

# Bitwise Operations and Modular Arithmetic

Brian Huffman

December 14, 2006

## Contents

<b>1</b>	<b>Numeral Syntax for Types</b>	<b>2</b>
1.1	Type constructors . . . . .	2
1.2	Syntax . . . . .	2
1.3	Types with <i>int</i> values . . . . .	3
1.4	Types with <i>nat</i> values . . . . .	5
<b>2</b>	<b>Integers mod n</b>	<b>5</b>
2.1	Locale for modular arithmetic subtypes . . . . .	6
2.2	Type Constructor for Integers mod n . . . . .	8
<b>3</b>	<b>Boolean Algebras</b>	<b>10</b>
<b>4</b>	<b>Integers as an Inductive Datatype</b>	<b>15</b>
4.1	Inductive property of BIT . . . . .	15
4.2	Injectivity of BIT . . . . .	16
4.3	Testing bit positions . . . . .	18
4.4	Syntactic class for bitwise operations . . . . .	20
4.5	Bitwise operations on type <i>bit</i> . . . . .	20
4.6	Bitwise operations on type <i>int</i> . . . . .	21
4.7	Bitwise ops with <i>getbit</i> , <i>lsb</i> , <i>div2</i> . . . . .	24
4.8	Axiomatic Class for Bit-Vector Types . . . . .	25
4.9	Integers modulo powers of 2 . . . . .	27
4.10	<i>bit-ring</i> instance . . . . .	30

# 1 Numeral Syntax for Types

```
theory NumeralType imports Main begin
```

## 1.1 Type constructors

Define types without any constants

(`typedecl` would work just as well)

```
typedef pls = UNIV::unit set ..
typedef min = UNIV::unit set ..
typedef 'a bit0 = UNIV::unit set ..
typedef 'a bit1 = UNIV::unit set ..
```

## 1.2 Syntax

**syntax**

```
-NumeralType :: num-const  $\Rightarrow$  type (-)
-NumeralType0 :: type (0)
-NumeralType1 :: type (1)
```

**translations**

```
-NumeralType1  $\Rightarrow$  (type) pls bit1
-NumeralType0  $\Rightarrow$  (type) pls
(type) -1  $\leq$  (type) min
```

**parse-translation**  $\ll$

*let*

```
val pls-const = Syntax.const NumeralType.pls;
val min-const = Syntax.const NumeralType.min;
val B0-const = Syntax.const NumeralType.bit0;
val B1-const = Syntax.const NumeralType.bit1;
```

*fun mk-bintype n =*

*let*

*fun mk-bit n = if n = 0 then B0-const else B1-const;*

*fun bin-of n =*

*if n = 0 then pls-const*

*else if n = ~1 then min-const*

*else*

*let val (q, r) = IntInf.divMod (n, 2);*

*in mk-bit r \$ bin-of q end;*

*in bin-of n end;*

*fun numeral-tr (\*-NumeralType\*) [Const (str, -)] =*

*mk-bintype (valOf (IntInf.fromString str))*

*| numeral-tr (\*-NumeralType\*) ts = raise TERM (numeral-tr, ts);*

```

in [(-NumeralType, numeral-tr)] end;
>>

print-translation <<
let
fun prefix-len - [] = 0
  | prefix-len pred (x :: xs) =
    if pred x then 1 + prefix-len pred xs else 0;

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

fun bin-of (Const (pls, -)) = []
  | bin-of (Const (min, -)) = [~1]
  | bin-of (Const (bit0, -) $ bs) = 0 :: bin-of bs
  | bin-of (Const (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 (sign, zs) =
  (case rev rev-digs of
   ~1 :: bs => (-, prefix-len (equal 1) bs)
  | bs => (, prefix-len (equal 0) bs));
val i = int-of rev-digs;
val num = IntInf.toString (IntInf.abs i);
val str = sign ^ implode (replicate zs 0) ^ num;
in
  Syntax.const -NumeralType $ Syntax.free str
end
| bit-tr' b - = raise Match;

in [(bit0, bit-tr' 0), (bit1, bit-tr' 1)] end;
>>

```

### 1.3 Types with *int* values

```
axclass numeral < type
```

```
consts bin-of-type :: 'a::numeral itself => int
```

```
instance pls :: numeral ..
```

```
instance min :: numeral ..
```

```
instance bit0 :: (numeral) numeral ..
```

```
instance bit1 :: (numeral) numeral ..
```

```
defs (overloaded)
```

```
  bin-of-type-pls-def:
```

*bin-of-type* (*t::pls* *itself*)  $\equiv$  *Numeral.Pls*

*bin-of-type-min-def*:  
*bin-of-type* (*t::min* *itself*)  $\equiv$  *Numeral.Min*

*bin-of-type-bit0-def*:  
*bin-of-type* (*t::'a::numeral bit0* *itself*)  $\equiv$   
*bin-of-type* *TYPE('a)* *BIT* *bit.B0*

*bin-of-type-bit1-def*:  
*bin-of-type* (*t::'a::numeral bit1* *itself*)  $\equiv$   
*bin-of-type* *TYPE('a)* *BIT* *bit.B1*

**lemmas** *bin-of-type-defs* [*simp*] =  
*bin-of-type-pls-def*  
*bin-of-type-min-def*  
*bin-of-type-bit0-def*  
*bin-of-type-bit1-def*

**definition**

*int-of-type* :: *'a::numeral* *itself*  $\Rightarrow$  *int* **where**  
*int-of-type* *t* = *number-of* (*bin-of-type* *t*)

**axclass** *numeral0* < *numeral*  
*zero-le-int-of-type*:  $0 \leq$  *int-of-type* *TYPE('a::numeral)*

**axclass** *numeral1* < *numeral*  
*one-le-int-of-type*:  $1 \leq$  *int-of-type* *TYPE('a::numeral)*

**axclass** *numeral2* < *numeral*  
*two-le-int-of-type*:  $2 \leq$  *int-of-type* *TYPE('a::numeral)*

**lemma** *zero-less-int-of-type*:  $0 <$  *int-of-type* *TYPE('a::numeral1)*  
**apply** (*rule* *int-one-le-iff-zero-less* [*THEN* *iffD1*])  
**apply** (*rule* *one-le-int-of-type*)  
**done**

**lemma** *one-less-int-of-type*:  $1 <$  *int-of-type* *TYPE('a::numeral2)*  
**by** (*cut-tac* *'a = 'a* **in** *two-le-int-of-type*, *simp*)

**instance** *numeral1* < *numeral0*

**proof**

**have**  $0 <$  *int-of-type* *TYPE('a)* **by** (*rule* *zero-less-int-of-type*)  
**thus**  $0 \leq$  *int-of-type* *TYPE('a)* **by** (*rule* *order-less-imp-le*)

**qed**

**instance** *numeral2* < *numeral1*

**proof**

**have**  $1 <$  *int-of-type* *TYPE('a)* **by** (*rule* *one-less-int-of-type*)

**thus**  $1 \leq \text{int-of-type } \text{TYPE}('a)$  **by** (rule order-less-imp-le)  
**qed**

**instance** *pls* :: numeral0  
**by** *intro-classes* (simp add: int-of-type-def)

**instance** *bit0* :: (numeral0) numeral0  
**proof**  
**have**  $0 \leq \text{int-of-type } \text{TYPE}('a)$  **by** (rule zero-le-int-of-type)  
**thus**  $0 \leq \text{int-of-type } \text{TYPE}('a \text{ bit0})$  **by** (simp add: int-of-type-def)  
**qed**

**instance** *bit0* :: (numeral1) numeral2  
**proof**  
**have**  $1 \leq \text{int-of-type } \text{TYPE}('a)$  **by** (rule one-le-int-of-type)  
**thus**  $2 \leq \text{int-of-type } \text{TYPE}('a \text{ bit0})$  **by** (simp add: int-of-type-def)  
**qed**

**instance** *bit1* :: (numeral0) numeral1  
**proof**  
**have**  $0 \leq \text{int-of-type } \text{TYPE}('a)$  **by** (rule zero-le-int-of-type)  
**thus**  $1 \leq \text{int-of-type } \text{TYPE}('a \text{ bit1})$  **by** (simp add: int-of-type-def)  
**qed**

**instance** *bit1* :: (numeral1) numeral2  
**proof**  
**have**  $1 \leq \text{int-of-type } \text{TYPE}('a)$  **by** (rule one-le-int-of-type)  
**thus**  $2 \leq \text{int-of-type } \text{TYPE}('a \text{ bit1})$  **by** (simp add: int-of-type-def)  
**qed**

## 1.4 Types with *nat* values

### definition

*nat-of-type* :: 'a::numeral0 itself  $\Rightarrow$  nat **where**  
*nat-of-type* t = number-of (bin-of-type t)

**lemma** *nat-of-type-eq*: *nat-of-type* t = nat (int-of-type t)  
**by** (simp add: nat-of-type-def int-of-type-def)

**lemma** *zero-less-nat-of-type*:

$0 < \text{nat-of-type } \text{TYPE}('a::\text{numeral1})$   
**by** (simp add: nat-of-type-eq zero-less-int-of-type)

**end**

## 2 Integers mod n

**theory** *IntMod*

```

imports NumeralType
begin

```

## 2.1 Locale for modular arithmetic subtypes

```

locale mod-type =
  fixes n :: int
  and Rep :: 'a::{zero,one,plus,times,minus,power} ⇒ int
  and Abs :: int ⇒ 'a::{zero,one,plus,times,minus,power}
  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)
  and power-def: x ^ k ≡ Abs (Rep x ^ k mod n)

```

```

lemma (in mod-type) size0: 0 < n
by (cut-tac size1, simp)

```

```

lemmas (in mod-type) definitions =
  zero-def one-def add-def mult-def minus-def diff-def power-def

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma (in mod-type) Rep-Abs-1: Rep (Abs 1) = 1

```

**by** (*simp add: Abs-inverse size1*)

**lemma** (*in mod-type*) *Rep-1: Rep 1 = 1*  
**by** (*simp add: one-def Rep-Abs-1*)

**lemma** (*in mod-type*) *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**

The following 3 lemmas belong in Integ/IntDiv.thy

**lemma** *zminus-zmod: - ((x::int) mod m) mod m = - x mod m*  
**by** (*simp only: zmod-zminus1-eq-if mod-mod-trivial*)

**lemma** *zdiff-zmod-left: (x mod m - y) mod m = (x - y) mod (m::int)*  
**by** (*simp only: diff-def zmod-zadd-left-eq [symmetric]*)

**lemma** *zdiff-zmod-right: (x - y mod m) mod m = (x - y) mod (m::int)*  
**proof** -  
  **have**  $(x + - (y \text{ mod } m) \text{ mod } m) \text{ mod } m = (x + - y \text{ mod } m) \text{ mod } m$   
    **by** (*simp only: zminus-zmod*)  
  **hence**  $(x + - (y \text{ mod } m)) \text{ mod } m = (x + - y) \text{ mod } m$   
    **by** (*simp only: zmod-zadd-right-eq [symmetric]*)  
  **thus**  $(x - y \text{ mod } m) \text{ mod } m = (x - y) \text{ mod } m$   
    **by** (*simp only: diff-def*)  
**qed**

**lemmas** *zmod-simps =*  
  *IntDiv.zmod-zadd-left-eq [symmetric]*  
  *IntDiv.zmod-zadd-right-eq [symmetric]*  
  *IntDiv.zmod-zmult1-eq [symmetric]*  
  *IntDiv.zmod-zmult1-eq' [symmetric]*  
  *IntDiv.zpower-zmod*  
  *zminus-zmod zdiff-zmod-left zdiff-zmod-right*

**lemmas** (*in mod-type*) *Rep-simps =*  
  *Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1*

**lemma** (*in mod-type*) *comm-ring-1: OFCLASS('a, comm-ring-1-class)*  
**apply** (*intro-classes, unfold definitions*)  
**apply** (*simp-all add: Rep-simps zmod-simps add-ac mult-ac ring-distrib*)  
**done**

**lemma** (*in mod-type*) *recpower: OFCLASS('a, recpower-class)*  
**apply** (*intro-classes, unfold definitions*)  
**apply** (*simp-all add: Rep-simps zmod-simps add-ac mult-assoc*  
  *mod-pos-pos-trivial size1*)  
**done**

```

locale (open) mod-ring = mod-type +
  constrains n :: int
  and Rep :: 'a::{number-ring,power} ⇒ int
  and Abs :: int ⇒ 'a::{number-ring,power}

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

lemma (in mod-ring) cases:
  assumes 1:  $\bigwedge z. \llbracket (x::'a) = \text{of-int } z; 0 \leq z; z < n \rrbracket \implies P$ 
  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 (in mod-ring) induct:
  ( $\bigwedge z. \llbracket 0 \leq z; z < n \rrbracket \implies P (\text{of-int } z)$ )  $\implies P (x::'a)$ 
by (cases x rule: cases simp)

```

## 2.2 Type Constructor for Integers mod n

```

typedef (open) 'a intmod = {0..int-of-type TYPE('a::numeral2)}
by (rule-tac x=0 in exI, simp add: zero-less-int-of-type)

constdefs
  Abs-intmod' :: int ⇒ 'a::numeral2 intmod
  Abs-intmod' x ≡ Abs-intmod (x mod (int-of-type TYPE('a)))

instance intmod :: (numeral1) {zero,one,plus,times,minus,power} ..

defs (overloaded)

```

*zero-intmod-def*:  $0 \equiv \text{Abs-intmod } 0$   
*one-intmod-def*:  $1 \equiv \text{Abs-intmod } 1$   
*add-intmod-def*:  $x + y \equiv \text{Abs-intmod}' (\text{Rep-intmod } x + \text{Rep-intmod } y)$   
*mult-intmod-def*:  $x * y \equiv \text{Abs-intmod}' (\text{Rep-intmod } x * \text{Rep-intmod } y)$   
*diff-intmod-def*:  $x - y \equiv \text{Abs-intmod}' (\text{Rep-intmod } x - \text{Rep-intmod } y)$   
*minus-intmod-def*:  $- x \equiv \text{Abs-intmod}' (- \text{Rep-intmod } x)$   
*power-intmod-def*:  $x ^ k \equiv \text{Abs-intmod}' (\text{Rep-intmod } x ^ k)$

**interpretation** *intmod*:

*mod-type* [*int-of-type* *TYPE*('a::numeral2)  
     *Rep-intmod*::'a::numeral2 *intmod*  $\Rightarrow$  *int*  
     *Abs-intmod*::*int*  $\Rightarrow$  'a::numeral2 *intmod*]  
**apply** (*rule mod-type.intro*)  
**apply** (*rule type-definition-intmod*)  
**apply** (*rule one-less-int-of-type*)  
**apply** (*rule zero-intmod-def*)  
**apply** (*rule one-intmod-def*)  
**apply** (*rule add-intmod-def* [*unfolded Abs-intmod'-def*])  
**apply** (*rule mult-intmod-def* [*unfolded Abs-intmod'-def*])  
**apply** (*rule diff-intmod-def* [*unfolded Abs-intmod'-def*])  
**apply** (*rule minus-intmod-def* [*unfolded Abs-intmod'-def*])  
**apply** (*rule power-intmod-def* [*unfolded Abs-intmod'-def*])  
**done**

**instance** *intmod* :: (numeral2) *comm-ring-1*  
**by** (*rule intmod.comm-ring-1*)

**instance** *intmod* :: (numeral2) *recpower*  
**by** (*rule intmod.recpower*)

**instance** *intmod* :: (numeral2) *number ..*

**defs** (**overloaded**)

*number-intmod-def*: *number-of* *w*::'a::numeral2 *intmod*  $\equiv$  *of-int* *w*

**instance** *intmod* :: (numeral2) *number-ring*  
**by** *intro-classes* (*simp only: number-intmod-def*)

**interpretation** *intmod*:

*mod-ring* [*int-of-type* *TYPE*('a::numeral2)  
     *Rep-intmod*::'a::numeral2 *intmod*  $\Rightarrow$  *int*  
     *Abs-intmod*::*int*  $\Rightarrow$  'a::numeral2 *intmod*] .

**lemmas** *intmod-cases* [*cases type: intmod, case-names of-int*] =  
*intmod.cases* [*unfolded int-of-type-def*]

**lemmas** *intmod-induct* [*induct type: intmod, case-names of-int*] =  
*intmod.induct* [*unfolded int-of-type-def*]

```

lemma intmod-iszero-number-of [unfolded int-of-type-def, simp]:
  iszero (number-of w::'a::numeral2 intmod) =
    (number-of w mod int-of-type TYPE('a) = 0)
by (rule intmod.iszero-number-of)

```

```

syntax
  intmod :: type  $\Rightarrow$  type (Z[-] [1000] 1000)

```

```

syntax (xsymbols)
  intmod :: type  $\Rightarrow$  type (Z. [1000] 1000)

```

```

end

```

### 3 Boolean Algebras

```

theory BooleanAlgebra
imports Main
begin

```

```

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

```

```

lemmas (in boolean) disj-ac =
  disj-assoc disj-commute
  mk-left-commute [of disj, OF disj-assoc disj-commute]

```

```

lemmas (in boolean) conj-ac =
  conj-assoc conj-commute
  mk-left-commute [of conj, OF conj-assoc conj-commute]

```

```

lemma (in boolean) 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

```

### Complement

**lemma** (in *boolean*) *complement-unique*:

```

assumes 1:  $a \sqcap x = \mathbf{0}$ 
assumes 2:  $a \sqcup x = \mathbf{1}$ 
assumes 3:  $a \sqcap y = \mathbf{0}$ 
assumes 4:  $a \sqcup y = \mathbf{1}$ 
shows  $x = y$ 
proof –
have  $(a \sqcap x) \sqcup (x \sqcap y) = (a \sqcap y) \sqcup (x \sqcap y)$  using 1 3 by simp
hence  $(x \sqcap a) \sqcup (x \sqcap y) = (y \sqcap a) \sqcup (y \sqcap x)$  using conj-commute by simp
hence  $x \sqcap (a \sqcup y) = y \sqcap (a \sqcup x)$  using conj-disj-distrib by simp
hence  $x \sqcap \mathbf{1} = y \sqcap \mathbf{1}$  using 2 4 by simp
thus  $x = y$  using conj-one-right by simp
qed

```

**lemma** (in *boolean*) *compl-unique*:  $\llbracket x \sqcap y = \mathbf{0}; x \sqcup y = \mathbf{1} \rrbracket \implies \sim x = y$   
**by** (rule *complement-unique* [*OF conj-cancel-right disj-cancel-right*])

**lemma** (in *boolean*) *double-compl*:  $\sim(\sim x) = x$

```

proof (rule compl-unique)
from conj-cancel-right show  $\sim x \sqcap x = \mathbf{0}$  by (simp add: conj-commute)
from disj-cancel-right show  $\sim x \sqcup x = \mathbf{1}$  by (simp add: disj-commute)
qed

```

**lemma** (in *boolean*) *compl-eq-compl-iff*:  $(\sim x = \sim y) = (x = y)$   
**by** (rule *inj-eq* [*OF inj-on-inverseI*], rule *double-compl*)

### Conjunction

**lemma** (in *boolean*) *conj-absorb*:  $x \sqcap x = x$

```

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
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** (in *boolean*) *conj-zero-right*:  $x \sqcap \mathbf{0} = \mathbf{0}$

**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*

**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** (*in boolean*) *compl-one*:  $\sim \mathbf{1} = \mathbf{0}$

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

**lemma** (*in boolean*) *conj-zero-left*:  $\mathbf{0} \sqcap x = \mathbf{0}$

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

**lemma** (*in boolean*) *conj-one-left*:  $\mathbf{1} \sqcap x = x$

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

**lemma** (*in boolean*) *conj-cancel-left*:  $\sim x \sqcap x = \mathbf{0}$

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

**lemma** (*in boolean*) *conj-left-absorb*:  $x \sqcap (x \sqcap y) = x \sqcap y$

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

**lemma** (*in boolean*) *conj-disj-distrib2*:

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

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

**lemmas** (*in boolean*) *conj-disj-distrib* =

*conj-disj-distrib conj-disj-distrib2*

Disjunction

**lemma** (*in boolean*) *disj-absorb*:  $x \sqcup x = x$

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

**lemma** (*in boolean*) *disj-one-right*:  $x \sqcup \mathbf{1} = \mathbf{1}$

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

**lemma** (*in boolean*) *compl-zero*:  $\sim \mathbf{0} = \mathbf{1}$

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

**lemma** (*in boolean*) *disj-zero-left*:  $\mathbf{0} \sqcup x = x$

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

**lemma** (*in boolean*) *disj-one-left*:  $\mathbf{1} \sqcup x = \mathbf{1}$

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

**lemma** (*in boolean*) *disj-cancel-left*:  $\sim x \sqcup x = \mathbf{1}$

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

**lemma** (in *boolean*) *disj-left-absorb*:  $x \sqcup (x \sqcup y) = x \sqcup y$   
**by** (rule *boolean.conj-left-absorb [OF dual]*)

**lemma** (in *boolean*) *disj-conj-distrib2*:  
 $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$   
**by** (rule *boolean.conj-disj-distrib2 [OF dual]*)

**lemmas** (in *boolean*) *disj-conj-distrib* =  
*disj-conj-distrib disj-conj-distrib2*

De Morgan's Laws

**lemma** (in *boolean*) *de-Morgan-conj*:  $\sim (x \sqcap y) = \sim x \sqcup \sim y$

**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 add: conj-ac*)

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

**by** (*simp add: 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 add: disj-ac*)

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

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

**qed**

**lemma** (in *boolean*) *de-Morgan-disj*:  $\sim (x \sqcup y) = \sim x \sqcap \sim y$   
**by** (rule *boolean.de-Morgan-conj [OF dual]*)

Symmetric Difference

**locale** *boolean-xor* = *boolean* +

**fixes** *xor* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a (**infixr**  $\oplus$  65)

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

**lemma** (in *boolean-xor*) *xor-def2*:

$x \oplus y = (x \sqcup y) \sqcap (\sim x \sqcup \sim y)$

**by** (*simp add: xor-def conj-disj-distrib*)

*disj-ac conj-ac conj-cancel-right disj-zero-left*)

**lemma** (in *boolean-xor*) *xor-commute*:  $x \oplus y = y \oplus x$

**by** (*simp add: xor-def conj-commute disj-commute*)

**lemma** (in *boolean-xor*) *xor-assoc*:  $(x \oplus y) \oplus z = x \oplus (y \oplus z)$

**proof** –

**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 ⊔ (x ⊓ y ⊓ ~ y) ⊔ (x ⊓ z ⊓ ~ z)
    by (simp add: conj-cancel-right conj-zero-right)
  thus (x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)
    apply (simp add: xor-def de-Morgan-disj de-Morgan-conj double-compl)
    apply (simp add: conj-disj-distrib conj-ac disj-ac)
  done
qed

lemmas (in boolean-xor) xor-ac =
  xor-assoc xor-commute
  mk-left-commute [of xor, OF xor-assoc xor-commute]

lemma (in boolean-xor) xor-zero-right: x ⊕ 0 = x
by (simp add: xor-def compl-zero conj-one-right conj-zero-right disj-zero-right)

lemma (in boolean-xor) xor-zero-left: 0 ⊕ x = x
by (subst xor-commute) (rule xor-zero-right)

lemma (in boolean-xor) xor-one-right: x ⊕ 1 = ~ x
by (simp add: xor-def compl-one conj-zero-right conj-one-right disj-zero-left)

lemma (in boolean-xor) xor-one-left: 1 ⊕ x = ~ x
by (subst xor-commute) (rule xor-one-right)

lemma (in boolean-xor) xor-self: x ⊕ x = 0
by (simp add: xor-def conj-cancel-right conj-cancel-left disj-zero-right)

lemma (in boolean-xor) xor-left-self: x ⊕ (x ⊕ y) = y
by (simp add: xor-assoc [symmetric] xor-self xor-zero-left)

lemma (in boolean-xor) xor-compl-left: ~ x ⊕ y = ~ (x ⊕ y)
  apply (simp add: xor-def de-Morgan-disj de-Morgan-conj double-compl)
  apply (simp add: conj-disj-distrib)
  apply (simp add: conj-cancel-right conj-cancel-left)
  apply (simp add: disj-zero-left disj-zero-right)
  apply (simp add: disj-ac conj-ac)
done

lemma (in boolean-xor) xor-compl-right: x ⊕ ~ y = ~ (x ⊕ y)
  apply (simp add: xor-def de-Morgan-disj de-Morgan-conj double-compl)
  apply (simp add: conj-disj-distrib)
  apply (simp add: conj-cancel-right conj-cancel-left)
  apply (simp add: disj-zero-left disj-zero-right)
  apply (simp add: disj-ac conj-ac)
done

lemma (in boolean-xor) xor-cancel-right: x ⊕ ~ x = 1
by (simp add: xor-compl-right xor-self compl-zero)

```

**lemma** (in *boolean-xor*) *xor-cancel-left*:  $\sim x \oplus x = \mathbf{1}$   
 by (*subst xor-commute*) (*rule xor-cancel-right*)

**lemma** (in *boolean-xor*) *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 add: 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) add:*  
*xor-def de-Morgan-disj de-Morgan-conj double-compl*  
*conj-disj-distrib conj-ac disj-ac*)

**qed**

**lemma** (in *boolean-xor*) *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 add: conj-commute*)

**qed**

**lemmas** (in *boolean-xor*) *conj-xor-distrib* =  
*conj-xor-distrib conj-xor-distrib2*

**end**

## 4 Integers as an Inductive Datatype

**theory** *BinInduct* **imports** *Main* **begin**

### 4.1 Inductive property of BIT

**consts** *Numerals* :: *int set*

**inductive** *Numerals*

**intros**

*Numerals-Pls*: *Numeral.Pl* ∈ *Numerals*

*Numerals-Min*: *Numeral.Min* ∈ *Numerals*

*Numerals-B0*:  $z \in \text{Numerals} \implies z \text{ BIT bit.B0} \in \text{Numerals}$

*Numerals-B1*:  $z \in \text{Numerals} \implies z \text{ BIT bit.B1} \in \text{Numerals}$

**lemma** *Numerals-succ*:  $z \in \text{Numerals} \implies \text{Numeral.succ } z \in \text{Numerals}$   
**by** (*erule Numerals.induct, simp-all add: Numerals.intros*)

**lemma** *Numerals-pred*:  $z \in \text{Numerals} \implies \text{Numeral.pred } z \in \text{Numerals}$   
**by** (*erule Numerals.induct, simp-all add: Numerals.intros*)

**lemma** *Numerals-uminus*:  $z \in \text{Numerals} \implies \text{uminus } z \in \text{Numerals}$   
**by** (*erule Numerals.induct*, *simp-all add: Numerals.intros Numerals-pred*)

**lemma** *Numerals-int*:  $\text{int } n \in \text{Numerals}$   
**apply** (*induct n*)  
**apply** (*simp add: Numerals-Pls [unfolded Numeral.Pl-def]*)  
**apply** (*drule Numerals-succ [unfolded Numeral.succ-def]*)  
**apply** (*simp add: add-commute*)  
**done**

**lemma** *mem-Numerals*:  $z \in \text{Numerals}$   
**by** (*induct z*) (*simp-all only: Numerals-int Numerals-uminus*)

**lemma** *BIT-induct* [*case-names Pls Min B0 B1*]:  
**assumes** *Pls*:  $P \text{ Numeral.Pl}$   
**assumes** *Min*:  $P \text{ Numeral.Min}$   
**assumes** *B0*:  $\bigwedge x. \llbracket P x; x \neq \text{Numeral.Pl} \rrbracket \implies P (x \text{ BIT bit.B0})$   
**assumes** *B1*:  $\bigwedge x. \llbracket P x; x \neq \text{Numeral.Min} \rrbracket \implies P (x \text{ BIT bit.B1})$   
**shows**  $P x$   
**apply** (*induct x rule: Numerals.induct [OF mem-Numerals]*)  
**apply** (*rule Pls*)  
**apply** (*rule Min*)  
**apply** (*case-tac z = Numeral.Pl, simp*)  
**apply** (*erule (1) B0*)  
**apply** (*case-tac z = Numeral.Min, simp*)  
**apply** (*erule (1) B1*)  
**done**

## 4.2 Injectivity of BIT

**lemma** *BIT-inject*:  $x \text{ BIT } a = y \text{ BIT } b \implies x = y \wedge a = b$   
**apply** (*drule-tac f=number-of :: int  $\Rightarrow$  int in arg-cong*)  
**apply** (*case-tac a, case-tac [!] b*)  
**apply** (*simp-all add: succ-def pred-def*)  
**apply** (*simp-all add: number-of-is-id iszero-def*)  
**done**

**lemma** *BIT-eq*:  $(x \text{ BIT } a = y \text{ BIT } b) = (x = y \wedge a = b)$   
**by** (*safe dest!: BIT-inject*)

**lemma** *BIT-eq-Pls*:  $(w \text{ BIT } b = \text{Numeral.Pl}) = (w = \text{Numeral.Pl} \wedge b = \text{bit.B0})$   
**by** (*subst Pls-0-eq [symmetric]*, *simp only: BIT-eq*)

**lemma** *BIT-eq-Min*:  $(w \text{ BIT } b = \text{Numeral.Min}) = (w = \text{Numeral.Min} \wedge b = \text{bit.B1})$   
**by** (*subst Min-1-eq [symmetric]*, *simp only: BIT-eq*)

**lemma** *Pls-eq-BIT*:  $(\text{Numeral.Pl} = w \text{ BIT } b) = (w = \text{Numeral.Pl} \wedge b = \text{bit.B0})$   
**by** (*subst eq-commute*, *rule BIT-eq-Pls*)

**lemma** *Min-eq-BIT*:  $(\text{Numeral.Min} = w \text{ BIT } b) = (w = \text{Numeral.Min} \wedge b = \text{bit.B1})$

**by** (*subst eq-commute*, *rule BIT-eq-Min*)

**lemma** *Min-neq-Pls*:  $\text{Numeral.Min} \neq \text{Numeral.Pl}s$

**by** (*unfold Min-def Pls-def*) *simp*

**lemma** *Pls-neq-Min*:  $\text{Numeral.Pl}s \neq \text{Numeral.Min}$

**by** (*unfold Min-def Pls-def*) *simp*

**lemmas** *bin-injects* [*simp*] =

*BIT-eq BIT-eq-Pls BIT-eq-Min Pls-eq-BIT Min-eq-BIT Min-neq-Pls Pls-neq-Min*

**lemma** *BIT-exhausts*:  $\exists w b. x = w \text{ BIT } b$

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

**apply** (*fast intro: Pls-0-eq [symmetric] Min-1-eq [symmetric]*)+

**done**

**definition**

*div2* :: *int*  $\Rightarrow$  *int* **where**

*div2* *x* = (*THE* *w*.  $\exists b. x = w \text{ BIT } b$ )

**definition**

*lsb* :: *int*  $\Rightarrow$  *bit* **where**

*lsb* *x* = (*THE* *b*.  $\exists w. x = w \text{ BIT } b$ )

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

**by** (*unfold div2-def*, *rule the-equality*, *fast*, *simp*)

**lemma** *div2-Pls* [*simp*]:  $\text{div2 } \text{Numeral.Pl}s = \text{Numeral.Pl}s$

**by** (*subst Pls-0-eq [symmetric]*, *rule div2-BIT*)

**lemma** *div2-Min* [*simp*]:  $\text{div2 } \text{Numeral.Min} = \text{Numeral.Min}$

**by** (*subst Min-1-eq [symmetric]*, *rule div2-BIT*)

**lemma** *lsb-BIT* [*simp*]:  $\text{lsb } (w \text{ BIT } b) = b$

**by** (*unfold lsb-def*, *rule the-equality*, *fast*, *simp*)

**lemma** *lsb-Pls* [*simp*]:  $\text{lsb } \text{Numeral.Pl}s = \text{bit.B0}$

**by** (*subst Pls-0-eq [symmetric]*, *rule lsb-BIT*)

**lemma** *lsb-Min* [*simp*]:  $\text{lsb } \text{Numeral.Min} = \text{bit.B1}$

**by** (*subst Min-1-eq [symmetric]*, *rule lsb-BIT*)

**lemma** *div2-BIT-lsb*:  $(\text{div2 } x) \text{ BIT } (\text{lsb } x) = x$

**by** (*induct x rule: BIT-induct*) *simp-all*

**lemmas** *div2-lsb-simps* =

*div2-BIT div2-Pls div2-Min*  
*lsb-BIT lsb-Pls lsb-Min*  
*div2-BIT-lsb*

**lemma** *wf-div2*:  $wf \{(div2\ w, w) \mid w. w \neq Numeral.Pl\} \wedge w \neq Numeral.Min\}$   
**apply** (*rule wfUNIVI, simp (no-asm-use)*)  
**apply** (*rename-tac z, induct-tac z rule: BIT-induct*)  
  **apply** (*drule spec, erule mp, simp*)  
  **apply** (*drule spec, erule mp, simp*)  
  **apply** (*drule spec, erule mp, simp*)  
  **apply** (*drule spec, erule mp, simp*)  
**done**

### 4.3 Testing bit positions

**consts** *getbit* :: *int*  $\Rightarrow$  *nat*  $\Rightarrow$  *bit*

**primrec**

*getbit* *x* 0 = *lsb* *x*  
*getbit* *x* (*Suc* *n*) = *getbit* (*div2* *x*) *n*

**lemma** *getbit-Pls*: *getbit* *Numeral.Pl* *n* = *bit.B0*  
**by** (*induct* *n*) *simp-all*

**lemma** *getbit-Min*: *getbit* *Numeral.Min* *n* = *bit.B1*  
**by** (*induct* *n*) *simp-all*

**lemma** *getbit-Pls-rev*:

$(\forall n. \text{getbit } x\ n = \text{bit.B0}) \implies x = \text{Numeral.Pl}$

**proof** (*induct* *x* *rule: BIT-induct*)

**case** *Pls*

**show** ?*case* **by** *simp*

**next**

**case** *Min*

**thus** ?*case* **by** (*simp add: getbit-Min*)

**next**

**case** (*B0* *z*)

**have**  $\forall n. \text{getbit } (z\ \text{BIT}\ \text{bit.B0})\ (\text{Suc } n) = \text{bit.B0}$  **using** *B0(3)* **by** *fast*

**hence**  $\forall n. \text{getbit } z\ n = \text{bit.B0}$  **by** *simp*

**hence**  $z = \text{Numeral.Pl}$  **by** (*rule B0(1)*)

**thus** ?*case* **by** *simp*

**next**

**case** (*B1* *z*)

**have** *getbit* (*z* *BIT* *bit.B1*) 0 = *bit.B0* **using** *B1(3)* **by** *fast*

**thus** ?*case* **by** *simp*

**qed**

**lemma** *getbit-Min-rev*:

$(\forall n. \text{getbit } x\ n = \text{bit.B1}) \implies x = \text{Numeral.Min}$

```

proof (induct x rule: BIT-induct)
  case Pls
  thus ?case by (simp add: getbit-Pls)
next
  case Min
  show ?case by simp
next
  case (B0 z)
  have getbit (z BIT bit.B0) 0 = bit.B1 using B0(3) by fast
  thus ?case by simp
next
  case (B1 z)
  have  $\forall n. \text{getbit } (z \text{ BIT bit.B1}) (\text{Suc } n) = \text{bit.B1}$  using B1(3) by fast
  hence  $\forall n. \text{getbit } z \ n = \text{bit.B1}$  by simp
  hence  $z = \text{Numeral.Min}$  by (rule B1(1))
  thus ?case by simp
qed

```

**lemma** *getbit-ext*:  $\bigwedge x. \forall n. \text{getbit } x \ n = \text{getbit } y \ n \implies (x::\text{int}) = y$

```

proof (induct y rule: BIT-induct)
  case Pls
  hence  $\forall n. \text{getbit } x \ n = \text{bit.B0}$  by (simp add: getbit-Pls)
  thus ?case by (rule getbit-Pls-rev)
next
  case Min
  hence  $\forall n. \text{getbit } x \ n = \text{bit.B1}$  by (simp add: getbit-Min)
  thus ?case by (rule getbit-Min-rev)
next
  case (B0 z)
  have  $\forall n. \text{getbit } x \ (\text{Suc } n) = \text{getbit } (z \text{ BIT bit.B0}) (\text{Suc } n)$ 
    using B0(3) by fast
  hence  $\forall n. \text{getbit } (\text{div2 } x) \ n = \text{getbit } z \ n$  by simp
  hence 1:  $\text{div2 } x = z$  by (rule B0(1))
  have  $\text{getbit } x \ 0 = \text{getbit } (z \text{ BIT bit.B0}) \ 0$  using B0(3) by fast
  hence 2:  $\text{lsb } x = \text{bit.B0}$  by simp
  from 1 2 have  $\text{div2 } x \text{ BIT lsb } x = z \text{ BIT bit.B0}$  by simp
  thus  $x = z \text{ BIT bit.B0}$  by (simp only: div2-BIT-lsb)
next
  case (B1 z)
  have  $\forall n. \text{getbit } x \ (\text{Suc } n) = \text{getbit } (z \text{ BIT bit.B1}) (\text{Suc } n)$ 
    using B1(3) by fast
  hence  $\forall n. \text{getbit } (\text{div2 } x) \ n = \text{getbit } z \ n$  by simp
  hence 1:  $\text{div2 } x = z$  by (rule B1(1))
  have  $\text{getbit } x \ 0 = \text{getbit } (z \text{ BIT bit.B1}) \ 0$  using B1(3) by fast
  hence 2:  $\text{lsb } x = \text{bit.B1}$  by simp
  from 1 2 have  $\text{div2 } x \text{ BIT lsb } x = z \text{ BIT bit.B1}$  by simp
  thus  $x = z \text{ BIT bit.B1}$  by (simp only: div2-BIT-lsb)
qed

```

**lemma** *expand-getbit-eq*:  
 $((x::int) = y) = (\forall n. \text{getbit } x \ n = \text{getbit } y \ n)$   
**by** (*safe intro!*: *getbit-ext*)

**end**

**theory** *BitRing*  
**imports** *BooleanAlgebra BinInduct*  
**begin**

#### 4.4 Syntactic class for bitwise operations

**axclass** *bit*  $\subseteq$  *type*

**consts**

*bitNOT* ::  $'a::\text{bit} \Rightarrow 'a$  (*NOT* - [81] 80)  
*bitAND* ::  $'a::\text{bit} \Rightarrow 'a \Rightarrow 'a$  (**infixr** *AND* 70)  
*bitOR* ::  $'a::\text{bit} \Rightarrow 'a \Rightarrow 'a$  (**infixr** *OR* 65)  
*bitXOR* ::  $'a::\text{bit} \Rightarrow 'a \Rightarrow 'a$  (**infixr** *XOR* 65)  
*testbit* ::  $'a::\text{bit} \Rightarrow \text{nat} \Rightarrow \text{bool}$   
*shiftL* ::  $'a::\text{bit} \Rightarrow \text{nat} \Rightarrow 'a$   
*shiftR* ::  $'a::\text{bit} \Rightarrow \text{nat} \Rightarrow 'a$

**syntax** (*xsymbols*)

*bitNOT* ::  $'a::\text{bit} \Rightarrow 'a$  ( $\neg_b$  - [81] 80)  
*bitAND* ::  $'a::\text{bit} \Rightarrow 'a \Rightarrow 'a$  (**infixr**  $\wedge_b$  70)  
*bitOR* ::  $'a::\text{bit} \Rightarrow 'a \Rightarrow 'a$  (**infixr**  $\vee_b$  65)  
*bitXOR* ::  $'a::\text{bit} \Rightarrow 'a \Rightarrow 'a$  (**infixr**  $\oplus_b$  65)

**syntax** (*HTML output*)

*bitNOT* ::  $'a::\text{bit} \Rightarrow 'a$  ( $\neg_b$  - [81] 80)  
*bitAND* ::  $'a::\text{bit} \Rightarrow 'a \Rightarrow 'a$  (**infixr**  $\wedge_b$  70)  
*bitOR* ::  $'a::\text{bit} \Rightarrow 'a \Rightarrow 'a$  (**infixr**  $\vee_b$  65)  
*bitXOR* ::  $'a::\text{bit} \Rightarrow 'a \Rightarrow 'a$  (**infixr**  $\oplus_b$  65)

#### 4.5 Bitwise operations on type *bit*

**instance** *bit* :: *bit* ..

**defs** (*overloaded*)

*NOT-bit-def*:  $\text{NOT } x \equiv \text{case } x \text{ of } \text{bit}.B0 \Rightarrow \text{bit}.B1 \mid \text{bit}.B1 \Rightarrow \text{bit}.B0$   
*AND-bit-def*:  $x \ \text{AND} \ y \equiv \text{case } x \text{ of } \text{bit}.B0 \Rightarrow \text{bit}.B0 \mid \text{bit}.B1 \Rightarrow y$   
*OR-bit-def*:  $x \ \text{OR} \ y :: \text{bit} \equiv \text{NOT } (\text{NOT } x \ \text{AND} \ \text{NOT } y)$   
*XOR-bit-def*:  $x \ \text{XOR} \ y :: \text{bit} \equiv (x \ \text{AND} \ \text{NOT } y) \ \text{OR} \ (\text{NOT } x \ \text{AND} \ y)$

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

$\text{NOT } \text{bit}.B0 = \text{bit}.B1$   
 $\text{NOT } \text{bit}.B1 = \text{bit}.B0$   
 $\text{bit}.B0 \ \text{AND} \ y = \text{bit}.B0$   
 $\text{bit}.B1 \ \text{AND} \ y = y$

*bit.B0 OR y = y*  
*bit.B1 OR y = bit.B1*  
*bit.B0 XOR y = y*  
*bit.B1 XOR y = NOT y*  
**by** (*simp-all add: NOT-bit-def AND-bit-def OR-bit-def XOR-bit-def split: bit.split*)

**lemma** *bit-NOT-NOT*:  $NOT (NOT (b::bit)) = b$   
**by** (*induct b*) *simp-all*

## 4.6 Bitwise operations on type *int*

**instance** *int :: bit ..*

**consts** *intAND* :: *int*  $\Rightarrow$  *int*  $\Rightarrow$  *int*

**defs** (**overloaded**)

*NOT-int-def*:  $NOT (x::int) \equiv - (x + 1)$   
*AND-int-def*:  $x AND y \equiv intAND x y$   
*OR-int-def*:  $x OR y :: int \equiv NOT (NOT x AND NOT y)$   
*XOR-int-def*:  $x XOR y :: int \equiv (x AND NOT y) OR (NOT x AND y)$

**lemma** *NOT-int-pred*:  $NOT x = Numeral.pred (- x)$   
**by** (*simp add: NOT-int-def pred-def*)

**lemma** *NOT-int-succ*:  $NOT x = - Numeral.succ x$   
**by** (*simp add: NOT-int-def succ-def*)

**lemma** *NOT-Pls*:  $NOT Numeral.Pls = Numeral.Min$   
**by** (*simp add: NOT-int-succ*)

**lemma** *NOT-Min*:  $NOT Numeral.Min = Numeral.Pls$   
**by** (*simp add: NOT-int-succ*)

**lemma** *NOT-BIT-0*:  $NOT (w BIT bit.B0) = (NOT w) BIT bit.B1$   
**by** (*simp add: NOT-int-pred*)

**lemma** *NOT-BIT-1*:  $NOT (w BIT bit.B1) = (NOT w) BIT bit.B0$   
**by** (*simp add: NOT-int-succ*)

**lemma** *NOT-BIT*:  $NOT (x BIT b) = (NOT x) BIT (NOT b)$   
**by** (*induct b*) (*simp-all add: NOT-BIT-0 NOT-BIT-1*)

**lemma** *int-NOT-NOT*:  $NOT (NOT (x::int)) = x$   
**apply** (*induct x rule: BIT-induct*)  
**apply** (*simp-all add: NOT-Pls NOT-Min NOT-BIT*)  
**done**

**recdef** *intAND*  $\{(div2 w, w) \mid w. w \neq Numeral.Pls \wedge w \neq Numeral.Min\}$

```

intAND x = (λy.
  if x = Numeral.Pls then Numeral.Pls else
  if x = Numeral.Min then y else
  (intAND (div2 x) (div2 y) BIT (lsb x AND lsb y)))
(hints recdef-wf: wf-div2)

declare intAND.simps [simp del]

lemma intAND-Pls: intAND Numeral.Pls y = Numeral.Pls
by (subst intAND.simps) simp

lemma intAND-Min: intAND Numeral.Min y = y
by (subst intAND.simps) simp

lemma intAND-BIT-BIT:
  intAND (x BIT a) (y BIT b) = (intAND x y) BIT (a AND b)
apply (subst intAND.simps)
apply (simp add: intAND-Pls intAND-Min)
done

lemma AND-Pls: Numeral.Pls AND x = Numeral.Pls
by (simp add: AND-int-def intAND-Pls)

lemma AND-Min: Numeral.Min AND x = x
by (simp add: AND-int-def intAND-Min)

lemma AND-BIT-0: x BIT bit.B0 AND y BIT b = (x AND y) BIT bit.B0
by (simp add: AND-int-def intAND-BIT-BIT)

lemma AND-BIT-1: x BIT bit.B1 AND y BIT b = (x AND y) BIT b
by (simp add: AND-int-def intAND-BIT-BIT)

lemma AND-Pls-right: x AND Numeral.Pls = Numeral.Pls
apply (induct x rule: BIT-induct)
apply (rule AND-Pls)
apply (rule AND-Min)
apply (subst Pls-0-eq [symmetric], subst AND-BIT-0, simp)
apply (subst Pls-0-eq [symmetric], subst AND-BIT-1, simp)
done

lemma AND-Min-right: x AND Numeral.Min = x
apply (induct x rule: BIT-induct)
apply (rule AND-Pls)
apply (rule AND-Min)
apply (subst Min-1-eq [symmetric], subst AND-BIT-0, simp)
apply (subst Min-1-eq [symmetric], subst AND-BIT-1, simp)
done

lemma OR-Pls: Numeral.Pls OR x = x

```

**by** (*simp add: OR-int-def NOT-Pls NOT-Min AND-Min int-NOT-NOT*)

**lemma** *OR-Min: Numeral.Min OR x = Numeral.Min*  
**by** (*simp add: OR-int-def NOT-Pls NOT-Min AND-Pls*)

**lemma** *OR-BIT-0: x BIT bit.B0 OR y BIT b = (x OR y) BIT b*  
**by** (*simp add: OR-int-def NOT-BIT AND-BIT-1 bit-NOT-NOT*)

**lemma** *OR-BIT-1: x BIT bit.B1 OR y BIT b = (x OR y) BIT bit.B1*  
**by** (*simp add: OR-int-def NOT-BIT AND-BIT-0*)

**lemma** *OR-Pls-right: x OR Numeral.Pls = x*  
**by** (*simp add: OR-int-def NOT-Pls AND-Min-right int-NOT-NOT*)

**lemma** *OR-Min-right: x OR Numeral.Min = Numeral.Min*  
**by** (*simp add: OR-int-def NOT-Min AND-Pls-right NOT-Pls*)

**lemma** *XOR-Pls: Numeral.Pls XOR x = x*  
**by** (*simp add: XOR-int-def AND-Pls AND-Min OR-Pls NOT-Pls*)

**lemma** *XOR-Min: Numeral.Min XOR x = NOT x*  
**by** (*simp add: XOR-int-def AND-Pls AND-Min NOT-Min OR-Pls-right*)

**lemma** *XOR-BIT-0: x BIT bit.B0 XOR y BIT b = (x XOR y) BIT b*  
**by** (*simp add: XOR-int-def NOT-BIT AND-BIT-0 AND-BIT-1 OR-BIT-0*)

**lemma** *XOR-BIT-10: x BIT bit.B1 XOR y BIT bit.B0 = (x XOR y) BIT bit.B1*  
**by** (*simp add: XOR-int-def NOT-BIT AND-BIT-0 AND-BIT-1 OR-BIT-1*)

**lemma** *XOR-BIT-11: x BIT bit.B1 XOR y BIT bit.B1 = (x XOR y) BIT bit.B0*  
**by** (*simp add: XOR-int-def NOT-BIT AND-BIT-0 AND-BIT-1 OR-BIT-0*)

**lemma** *XOR-Pls-right: x XOR Numeral.Pls = x*  
**by** (*simp add: XOR-int-def NOT-Pls AND-Min-right AND-Pls-right OR-Pls-right*)

**lemma** *XOR-Min-right: x XOR Numeral.Min = NOT x*  
**by** (*simp add: XOR-int-def NOT-Min AND-Pls-right AND-Min-right OR-Pls*)

**lemmas** *bitwise-arith-simps [simp] =*  
*NOT-Pls NOT-Min NOT-BIT-0 NOT-BIT-1*  
*AND-Pls AND-Min AND-BIT-0 AND-BIT-1*  
*AND-Pls-right AND-Min-right*  
*OR-Pls OR-Min OR-BIT-0 OR-BIT-1*  
*OR-Pls-right OR-Min-right*  
*XOR-Pls XOR-Min XOR-BIT-0 XOR-BIT-10 XOR-BIT-11*  
*XOR-Pls-right XOR-Min-right*

## 4.7 Bitwise ops with getbit, lsb, div2

**lemma** *lsb-NOT*:  $lsb (NOT\ x) = NOT\ (lsb\ x)$

**by** (*induct x rule: BIT-induct*) *simp-all*

**lemma** *div2-NOT*:  $div2 (NOT\ x) = NOT\ (div2\ x)$

**by** (*induct x rule: BIT-induct*) *simp-all*

**lemma** *lsb-AND*:  $lsb (x\ AND\ y) = lsb\ x\ AND\ lsb\ y$

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

**apply** *simp*

**apply** *simp*

**apply** (*subst div2-BIT-lsb [symmetric, of y], simp*)

**apply** (*subst div2-BIT-lsb [symmetric, of y], simp*)

**done**

**lemma** *div2-AND*:  $div2 (x\ AND\ y) = div2\ x\ AND\ div2\ y$

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

**apply** *simp*

**apply** *simp*

**apply** (*subst div2-BIT-lsb [symmetric, of y], simp*)

**apply** (*subst div2-BIT-lsb [symmetric, of y], simp*)

**done**

**lemma** *lsb-OR*:  $lsb (x\ OR\ y) = lsb\ x\ OR\ lsb\ y$

**by** (*simp add: OR-int-def OR-bit-def lsb-NOT lsb-AND*)

**lemma** *div2-OR*:  $div2 (x\ OR\ y) = div2\ x\ OR\ div2\ y$

**by** (*simp add: OR-int-def div2-NOT div2-AND*)

**lemma** *lsb-XOR*:  $lsb (x\ XOR\ y) = lsb\ x\ XOR\ lsb\ y$

**by** (*simp add: XOR-int-def XOR-bit-def lsb-NOT lsb-AND lsb-OR*)

**lemma** *div2-XOR*:  $div2 (x\ XOR\ y) = div2\ x\ XOR\ div2\ y$

**by** (*simp add: XOR-int-def div2-NOT div2-AND div2-OR*)

**lemma** *getbit-NOT*:  $getbit (NOT\ x)\ n = NOT\ (getbit\ x\ n)$

**by** (*induct n arbitrary: x, simp add: lsb-NOT, simp add: div2-NOT*)

**lemma** *getbit-AND*:  $getbit (x\ AND\ y)\ n = getbit\ x\ n\ AND\ getbit\ y\ n$

**by** (*induct n arbitrary: x y, simp add: lsb-AND, simp add: div2-AND*)

**lemma** *getbit-OR*:  $getbit (x\ OR\ y)\ n = getbit\ x\ n\ OR\ getbit\ y\ n$

**by** (*induct n arbitrary: x y, simp add: lsb-OR, simp add: div2-OR*)

**lemma** *getbit-XOR*:  $getbit (x\ XOR\ y)\ n = getbit\ x\ n\ XOR\ getbit\ y\ n$

**by** (*induct n arbitrary: x y, simp add: lsb-XOR, simp add: div2-XOR*)

**defs** (**overloaded**)

*testbit-int-def*:

$testbit\ x\ n \equiv case\ getbit\ x\ n\ of\ bit.B0 \Rightarrow False \mid bit.B1 \Rightarrow True$

**lemma** *testbit-int-NOT*:  $testbit\ (NOT\ x::int)\ n = (\neg\ testbit\ x\ n)$   
**by** (*simp add: testbit-int-def getbit-NOT split: bit.split*)

**lemma** *testbit-int-AND*:  
 $testbit\ (x\ AND\ y::int)\ n = (testbit\ x\ n \wedge testbit\ y\ n)$   
**by** (*simp add: testbit-int-def getbit-AND split: bit.split*)

**lemma** *testbit-int-OR*:  
 $testbit\ (x\ OR\ y::int)\ n = (testbit\ x\ n \vee testbit\ y\ n)$   
**by** (*simp add: testbit-int-def getbit-OR split: bit.split*)

**lemma** *testbit-int-XOR*:  
 $testbit\ (x\ XOR\ y::int)\ n = (testbit\ x\ n \neq testbit\ y\ n)$   
**by** (*simp add: testbit-int-def getbit-XOR split: bit.split*)

**lemma** *testbit-int-0*:  $testbit\ (0::int)\ n = False$   
**by** (*simp add: Pls-def [symmetric] testbit-int-def getbit-Pls*)

**lemma** *testbit-int-1*:  $testbit\ (-1::int)\ n = True$   
**by** (*simp add: number-of-is-id testbit-int-def getbit-Min*)

**lemmas** *testbit-int-simps* =  
*testbit-int-NOT testbit-int-AND*  
*testbit-int-OR testbit-int-XOR*  
*testbit-int-0 testbit-int-1*

**lemma** *testbit-expand-int-eq*:  
 $((x::int) = y) = (\forall n. testbit\ x\ n = testbit\ y\ n)$   
**apply** (*safe intro!: getbit-ext*)  
**apply** (*drule-tac x=n in spec*)  
**apply** (*simp add: testbit-int-def split: bit.splits*)  
**done**

## 4.8 Axiomatic Class for Bit-Vector Types

The following class includes boolean algebra types where we can compute bitwise operations on numbers as if they were integers.

**axclass** *bit-ring*  $\subseteq$  *bit*, *number-ring*  
*of-int-NOT*:  $of-int\ (NOT\ a) = NOT\ (of-int\ a)$   
*of-int-AND*:  $of-int\ (a\ AND\ b) = of-int\ a\ AND\ of-int\ b$   
*AND-assoc*:  $(x\ AND\ y)\ AND\ z = x\ AND\ (y\ AND\ z)$   
*OR-assoc*:  $(x\ OR\ y)\ OR\ z = x\ OR\ (y\ OR\ z)$   
*AND-commute*:  $x\ AND\ y = y\ AND\ x$   
*OR-commute*:  $x\ OR\ y = y\ OR\ x$   
*AND-OR-distrib*:  $x\ AND\ (y\ OR\ z) = (x\ AND\ y)\ OR\ (x\ AND\ z)$   
*OR-AND-distrib*:  $x\ OR\ (y\ AND\ z) = (x\ OR\ y)\ AND\ (x\ OR\ z)$   
*AND-1-right [simp]*:  $x\ AND\ -1 = x$

$OR-0-right$  [simp]:  $x OR 0 = x$   
 $AND-cancel-right$  [simp]:  $x AND NOT x = 0$   
 $OR-cancel-right$  [simp]:  $x OR NOT x = -1$   
 $XOR-def$ :  $x XOR y = x AND NOT y OR NOT x AND y$

**instance**  $int :: bit-ring$   
**apply** ( $intro-classes$ ,  $simp$ ,  $simp$ )  
**apply** ( $auto$   $simp$   $add$ :  $testbit-expand-int-eq$   $testbit-int-simps$ )  
**done**

**interpretation**  $bit-ring$ :

$boolean-xor$  [ $bitAND$   $bitOR$   $bitNOT$   $0::'a::bit-ring$   $-1$   $bitXOR$ ]  
**by**  $unfold-locales$  ( $rule$   $bit-ring-class.axioms$ ) $+$

**lemmas**  $NOT-0$  [simp] =  $bit-ring.compl-zero$   
**lemmas**  $NOT-1$  [simp] =  $bit-ring.compl-one$   
**lemmas**  $AND-0-right$  [simp] =  $bit-ring.conj-zero-right$   
**lemmas**  $AND-0-left$  [simp] =  $bit-ring.conj-zero-left$   
**lemmas**  $AND-1-left$  [simp] =  $bit-ring.conj-one-left$   
**lemmas**  $OR-0-left$  [simp] =  $bit-ring.disj-zero-left$   
**lemmas**  $OR-1-right$  [simp] =  $bit-ring.disj-one-right$   
**lemmas**  $OR-1-left$  [simp] =  $bit-ring.disj-one-left$   
**lemmas**  $XOR-0-right$  [simp] =  $bit-ring.xor-zero-right$   
**lemmas**  $XOR-0-left$  [simp] =  $bit-ring.xor-zero-left$   
**lemmas**  $XOR-1-right$  [simp] =  $bit-ring.xor-one-right$   
**lemmas**  $XOR-1-left$  [simp] =  $bit-ring.xor-one-left$

**lemmas**  $NOT-unique$  =  $bit-ring.compl-unique$   
**lemmas**  $double-NOT$  [simp] =  $bit-ring.double-compl$   
**lemmas**  $NOT-eq-NOT-iff$  [iff] =  $bit-ring.compl-eq-compl-iff$

**lemmas**  $AND-absorb$  [simp] =  $bit-ring.conj-absorb$   
**lemmas**  $AND-left-absorb$  [simp] =  $bit-ring.conj-left-absorb$   
**lemmas**  $AND-cancel-left$  [simp] =  $bit-ring.conj-cancel-left$   
**lemmas**  $AND-ac$  =  $bit-ring.conj-ac$   
**lemmas**  $AND-OR-distrib2$  =  $bit-ring.conj-disj-distrib2$   
**lemmas**  $AND-OR-distrib$  =  $bit-ring.conj-disj-distrib$   
**lemmas**  $de-Morgan-AND$  =  $bit-ring.de-Morgan-conj$

**lemmas**  $OR-absorb$  [simp] =  $bit-ring.disj-absorb$   
**lemmas**  $OR-left-absorb$  [simp] =  $bit-ring.disj-left-absorb$   
**lemmas**  $OR-cancel-left$  [simp] =  $bit-ring.disj-cancel-left$   
**lemmas**  $OR-ac$  =  $bit-ring.disj-ac$   
**lemmas**  $OR-AND-distrib2$  =  $bit-ring.disj-conj-distrib2$   
**lemmas**  $OR-AND-distrib$  =  $bit-ring.disj-conj-distrib$   
**lemmas**  $de-Morgan-OR$  =  $bit-ring.de-Morgan-disj$

**lemmas**  $XOR-def$  =  $bit-ring.xor-def$   
**lemmas**  $XOR-def2$  =  $bit-ring.xor-def2$

**lemmas** *XOR-cancel-right* [*simp*] = *bit-ring.xor-cancel-right*  
**lemmas** *XOR-cancel-left* [*simp*] = *bit-ring.xor-cancel-left*  
**lemmas** *XOR-NOT-right* = *bit-ring.xor-compl-right*  
**lemmas** *XOR-NOT-left* = *bit-ring.xor-compl-right*  
**lemmas** *XOR-self* [*simp*] = *bit-ring.xor-self*  
**lemmas** *XOR-left-self* [*simp*] = *bit-ring.xor-left-self*  
**lemmas** *XOR-commute* = *bit-ring.xor-commute*  
**lemmas** *XOR-assoc* = *bit-ring.xor-assoc*  
**lemmas** *XOR-ac* = *bit-ring.xor-ac*  
**lemmas** *AND-XOR-distrib* = *bit-ring.conj.xor-distrib*  
**lemmas** *AND-XOR-distrib2* = *bit-ring.conj.xor-distrib2*  
**lemmas** *AND-XOR-distrib3* = *bit-ring.conj.xor-distrib3*

**lemma** *number-of-NOT*:  
 $(\text{number-of } (\text{NOT } a)::'a::\text{bit-ring}) = \text{NOT } (\text{number-of } a)$   
**by** (*simp only: number-of-eq of-int-NOT*)

**lemma** *number-of-AND*:  
 $(\text{number-of } (a \text{ AND } b)::'a::\text{bit-ring}) = \text{number-of } a \text{ AND } \text{number-of } b$   
**by** (*simp only: number-of-eq of-int-AND*)

**lemma** *number-of-OR*:  
 $(\text{number-of } (a \text{ OR } b)::'a::\text{bit-ring}) = \text{number-of } a \text{ OR } \text{number-of } b$   
**apply** (*rule NOT-eq-NOT-iff [THEN iffD1]*)  
**apply** (*simp only: de-Morgan-OR*  
 $\text{number-of-NOT } [\text{symmetric}]$   
 $\text{number-of-AND } [\text{symmetric}]$ )

**done**

**lemma** *number-of-XOR*:  
 $(\text{number-of } (a \text{ XOR } b)::'a::\text{bit-ring}) = \text{number-of } a \text{ XOR } \text{number-of } b$   
**by** (*simp add: XOR-def number-of-NOT number-of-AND number-of-OR*)

**lemmas** *bitwise-arith-extra-simps* [*simp*] =  
 $\text{number-of-AND } [\text{symmetric}]$   
 $\text{number-of-OR } [\text{symmetric}]$   
 $\text{number-of-NOT } [\text{symmetric}]$   
 $\text{number-of-XOR } [\text{symmetric}]$

## 4.9 Integers modulo powers of 2

**lemma** *ex-power2-eq-number*:  
 $\exists w. (2::\text{int}) ^ n = \text{number-of } w \wedge \neg \text{neg } (\text{number-of } w::\text{int})$   
**apply** (*induct n*)  
**apply** (*rule-tac x=Numeral.Pls BIT bit.B1 in exI, simp*)  
**apply** (*erule exE, rule-tac x=w BIT bit.B0 in exI, simp*)  
**done**

**lemma** *zmod-power2-lemma*:

```

    (number-of x::int) mod 2 ^ n = number-of x AND (2 ^ n - 1)
  apply (induct n arbitrary: x, simp-all)
  apply (cut-tac x=x in div2-BIT-lsb, erule subst)
  apply (cut-tac n=n in ex-power2-eq-number, erule exE)
  apply (simp split: bit.split)
done

lemma zmod-power2: (x::int) mod 2 ^ n = x AND (2 ^ n - 1)
by (cut-tac zmod-power2-lemma, simp add: number-of-is-id)

lemma zmod-NOT: (NOT (x mod 2 ^ n)) mod 2 ^ n = (NOT x::int) mod 2 ^ n
by (simp add: zmod-power2 de-Morgan-AND AND-OR-distrib)

lemma zmod-AND-left: ((x mod 2 ^ n) AND y) mod 2 ^ n = (x AND y::int) mod
2 ^ n
by (simp add: zmod-power2 AND-ac)

lemma zmod-AND-right: (x AND (y mod 2 ^ n)) mod 2 ^ n = (x AND y::int) mod
2 ^ n
by (simp add: zmod-power2 AND-ac)

lemma zmod-OR-left: ((x mod 2 ^ n) OR y) mod 2 ^ n = (x OR y::int) mod 2 ^ n
by (simp add: zmod-power2 AND-OR-distrib AND-ac)

lemma zmod-OR-right: (x OR (y mod 2 ^ n)) mod 2 ^ n = (x OR y::int) mod 2 ^ n
by (simp add: zmod-power2 AND-OR-distrib AND-ac)

lemmas zmod-bit-simps =
  zmod-NOT zmod-AND-left zmod-AND-right zmod-OR-left zmod-OR-right

end

theory Bits
imports IntMod BitRing
begin

typedef (open) 'a bits = {0::int..<2 ^ nat-of-type TYPE('a::numeral1)}
by (rule-tac x=0 in exI, simp add: zero-less-power)

lemma one-less-power: — belongs in Power.thy
   $\llbracket 1 < (a::'a::\{\text{ordered-semidom,recpower}\}); 0 < n \rrbracket \implies 1 < a ^ n$ 
by (cases n, simp, simp add: power-gt1)

constdefs
  Abs-bits' :: int  $\Rightarrow$  'a::numeral1 bits
  Abs-bits' x  $\equiv$  Abs-bits (x mod (2 ^ nat-of-type TYPE('a)))

instance bits :: (numeral1) {zero,one,plus,times,minus,power,bit} ..

```

**defs (overloaded)**

*zero-bits-def*:  $0 \equiv \text{Abs-bits } 0$   
*one-bits-def*:  $1 \equiv \text{Abs-bits } 1$   
*add-bits-def*:  $x + y \equiv \text{Abs-bits}' (\text{Rep-bits } x + \text{Rep-bits } y)$   
*mult-bits-def*:  $x * y \equiv \text{Abs-bits}' (\text{Rep-bits } x * \text{Rep-bits } y)$   
*diff-bits-def*:  $x - y \equiv \text{Abs-bits}' (\text{Rep-bits } x - \text{Rep-bits } y)$   
*minus-bits-def*:  $- x \equiv \text{Abs-bits}' (- \text{Rep-bits } x)$   
*power-bits-def*:  $x ^ n \equiv \text{Abs-bits}' (\text{Rep-bits } x ^ n)$   
*NOT-bits-def*:  $\text{NOT } x \equiv \text{Abs-bits}' (\text{NOT } \text{Rep-bits } x)$   
*AND-bits-def*:  $x \text{ AND } y \equiv \text{Abs-bits}' (\text{Rep-bits } x \text{ AND } \text{Rep-bits } y)$   
*OR-bits-def*:  $x \text{ OR } y \equiv \text{Abs-bits}' (\text{Rep-bits } x \text{ OR } \text{Rep-bits } y)$   
*XOR-bits-def*:  $x \text{ XOR } y \equiv \text{Abs-bits}' (\text{Rep-bits } x \text{ XOR } \text{Rep-bits } y)$

**interpretation bits:**

*mod-type* [ $2 ^ \text{nat-of-type TYPE}('a::\text{numeral1})$   
 $\text{Rep-bits}::'a::\text{numeral1 bits} \Rightarrow \text{int}$   
 $\text{Abs-bits}::\text{int} \Rightarrow 'a::\text{numeral1 bits}$ ]

**apply** (*rule mod-type.intro*)  
**apply** (*rule type-definition-bits*)  
**apply** (*rule one-less-power, simp*)  
**apply** (*rule zero-less-nat-of-type*)  
**apply** (*rule zero-bits-def*)  
**apply** (*rule one-bits-def*)  
**apply** (*rule add-bits-def [unfolded Abs-bits'-def]*)  
**apply** (*rule mult-bits-def [unfolded Abs-bits'-def]*)  
**apply** (*rule diff-bits-def [unfolded Abs-bits'-def]*)  
**apply** (*rule minus-bits-def [unfolded Abs-bits'-def]*)  
**apply** (*rule power-bits-def [unfolded Abs-bits'-def]*)  
**done**

**instance** *bits* :: (*numeral1*) *comm-ring-1*  
**by** (*rule bits.comm-ring-1*)

**instance** *bits* :: (*numeral1*) *recpower*  
**by** (*rule bits.recpower*)

**instance** *bits* :: (*numeral1*) *number ..*

**defs (overloaded)**

*number-bits-def*: *number-of w::- bits*  $\equiv$  *of-int w*

**instance** *bits* :: (*numeral1*) *number-ring*  
**by** *intro-classes (simp add: number-bits-def)*

**interpretation bits:**

*mod-ring* [ $2 ^ \text{nat-of-type TYPE}('a::\text{numeral1})$   
 $\text{Rep-bits}::'a::\text{numeral1 bits} \Rightarrow \text{int}$   
 $\text{Abs-bits}::\text{int} \Rightarrow 'a::\text{numeral1 bits}$ ].

```

lemma bits-iszero-number-of [unfolded nat-of-type-def, simp]:
  iszero (number-of w::'a::numeral1 bits) =
    (number-of w mod 2 ^ nat-of-type TYPE('a) = (0::int))
by (rule bits.iszero-number-of)

```

#### 4.10 bit-ring instance

```

lemmas bits-bit-defs =
  NOT-bits-def AND-bits-def OR-bits-def XOR-bits-def

```

```

instance bits :: (numeral1) bit-ring
apply (intro-classes)
apply (unfold bits-bit-defs Abs-bits'-def)
apply (simp-all add: bits.Rep-inject-sym bits.of-int-eq
  bits.Rep-Abs-mod bits.Rep-0 bits.Rep-number-of)
apply (simp-all add: zmod-bit-simps AND-ac OR-ac XOR-def bits.Rep-mod)
apply (simp add: AND-OR-distrib)
apply (simp add: OR-AND-distrib)
done

```

```

end

```

```

theory Examples imports Bits begin

```

```

lemma (123::Z[5]) = (8::Z[5])
by simp

```

```

lemma 1 + 2 * 3 = (0::Z[7])
by simp

```

```

lemma 0b10101 AND 0b11011 = (0b10001 :: int)
by simp

```

```

lemma 0b10100 OR 0b01001 = (0b01101 :: 4 bits)
by simp

```

```

lemma 0b10100 OR 0b01001 ≠ (0b01101 :: 5 bits)
by simp

```

```

lemma (x AND 0xff00) OR (x AND 0x00ff) = (x::16 bits)

```

```

proof –

```

```

  have (x AND 0xff00) OR (x AND 0x00ff) = x AND (0xff00 OR 0x00ff)

```

```

    by (simp only: AND-OR-distrib)

```

```

  also have 0xff00 OR 0x00ff = (-1::16 bits)

```

```

    by simp

```

```

  also have x AND -1 = x

```

```

    by simp

```

```

  finally show ?thesis .

```

```

qed

```

end