas *segr* = *eq-reflection* [**where** $x = size\ w$, *standard*]

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

12.3 Repeated splitting or concatenation

lemma *sclm*:
 $size\ (concat\ (map\ (bin-to-bl\ n)\ xs)) = length\ xs * n$
by (*induct xs*) *auto*

lemma *bin-cat-foldl-lem* [*rule-format*] :
 $ALL\ x.\ foldl\ (\%u.\ bin-cat\ u\ n)\ x\ xs =$
 $bin-cat\ x\ (size\ xs * n)\ (foldl\ (\%u.\ bin-cat\ u\ n)\ y\ xs)$
apply (*induct xs*)
apply *simp*
apply *clarify*
apply (*simp* (*no-asm*))
apply (*frule asm-rl*)
apply (*drule spec*)
apply (*erule trans*)
apply (*drule-tac x = bin-cat y n a in spec*)
apply (*simp add : bin-cat-assoc-sym min-max.inf-absorb2*)
done

lemma *bin-rcat-bl*:
 $(bin-rcat\ n\ wl) = bl-to-bin\ (concat\ (map\ (bin-to-bl\ n)\ wl))$
apply (*unfold bin-rcat-def*)
apply (*rule sym*)
apply (*induct wl*)
apply (*auto simp add : bl-to-bin-append*)
apply (*simp add : bl-to-bin-aux-alt sclm*)

```

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, standard]
lemmas th-if-simp2 = split-if [where P = op = l,
  THEN iffD1, THEN conjunct2, THEN mp, standard]

lemmas rsplit-aux-simp1s = rsplit-aux-simps [THEN th-if-simp1]

lemmas rsplit-aux-simp2ls = rsplit-aux-simps [THEN th-if-simp2]

lemmas bin-rsplit-aux-simp2s [simp] = rsplit-aux-simp2ls [unfolded Let-def]
lemmas rbscl = bin-rsplit-aux-simp2s (2)

lemmas rsplit-aux-0-simps [simp] =
  rsplit-aux-simp1s [OF disjI1] rsplit-aux-simp1s [OF disjI2]

lemma bin-rsplit-aux-append:
  bin-rsplit-aux n 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 ==> bin-split (Suc (n - 1)) w = bin-split n w
  by auto

```

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

lemma *bin-split-pred-simp* [*simp*]:
 $(0::nat) < \text{number-of bin} \implies$
 $\text{bin-split } (\text{number-of bin}) w =$
 $(\text{let } (w1, w2) = \text{bin-split } (\text{number-of } (\text{Int.pred bin})) (\text{bin-rest } w)$
 $\text{in } (w1, w2 \text{ BIT bin-last } w))$
by (*simp only: nobm1 bin-split-minus-simp*)

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

lemma *bin-rsplit-aux-simp-alt*:
 $\text{bin-rsplit-aux } n m c bs =$
 $(\text{if } m = 0 \vee n = 0$
 $\text{then } bs$
 $\text{else let } (a, b) = \text{bin-split } n c \text{ in } \text{bin-rsplit } n (m - n, a) @ b \# bs)$
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, standard]

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

lemma *bin-rsplit-size-sign'* [*rule-format*] :
 $n > 0 \implies (\text{ALL } nw w. \text{rev } sw = \text{bin-rsplit } n (nw, w) \dashrightarrow$
 $(\text{ALL } v: \text{set } sw. \text{bintrunc } n v = v))$
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 impE, rule exI, erule exI*)
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]],
standard]

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

```

n > 0 ==> m < n ==> (ALL w k nw. rev sw = bin-rsplit n (nw, w) -->
  k < size sw --> bin-nth (sw ! k) m = bin-nth w (k * n + m))
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* [rule-format] :
 $n \neq 0 \dashrightarrow \text{length } (\text{bin-rsplit-aux } n \text{ } nw \text{ } w \text{ } cs) =$
 $(nw + n - 1) \text{ div } n + \text{length } cs$
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: $v \text{ } bs$ 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)
begin

```

see *Examples/WordExamples.thy* for examples

13.1 Type definition

```

typedef (open word) 'a word = {(0::int) ..< 2^len-of TYPE('a::len0)}
morphisms uint Abs-word by auto

```

definition *word-of-int* :: int ⇒ 'a::len0 word **where**
 — representation of words using unsigned or signed bins, only difference in these
 is the type class
word-of-int w = Abs-word (bintrunc (len-of TYPE ('a)) w)

```

lemma uint-word-of-int [code]: uint (word-of-int w :: 'a::len0 word) = w mod 2 ^
  len-of TYPE('a)
by (auto simp add: word-of-int-def bintrunc-mod2p intro: Abs-word-inverse)

```

```

code-datatype word-of-int

```

```

notation fcomp (infixl o> 60)
notation scomp (infixl o→ 60)

```

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

definition

random-word *i* = *Random.range* (*max i* ($2 \wedge \text{len-of TYPE}('a)$)) $\circ \rightarrow$ (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)

13.2 Type conversions and casting

definition *sint* :: 'a :: *len word* => *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* => *nat* **where**

unat w = *nat* (*uint w*)

definition *uints* :: *nat* => *int set* **where**

— the sets of integers representing the words

uints n = *range* (*bintrunc n*)

definition *sints* :: *nat* => *int set* **where**

sints n = *range* (*sbintrunc* (*n* - 1))

definition *unats* :: *nat* => *nat set* **where**

unats n = {*i*. *i* < $2 \wedge n$ }

definition *norm-sint* :: *nat* => *int* => *int* **where**

norm-sint n w = (*w* + $2 \wedge (n - 1)$) *mod* $2 \wedge n - 2 \wedge (n - 1)$

definition *scast* :: 'a :: *len word* => 'b :: *len word* **where**

— cast a word to a different length

scast w = *word-of-int* (*sint w*)

definition *ucast* :: 'a :: *len0 word* => 'b :: *len0 word* **where**

ucast w = *word-of-int* (*uint w*)

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

begin

definition

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

instance ..

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)

syntax

of-int :: int => 'a

translations

case x of CONST of-int y => b == CONST word-int-case (%y. b) x

13.3 Arithmetic operations

instantiation *word* :: (len0) {number, uminus, minus, plus, one, zero, times, Divides.div, ord, bit}

begin

definition

word-0-wi: 0 = word-of-int 0

definition

word-1-wi: 1 = word-of-int 1

definition

word-add-def: a + b = word-of-int (uint a + uint b)

definition

word-sub-wi: $a - b = \text{word-of-int } (\text{uint } a - \text{uint } b)$

definition

word-minus-def: $- a = \text{word-of-int } (- \text{uint } a)$

definition

word-mult-def: $a * b = \text{word-of-int } (\text{uint } a * \text{uint } b)$

definition

word-div-def: $a \text{ div } b = \text{word-of-int } (\text{uint } a \text{ div } \text{uint } b)$

definition

word-mod-def: $a \text{ mod } b = \text{word-of-int } (\text{uint } a \text{ mod } \text{uint } b)$

definition

word-number-of-def: $\text{number-of } w = \text{word-of-int } w$

definition

word-le-def: $a \leq b \longleftrightarrow \text{uint } a \leq \text{uint } b$

definition

word-less-def: $x < y \longleftrightarrow x \leq y \wedge x \neq (y :: 'a \text{ word})$

definition

word-and-def:
 $(a :: 'a \text{ word}) \text{ AND } b = \text{word-of-int } (\text{uint } a \text{ AND } \text{uint } b)$

definition

word-or-def:
 $(a :: 'a \text{ word}) \text{ OR } b = \text{word-of-int } (\text{uint } a \text{ OR } \text{uint } b)$

definition

word-xor-def:
 $(a :: 'a \text{ word}) \text{ XOR } b = \text{word-of-int } (\text{uint } a \text{ XOR } \text{uint } b)$

definition

word-not-def:
 $\text{NOT } (a :: 'a \text{ word}) = \text{word-of-int } (\text{NOT } (\text{uint } a))$

instance ..

end

definition

word-succ :: $'a :: \text{len0 word} \Rightarrow 'a \text{ word}$

where

word-succ $a = \text{word-of-int } (\text{Int.succ } (\text{uint } a))$

definition

word-pred :: 'a :: len0 word => 'a word

where

word-pred a = *word-of-int* (*Int.pred* (*uint* a))

definition *udvd* :: 'a::len word => 'a::len word => bool (**infixl** *udvd* 50) **where**

a *udvd* b = (*EX* n >= 0. *uint* b = n * *uint* a)

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

a <=s b = (*sint* a <= *sint* b)

definition *word-sless* :: 'a :: len word => 'a word => bool ((-/ <s -) [50, 51] 50)**where**

(x <s y) = (x <=s y & x ~ = y)

13.4 Bit-wise operations

instantiation *word* :: (len0) bits

begin**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

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

definition

word-msb-def:
 $msb\ a \longleftrightarrow bin\text{-}sign\ (sint\ a) = Int.Min$

instance ..

end

lemma [*code*]:

$msb\ a \longleftrightarrow bin\text{-}sign\ (sint\ a) = (-\ 1 :: int)$
by (*simp only*: *word-msb-def Min-def*)

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

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

13.5 Shift operations

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

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

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

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

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

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

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

13.6 Rotation

definition *rotater1* :: 'a *list* => 'a *list* **where**

rotater1 *ys* =
 (case *ys* of [] => [] | *x* # *xs* => last *ys* # butlast *ys*)

definition *rotater* :: nat => 'a list => 'a list **where**
rotater *n* = *rotater1* ^^ *n*

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

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

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

13.7 Split and cat operations

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

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

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

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

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

primrec *of-bool* :: bool => 'a::len word **where**
of-bool False = 0
 | *of-bool* True = 1

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

lemmas *word-size-gt-0* [iff] =
xtr1 [OF *word-size len-gt-0*, *standard*]
lemmas *lens-gt-0* = *word-size-gt-0 len-gt-0*

lemmas *lens-not-0* [iff] = *lens-gt-0* [THEN *gr-implies-not0*, *standard*]

lemma *uints-num*: $uints\ n = \{i. 0 \leq i \wedge i < 2 \wedge n\}$
by (*simp add: uints-def range-bintrunc*)

lemma *sints-num*: $sints\ n = \{i. -(2 \wedge (n - 1)) \leq i \wedge i < 2 \wedge (n - 1)\}$
by (*simp add: sints-def range-sbintrunc*)

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

lemma *mod-in-reps*: $m > 0 \implies y \bmod m : \{0::int ..< m\}$
unfolding *atLeastLessThan-alt* **by** *auto*

lemma
uint-0:0 <= *uint x* **and**
uint-lt: $uint\ (x::'a::len0\ word) < 2 \wedge len-of\ TYPE('a)$
by (*auto simp: uint [simplified]*)

lemma *uint-mod-same*:
 $uint\ x \bmod 2 \wedge len-of\ TYPE('a) = uint\ (x::'a::len0\ word)$
by (*simp add: int-mod-eq uint-lt uint-0*)

lemma *td-ext-uint*:
 $td-ext\ (uint :: 'a\ word \implies int)\ word-of-int\ (uints\ (len-of\ TYPE('a::len0)))$
 $(\%w::int. w \bmod 2 \wedge len-of\ TYPE('a))$
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

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

interpretation *word-uint*:
 $td-ext\ uint::'a::len0\ word \implies int$
word-of-int
 $uints\ (len-of\ TYPE('a::len0))$
 $\lambda w. w \bmod 2 \wedge len-of\ TYPE('a::len0)$
by (*rule td-ext-uint*)

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

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

interpretation *word-ubin*:
 $td-ext\ uint::'a::len0\ word \implies int$
word-of-int

```

      uints (len-of TYPE('a::len0))
      bintrunc (len-of TYPE('a::len0))
    by (rule td-ext-ubin)

```

```

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':
  n >= size w  $\implies$  bintrunc n (uint w) = uint w
apply (unfold word-size)
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':
  wb = word-of-int bin  $\implies$  n >= size wb  $\implies$ 
    word-of-int (bintrunc n bin) = wb
unfolding word-size
by (clarsimp simp add: word-ubin.norm-eq-iff [symmetric] min-max.inf-absorb1)

```

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

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

```

lemma td-ext-sbin:
  td-ext (sint :: 'a word => int) word-of-int (sints (len-of TYPE('a::len)))
    (sbintrunc (len-of TYPE('a) - 1))
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, standard*]

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

lemma *word-number-of-alt* [*code-unfold-post*]:

```

number-of b = word-of-int (number-of b)
  by (simp add: number-of-eq word-number-of-def)

```

lemma *word-no-wi*: *number-of* = *word-of-int*

```

  by (auto simp: word-number-of-def)

```

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 number-of w, standard*]

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

lemma *uint-bintrunc*: *uint* (*number-of bin* :: 'a word) =

```

  number-of (bintrunc (len-of TYPE ('a :: len0)) bin)
  unfolding word-number-of-def number-of-eq
  by (auto intro: word-ubin.eq-norm)

```

lemma *sint-sbintrunc*: *sint* (*number-of bin* :: 'a word) =

```

  number-of (sbintrunc (len-of TYPE ('a :: len) - 1) bin)
  unfolding word-number-of-def number-of-eq
  by (subst word-sbin.eq-norm) simp

```

lemma *unat-bintrunc*:

```

unat (number-of bin :: 'a :: len0 word) =
  number-of (bintrunc (len-of TYPE('a)) bin)
  unfolding unat-def nat-number-of-def

```

by (*simp only: uint-bintrunc*)

declare

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

lemma *size-0-eq*: $\text{size } (w :: 'a :: \text{len0 word}) = 0 \implies v = w$

apply (*unfold word-size*)
apply (*rule word-uint.Rep-eqD*)
apply (*rule box-equals*)
defer
apply (*rule word-ubin.norm-Rep*) +
apply *simp*
done

lemmas *uint-lem* = *word-uint.Rep* [*unfolded uints-num mem-Collect-eq*]

lemmas *sint-lem* = *word-sint.Rep* [*unfolded sints-num mem-Collect-eq*]

lemmas *uint-ge-0* [*iff*] = *uint-lem* [*THEN conjunct1, standard*]

lemmas *uint-lt2p* [*iff*] = *uint-lem* [*THEN conjunct2, standard*]

lemmas *sint-ge* = *sint-lem* [*THEN conjunct1, standard*]

lemmas *sint-lt* = *sint-lem* [*THEN conjunct2, standard*]

lemma *sign-uint-Pls* [*simp*]:

bin-sign (*uint x*) = *Int.Pl*
by (*simp add: sign-Pls-ge-0 number-of-eq*)

lemmas *uint-m2p-neg* = *iffD2* [*OF diff-less-0-iff-less uint-lt2p, standard*]

lemmas *uint-m2p-not-non-neg* =
iffD2 [*OF linorder-not-le uint-m2p-neg, standard*]

lemma *lt2p-lem*:

len-of TYPE('a) $\leq n \implies \text{uint } (w :: 'a :: \text{len0 word}) < 2 \wedge n$
by (*rule xtr8* [*OF - uint-lt2p*]) *simp*

lemmas *uint-le-0-iff* [*simp*] =

uint-ge-0 [*THEN leD, THEN linorder-antisym-conv1, standard*]

lemma *uint-nat*: $\text{uint } w = \text{int } (\text{unat } w)$

unfolding *unat-def* **by** *auto*

lemma *uint-number-of*:

$\text{uint } (\text{number-of } b :: 'a :: \text{len0 word}) = \text{number-of } b \bmod 2 \wedge \text{len-of TYPE('a)}$

unfolding *word-number-of-alt*

by (*simp only: int-word-uint*)

lemma *unat-number-of*:

bin-sign *b* = *Int.Pl* \implies

```

unat (number-of b::'a::len0 word) = number-of b mod  $2 \wedge \text{len-of TYPE } ('a)$ 
apply (unfold unat-def)
apply (clarsimp simp only: uint-number-of)
apply (rule nat-mod-distrib [THEN trans])
  apply (erule sign-Pls-ge-0 [THEN iffD1])
  apply (simp-all add: nat-power-eq)
done

```

```

lemma sint-number-of: sint (number-of b::'a::len word) = (number-of b +
 $2 \wedge (\text{len-of TYPE } ('a) - 1) \text{ mod } 2 \wedge \text{len-of TYPE } ('a) -$ 
 $2 \wedge (\text{len-of TYPE } ('a) - 1)$ )
unfolding word-number-of-alt by (rule int-word-sint)

```

```

lemma word-of-int-bin [simp] :
(word-of-int (number-of bin) :: 'a::len0 word) = (number-of bin)
unfolding word-number-of-alt by auto

```

```

lemma word-int-case-wi:
word-int-case f (word-of-int i :: 'b word) =
  f (i mod  $2 \wedge \text{len-of TYPE } ('b::\text{len0})$ )
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) &
 $0 \leq i \text{ \& } i < 2 \wedge \text{len-of TYPE } ('b)$   $\longrightarrow P (f i)$ )
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) =
  ( $\sim (EX n. x = (\text{word-of-int } n :: 'b::\text{len0 } \text{word}) \ \&$ 
 $0 \leq n \ \& \ n < 2 \wedge \text{len-of TYPE } ('b::\text{len0}) \ \& \ \sim P (f n))$ )
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, standard]
lemmas sint-range' = word-sint.Rep [unfolded One-nat-def
  sints-num mem-Collect-eq, standard]

```

```

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')

```

```

lemmas sint-above-size = sint-range-size

```

[*THEN* *conjunct2*, *THEN* [2] *xtr8*, *folded One-nat-def*, *standard*]

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

lemma *test-bit-eq-iff*: (*test-bit (u::'a::len0 word) = test-bit v*) = (*u = v*)
unfolding *word-test-bit-def* **by** (*simp add: bin-nth-eq-iff*)

lemma *test-bit-size* [*rule-format*] : (*w::'a::len0 word*) !! *n* \longrightarrow *n* < *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-eqI* [*rule-format*] :
fixes *u :: 'a::len0 word*
shows (*ALL n. n < size u \longrightarrow u !! n = v !! n*) \implies *u = v*
apply (*rule test-bit-eq-iff [THEN iffD1]*)
apply (*rule ext*)
apply (*erule allE*)
apply (*erule impCE*)
prefer 2
apply *assumption*
apply (*auto dest!: test-bit-size simp add: word-size*)
done

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

lemma *test-bit-bin'*: *w !! n = (n < size w & bin-nth (uint w) n)*
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) n \longrightarrow n < size w*
apply (*unfold word-size*)
apply (*rule impI*)
apply (*rule nth-bintr [THEN iffD1, THEN conjunct1]*)
apply (*subst word-ubin.norm-Rep*)
apply *assumption*
done

lemma *bin-nth-sint'*:
n \geq size w \longrightarrow bin-nth (sint w) n = bin-nth (sint w) (size w - 1)
apply (*rule impI*)
apply (*subst word-sbin.norm-Rep [symmetric]*)
apply (*simp add : nth-sbintr word-size*)
apply *auto*

done

lemmas $\text{bin-nth-uint-imp} = \text{bin-nth-uint-imp}'$ [rule-format, unfolded word-size]

lemmas $\text{bin-nth-sint} = \text{bin-nth-sint}'$ [rule-format, unfolded word-size]

lemma *td-bl*:

type-definition $(\text{to-bl} :: 'a::\text{len0 word} \Rightarrow \text{bool list})$

of-bl

$\{\text{bl. length bl} = \text{len-of TYPE('a)}\}$

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 $\text{to-bl} :: 'a::\text{len0 word} \Rightarrow \text{bool list}$

of-bl

$\{\text{bl. length bl} = \text{len-of TYPE('a::len0)}\}$

by (*rule td-bl*)

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

unfolding *word-size* **by** *auto*

lemma *to-bl-use-of-bl*:

$(\text{to-bl } w = \text{bl}) = (w = \text{of-bl bl} \wedge \text{length bl} = \text{length } (\text{to-bl } w))$

by (*fastforce elim!: word-bl.Abs-inverse [simplified]*)

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

unfolding *word-reverse-def* **by** (*simp add: word-bl.Abs-inverse*)

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

unfolding *word-reverse-def* **by** (*simp add: word-bl.Abs-inverse*)

lemma *word-rev-gal*: $\text{word-reverse } w = u \Longrightarrow \text{word-reverse } u = w$

by *auto*

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

lemmas *length-bl-gt-0* [*iff*] = *xtr1* [OF *word-bl.Rep' len-gt-0*, *standard*]

lemmas *bl-not-Nil* [*iff*] =

length-bl-gt-0 [THEN *length-greater-0-conv* [THEN *iffD1*], *standard*]

lemmas *length-bl-neq-0* [*iff*] = *length-bl-gt-0* [THEN *gr-implies-not0*]

lemma *hd-bl-sign-sint*: $\text{hd } (\text{to-bl } w) = (\text{bin-sign } (\text{sint } w) = \text{Int.Min})$

apply (*unfold to-bl-def sint-uint*)

apply (*rule trans* [OF - *bl-sbin-sign*])

apply *simp*
done

lemma *of-bl-drop'*:
 $lend = length\ bl - len-of\ TYPE\ ('a :: len0) \implies$
 $of-bl\ (drop\ lend\ bl) = (of-bl\ bl :: 'a\ word)$
apply (*unfold of-bl-def*)
apply (*clarsimp simp add : trunc-bl2bin [symmetric]*)
done

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

lemma *test-bit-of-bl*:
 $(of-bl\ bl :: 'a :: len0\ word) !!\ n = (rev\ bl !\ n \wedge\ n < len-of\ TYPE('a) \wedge\ n < length$
 $bl)$
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*:
 $(number-of\ bin :: 'a :: len0\ word) = of-bl\ (bin-to-bl\ (len-of\ TYPE('a))\ bin)$
unfolding *word-size of-bl-no by (simp add : word-number-of-def)*

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

lemma *to-bl-bin*: $bl-to-bin\ (to-bl\ w) = uint\ w$
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)*

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

lemma *to-bl-to-bin [simp]* : $bl-to-bin\ (to-bl\ w) = uint\ w$
unfolding *uint-bl by (simp add : word-size)*

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

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

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

lemma *uints-unats*: $uints\ n = int\ 'unats\ n$
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
  [symmetric, folded word-number-of-def, standard]
lemmas sbintr-num = word-sbin.norm-eq-iff
  [symmetric, folded word-number-of-def, standard]

```

```

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

```

```

lemma num-of-bintr':
  bintrunc (len-of TYPE('a :: len0)) a = b  $\implies$ 
    number-of a = (number-of b :: 'a word)
  apply safe
  apply (rule-tac num-of-bintr [symmetric])
  done

```

```

lemma num-of-sbintr':
  sbintrunc (len-of TYPE('a :: len) - 1) a = b  $\implies$ 
    number-of a = (number-of b :: 'a word)
  apply safe
  apply (rule-tac num-of-sbintr [symmetric])
  done

```

```

lemmas num-abs-bintr = sym [THEN trans,
  OF num-of-bintr word-number-of-def, standard]
lemmas num-abs-sbintr = sym [THEN trans,
  OF num-of-sbintr word-number-of-def, standard]

```

```

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 (number-of w :: 'a :: len0 word) =
number-of (bintrunc (len-of TYPE('a)) w)
unfolding *ucast-def* **by** *simp*

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

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

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

lemma *down-cast-same'*: *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'*:
r = to-bl (of-bl bl) \implies r = rev (takefill False (length r) (rev bl))
unfolding *of-bl-def uint-bl*
by (*clarsimp simp add: bl-bin-bl-rtf word-ubin.eq-norm word-size*)

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

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

lemma *to-bl-ucast*:

```

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

```

lemma *ucast-up-app'*:

```

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'*:

```

uc = ucast  $\implies$  source-size uc = target-size uc + n  $\implies$ 
  to-bl (uc w) = drop n (to-bl w)
by (auto simp add : source-size target-size to-bl-ucast)

```

lemma *scast-down-drop'*:

```

sc = scast  $\implies$  source-size sc = target-size sc + n  $\implies$ 
  to-bl (sc w) = drop n (to-bl w)
apply (subgoal-tac sc = ucast)
apply safe
apply simp
apply (erule refl [THEN ucast-down-drop'])
apply (rule refl [THEN down-cast-same', symmetric])
apply (simp add : source-size target-size is-down)
done

```

lemma *sint-up-scast'*:

```

sc = scast  $\implies$  is-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'*:

```

uc = ucast  $\implies$  is-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

```

```

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

```

```

lemma ucast-up-ucast': uc = ucast  $\implies$  is-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': 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':
  w = of-bl bl  $\implies$  size bl  $\leq$  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 = refl [THEN ucast-up-ucast^]
lemmas scast-up-scast = refl [THEN scast-up-scast^]
lemmas ucast-of-bl-up = refl [THEN ucast-of-bl-up^]

```

```

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

```

```

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

```

```

lemma up-ucast-surj:
  is-up (ucast :: 'b::len0 word => 'a::len0 word)  $\implies$ 
  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) ==>

inj-on (ucast :: 'a word => 'b word) A

by (rule inj-on-inverseI, erule scast-down-scast-id)

lemma down-ucast-inj:

is-down (ucast :: 'b::len0 word => 'a::len0 word) ==>

inj-on (ucast :: 'a word => 'b word) A

by (rule inj-on-inverseI, erule ucast-down-ucast-id)

lemma of-bl-append-same: of-bl (X @ to-bl w) = w

by (rule word-bl.Rep-eqD) (simp add: word-rep-drop)

lemma ucast-down-no':

uc = *ucast* ==> *is-down uc* ==> *uc* (number-of bin) = number-of bin

apply (unfold word-number-of-def *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

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

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

unfolding of-bl-no **by** clarify (erule ucast-down-no)

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

lemmas slice-def' = slice-def [unfolded word-size]

lemmas test-bit-def' = word-test-bit-def [THEN fun-cong]

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

lemmas word-log-bin-defs = word-log-defs

Executable equality

instantiation word :: (len0) equal

begin

definition equal-word :: 'a word => 'a word => bool **where**

equal-word k l <=> HOL.equal (uint k) (uint l)

instance proof

qed (simp add: equal equal-word-def)

end

13.8 Word Arithmetic

lemma *word-less-alt*: $(a < b) = (\text{uint } a < \text{uint } b)$

unfolding *word-less-def word-le-def*

by (*auto simp del: word-uint.Rep-inject*
simp: word-uint.Rep-inject [symmetric])

lemma *signed-linorder*: *class.linorder word-sle word-sless*

proof

qed (*unfold word-sle-def word-sless-def, auto*)

interpretation *signed*: *linorder word-sle word-sless*

by (*rule signed-linorder*)

lemmas *word-arith-wis* =

word-add-def word-mult-def word-minus-def
word-succ-def word-pred-def word-0-wi word-1-wi

lemma *udvdI*:

$0 \leq n \implies \text{uint } b = n * \text{uint } a \implies a \text{ udvd } b$

by (*auto simp: udvd-def*)

lemmas *word-div-no [simp]* =

word-div-def [of number-of a number-of b, standard]

lemmas *word-mod-no [simp]* =

word-mod-def [of number-of a number-of b, standard]

lemmas *word-less-no [simp]* =

word-less-def [of number-of a number-of b, standard]

lemmas *word-le-no [simp]* =

word-le-def [of number-of a number-of b, standard]

lemmas *word-sless-no [simp]* =

word-sless-def [of number-of a number-of b, standard]

lemmas *word-sle-no [simp]* =

word-sle-def [of number-of a number-of b, standard]

lemmas *word-0-wi-Pls* = *word-0-wi [folded Pls-def]*

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

lemma *int-one-bin*: $(1 :: \text{int}) = (\text{Int.Pls BIT } 1)$

unfolding *Pls-def Bit-def* **by** *auto*

lemma *word-1-no*:

$(1 :: 'a :: \text{len0 word}) = \text{number-of } (\text{Int.Pls BIT } 1)$

unfolding *word-1-wi word-number-of-def int-one-bin* **by** *auto*

```

lemma word-m1-wi:  $-1 = \text{word-of-int } -1$ 
  by (rule word-number-of-alt)

lemma word-m1-wi-Min:  $-1 = \text{word-of-int Int.Min}$ 
  by (simp add: word-m1-wi number-of-eq)

lemma word-0-bl:  $\text{of-bl } [] = 0$ 
  unfolding word-0-wi of-bl-def by (simp add : Pls-def)

lemma word-1-bl:  $\text{of-bl } [True] = 1$ 
  unfolding word-1-wi of-bl-def
  by (simp add : bl-to-bin-def Bit-def Pls-def)

lemma uint-eq-0 [simp]:  $(\text{uint } 0 = 0)$ 
  unfolding word-0-wi
  by (simp add: word-ubin.eq-norm Pls-def [symmetric])

lemma of-bl-0 [simp]:  $\text{of-bl } (\text{replicate } n \text{ False}) = 0$ 
  by (simp add : word-0-wi of-bl-def bl-to-bin-rep-False Pls-def)

lemma to-bl-0:
   $\text{to-bl } (0 :: 'a :: \text{len0 word}) = \text{replicate } (\text{len-of TYPE('a)}) \text{ False}$ 
  unfolding uint-bl
  by (simp add : word-size bin-to-bl-Pls Pls-def [symmetric])

lemma uint-0-iff:  $(\text{uint } x = 0) = (x = 0)$ 
  by (auto intro!: word-uint.Rep-eqD)

lemma unat-0-iff:  $(\text{unat } x = 0) = (x = 0)$ 
  unfolding unat-def by (auto simp add : nat-eq-iff uint-0-iff)

lemma unat-0 [simp]:  $\text{unat } 0 = 0$ 
  unfolding unat-def by auto

lemma size-0-same':  $\text{size } w = 0 \implies w = (v :: 'a :: \text{len0 word})$ 
  apply (unfold word-size)
  apply (rule box-equals)
  defer
  apply (rule word-uint.Rep-inverse)+
  apply (rule word-ubin.norm-eq-iff [THEN iffD1])
  apply simp
  done

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

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

```

lemma *unat-gt-0*: $(0 < \text{unat } x) = (x \sim = 0)$
by (*auto simp: unat-0-iff [symmetric]*)

lemma *ucast-0 [simp]*: $\text{ucast } 0 = 0$
unfolding *ucast-def*
by *simp (simp add: word-0-wi)*

lemma *sint-0 [simp]*: $\text{sint } 0 = 0$
unfolding *sint-wint*
by (*simp add: Pls-def [symmetric]*)

lemma *scast-0 [simp]*: $\text{scast } 0 = 0$
apply (*unfold scast-def*)
apply *simp*
apply (*simp add: word-0-wi*)
done

lemma *sint-n1 [simp]*: $\text{sint } -1 = -1$
apply (*unfold word-m1-wi-Min*)
apply (*simp add: word-sbin.eq-norm*)
apply (*unfold Min-def number-of-eq*)
apply *simp*
done

lemma *scast-n1 [simp]*: $\text{scast } -1 = -1$
apply (*unfold scast-def sint-n1*)
apply (*unfold word-number-of-alt*)
apply (*rule refl*)
done

lemma *uint-1 [simp]*: $\text{uint } (1 :: 'a :: \text{len word}) = 1$
unfolding *word-1-wi*
by (*simp add: word-ubin.eq-norm int-one-bin bintrunc-minus-simps*)

lemma *unat-1 [simp]*: $\text{unat } (1 :: 'a :: \text{len word}) = 1$
by (*unfold unat-def uint-1 auto*)

lemma *ucast-1 [simp]*: $\text{ucast } (1 :: 'a :: \text{len word}) = 1$
unfolding *ucast-def word-1-wi*
by (*simp add: word-ubin.eq-norm int-one-bin bintrunc-minus-simps*)

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

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-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: $\text{word-succ } (\text{word-of-int } a) = \text{word-of-int } (\text{Int.succ } a)$ **and**
wi-hom-pred: $\text{word-pred } (\text{word-of-int } a) = \text{word-of-int } (\text{Int.pred } a)$
by (*auto simp*: *word-arith-wis arths*)

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

lemma *word-sub-def*: $a - b = a + - (b :: 'a :: \text{len0 word})$
unfolding *word-sub-wi diff-minus*
by (*simp only* : *word-int.Rep-inverse wi-hom-syms*)

lemmas *word-diff-minus* = *word-sub-def* [*standard*]

lemma *word-of-int-sub-hom*:
 $(\text{word-of-int } a) - \text{word-of-int } b = \text{word-of-int } (a - b)$
unfolding *word-sub-def diff-minus* **by** (*simp only* : *wi-homs*)

lemmas *new-word-of-int-homs* =
word-of-int-sub-hom wi-homs word-0-wi word-1-wi

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

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

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

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

lemmas *word-arith-alts* =
word-sub-wi [*unfolded succ-def pred-def, standard*]
word-arith-wis [*unfolded succ-def pred-def, standard*]

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

lemmas *word-pred-alt* = *word-arith-alt*s (6)

lemmas *word-0-alt* = *word-arith-alt*s (7)

lemmas *word-1-alt* = *word-arith-alt*s (8)

13.9 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 (*rule* *word-uint.Abs-cases* [of b],

rule *word-uint.Abs-cases* [of a],

simp *add*: *pred-def succ-def add-commute mult-commute*

*ring-distrib*s *new-word-of-int-homs*)+

lemmas *uint-cong* = *arg-cong* [**where** $f = \text{uint}$]

lemmas *uint-word-ariths* =

*word-arith-alt*s [*THEN* *trans* [*OF* *uint-cong int-word-uint*], *standard*]

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

lemmas *sint-word-ariths* = *uint-word-arith-bintrs*

[*THEN* *uint-sint* [*symmetric*, *THEN* *trans*],

unfolded uint-sint bintr-arith1s bintr-ariths

len-gt-0 [*THEN* *bin-sbin-eq-iff*'] *word-sbin.norm-Rep*, *standard*]

lemmas *uint-div-alt* = *word-div-def*

[*THEN* *trans* [*OF* *uint-cong int-word-uint*], *standard*]

lemmas *uint-mod-alt* = *word-mod-def*

[*THEN* *trans* [*OF* *uint-cong int-word-uint*], *standard*]

lemma *word-pred-0-n1*: *word-pred* $0 = \text{word-of-int } -1$

unfolding *word-pred-def number-of-eq*

by (*simp* *add* : *pred-def word-no-wi*)

lemma *word-pred-0-Min*: *word-pred* $0 = \text{word-of-int Int.Min}$

by (*simp* *add*: *word-pred-0-n1 number-of-eq*)

lemma *word-m1-Min*: $-1 = \text{word-of-int Int.Min}$

unfolding *Min-def* **by** (*simp* *only*: *word-of-int-hom-syms*)

lemma *succ-pred-no* [*simp*]:

word-succ (*number-of* bin) = *number-of* (*Int.succ* bin) &

word-pred (*number-of* bin) = *number-of* (*Int.pred* bin)

unfolding *word-number-of-def* **by** (*simp add : new-word-of-int-homs*)

lemma *word-sp-01* [*simp*] :

word-succ -1 = 0 & word-succ 0 = 1 & word-pred 0 = -1 & word-pred 1 = 0
by (*unfold word-0-no word-1-no*) *auto*

lemma *word-of-int-Ex*:

$\exists y. x = \text{word-of-int } y$
by (*rule-tac x=uint x in exI*) *simp*

lemma *word-arith-eqs*:

fixes *a :: 'a::len0 word*

fixes *b :: 'a::len0 word*

shows

word-add-0: $0 + a = a$ **and**

word-add-0-right: $a + 0 = a$ **and**

word-mult-1: $1 * a = a$ **and**

word-mult-1-right: $a * 1 = a$ **and**

word-add-commute: $a + b = b + a$ **and**

word-add-assoc: $a + b + c = a + (b + c)$ **and**

word-add-left-commute: $a + (b + c) = b + (a + c)$ **and**

word-mult-commute: $a * b = b * a$ **and**

word-mult-assoc: $a * b * c = a * (b * c)$ **and**

word-mult-left-commute: $a * (b * c) = b * (a * c)$ **and**

word-left-distrib: $(a + b) * c = a * c + b * c$ **and**

word-right-distrib: $a * (b + c) = a * b + a * c$ **and**

word-left-minus: $- a + a = 0$ **and**

word-diff-0-right: $a - 0 = a$ **and**

word-diff-self: $a - a = 0$

using *word-of-int-Ex* [*of a*]

word-of-int-Ex [*of b*]

word-of-int-Ex [*of c*]

by (*auto simp: word-of-int-hom-syms* [*symmetric*]

add-0-right add-commute add-assoc add-left-commute

mult-commute mult-assoc mult-left-commute

left-distrib right-distrib)

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

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

lemmas *word-plus-ac0 = word-add-0 word-add-0-right word-add-ac*

lemmas *word-times-ac1 = word-mult-1 word-mult-1-right word-mult-ac*

13.10 Order on fixed-length words

lemma *word-order-trans*: $x \leq y \implies y \leq z \implies x \leq (z :: 'a :: \text{len0 word})$

unfolding *word-le-def* **by** *auto*

lemma *word-order-refl*: $z \leq (z :: 'a :: \text{len0 word})$
unfolding *word-le-def* **by** *auto*

lemma *word-order-antisym*: $x \leq y \implies y \leq x \implies x = (y :: 'a :: \text{len0 word})$
unfolding *word-le-def* **by** (*auto intro!*: *word-uint.Rep-eqD*)

lemma *word-order-linear*:
 $y \leq x \mid x \leq (y :: 'a :: \text{len0 word})$
unfolding *word-le-def* **by** *auto*

lemma *word-zero-le* [*simp*] :
 $0 \leq (y :: 'a :: \text{len0 word})$
unfolding *word-le-def* **by** *auto*

instance *word* :: (*len0*) *semigroup-add*
by *intro-classes* (*simp add*: *word-add-assoc*)

instance *word* :: (*len0*) *linorder*
by *intro-classes* (*auto simp*: *word-less-def word-le-def*)

instance *word* :: (*len0*) *ring*
by *intro-classes*
(*auto simp*: *word-arith-eqs word-diff-minus*
word-diff-self [*unfolded word-diff-minus*])

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

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

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

lemma *word-gt-0*: $0 < y = (0 \sim = (y :: 'a :: \text{len0 word}))$
unfolding *word-less-def* **by** *auto*

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

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*:

(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*:

(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*, *standard*]

lemma *word-zero-neq-one*: $0 < \text{len-of TYPE}('a :: \text{len0}) \implies (0 :: 'a \text{ word}) \sim = 1$

unfolding *word-arith-wis*
by (auto simp add: word-ubin.norm-eq-iff [symmetric] *gr0-conv-Suc*)

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

lemma *no-no* [simp]: $\text{number-of} (\text{number-of } b) = \text{number-of } b$

by (simp add: *number-of-eq*)

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, standard]
lemmas *uint-mult-ge0* [*simp*] =
mult-nonneg-nonneg [OF uint-ge-0 uint-ge-0, standard]

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*

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

13.12 Definition of uint_arith

lemma word-of-int-inverse:
 $word-of-int\ r = a \implies 0 \leq r \implies r < 2^{\wedge} len-of\ TYPE('a) \implies$
 $uint\ (a::'a::len0\ word) = r$
apply (erule word-uint.Abs-inverse' [rotated])
apply (simp add: uints-num)
done

lemma uint-split:
fixes $x::'a::len0\ word$
shows $P\ (uint\ x) =$
 $(ALL\ i.\ word-of-int\ i = x \ \&\ 0 \leq i \ \&\ i < 2^{\wedge} len-of\ TYPE('a) \longrightarrow P\ i)$
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 $P\ (uint\ x) =$
 $(\sim (EX\ i.\ word-of-int\ i = x \ \&\ 0 \leq i \ \&\ i < 2^{\wedge} len-of\ TYPE('a) \ \&\ \sim P\ i))$
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: $False \implies a^{\wedge} b = c^{\wedge} d$
by auto

ML ⟨⟨
fun uint-arith-ss-of ss =
ss addsimps @ { thms uint-arith-simps }
delsimps @ { thms word-uint.Rep-inject }
addsplits @ { thms split-if-asm }
addcongs @ { 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 - => Seq.empty;
  in
    [ clarify-tac ctxt 1,
      full-simp-tac (uint-arith-ss-of (simpset-of ctxt)) 1,
      ALLGOALS (full-simp-tac (HOL-ss addsplits @ { thms uint-splits }
        addcongs @ { 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

```

13.13 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: word-add-ac add-ac no-olen-add)

```

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

lemmas *uint-plus-simple-iff* = *trans* [OF *no-olen-add uint-add-lem, standard*]

lemmas *uint-plus-simple* = *uint-plus-simple-iff* [THEN *iffD1, standard*]

lemmas *uint-minus-simple-iff* = *trans* [OF *no-ulen-sub uint-sub-lem, standard*]

lemmas *uint-minus-simple-alt* = *uint-sub-lem* [folded word-le-def]

lemmas *word-sub-le-iff* = *no-ulen-sub* [*folded word-le-def*]
lemmas *word-sub-le* = *word-sub-le-iff* [*THEN iffD2, standard*]

lemma *word-less-sub1*:
 $(x :: 'a :: \text{len0 word}) \sim = 0 \implies (1 < x) = (0 < x - 1)$
by *uint-arith*

lemma *word-le-sub1*:
 $(x :: 'a :: \text{len0 word}) \sim = 0 \implies (1 \leq x) = (0 \leq x - 1)$
by *uint-arith*

lemma *sub-wrap-lt*:
 $((x :: 'a :: \text{len0 word}) < x - z) = (x < z)$
by *uint-arith*

lemma *sub-wrap*:
 $((x :: 'a :: \text{len0 word}) \leq x - z) = (z = 0 \mid x < z)$
by *uint-arith*

lemma *plus-minus-not-NULL-ab*:
 $(x :: 'a :: \text{len0 word}) \leq ab - c \implies c \leq ab \implies c \sim = 0 \implies x + c \sim = 0$
by *uint-arith*

lemma *plus-minus-no-overflow-ab*:
 $(x :: 'a :: \text{len0 word}) \leq ab - c \implies c \leq ab \implies x \leq x + c$
by *uint-arith*

lemma *le-minus'*:
 $(a :: 'a :: \text{len0 word}) + c \leq b \implies a \leq a + c \implies c \leq b - a$
by *uint-arith*

lemma *le-plus'*:
 $(a :: 'a :: \text{len0 word}) \leq b \implies c \leq b - a \implies a + c \leq b$
by *uint-arith*

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

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

lemma *word-plus-mono-right*:
 $(y :: 'a :: \text{len0 word}) \leq z \implies x \leq x + z \implies x + y \leq x + z$
by *uint-arith*

lemma *word-less-minus-cancel*:
 $y - x < z - x \implies x \leq z \implies (y :: 'a :: \text{len0 word}) < z$
by *uint-arith*

lemma *word-less-minus-mono-left*:
 $(y :: 'a :: \text{len0 word}) < z \implies x \leq y \implies y - x < z - x$

by *uint-arith*

lemma *word-less-minus-mono*:

$$a < c \implies d < b \implies a - b < a \implies c - d < c \\ \implies a - b < c - (d :: 'a :: \text{len } \text{word})$$

by *uint-arith*

lemma *word-le-minus-cancel*:

$$y - x \leq z - x \implies x \leq z \implies (y :: 'a :: \text{len } \text{word}) \leq z$$

by *uint-arith*

lemma *word-le-minus-mono-left*:

$$(y :: 'a :: \text{len } \text{word}) \leq z \implies x \leq y \implies y - x \leq z - x$$

by *uint-arith*

lemma *word-le-minus-mono*:

$$a \leq c \implies d \leq b \implies a - b \leq a \implies c - d \leq c \\ \implies a - b \leq c - (d :: 'a :: \text{len } \text{word})$$

by *uint-arith*

lemma *plus-le-left-cancel-wrap*:

$$(x :: 'a :: \text{len } \text{word}) + y' < x \implies x + y < x \implies (x + y' < x + y) = (y' < y)$$

by *uint-arith*

lemma *plus-le-left-cancel-nowrap*:

$$(x :: 'a :: \text{len } \text{word}) \leq x + y' \implies x \leq x + y \implies \\ (x + y' < x + y) = (y' < y)$$

by *uint-arith*

lemma *word-plus-mono-right2*:

$$(a :: 'a :: \text{len } \text{word}) \leq a + b \implies c \leq b \implies a \leq a + c$$

by *uint-arith*

lemma *word-less-add-right*:

$$(x :: 'a :: \text{len } \text{word}) < y - z \implies z \leq y \implies x + z < y$$

by *uint-arith*

lemma *word-less-sub-right*:

$$(x :: 'a :: \text{len } \text{word}) < y + z \implies y \leq x \implies x - y < z$$

by *uint-arith*

lemma *word-le-plus-either*:

$$(x :: 'a :: \text{len } \text{word}) \leq y \mid x \leq z \implies y \leq y + z \implies x \leq y + z$$

by *uint-arith*

lemma *word-less-nowrapI*:

$$(x :: 'a :: \text{len } \text{word}) < z - k \implies k \leq z \implies 0 < k \implies x < x + k$$

by *uint-arith*

lemma *inc-le*: $(i :: 'a :: \text{len word}) < m \implies i + 1 \leq m$
by *uint-arith*

lemma *inc-i*:
 $(1 :: 'a :: \text{len word}) \leq i \implies i < m \implies 1 \leq (i + 1) \ \& \ i + 1 \leq m$
by *uint-arith*

lemma *udvd-incr-lem*:
 $up < uq \implies up = ua + n * \text{uint } K \implies$
 $uq = ua + n' * \text{uint } K \implies up + \text{uint } K \leq uq$
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, simplified*]

lemmas *udvd-incr0* = *udvd-incr'* [**where** *ua=0, simplified*]

lemmas *udvd-decr0* = *udvd-decr'* [**where** *ua=0, simplified*]

lemma *udvd-minus-le'*:
 $xy < k \implies z \text{ udvd } xy \implies z \text{ udvd } k \implies xy \leq k - z$
apply (*unfold udvd-def*)
apply *clarify*
apply (*erule* (2) *udvd-decr0*)
done

ML \ll *Delsimprocs Numeral-Simprocs.cancel-factors* \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 Numeral-Simprocs.cancel-factors \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)

13.14 Arithmetic type class instantiations

instance *word* :: (*len0*) *comm-monoid-add* ..

instance *word* :: (*len0*) *comm-monoid-mult*
apply (*intro-classes*)
apply (*simp add: word-mult-commute*)
apply (*simp add: word-mult-1*)
done

instance *word* :: (*len0*) *comm-semiring*
by (*intro-classes*) (*simp add : word-left-distrib*)

instance *word* :: (*len0*) *ab-group-add* ..

instance *word* :: (*len0*) *comm-ring* ..

instance *word* :: (*len*) *comm-semiring-1*
by (*intro-classes*) (*simp add: lenw1-zero-neq-one*)

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

instance *word* :: (*len0*) *comm-semiring-0* ..

instance *word* :: (*len0*) *order* ..

lemma *zero-bintrunc*:

$iszero (number-of x :: 'a :: len word) =$
 $(bintrunc (len-of TYPE('a)) x = Int.Pls)$
apply (*unfold iszero-def word-0-wi word-no-wi*)
apply (*rule word-ubin.norm-eq-iff [symmetric, THEN trans]*)
apply (*simp add : Pls-def [symmetric]*)
done

lemmas *word-le-0-iff* [*simp*] =
word-zero-le [THEN leD, THEN linorder-antisym-conv1]

lemma *word-of-nat*: $of-nat n = word-of-int (int n)$

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

lemma *word-of-int: of-int = word-of-int*
apply (*rule ext*)
apply (*unfold of-int-def*)
apply (*rule the-elemI*)
apply *safe*
apply (*simp-all add: word-of-nat word-of-int-homs*)
defer
apply (*rule Rep-Integ-ne [THEN nonemptyE]*)
apply (*rule beXI*)
prefer 2
apply *assumption*
apply (*auto simp add: RI-eq-diff*)
done

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 *word-number-of-eq:*
 $\text{number-of } w = (\text{of-int } w :: 'a :: \text{len word})$
unfolding *word-number-of-def word-of-int* **by** *auto*

instance *word :: (len) number-ring*
by (*intro-classes*) (*simp add : word-number-of-eq*)

lemma *iszero-word-no [simp] :*
 $\text{iszero } (\text{number-of } \text{bin} :: 'a :: \text{len word}) =$
 $\text{iszero } (\text{number-of } (\text{bintrunc } (\text{len-of TYPE}('a)) \text{bin}) :: \text{int})$
apply (*simp add: zero-bintrunc number-of-is-id*)
apply (*unfold iszero-def Pls-def*)
apply (*rule refl*)
done

13.15 Word and nat

lemma *td-ext-unat!:*
 $n = \text{len-of TYPE } ('a :: \text{len}) \implies$
 $\text{td-ext } (\text{unat} :: 'a \text{ word} \Rightarrow \text{nat}) \text{ of-nat}$
 $(\text{unats } n) (\%i. i \bmod 2 \wedge n)$
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 *td-ext-unat = refl [THEN td-ext-unat]*

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

interpretation *word-unat*:

```

td-ext unat::'a::len word => nat
  of-nat
  unats (len-of TYPE('a::len))
  %i. i mod 2 ^ len-of TYPE('a::len)
by (rule td-ext-unat)

```

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

lemmas *unat-lt2p [iff] = word-unat.Rep [unfolded unats-def mem-Collect-eq]*

lemma *unat-le: y <= unat (z :: 'a :: len word) ==> y : unats (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 :: len word) = of-nat n & n < 2 ^ len-of TYPE ('a)
apply (rule allI)
apply (rule word-unat.Abs-cases)
apply (unfold unats-def)
apply auto
done

```

lemma *of-nat-eq*:

```

fixes w :: 'a::len word
shows (of-nat n = w) = ( $\exists q. n = \text{unat } w + q * 2 ^ \text{len-of TYPE('a)}$ )
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*:

```

(of-nat n = w) = (EX q. n = unat w + q * 2 ^ size w)
unfolding word-size by (rule of-nat-eq)

```

lemma *of-nat-0*:

```

(of-nat m = (0::'a::len word)) = ( $\exists q. m = q * 2 ^ \text{len-of TYPE('a)}$ )
by (simp add: of-nat-eq)

```

lemmas *of-nat-2p = mult-1 [symmetric, THEN iffD2 [OF of-nat-0 exI]]*

lemma *of-nat-gt-0: of-nat k ~ = 0 ==> 0 < k*

```

by (cases k) auto

```

lemma *of-nat-neq-0*:

$0 < k \implies k < 2 \wedge \text{len-of TYPE } ('a :: \text{len}) \implies \text{of-nat } k \approx = (0 :: 'a \text{ word})$
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 word-of-int-mult-hom 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 word-of-int-succ-hom add-ac*)

lemma *Abs-fnat-hom-0*: $(0 :: 'a :: \text{len word}) = \text{of-nat } 0$

by (*simp add: word-of-nat word-0-wi*)

lemma *Abs-fnat-hom-1*: $(1 :: 'a :: \text{len word}) = \text{of-nat } (\text{Suc } 0)$

by (*simp add: word-of-nat word-1-wi*)

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: Abs-fnat-hom-mult [symmetric]*)

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

lemmas *unat-cong* = *arg-cong* [**where** $f = \text{unat}$]

lemmas *unat-word-ariths* = *word-arith-nat-defs*
 [THEN *trans* [OF *unat-cong unat-of-nat*], *standard*]

lemmas *word-sub-less-iff* = *word-sub-le-iff*
 [*simplified linorder-not-less* [*symmetric*], *simplified*]

lemma *unat-add-lem*:
 ($\text{unat } x + \text{unat } y < 2 \wedge \text{len-of TYPE('a)} =$
 $\text{unat } (x + y :: 'a :: \text{len word}) = \text{unat } x + \text{unat } y$)
unfolding *unat-word-ariths*
by (*auto intro!*: *trans* [OF - *nat-mod-lem*])

lemma *unat-mult-lem*:
 ($\text{unat } x * \text{unat } y < 2 \wedge \text{len-of TYPE('a)} =$
 $\text{unat } (x * y :: 'a :: \text{len word}) = \text{unat } x * \text{unat } y$)
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*, *standard*]

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-eqv* [THEN *iffD1*])
done

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

lemma *unat-sub-if-size*:
 $\text{unat } (x - y) = (\text{if } \text{unat } y \leq \text{unat } x$
 $\text{then } \text{unat } x - \text{unat } y$
 $\text{else } \text{unat } x + 2 \wedge \text{size } x - \text{unat } y)$
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-eq)
  apply auto
apply uint-arith
done

```

lemmas unat-sub-if' = unat-sub-if-size [unfolded word-size]

```

lemma unat-div: unat ((x :: 'a :: len word) div y) = unat x div unat y
apply (simp add : unat-word-ariths)
apply (rule unat-lt2p [THEN xtr7, THEN nat-mod-eq])
apply (rule div-le-dividend)
done

```

```

lemma unat-mod: unat ((x :: 'a :: len word) mod y) = unat x mod 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: uint ((x :: 'a :: len word) div y) = uint x div uint y
unfolding uint-nat by (simp add : unat-div zdiv-int)

```

```

lemma uint-mod: uint ((x :: 'a :: len word) mod y) = uint x mod uint y
unfolding uint-nat by (simp add : unat-mod zmod-int)

```

13.16 Definition of unat_arith tactic

```

lemma unat-split:
  fixes x::'a::len word
  shows P (unat x) =
    (ALL n. of-nat n = x & n < 2^len-of TYPE('a) --> P n)
  by (auto simp: unat-of-nat)

```

```

lemma unat-split-asm:
  fixes x::'a::len word
  shows P (unat x) =
    (~(EX n. of-nat n = x & n < 2^len-of TYPE('a) & ~ P n))
  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 =
  ss addsimps @{\thms unat-arith-simps}
  delsimps @{\thms word-unat.Rep-inject}
  addsplits @{\thms split-if-asm}
  addcongs @{\thms power-False-cong}

fun unat-arith-tacs ctxt =
  let
    fun arith-tac' n t =
      Arith-Data.verbose-arith-tac 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 addsplits @{\thms unat-splits}
                             addcongs @{\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))
 $\rangle\rangle$ 

```

```

method-setup unat-arith =
   $\langle\langle$  Scan.succeed (SIMPLE-METHOD' o unat-arith-tac)  $\rangle\rangle$ 
  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, standard]

```

```

lemma word-div-mult:
  (0 :: 'a :: len word) < y  $\implies$  unat x * unat y < 2 ^ len-of TYPE('a)  $\implies$ 
  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  $\implies$ 
  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  $\implies$  0 < x  $\implies$  i * x < k
apply (frule div-lt'' [THEN unat-mult-lem [THEN iffD1]])
apply (simp add: unat-arith-simps)
apply (drule (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, standard]
lemmas word-diff-ls = mcs [where z = w + x, unfolded add-diff-cancel, standard]
lemmas word-plus-mcs = word-diff-ls
  [where y = v + x, unfolded add-diff-cancel, standard]

lemmas le-unat-voi = unat-le [THEN word-unat.Abs-inverse]

lemmas thd = refl [THEN [2] split-div-lemma [THEN iffD2], THEN conjunct1]

lemma thd1:
  a div b * b ≤ (a::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-voi, standard] =
  mod-le-divisor div-le-dividend thd1

lemma word-mod-div-equality:
  (n div b) * b + (n mod b) = (n :: 'a :: len word)
apply (unfold word-less-nat-alt word-arith-nat-defs)
apply (cut-tac y=unat b in gt-or-eq-0)
apply (erule disjE)
  apply (simp 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*: $0 < n \implies m \bmod n < (n :: 'a :: \text{len word})$
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*: *word-of-int-hom-syms*)

lemma *word-arith-power-alt*:
 $a \wedge n = (\text{word-of-int } (\text{wint } a \wedge n) :: 'a :: \text{len word})$
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-no [unfolded word-number-of-def]*
word-less-alt word-number-of-alt)
apply *safe*
apply (*simp (no-asm) add*: *word-of-int-power-hom word-wint.eq-norm*
del: *word-of-int-bin*)
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

13.17 Cardinality, finiteness of set of words

lemmas *card-lessThan' = card-lessThan* [*unfolded lessThan-def*]

lemmas *card-eq = word-unat.Abs-inj-on* [*THEN card-image,*
unfolded word-unat.image, unfolded unats-def, standard]

lemmas *card-word = trans* [*OF card-eq card-lessThan', standard*]

lemma *finite-word-UNIV*: *finite* (*UNIV* :: *'a* :: *len word set*)
apply (*rule contrapos-np*)
prefer 2
apply (*erule card-infinite*)
apply (*simp add*: *card-word*)
done

lemma *card-word-size*:
 $\text{card } (\text{UNIV} :: 'a :: \text{len word set}) = (2 \wedge \text{size } (x :: 'a \text{ word}))$
unfolding *word-size* **by** (*rule card-word*)

13.18 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, standard]

lemmas *word-log-binary-defs* =
word-and-def word-or-def word-xor-def

lemmas *word-no-log-defs* [*simp*] =
word-not-def [**where** *a=number-of a*,
unfolded word-no-wi wils1, folded word-no-wi, standard]
word-log-binary-defs [**where** *a=number-of a and b=number-of b*,
unfolded word-no-wi wils1, folded word-no-wi, standard]

lemmas *word-wi-log-defs* = *word-no-log-defs* [*unfolded word-no-wi*]

lemma *uint-or*: *uint (x OR y) = (uint x) OR (uint y)*
by (*simp add: word-or-def word-no-wi* [*symmetric*] *number-of-is-id*
bin-trunc-ao(2) [*symmetric*])

lemma *uint-and*: *uint (x AND y) = (uint x) AND (uint y)*
by (*simp add: word-and-def number-of-is-id word-no-wi* [*symmetric*]
bin-trunc-ao(1) [*symmetric*])

lemma *word-ops-nth-size*:
 $n < \text{size } (x::'a::\text{len0 word}) \implies$
 $(x \text{ OR } y) !! n = (x !! n | y !! n) \ \&$
 $(x \text{ AND } y) !! n = (x !! n \ \& \ y !! n) \ \&$
 $(x \text{ XOR } y) !! n = (x !! n \ \sim = \ y !! n) \ \&$
 $(\text{NOT } x) !! n = (\sim x !! n)$
unfolding *word-size word-no-wi word-test-bit-def word-log-defs*
by (*clarsimp simp add : word-ubin.eq-norm nth-bintr bin-nth-ops*)

lemma *word-ao-nth*:
fixes $x :: 'a::\text{len0 word}$
shows $(x \text{ OR } y) !! n = (x !! n | y !! n) \ \&$
 $(x \text{ AND } y) !! n = (x !! n \ \& \ y !! n) \ \&$
apply (*cases n < size x*)
apply (*drule-tac y = y in word-ops-nth-size*)
apply *simp*
apply (*simp add : test-bit-bin word-size*)
done

lemmas *bwsimps* =
word-of-int-homs(2)
word-0-wi-Pls
word-m1-wi-Min
word-wi-log-defs

lemma *word-bw-assocs*:
fixes $x :: 'a::len0$ *word*
shows
 $(x \text{ AND } y) \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $(x \text{ OR } y) \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $(x \text{ XOR } y) \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
using *word-of-int-Ex* [**where** $x=x$]
word-of-int-Ex [**where** $x=y$]
word-of-int-Ex [**where** $x=z$]
by (*auto simp: bwsimps bbw-assocs*)

lemma *word-bw-comms*:
fixes $x :: 'a::len0$ *word*
shows
 $x \text{ AND } y = y \text{ AND } x$
 $x \text{ OR } y = y \text{ OR } x$
 $x \text{ XOR } y = y \text{ XOR } x$
using *word-of-int-Ex* [**where** $x=x$]
word-of-int-Ex [**where** $x=y$]
by (*auto simp: bwsimps bin-ops-comm*)

lemma *word-bw-lcs*:
fixes $x :: 'a::len0$ *word*
shows
 $y \text{ AND } x \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $y \text{ OR } x \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $y \text{ XOR } x \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
using *word-of-int-Ex* [**where** $x=x$]
word-of-int-Ex [**where** $x=y$]
word-of-int-Ex [**where** $x=z$]
by (*auto simp: bwsimps*)

lemma *word-log-esimps* [*simp*]:
fixes $x :: 'a::len0$ *word*
shows
 $x \text{ AND } 0 = 0$
 $x \text{ AND } -1 = x$
 $x \text{ OR } 0 = x$
 $x \text{ OR } -1 = -1$
 $x \text{ XOR } 0 = x$
 $x \text{ XOR } -1 = \text{NOT } x$
 $0 \text{ AND } x = 0$

$-1 \text{ AND } x = x$
 $0 \text{ OR } x = x$
 $-1 \text{ OR } x = -1$
 $0 \text{ XOR } x = x$
 $-1 \text{ XOR } x = \text{NOT } x$
using *word-of-int-Ex* [**where** $x=x$]
by (*auto simp: bwsimps*)

lemma *word-not-dist*:
fixes $x :: 'a::\text{len0 word}$
shows
 $\text{NOT } (x \text{ OR } y) = \text{NOT } x \text{ AND } \text{NOT } y$
 $\text{NOT } (x \text{ AND } y) = \text{NOT } x \text{ OR } \text{NOT } y$
using *word-of-int-Ex* [**where** $x=x$]
word-of-int-Ex [**where** $x=y$]
by (*auto simp: bwsimps bbw-not-dist*)

lemma *word-bw-same*:
fixes $x :: 'a::\text{len0 word}$
shows
 $x \text{ AND } x = x$
 $x \text{ OR } x = x$
 $x \text{ XOR } x = 0$
using *word-of-int-Ex* [**where** $x=x$]
by (*auto simp: bwsimps*)

lemma *word-ao-absorbs* [*simp*]:
fixes $x :: 'a::\text{len0 word}$
shows
 $x \text{ AND } (y \text{ OR } x) = x$
 $x \text{ OR } y \text{ AND } x = x$
 $x \text{ AND } (x \text{ OR } y) = x$
 $y \text{ AND } x \text{ OR } x = x$
 $(y \text{ OR } x) \text{ AND } x = x$
 $x \text{ OR } x \text{ AND } y = x$
 $(x \text{ OR } y) \text{ AND } x = x$
 $x \text{ AND } y \text{ OR } x = x$
using *word-of-int-Ex* [**where** $x=x$]
word-of-int-Ex [**where** $x=y$]
by (*auto simp: bwsimps*)

lemma *word-not-not* [*simp*]:
 $\text{NOT } \text{NOT } (x :: 'a::\text{len0 word}) = x$
using *word-of-int-Ex* [**where** $x=x$]
by (*auto simp: bwsimps*)

lemma *word-ao-dist*:
fixes $x :: 'a::\text{len0 word}$
shows $(x \text{ OR } y) \text{ AND } z = x \text{ AND } z \text{ OR } y \text{ AND } z$

using *word-of-int-Ex* [**where** $x=x$]
word-of-int-Ex [**where** $x=y$]
word-of-int-Ex [**where** $x=z$]
by (*auto simp: bwsimps bbw-ao-dist*)

lemma *word-oa-dist*:
fixes $x :: 'a::len0\ word$
shows $x\ AND\ y\ OR\ z = (x\ OR\ z)\ AND\ (y\ OR\ z)$
using *word-of-int-Ex* [**where** $x=x$]
word-of-int-Ex [**where** $x=y$]
word-of-int-Ex [**where** $x=z$]
by (*auto simp: bwsimps bbw-oa-dist*)

lemma *word-add-not* [*simp*]:
fixes $x :: 'a::len0\ word$
shows $x + NOT\ x = -1$
using *word-of-int-Ex* [**where** $x=x$]
by (*auto simp: bwsimps bin-add-not*)

lemma *word-plus-and-or* [*simp*]:
fixes $x :: 'a::len0\ word$
shows $(x\ AND\ y) + (x\ OR\ y) = x + y$
using *word-of-int-Ex* [**where** $x=x$]
word-of-int-Ex [**where** $x=y$]
by (*auto simp: bwsimps plus-and-or*)

lemma *leoa*:
fixes $x :: 'a::len0\ word$
shows $(w = (x\ OR\ y)) \implies (y = (w\ AND\ y))$ **by** *auto*

lemma *leao*:
fixes $x' :: 'a::len0\ word$
shows $(w' = (x'\ AND\ y')) \implies (x' = (x'\ OR\ w'))$ **by** *auto*

lemmas *word-ao-equiv = leao* [*COMP leoa* [*COMP iffI*]]

lemma *le-word-or2*: $x \leq x\ OR\ (y :: 'a::len0\ word)$
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, standard*]

lemmas *word-and-le1 =*
xtr3 [*OF word-ao-absorbs* (4) [*symmetric*] *le-word-or2, standard*]

lemmas *word-and-le2 =*
xtr3 [*OF word-ao-absorbs* (8) [*symmetric*] *le-word-or2, standard*]

lemma *bl-word-not*: $to-bl\ (NOT\ w) = map\ Not\ (to-bl\ w)$
unfolding *to-bl-def word-log-defs*
by (*simp add: bl-not-bin number-of-is-id word-no-wi* [*symmetric*] *bin-to-bl-def* [*symmetric*])

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: number-of-is-id word-no-wi [symmetric]*)

lemma *bl-word-or*: $to-bl (v OR w) = map2\ op\ | (to-bl\ v)\ (to-bl\ w)$
unfolding *to-bl-def word-log-defs*
by (*simp add: bl-or-bin number-of-is-id word-no-wi [symmetric]*)

lemma *bl-word-and*: $to-bl (v AND w) = map2\ op\ \& (to-bl\ v)\ (to-bl\ w)$
unfolding *to-bl-def word-log-defs*
by (*simp add: bl-and-bin number-of-is-id word-no-wi [symmetric]*)

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*: $lsb (1::'a::len\ word) \& \sim lsb (0::'b::len0\ word)$
unfolding *word-lsb-def word-1-no word-0-no* **by** *auto*

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-mod* **by** *auto*

lemma *word-msb-sint*: $msb\ w = (sint\ w < 0)$
unfolding *word-msb-def*
by (*simp add : sign-Min-lt-0 number-of-is-id*)

lemma *word-msb-no'*:
 $w = number-of\ bin \implies msb (w::'a::len\ word) = bin-nth\ bin (size\ w - 1)$
unfolding *word-msb-def word-number-of-def*
by (*clarsimp simp add: word-sbin.eq-norm word-size bin-sign-lem*)

lemmas *word-msb-no = refl [THEN word-msb-no', unfolded word-size]*

lemma *word-msb-nth'*: $msb (w::'a::len\ word) = bin-nth (uint\ w) (size\ w - 1)$
apply (*unfold word-size*)
apply (*rule trans [OF - word-msb-no]*)
apply (*simp add : word-number-of-def*)
done

lemmas *word-msb-nth = word-msb-nth' [unfolded word-size]*

lemma *word-msb-alt*: $msb (w::'a::len\ word) = hd (to-bl\ w)$

```

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*:

```

set-bit w n (test-bit w n) = (w::'a::len0 word)
unfolding word-test-bit-def word-set-bit-def by auto

```

lemma *bin-nth-uint'*:

```

bin-nth (uint w) n = (rev (bin-to-bl (size w) (uint w)) ! n & n < size w)
apply (unfold word-size)
apply (safe elim!: bin-nth-uint-imp)
apply (frule bin-nth-uint-imp)
apply (fast dest!: bin-nth-bl)+
done

```

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

lemma *test-bit-bl*: $w !! n = (rev (to-bl w) ! n \ \& \ n < size \ w)$

```

unfolding to-bl-def word-test-bit-def word-size
by (rule bin-nth-uint)

```

lemma *to-bl-nth*: $n < size \ w \implies to-bl \ w \ ! \ n = w \ !! (size \ w - Suc \ n)$

```

apply (unfold test-bit-bl)
apply clarsimp
apply (rule trans)
apply (rule nth-rev-alt)
apply (auto simp add: word-size)
done

```

lemma *test-bit-set*:

```

fixes w :: 'a::len0 word
shows (set-bit w n x) !! n = (n < 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 !! (size w - 1)*
unfolding *word-msb-nth' word-test-bit-def* **by** *simp*

lemmas *msb-nth = msb-nth' [unfolded word-size]*

lemmas *msb0 = len-gt-0 [THEN diff-Suc-less, THEN*
word-ops-nth-size [unfolded word-size], standard]

lemmas *msb1 = msb0 [where i = 0]*

lemmas *word-ops-msb = msb1 [unfolded msb-nth [symmetric, unfolded One-nat-def]]*

lemmas *lsb0 = len-gt-0 [THEN word-ops-nth-size [unfolded word-size], standard]*

lemmas *word-ops-lsb = lsb0 [unfolded word-lsb-alt]*

lemma *td-ext-nth'*:
n = size (w::'a::len0 word) \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 del: subset-antisym*)
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

lemmas *td-ext-nth = td-ext-nth' [OF refl refl refl, unfolded word-size]*

interpretation *test-bit*:

td-ext op !! :: 'a::len0 word => nat => bool
set-bits
{f. $\forall i. f i \implies i < \text{len-of TYPE('a::len0)}$ }
($\lambda h i. h i \wedge i < \text{len-of TYPE('a::len0)}$)
by (*rule td-ext-nth*)

declare *test-bit.Rep'* [*simp del*]

declare *test-bit.Rep'* [*rule del*]

lemmas *td-nth = test-bit.td-thm*

lemma *word-set-set-same:*

fixes *w :: 'a::len0 word*
shows *set-bit (set-bit w n x) n y = set-bit w n y*
by (*rule word-eqI*) (*simp add : test-bit-set-gen word-size*)

lemma *word-set-set-diff:*

fixes *w :: 'a::len0 word*
assumes *m ~ = n*
shows *set-bit (set-bit w m x) n y = set-bit (set-bit w n y) m x*
by (*rule word-eqI*) (*clarsimp simp add: test-bit-set-gen word-size assms*)

lemma *test-bit-no':*

fixes *w :: 'a::len0 word*
shows *w = number-of bin \implies test-bit w n = (n < size w & bin-nth bin n)*
unfolding *word-test-bit-def word-number-of-def word-size*
by (*simp add : nth-bintr [symmetric] word-ubin.eq-norm*)

lemmas *test-bit-no =*

refl [THEN test-bit-no', unfolded word-size, THEN eq-reflection, standard]

lemma *nth-0: ~ (0::'a::len0 word) !! n*

unfolding *test-bit-no word-0-no* **by** *auto*

lemma *nth-sint:*

fixes *w :: 'a::len word*
defines *l \equiv len-of TYPE ('a)*
shows *bin-nth (sint w) n = (if n < l - 1 then w !! n else w !! (l - 1))*
unfolding *sint-uint l-def*
by (*clarsimp simp add: nth-sbintr word-test-bit-def [symmetric]*)

lemma *word-lsb-no:*

lsb (number-of bin :: 'a :: len word) = (bin-last bin = 1)
unfolding *word-lsb-alt test-bit-no* **by** *auto*

lemma *word-set-no:*

set-bit (number-of bin::'a::len0 word) n b =
number-of (bin-sc n (if b then 1 else 0) bin)
apply (*unfold word-set-bit-def word-number-of-def [symmetric]*)
apply (*rule word-eqI*)
apply (*clarsimp simp: word-size bin-nth-sc-gen number-of-is-id*
test-bit-no nth-bintr)
done

lemma *setBit-no:*

setBit (number-of bin) n = number-of (bin-sc n 1 bin)
by (*simp add: setBit-def word-set-no*)

lemma *clearBit-no*:

clearBit (number-of bin) *n* = number-of (bin-sc *n* 0 bin)
by (*simp* add: *clearBit-def* word-set-no)

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 test-bit-no)
apply (*auto simp* add: word-bl.Abs-inverse test-bit-bl word-size)
done

lemma *word-msb-n1*: *msb* (-1::'a::len word)

unfolding *word-msb-alt* *to-bl-n1* **by** *simp*

declare *word-set-set-same* [*simp*] *word-set-nth* [*simp*]

test-bit-no [*simp*] *word-set-no* [*simp*] *nth-0* [*simp*]
setBit-no [*simp*] *clearBit-no* [*simp*]
word-lsb-no [*simp*] *word-msb-no* [*simp*] *word-msb-n1* [*simp*] *word-lsb-1-0* [*simp*]

lemma *word-set-nth-iff*:

(*set-bit* *w* *n* *b* = *w*) = (*w* !! *n* = *b* | *n* >= size (*w*::'a::len0 word))
apply (rule *iffI*)
apply (rule *disjCI*)
apply (drule *word-eqD*)
apply (erule *sym* [THEN *trans*])
apply (*simp* add: *test-bit-set*)
apply (erule *disjE*)
apply *clarsimp*
apply (rule *word-eqI*)
apply (*clarsimp simp* add : *test-bit-set-gen*)
apply (drule *test-bit-size*)
apply *force*
done

lemma *test-bit-2p'*:

w = word-of-int ($2 \wedge n$) \implies
w !! *m* = (*m* = *n* & *m* < size (*w*::'a::len word))
unfolding *word-test-bit-def* word-size
by (*auto simp* add: *word-ubin.eq-norm* *nth-bintr* *nth-2p-bin*)

lemmas *test-bit-2p* = refl [THEN *test-bit-2p'*, unfolded word-size]

lemma *nth-w2p*:

((2::'a::len word) $\wedge n$) !! *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 simp add : word-1-wi [symmetric])
  apply (clarsimp simp add : word-0-wi [symmetric])
  apply (drule word-gt-0 [THEN iffD1])
  apply (safe intro!: word-eqI bin-nth-lem ext)
  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 (clarsimp simp add : number-of-is-id)
  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 <= 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

```

13.19 Shifting, Rotating, and Splitting Words

```

lemma shiftl1-number [simp] :
  shiftl1 (number-of w) = number-of (w BIT 0)
apply (unfold shiftl1-def word-number-of-def)
apply (simp add: word-ubin.norm-eq-iff [symmetric] word-ubin.eq-norm
  del: BIT-simps)
apply (subst refl [THEN bintrunc-BIT-I, symmetric])
apply (subst bintrunc-bintrunc-min)
apply simp
done

```

```

lemma shiftl1-0 [simp] : shiftl1 0 = 0
unfolding word-0-no shiftl1-number by auto

```

```

lemmas shiftl1-def-u = shiftl1-def [folded word-number-of-def]

```

```

lemma shiftl1-def-s: shiftl1 w = number-of (sint w BIT 0)
by (rule trans [OF - shiftl1-number]) simp

```

```

lemma shiftr1-0 [simp] : shiftr1 0 = 0
unfolding shiftr1-def
by simp (simp add: word-0-wi)

```

```

lemma sshiftr1-0 [simp] : sshiftr1 0 = 0
apply (unfold sshiftr1-def)
apply simp
apply (simp add : word-0-wi)
done

```

```

lemma sshiftr1-n1 [simp] : sshiftr1 -1 = -1
unfolding sshiftr1-def by auto

```

```

lemma shiftl-0 [simp] : (0::'a::len0 word) << n = 0
unfolding shiftl-def by (induct n) auto

```

```

lemma shiftr-0 [simp] : (0::'a::len0 word) >> n = 0
unfolding shiftr-def by (induct n) auto

```

```

lemma sshiftr-0 [simp] : 0 >>> n = 0
unfolding sshiftr-def by (induct n) auto

```

```

lemma sshiftr-n1 [simp] : -1 >>> n = -1
unfolding sshiftr-def by (induct n) auto

```

```

lemma nth-shiftl1: shiftl1 w !! n = (n < size w & n > 0 & w !! (n - 1))
  apply (unfold shiftl1-def word-test-bit-def)
  apply (simp add: nth-bintr word-ubin.eq-norm word-size)
  apply (cases n)
  apply auto
done

```

```

lemma nth-shiftl' [rule-format]:
  ALL n. ((w::'a::len0 word) << m) !! n = (n < size w & n >= m & w !! (n - m))
  apply (unfold shiftl-def)
  apply (induct m)
  apply (force elim!: test-bit-size)
  apply (clarsimp simp add : nth-shiftl1 word-size)
  apply arith
done

```

```

lemmas nth-shiftl = nth-shiftl' [unfolded word-size]

```

```

lemma nth-shiftr1: shiftr1 w !! n = w !! Suc n
  apply (unfold shiftr1-def word-test-bit-def)
  apply (simp add: nth-bintr word-ubin.eq-norm)
  apply safe
  apply (drule bin-nth.Suc [THEN iffD2, THEN bin-nth-uint-imp])
  apply simp
done

```

```

lemma nth-shiftr:
  ∧n. ((w::'a::len0 word) >> m) !! n = w !! (n + m)
  apply (unfold shiftr-def)
  apply (induct m)
  apply (auto simp add : nth-shiftr1)
done

```

```

lemma uint-shiftr1: uint (shiftr1 w) = bin-rest (uint w)
  apply (unfold shiftr1-def word-ubin.eq-norm bin-rest-trunc-i)
  apply (subst bintr-uint [symmetric, OF order-refl])
  apply (simp only : bintrunc-bintrunc-l)
  apply simp
done

```

```

lemma nth-sshiftr1:
  sshiftr1 w !! n = (if n = size w - 1 then w !! n else w !! Suc n)
  apply (unfold sshiftr1-def word-test-bit-def)
  apply (simp add: nth-bintr word-ubin.eq-norm
    bin-nth.Suc [symmetric] word-size
    del: bin-nth.simps)

```

```

apply (simp add: nth-bintr uint-sint del : bin-nth.simps)
apply (auto simp add: bin-nth-sint)
done

```

```

lemma nth-sshiftr [rule-format] :
  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-div)
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-div [symmetric])
apply (simp add: word-sbin.eq-norm)
apply (rule trans)
  defer
  apply (subst word-sbin.norm-Rep [symmetric])
  apply (rule refl)
  apply (subst word-sbin.norm-Rep [symmetric])
apply (unfold One-nat-def)
apply (rule sbintrunc-rest)
done

```

```

lemma shiftr-div-2n: uint (shiftr w n) = uint w div 2 ^ n
  apply (unfold shiftr-def)
  apply (induct n)
  apply simp
  apply (simp add: shiftr1-div-2 mult-commute
          zdiv-zmult2-eq [symmetric])
done

```

```

lemma sshiftr-div-2n: sint (sshiftr w n) = sint w div 2 ^ n
  apply (unfold sshiftr-def)
  apply (induct n)
  apply simp
  apply (simp add: sshiftr1-div-2 mult-commute
          zdiv-zmult2-eq [symmetric])
done

```

13.19.1 shift functions in terms of lists of bools

```

lemmas bshiftr1-no-bin [simp] =
  bshiftr1-def [where w=number-of w, unfolded to-bl-no-bin, standard]

```

```

lemma bshiftr1-bl: to-bl (bshiftr1 b w) = b # butlast (to-bl w)
  unfolding bshiftr1-def by (rule word-bl.Abs-inverse) simp

```

```

lemma shiftl1-of-bl: shiftl1 (of-bl bl) = of-bl (bl @ [False])
  unfolding uint-bl of-bl-no
  by (simp add: bl-to-bin-aux-append bl-to-bin-def)

```

```

lemma shiftl1-bl: shiftl1 (w :: 'a :: len0 word) = of-bl (to-bl w @ [False])

```

proof –

```

  have shiftl1 w = shiftl1 (of-bl (to-bl w)) by simp
  also have ... = of-bl (to-bl w @ [False]) by (rule shiftl1-of-bl)
  finally show ?thesis .

```

qed

```

lemma bl-shiftl1:
  to-bl (shiftl1 (w :: 'a :: len 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 shiftr1-bl: shiftr1 w = of-bl (butlast (to-bl w))
  apply (unfold shiftr1-def uint-bl of-bl-def)
  apply (simp add: butlast-rest-bin word-size)
  apply (simp add: bin-rest-trunc [symmetric, unfolded One-nat-def])
done

```

```

lemma bl-shiftr1:
  to-bl (shiftr1 (w :: 'a :: len word)) = False # butlast (to-bl w)

```

unfolding *shiftr1-bl*
by (*simp add : word-rep-drop len-gt-0 [THEN Suc-leI]*)

lemma *shiffl1-rev*:
shiffl1 (w :: 'a :: len word) = word-reverse (shiftr1 (word-reverse w))
apply (*unfold word-reverse-def*)
apply (*rule word-bl.Rep-inverse' [symmetric]*)
apply (*simp add: bl-shiffl1 bl-shiftr1 word-bl.Abs-inverse*)
apply (*cases to-bl w*)
apply *auto*
done

lemma *shiffl-rev*:
shiffl (w :: 'a :: len word) n = word-reverse (shiftr (word-reverse w) n)
apply (*unfold shiffl-def shiftr-def*)
apply (*induct n*)
apply (*auto simp add : shiffl1-rev*)
done

lemmas *rev-shiffl* =
shiffl-rev [where w = word-reverse w, simplified, standard]

lemmas *shiftr-rev* = *rev-shiffl [THEN word-rev-gal', standard]*
lemmas *rev-shiftr* = *shiffl-rev [THEN word-rev-gal', standard]*

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 [rule-format] :
  n <= size (w :: 'a :: len word) --> take n (to-bl (w >> n)) =
    replicate n False
apply (unfold shiftr-def)
apply (induct n)
prefer 2
apply (simp add: bl-shiftr1)
apply (rule impI)
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, standard]
lemmas take-sshiftr = take-sshiftr' [THEN conjunct2, standard]

```

```

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 shiffl-of-bl: of-bl bl << n = of-bl (bl @ replicate n False)
  unfolding shiffl-def
  by (induct n) (auto simp: shiffl1-of-bl replicate-app-Cons-same)

lemma shiffl-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 shiffl-of-bl)
  finally show ?thesis .
qed

lemmas shiffl-number [simp] = shiffl-def [where w=number-of w, standard]

lemma bl-shiffl:
  to-bl (w << n) = drop n (to-bl w) @ replicate (min (size w) n) False
  by (simp add: shiffl-bl word-rep-drop word-size)

lemma shiffl-zero-size:
  fixes x :: 'a::len0 word
  shows size x <= n  $\implies$  x << n = 0
  apply (unfold word-size)
  apply (rule word-eqI)
  apply (clarsimp simp add: shiffl-bl word-size test-bit-of-bl nth-append)
  done

lemma shiffl1-2t: shiffl1 (w :: 'a :: len word) = 2 * w
  apply (simp add: shiffl1-def-u)
  apply (simp only: double-number-of-Bit0 [symmetric])
  apply simp
  done

lemma shiffl1-p: shiffl1 (w :: 'a :: len word) = w + w
  apply (simp add: shiffl1-def-u)
  apply (simp only: double-number-of-Bit0 [symmetric])
  apply simp
  done

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 (number-of w) :: 'a :: len0 word) =
    number-of (bin-rest (bintrunc (len-of TYPE ('a)) w))
  unfolding shiftr1-def word-number-of-def

```

by (simp add : word-ubin.eq-norm)

lemma *sshiftr1-sbintr* [simp] :
 (sshiftr1 (number-of w) :: 'a :: len word) =
 number-of (bin-rest (sbintrunc (len-of TYPE ('a) - 1) w))
unfolding *sshiftr1-def word-number-of-def*
 by (simp add : word-sbin.eq-norm)

lemma *shiftr-no'*:
 $w = \text{number-of bin} \implies$
 $(w::'a::\text{len } 0 \text{ word}) \gg n = \text{number-of } ((\text{bin-rest } \wedge \wedge n) (\text{bintrunc } (\text{size } w) \text{ bin}))$
apply *clarsimp*
apply (*rule word-eqI*)
apply (*auto simp: nth-shiftr nth-rest-power-bin nth-bintr word-size*)
done

lemma *sshiftr-no'*:
 $w = \text{number-of bin} \implies w \gg \gg n = \text{number-of } ((\text{bin-rest } \wedge \wedge n)$
 $(\text{sbintrunc } (\text{size } w - 1) \text{ bin}))$
apply *clarsimp*
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

lemmas *sshiftr-no* [simp] =
sshiftr-no' [where $w = \text{number-of } w$, OF *refl*, *unfolded word-size*, *standard*]

lemmas *shiftr-no* [simp] =
shiftr-no' [where $w = \text{number-of } w$, OF *refl*, *unfolded word-size*, *standard*]

lemma *shiftr1-bl-of'*:
 $us = \text{shiftr1 } (\text{of-bl } bl) \implies \text{length } bl \leq \text{size } us \implies$
 $us = \text{of-bl } (\text{butlast } bl)$
by (*clarsimp simp: shiftr1-def of-bl-def word-size butlast-rest-bl2bin*
word-ubin.eq-norm trunc-bl2bin)

lemmas *shiftr1-bl-of* = *refl* [THEN *shiftr1-bl-of'*, *unfolded word-size*]

lemma *shiftr-bl-of'* [rule-format]:
 $us = \text{of-bl } bl \gg n \implies \text{length } bl \leq \text{size } us \dashrightarrow$
 $us = \text{of-bl } (\text{take } (\text{length } bl - n) \text{ bl})$
apply (*unfold shiftr-def*)
apply *hypsubst*
apply (*unfold word-size*)
apply (*induct n*)
apply *clarsimp*
apply *clarsimp*
apply (*subst shiftr1-bl-of*)

```

apply simp
apply (simp add: butlast-take)
done

lemmas shiftr-bl-of = refl [THEN shiftr-bl-of', unfolded word-size]

lemmas shiftr-bl = word-bl.Rep' [THEN eq-imp-le, THEN shiftr-bl-of,
  simplified word-size, simplified, THEN eq-reflection, standard]

lemma msb-shift': msb (w::'a::len word) <-> (w >> (size w - 1)) ~ = 0
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 word-0-bl
  of-bl-rep-False [where n=1 and bs=[], simplified])
done

lemmas msb-shift = msb-shift' [unfolded word-size]

lemma align-lem-or [rule-format] :
  ALL x m. length x = n + m --> length y = n + m -->
    drop m x = replicate n False --> take m y = replicate m False -->
    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 --> length y = n + m -->
    drop m x = replicate n False --> take m y = replicate m False -->
    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':
  size x - n = m ==> n <= size x ==> drop m (to-bl x) = replicate n False ==>

```

```

    take m (to-bl y) = replicate m False ==>
    to-bl (x + y) = take m (to-bl x) @ drop m (to-bl y)
apply (subgoal-tac x AND y = 0)
prefer 2
apply (rule word-bl.Rep-eqD)
apply (simp add: bl-word-and to-bl-0)
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

```

lemmas *aligned-bl-add-size* = refl [THEN *aligned-bl-add-size*]

13.19.2 Mask

```

lemma nth-mask': m = mask n ==> test-bit m i = (i < n & i < size m)
apply (unfold mask-def test-bit-bl)
apply (simp only: word-1-bl [symmetric] shiftl-of-bl)
apply (clarsimp simp add: word-size)
apply (simp only: of-bl-no mask-lem number-of-succ add-diff-cancel2)
apply (fold of-bl-no)
apply (simp add: word-1-bl)
apply (rule test-bit-of-bl [THEN trans, unfolded test-bit-bl word-size])
apply auto
done

```

lemmas *nth-mask* [simp] = refl [THEN *nth-mask*]

```

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 = number-of (bintrunc n Int.Min)
  by (auto simp add: nth-bintr word-size intro: word-eqI)

```

```

lemma and-mask-bintr: w AND mask n = number-of (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-no: number-of i AND mask n = number-of (bintrunc n i)
  by (auto simp add : nth-bintr word-size word-ops-nth-size intro: word-eqI)

```

lemmas *and-mask-wi* = *and-mask-no* [unfolded *word-number-of-def*]

lemma *bl-and-mask'*:

```

to-bl (w AND mask n :: 'a :: len word) =
  replicate (len-of TYPE('a) - n) False @
  drop (len-of TYPE('a) - n) (to-bl w)
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

```

lemmas *and-mask-mod-2p* =

```

and-mask-bintr [unfolded word-number-of-alt no-bintr-alt]

```

lemma *and-mask-lt-2p*: $\text{uint } (w \text{ AND } \text{mask } n) < 2 \wedge n$

```

apply (simp add : and-mask-bintr no-bintr-alt)
apply (rule xtr8)
prefer 2
apply (rule pos-mod-bound)
apply auto
done

```

lemmas *eq-mod-iff* = *trans* [*symmetric*, *OF int-mod-lem eq-sym-conv*]

lemma *mask-eq-iff*: $(w \text{ AND } \text{mask } n) = w \leftrightarrow \text{uint } w < 2 \wedge n$

```

apply (simp add: and-mask-bintr word-number-of-def)
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)
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-number-of-alt*)

apply (*clarsimp simp add: word-of-int-power-hom word-uint.eq-norm
int-mod-eq'*)

simp del: word-of-int-bin)

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-number-of-alt*)

apply (*clarsimp simp add: and-mask-mod-2p word-of-int-power-hom
word-uint.eq-norm*)

simp del: word-of-int-bin)

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], standard]

lemmas *and-mask-less'* =

iffD2 [OF word-2p-lem and-mask-lt-2p, simplified word-size, standard]

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'*:

$c = 2 \wedge n \implies c > 0 \implies x \text{ mod } c = (x :: 'a :: \text{len } \text{word}) \text{ AND } \text{mask } n$

by (*clarsimp simp add: word-mod-def uint-2p and-mask-mod-2p*)

lemmas *word-mod-2p-is-mask* = *refl [THEN word-mod-2p-is-mask']*

lemma *mask-egs*:

$(a \text{ AND } \text{mask } n) + b \text{ AND } \text{mask } n = a + b \text{ AND } \text{mask } n$

$a + (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a + b \text{ AND } \text{mask } n$

$(a \text{ AND } \text{mask } n) - b \text{ AND } \text{mask } n = a - b \text{ AND } \text{mask } n$

$a - (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a - b \text{ AND } \text{mask } n$

$a * (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a * b \text{ AND } \text{mask } n$

$(b \text{ AND } \text{mask } n) * a \text{ AND } \text{mask } n = b * a \text{ AND } \text{mask } n$

$(a \text{ AND } \text{mask } n) + (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a + b \text{ AND } \text{mask } n$

$(a \text{ AND } \text{mask } n) - (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a - b \text{ AND } \text{mask } n$

$(a \text{ AND } \text{mask } n) * (b \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = a * b \text{ AND } \text{mask } n$

$-(a \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = -a \text{ AND } \text{mask } n$

word-succ $(a \text{ AND } \text{mask } n) \text{ AND } \text{mask } n = \text{word-succ } a \text{ AND } \text{mask } n$

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 new-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*)

13.19.3 Recast

lemmas *revcast-def' = revcast-def* [*simplified*]
lemmas *revcast-def'' = revcast-def'* [*simplified word-size*]
lemmas *revcast-no-def* [*simp*] =
revcast-def' [**where** $w=\text{number-of } w, \text{ unfolded word-size, standard}$]

lemma *to-bl-revcast*:
 $\text{to-bl } (\text{revcast } w :: 'a :: \text{len0 word}) =$
 $\text{takefill False } (\text{len-of TYPE } ('a)) (\text{to-bl } w)$
apply (*unfold revcast-def' word-size*)
apply (*rule word-bl.Abs-inverse*)
apply *simp*
done

lemma *revcast-rev-ucast'*:
 $cs = [rc, uc] \implies rc = \text{revcast } (\text{word-reverse } w) \implies uc = \text{ucast } w \implies$
 $rc = \text{word-reverse } uc$
apply (*unfold ucast-def revcast-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

lemmas *revcast-rev-ucast = revcast-rev-ucast'* [*OF refl refl refl*]

lemmas *revcast-ucast = revcast-rev-ucast*
[**where** $w = \text{word-reverse } w, \text{ simplified word-rev-rev, standard}$]

lemmas *ucast-revcast = revcast-rev-ucast* [*THEN word-rev-gal', standard*]
lemmas *ucast-rev-revcast = revcast-ucast* [*THEN word-rev-gal', standard*]

— linking *revcast* and *cast* via *shift*

lemmas *wsst-TYs = source-size target-size word-size*

lemma *revcast-down-uu'*:
 $rc = \text{revcast} \implies \text{source-size } rc = \text{target-size } rc + n \implies$
 $rc (w :: 'a :: \text{len word}) = \text{ucast } (w \gg n)$
apply (*simp add: revcast-def'*)
apply (*rule word-bl.Rep-inverse'*)
apply (*rule trans, rule ucast-down-drop*)
prefer 2
apply (*rule trans, rule drop-shiftr*)

```

apply (auto simp: takefill-alt wsst-TYs)
done

```

lemma *revcast-down-us'*:

```

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'*:

```

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'*:

```

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

```

lemmas *revcast-down-uu* = refl [THEN *revcast-down-uu*']

lemmas *revcast-down-us* = refl [THEN *revcast-down-us*']

lemmas *revcast-down-su* = refl [THEN *revcast-down-su*']

lemmas *revcast-down-ss* = refl [THEN *revcast-down-ss*']

lemma *cast-down-rev*:

```

uc = ucast  $\implies$  source-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':
  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-shiffl [THEN trans])
apply (subst ucast-up-app)
apply (auto simp add: wsst-TYs)
done

```

```

lemmas revcast-up = refl [THEN revcast-up']

```

```

lemmas rc1 = revcast-up [THEN
  revcast-rev-ucast [symmetric, THEN trans, THEN word-rev-gal, symmetric]]
lemmas rc2 = revcast-down-uu [THEN
  revcast-rev-ucast [symmetric, THEN trans, THEN word-rev-gal, symmetric]]

```

```

lemmas ucast-up =
  rc1 [simplified rev-shiftr [symmetric] revcast-ucast [symmetric]]
lemmas ucast-down =
  rc2 [simplified rev-shiftr revcast-ucast [symmetric]]

```

13.19.4 Slices

```

lemma slice1-no-bin [simp]:
  slice1 n (number-of w :: 'b word) = of-bl (takefill False n (bin-to-bl (len-of
  TYPE('b :: len0)) w))
  by (simp add: slice1-def)

```

```

lemma slice-no-bin [simp]:
  slice n (number-of w :: 'b word) = of-bl (takefill False (len-of TYPE('b :: len0)
  - n)
  (bin-to-bl (len-of TYPE('b :: len0)) w))
  by (simp add: slice-def word-size)

```

```

lemma slice1-0 [simp] : slice1 n 0 = 0
  unfolding slice1-def by (simp add : to-bl-0)

```

```

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], standard*]

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*)

lemmas *slice-id* = *trans* [*OF* *ucast-slice* [*symmetric*] *ucast-id*]

lemma *revcast-slice1'*:

$rc = \text{revcast } w \implies \text{slice1 } (\text{size } rc) w = rc$

unfolding *slice1-def* *revcast-def'* **by** (*simp add* : *word-size*)

lemmas *revcast-slice1* = *refl* [*THEN* *revcast-slice1'*]

lemma *slice1-tf-tf'*:

$\text{to-bl } (\text{slice1 } n w :: 'a :: \text{len0 } \text{word}) =$

$\text{rev } (\text{takefill } \text{False } (\text{len-of } \text{TYPE}('a)) (\text{rev } (\text{takefill } \text{False } n (\text{to-bl } w))))$

unfolding *slice1-def* **by** (*rule* *word-rev-tf*)

lemmas *slice1-tf-tf* = *slice1-tf-tf'*

[*THEN* *word-bl.Rep-inverse'*, *symmetric*, *standard*]

lemma *rev-slice1*:

$n + k = \text{len-of } \text{TYPE}('a) + \text{len-of } \text{TYPE}('b) \implies$

$\text{slice1 } n (\text{word-reverse } w :: 'b :: \text{len0 } \text{word}) =$

$\text{word-reverse } (\text{slice1 } k w :: 'a :: \text{len0 } \text{word})$

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-slice1'*:

$\text{res} = \text{slice } n (\text{word-reverse } w) \implies n + k + \text{size } \text{res} = \text{size } w \implies$

$\text{res} = \text{word-reverse } (\text{slice } k w)$

apply (*unfold* *slice-def* *word-size*)

apply *clarify*

apply (*rule* *rev-slice1*)

apply *arith*

done

lemmas *rev-slice* = *refl* [*THEN* *rev-slice1'*, *unfolded* *word-size*]

lemmas *sym-notr* =

not-iff [*THEN* *iffD2*, *THEN* *not-sym*, *THEN* *not-iff* [*THEN* *iffD1*]]

— problem posed by TPHOLs referee: criterion for overflow of addition of signed integers

lemma *soft-test*:

$(\text{sint } (x :: 'a :: \text{len } \text{word}) + \text{sint } y = \text{sint } (x + y)) =$

```

      (((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

```

13.20 Split and cat

lemmas word-split-bin' = word-split-def

lemmas word-cat-bin' = word-cat-def

lemma word-rsplit-no:

```

(word-rsplit (number-of bin :: 'b :: len0 word) :: 'a word list) =
  map number-of (bin-rsplit (len-of TYPE('a :: len))
    (len-of TYPE('b), bintrunc (len-of TYPE('b)) bin))
apply (unfold word-rsplit-def word-no-wi)
apply (simp add: word-ubin.eq-norm)
done

```

lemmas word-rsplit-no-cl [simp] = word-rsplit-no
 [unfolded bin-rsplitl-def bin-rsplit-l [symmetric]]

lemma test-bit-cat:

```

wc = word-cat a b  $\implies$  wc !! n = (n < size wc &
  (if n < size b then b !! n else a !! (n - size b)))
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*: $word\text{-}cat\ a\ b = of\text{-}bl\ (to\text{-}bl\ a\ @\ to\text{-}bl\ b)$
apply (*unfold of-bl-def to-bl-def word-cat-bin'*)
apply (*simp add: bl-to-bin-app-cat*)
done

lemma *of-bl-append*:
 $(of\text{-}bl\ (xs\ @\ ys) :: 'a :: len\ word) = of\text{-}bl\ xs * 2^{length\ ys} + of\text{-}bl\ ys$
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] new-word-of-int-hom-syms*)
done

lemma *of-bl-False [simp]*:
 $of\text{-}bl\ (False\#xs) = of\text{-}bl\ xs$
by (*rule word-eqI*)
(auto simp add: test-bit-of-bl nth-append)

lemma *of-bl-True*:
 $(of\text{-}bl\ (True\#xs) :: 'a :: len\ word) = 2^{length\ xs} + of\text{-}bl\ xs$
by (*subst of-bl-append [where xs=[True], simplified]*)
(simp add: word-1-bl)

lemma *of-bl-Cons*:
 $of\text{-}bl\ (x\#xs) = of\text{-}bool\ x * 2^{length\ xs} + of\text{-}bl\ xs$
by (*cases x*) (*simp-all add: of-bl-True*)

lemma *split-uint-lem*: $bin\text{-}split\ n\ (uint\ (w :: 'a :: len0\ word)) = (a, b) \implies$
 $a = bintrunc\ (len\text{-}of\ TYPE('a) - n)\ a \ \&\ b = bintrunc\ (len\text{-}of\ TYPE('a))\ b$
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'*:
 $std = size\ c - size\ b \implies (word\text{-}split\ c = (a, b)) \implies$
 $(a = of\text{-}bl\ (take\ std\ (to\text{-}bl\ c)) \ \&\ b = of\text{-}bl\ (drop\ std\ (to\text{-}bl\ c)))$
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-uint-lem [THEN conjunct1]*)

```

apply (unfold word-size)
apply (cases len-of TYPE('a) >= len-of TYPE('b))
  defer
    apply (simp add: word-0-bl word-0-wi-Pls)
  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))) <->
  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) --> (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) ==>
  (∀ n::nat. b !! n <=> n < size b ∧ c !! n) ∧ (∀ m::nat. a !! m <=> m < size a
  ∧ c !! (m + size b))
  by (simp add: test-bit-split')

```

```

lemma test-bit-split-eq: word-split c = (a, b) <->
  ((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)
  ==>
  (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 ==> word-split w = (u, v) ==> word-cat u v = w
apply (rule word-eqI)
apply (drule test-bit-split)
apply (clarsimp simp add : test-bit-cat word-size)
apply safe
apply arith
done

```

```

lemmas word-cat-split-size =
  sym [THEN [2] word-cat-split-alt [symmetric], standard]

```

13.20.1 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'*: $wc = word-cat\ a\ b \implies size\ wc \geq size\ a + size\ b \implies slice\ (size\ b)\ wc = a$ **apply** *safe***apply** (*rule word-eqI*)**apply** (*simp add: nth-slice test-bit-cat word-size*)**done****lemmas** *slice-cat1* = *refl [THEN slice-cat1']***lemmas** *slice-cat2* = *trans [OF slice-id word-cat-id]***lemma** *cat-slices*: $a = slice\ n\ c \implies b = slice\ 0\ c \implies n = size\ b \implies$ $size\ a + size\ b \geq size\ c \implies word-cat\ a\ b = c$ **apply** *safe***apply** (*rule word-eqI*)**apply** (*simp add: nth-slice test-bit-cat word-size*)**apply** *safe***apply** *arith***done****lemma** *word-split-cat-alt*: $w = word-cat\ u\ v \implies size\ u + size\ v \leq size\ w \implies word-split\ w = (u, v)$ **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=number-of a***and** *b=number-of b,**unfolded to-bl-no-bin, standard]***lemmas** *word-split-bl-no-bin [simp]* =*word-split-bl-eq [where c=number-of c, unfolded to-bl-no-bin, standard]*

— this odd result arises from the fact that the statement of the result implies that the decoded words are of the same type, and therefore of the same length, as the original word

```

lemma word-rsplit-same: word-rsplit w = [w]
  unfolding word-rsplit-def by (simp add : bin-rsplit-all)

lemma word-rsplit-empty-iff-size:
  (word-rsplit w = []) = (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 = word-rsplit w  $\implies$  m < size (hd sw :: 'a :: len word)  $\implies$ 
    k < length sw  $\implies$  (rev sw ! k) !! m = (w !! (k * size (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, standard]

lemma nth-rcat-lem' [rule-format] :
  sw = size (hd wl :: 'a :: len word)  $\implies$  (ALL n. n < size wl * sw  $\longrightarrow$ 
    rev (concat (map to-bl wl)) ! n =
    rev (to-bl (rev wl ! (n div sw))) ! (n mod sw))
  apply (unfold word-size)
  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

lemmas *nth-rcat-lem* = *refl* [*THEN* *nth-rcat-lem'*, *unfolded word-size*]

lemma *test-bit-rcat*:

$sw = \text{size } (hd\ wl :: 'a :: \text{len } word) \implies rc = \text{word-rcat } wl \implies rc \text{ !! } n =$
 $(n < \text{size } rc \ \& \ n \text{ div } sw < \text{size } wl \ \& \ (\text{rev } wl) ! (n \text{ div } sw) \text{ !! } (n \text{ mod } sw))$

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* [*rule-format*] :

ALL *x*. *foldl* *op* + *x* *xs* = *foldr* *op* + (*x* # *xs*) (*0* :: '*a* :: *comm-monoid-add*)

by (*induct* *xs*) (*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'*:

$[u,v] = p \implies [su,sv] = q \implies \text{word-rsplit } u = su \implies$

$\text{word-rsplit } v = sv \implies \text{length } su = \text{length } sv$

apply (*unfold word-rsplit-def*)

apply (*auto simp add* : *bin-rsplit-len-indep*)

done

lemmas *word-rsplit-len-indep* = *word-rsplit-len-indep'* [*OF* *refl refl refl refl*]

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*: $\text{word-rcat } (\text{word-rsplit } w) = w$

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'*:

$\text{word-rcat } (ws :: 'a :: \text{len word list}) = \text{frcw} \implies$

$\text{size } \text{frcw} = \text{length } ws * \text{len-of } TYPE ('a) \implies$

$\text{size } (\text{hd } [\text{word-rsplit } \text{frcw}, ws]) = \text{size } ws$

apply (*clarsimp simp add : word-size length-word-rsplit-exp-size'*)

apply (*fast intro: given-quot-alt*)

done

lemmas *size-word-rsplit-rcat-size =*

size-word-rsplit-rcat-size' [simplified]

lemma *msreus*:

fixes *n::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'*:

$\text{word-rcat } (ws :: 'a :: \text{len word list}) = \text{frcw} \implies$

$\text{size } \text{frcw} = \text{length } ws * \text{len-of } TYPE ('a) \implies \text{word-rsplit } \text{frcw} = ws$

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*)

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

```

lemmas word-rsplit-rcat-size = refl [THEN word-rsplit-rcat-size¹]

13.21 Rotation

lemmas rotater-0' [simp] = rotater-def [where n = 0, simplified]

lemmas word-rot-defs = word-roti-def word-rotr-def word-rotl-def

lemma rotate-eq-mod:

```

  m mod length xs = n mod length xs  $\implies$  rotate m xs = rotate n xs
apply (rule box-equals)
  defer
  apply (rule rotate-conv-mod [symmetric])+
apply simp
done

```

lemmas rotate-egs [standard] =
 trans [OF rotate0 [THEN fun-cong] id-apply]
 rotate-rotate [symmetric]
 rotate-id
 rotate-conv-mod
 rotate-eq-mod

13.21.1 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*] : $\text{rotate1} (\text{rotater1 } l) = l$
unfolding *rotater1-def* **by** (*cases l*) *auto*

lemma *rotater1-rev'*: $\text{rotater1} (\text{rev } xs) = \text{rev} (\text{rotate1 } xs)$
apply (*cases xs*)
apply (*simp add : rotater1-def*)
apply (*simp add : rotate1-rl'*)
done

lemma *rotater-rev'*: $\text{rotater } n (\text{rev } xs) = \text{rev} (\text{rotate } n xs)$
unfolding *rotater-def* **by** (*induct n*) (*auto intro: rotater1-rev'*)

lemmas *rotater-rev = rotater-rev'* [**where** $xs = \text{rev } ys$, *simplified*, *standard*]

lemma *rotater-drop-take*:
 $\text{rotater } n xs =$
 $\text{drop} (\text{length } xs - n \text{ mod } \text{length } xs) xs @$
 $\text{take} (\text{length } xs - n \text{ mod } \text{length } xs) xs$
by (*clarsimp simp add : rotater-rev rotate-drop-take rev-take rev-drop*)

lemma *rotater-Suc* [*simp*] :
 $\text{rotater} (\text{Suc } n) xs = \text{rotater1} (\text{rotater } n xs)$
unfolding *rotater-def* **by** *auto*

lemma *rotate-inv-plus* [*rule-format*] :
 $\text{ALL } k. k = m + n \longrightarrow \text{rotater } k (\text{rotate } n xs) = \text{rotater } m xs \ \&$
 $\text{rotate } k (\text{rotater } n xs) = \text{rotate } m xs \ \&$
 $\text{rotater } n (\text{rotate } k xs) = \text{rotate } m xs \ \&$
 $\text{rotate } n (\text{rotater } k xs) = \text{rotater } m xs$
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*, *standard*]

lemmas *rotate-rl* [*simp*] =
 rotate-inv-eq [*THEN conjunct2*, *THEN conjunct1*, *standard*]

lemma *rotate-gal*: $(\text{rotater } n xs = ys) = (\text{rotate } n ys = xs)$
by *auto*

lemma *rotate-gal'*: $(ys = \text{rotater } n xs) = (xs = \text{rotate } n ys)$
by *auto*

lemma *length-rotater* [*simp*]:
 $\text{length} (\text{rotater } n xs) = \text{length } xs$
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*], *standard*]

lemmas *rrs1 = rrs0* [*THEN refl* [*THEN rev-iffD1*]]

lemmas *rotater-eqs = rrs1* [*simplified length-rotater*, *standard*]

lemmas *rotater-0 = rotater-eqs* (1)

lemmas *rotater-add = rotater-eqs* (2)

13.21.2 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*: $\text{rotater1} (\text{map } f \text{ } xs) = \text{map } f (\text{rotater1 } xs)$

unfolding *rotater1-def*

by (*cases xs*) (*auto simp add: last-map butlast-map*)

lemma *rotater-map*:

$\text{rotater } n (\text{map } f \text{ } xs) = \text{map } f (\text{rotater } n \text{ } xs)$

unfolding *rotater-def*

by (*induct n*) (*auto simp add : rotater1-map*)

lemma *but-last-zip* [*rule-format*] :

ALL ys. length xs = length ys \implies xs \sim [] \implies

last (zip xs ys) = (last xs, last ys) &

butlast (zip xs ys) = zip (butlast xs) (butlast ys)

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 \implies xs \sim [] \implies

last (map2 f xs ys) = f (last xs) (last ys) &

butlast (map2 f xs ys) = map2 f (butlast xs) (butlast ys)

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*:
 $length\ xs = length\ ys \implies$
 $rotate1\ (map2\ f\ xs\ ys) = map2\ f\ (rotate1\ xs)\ (rotate1\ ys)$
apply (*unfold map2-def*)
apply (*cases xs*)
apply (*cases ys, auto simp add : rotate1-def*)
done

lemmas *lth* = *box-equals* [*OF asm-rl length-rotate [symmetric]*
length-rotate [symmetric], *THEN rotate1-map2*]

lemma *rotate-map2*:
 $length\ xs = length\ ys \implies$
 $rotate\ n\ (map2\ f\ xs\ ys) = map2\ f\ (rotate\ n\ xs)\ (rotate\ n\ ys)$
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-eqs =
  blrs0 [simplified word-bl.Rep' word-bl.Rep-inject to-bl-rotl [symmetric]]

lemma to-bl-rotr:
  to-bl (word-rotr n w) = rotater n (to-bl w)
  by (simp add: word-bl.Abs-inverse' word-rotr-def)

lemmas brrs0 = rotater-eqs [THEN to-bl-rotr [THEN trans]]

lemmas word-rotr-eqs =
  brrs0 [simplified word-bl.Rep' word-bl.Rep-inject to-bl-rotr [symmetric]]

declare word-rotr-eqs (1) [simp]
declare word-rotl-eqs (1) [simp]

lemma
  word-rot-rl [simp]:
  word-rotl k (word-rotr k v) = v and
  word-rot-lr [simp]:
  word-rotr k (word-rotl k v) = v
  by (auto simp add: to-bl-rotr to-bl-rotl word-bl.Rep-inject [symmetric])

lemma
  word-rot-gal:
  (word-rotr n v = w) = (word-rotl n w = v) and
  word-rot-gal':
  (w = word-rotr n v) = (v = word-rotl n w)
  by (auto simp: to-bl-rotr to-bl-rotl word-bl.Rep-inject [symmetric]
    dest: sym)

lemma word-rotr-rev:
  word-rotr n w = word-reverse (word-rotl n (word-reverse w))
  by (simp add: 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, standard] =
  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: rdmods)
done

lemmas word-roti-conv-mod = word-roti-conv-mod' [unfolded word-size]

```

13.21.3 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, standard] = rotate-map rotater-map*

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', standard]*

lemmas *bl-word-rotr-dt = trans [OF to-bl-rotr rotater-drop-take,
simplified word-bl.Rep', standard]*

lemma *bl-word-roti-dt':*

n = nat ((- i) mod int (size (w :: 'a :: len word))) ⇒

to-bl (word-roti i w) = drop n (to-bl w) @ take n (to-bl w)

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], standard*]

lemmas *word-rotr-dt = bl-word-rotr-dt*
 [*THEN word-bl.Rep-inverse' [symmetric], standard*]

lemmas *word-roti-dt = bl-word-roti-dt*
 [*THEN word-bl.Rep-inverse' [symmetric], standard*]

lemma *word-rotx-0 [simp] : word-rotr i 0 = 0 & word-rotl i 0 = 0*
by (*simp add : word-rotr-dt word-rotl-dt to-bl-0 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=number-of w, unfolded to-bl-no-bin, standard]

lemmas *word-rotl-dt-no-bin' [simp] =*
word-rotl-dt [where w=number-of w, unfolded to-bl-no-bin, standard]

declare *word-roti-def [simp]*

13.22 Miscellaneous

declare *of-nat-2p [simp]*

lemma *word-int-cases:*

$$\llbracket \bigwedge n. \llbracket (x :: 'a :: \text{len } 0 \text{ word}) = \text{word-of-int } n; 0 \leq n; n < 2^{\text{len-of TYPE('a)}} \rrbracket \implies P \rrbracket$$

$$\implies P$$
by (*cases x rule: word-uint.Abs-cases*) (*simp add: uints-num*)

lemma *word-nat-cases [cases type: word]:*

$$\llbracket \bigwedge n. \llbracket (x :: 'a :: \text{len } \text{word}) = \text{of-nat } n; n < 2^{\text{len-of TYPE('a)}} \rrbracket \implies P \rrbracket$$

$$\implies P$$
by (*cases x rule: word-unat.Abs-cases*) (*simp add: unats-def*)

lemma *max-word-eq:*

$$(\text{max-word} :: 'a :: \text{len } \text{word}) = 2^{\text{len-of TYPE('a)}} - 1$$
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*:
 $word\text{-of-int } (2 \wedge len\text{-of } TYPE('a)) = (0::'a::len0 \text{ word})$
by (*subst word-uint.Abs-norm [symmetric]*)
(simp add: word-of-int-hom-syms)

lemma *word-pow-0*:
 $(2::'a::len \text{ word}) \wedge len\text{-of } TYPE('a) = 0$
proof –
have $word\text{-of-int } (2 \wedge len\text{-of } TYPE('a)) = (0::'a \text{ word})$
by (*rule word-of-int-2p-len*)
thus *?thesis* **by** (*simp add: word-of-int-2p*)
qed

lemma *max-word-wrap*: $x + 1 = 0 \implies x = max\text{-word}$
apply (*simp add: max-word-eq*)
apply *uint-arith*
apply *auto*
apply (*simp add: word-pow-0*)
done

lemma *max-word-minus*:
 $max\text{-word} = (-1::'a::len \text{ word})$
proof –
have $-1 + 1 = (0::'a \text{ word})$ **by** *simp*
thus *?thesis* **by** (*rule max-word-wrap [symmetric]*)
qed

lemma *max-word-bl [simp]*:
 $to\text{-bl } (max\text{-word}::'a::len \text{ word}) = replicate (len\text{-of } TYPE('a)) \text{ True}$
by (*subst max-word-minus to-bl-n1*) *+ simp*

lemma *max-test-bit [simp]*:
 $(max\text{-word}::'a::len \text{ word}) !! n = (n < len\text{-of } TYPE('a))$
by (*auto simp add: test-bit-bl word-size*)

lemma *word-and-max [simp]*:
 $x \text{ AND } max\text{-word} = x$
by (*rule word-eqI*) (*simp add: word-ops-nth-size word-size*)

lemma *word-or-max [simp]*:
 $x \text{ OR } max\text{-word} = max\text{-word}$
by (*rule word-eqI*) (*simp add: word-ops-nth-size word-size*)

lemma *word-ao-dist2*:
 $x \text{ AND } (y \text{ OR } z) = x \text{ AND } y \text{ OR } x \text{ AND } (z::'a::len0 \text{ word})$
by (*rule word-eqI*) (*auto simp add: word-ops-nth-size word-size*)

lemma *word-oa-dist2*:
 $x \text{ OR } y \text{ AND } z = (x \text{ OR } y) \text{ AND } (x \text{ OR } (z::'a::len0 \text{ word}))$

by (rule word-eqI) (auto simp add: word-ops-nth-size word-size)

lemma word-and-not [simp]:

$x \text{ AND } \text{NOT } x = (0::'a::\text{len } 0 \text{ word})$

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:

boolean (op AND) (op OR) bitNOT 0 max-word

apply (rule boolean.intro)

apply (rule word-bw-assocs)

apply (rule word-bw-assocs)

apply (rule word-bw-comms)

apply (rule word-bw-comms)

apply (rule word-ao-dist2)

apply (rule word-oa-dist2)

apply (rule word-and-max)

apply (rule word-log-esimps)

apply (rule word-and-not)

apply (rule word-or-not)

done

interpretation word-bool-alg:

boolean op AND op OR bitNOT 0 max-word

by (rule word-boolean)

lemma word-xor-and-or:

$x \text{ XOR } y = x \text{ AND } \text{NOT } y \text{ OR } \text{NOT } x \text{ AND } (y::'a::\text{len } 0 \text{ word})$

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]:

$(x::'a::\text{len } 0 \text{ word}) \gg 0 = x$

by (simp add: shiftr-bl)

lemma shiffl-x-0 [simp]:

$(x :: 'a :: \text{len } \text{word}) \ll 0 = x$

by (simp add: shiffl-t2n)

lemma *shiffl-1* [*simp*]:
 $(1::'a::\text{len word}) \ll n = 2^n$
by (*simp add: shiffl-t2n*)

lemma *uint-lt-0* [*simp*]:
 $\text{uint } x < 0 = \text{False}$
by (*simp add: linorder-not-less*)

lemma *shiftr1-1* [*simp*]:
 $\text{shiftr1 } (1::'a::\text{len word}) = 0$
by (*simp add: shiftr1-def word-0-alt*)

lemma *shiftr-1* [*simp*]:
 $(1::'a::\text{len word}) \gg n = (\text{if } n = 0 \text{ then } 1 \text{ else } 0)$
by (*induct n*) (*auto simp: shiftr-def*)

lemma *word-less-1* [*simp*]:
 $((x::'a::\text{len word}) < 1) = (x = 0)$
by (*simp add: word-less-nat-alt unat-0-iff*)

lemma *to-bl-mask*:
 $\text{to-bl } (\text{mask } n :: 'a::\text{len word}) =$
 $\text{replicate } (\text{len-of TYPE('a)} - n) \text{ False } @$
 $\text{replicate } (\text{min } (\text{len-of TYPE('a)}) n) \text{ True}$
by (*simp add: mask-bl word-rep-drop min-def*)

lemma *map-replicate-True*:
 $n = \text{length } xs \implies$
 $\text{map } (\lambda(x,y). x \ \& \ y) (\text{zip } xs (\text{replicate } n \text{ True})) = xs$
by (*induct xs arbitrary: n*) *auto*

lemma *map-replicate-False*:
 $n = \text{length } xs \implies \text{map } (\lambda(x,y). x \ \& \ y)$
 $(\text{zip } xs (\text{replicate } n \text{ False})) = \text{replicate } n \text{ False}$
by (*induct xs arbitrary: n*) *auto*

lemma *bl-and-mask*:
fixes $w :: 'a::\text{len word}$
fixes n
defines $n' \equiv \text{len-of TYPE('a)} - n$
shows $\text{to-bl } (w \ \text{AND} \ \text{mask } n) = \text{replicate } n' \text{ False } @ \text{drop } n' (\text{to-bl } w)$
proof –
note [*simp*] = *map-replicate-True map-replicate-False*
have $\text{to-bl } (w \ \text{AND} \ \text{mask } n) =$
 $\text{map2 } \text{op } \& (\text{to-bl } w) (\text{to-bl } (\text{mask } n::'a::\text{len word}))$
by (*simp add: bl-word-and*)
also
have $\text{to-bl } w = \text{take } n' (\text{to-bl } w) @ \text{drop } n' (\text{to-bl } w)$ **by** *simp*
also

```

have map2 op & ... (to-bl (mask n::'a::len word)) =
  replicate n' False @ drop n' (to-bl w)
unfolding to-bl-mask n'-def map2-def
by (subst zip-append) auto
finally
show ?thesis .
qed

```

```

lemma drop-rev-takefill:
  length xs ≤ n ⇒
  drop (n - length xs) (rev (takefill False n (rev xs))) = xs
by (simp add: takefill-alt rev-take)

```

```

lemma map-nth-0 [simp]:
  map (op !! (0::'a::len0 word)) xs = replicate (length xs) False
by (induct xs) auto

```

```

lemma uint-plus-if-size:
  uint (x + y) =
  (if uint x + uint y < 2^size x then
    uint x + uint y
  else
    uint x + uint y - 2^size x)
by (simp add: word-arith-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 :: len word
  shows (w ≠ 0) = (0 < w)
proof -
  have 0 ≤ w by (rule word-zero-le)
  thus ?thesis by (auto simp add: word-less-def)
qed

```

```

lemma max-lt:
  unat (max a b div c) = unat (max a b) div unat (c:: 'a :: 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*:

```

uint (x - y) =
  (if uint y ≤ uint x then
    uint x - uint y
  else
    uint x - uint y + 2size x)
by (simp add: word-arith-alts int-word-uint mod-sub-if-z
      word-size)

```

lemma *unat-sub*:

```

b ≤ a ⇒ unat (a - b) = unat a - unat b
by (simp add: unat-def uint-sub-if-size word-le-def nat-diff-distrib)

```

lemmas *word-less-sub1-numberof* [simp] =
word-less-sub1 [of number-of w, standard]

lemmas *word-le-sub1-numberof* [simp] =
word-le-sub1 [of number-of w, standard]

lemma *word-of-int-minus*:

```

word-of-int (2len-of TYPE('a) - i) = (word-of-int (-i)::'a::len word)

```

proof –

```

have x: 2len-of TYPE('a) - i = -i + 2len-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-eg*:

```

(x ::'z::len word) ≤ y = (x = y ∨ x < y)
by (auto simp add: word-less-def)

```

lemma *mod-plus-cong*:

```

assumes 1: (b::int) = b'
  and 2: x mod b' = x' mod b'
  and 3: y mod b' = y' mod b'

```

```

    and 4:  $x' + y' = z'$ 
  shows  $(x + y) \bmod b = z' \bmod b'$ 
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:  $(b::int) = b'$ 
    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 zmod-zsub-left-eq)
apply (subst zmod-zsub-right-eq)
apply (simp add: zmod-zsub-left-eq [symmetric] zmod-zsub-right-eq [symmetric])
done

```

lemma *word-induct-less*:

```

[[P (0::'a::len word);  $\bigwedge n. [n < m; P n] \implies P (1 + n)]] \implies P m$ 
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*:

```

[[P (0::'a::len word);  $\bigwedge n. P n \implies P (1 + n)]] \implies P m$ 
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

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*:

$f\ (\text{word-rec } z\ (\lambda-. f)\ n) = \text{word-rec } (f\ z)\ (\lambda-. f)\ n$
by (*induct n*) (*simp-all add: word-rec-0 word-rec-Suc*)

lemma *word-rec-in2*:

$f\ n\ (\text{word-rec } z\ f\ n) = \text{word-rec } (f\ 0\ z)\ (f\ \circ\ \text{op} + 1)\ n$
by (*induct n*) (*simp-all add: word-rec-0 word-rec-Suc*)

lemma *word-rec-twice*:

$m \leq n \implies \text{word-rec } z\ f\ n = \text{word-rec } (\text{word-rec } z\ f\ (n - m))\ (f\ \circ\ \text{op} + (n - m))\ m$

```

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  $f = \text{word-rec } ?a ?b$  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 (drule-tac  $x=x \circ op + 1$  in spec)
apply (drule-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: $\text{word-rec } z (\lambda \cdot. \text{id}) n = z$
by (induct n) (auto simp add: word-rec-0 word-rec-Suc)

lemma word-rec-id-eq: $\forall m < n. f m = \text{id} \implies \text{word-rec } z f n = z$
apply (erule rev-mp)
apply (induct n)
apply (auto simp add: word-rec-0 word-rec-Suc)
apply (drule spec, erule mp)
apply uint-arith
apply (drule-tac $x=n$ **in** spec, erule impE)
apply uint-arith
apply simp
done

lemma word-rec-max:
 $\forall m \geq n. m \neq -1 \implies f m = \text{id} \implies \text{word-rec } z f -1 = \text{word-rec } z f n$
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

lemmas word-no-1 [simp] = word-1-no [symmetric, unfolded BIT-simps]
lemmas word-no-0 [simp] = word-0-no [symmetric]

declare word-0-bl [simp]
declare bin-to-bl-def [simp]
declare to-bl-0 [simp]
declare of-bl-True [simp]

use ~/src/HOL/Word/Tools/smt-word.ML

setup « SMT-Word.setup »

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.