

Machine Words in Isabelle/HOL

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

April 19, 2009

Abstract

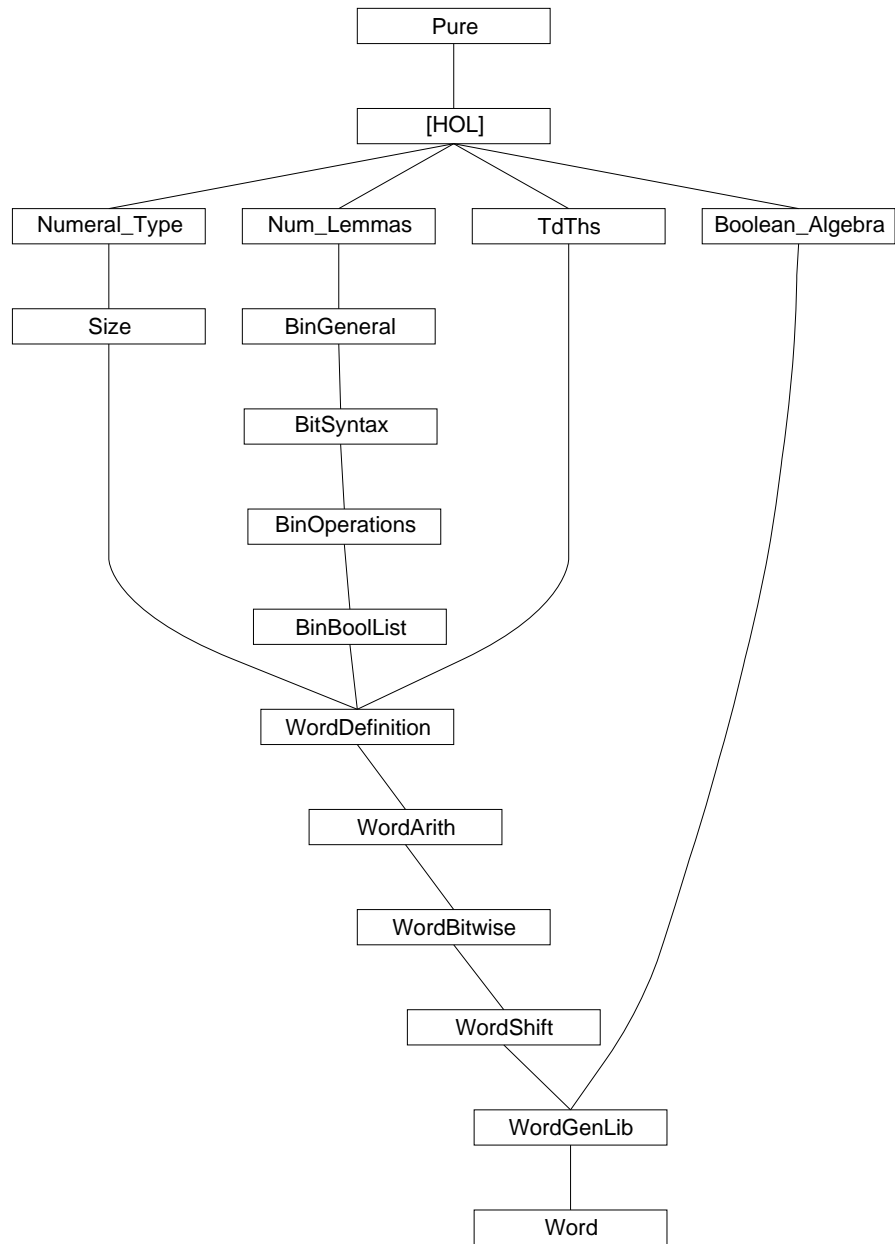
A formalisation of generic, fixed size machine words in Isabelle/HOL.
An earlier version of this formalisation is described in [1].

Contents

1	Numeral-Type: Numeral Syntax for Types	5
1.1	Preliminary lemmas	5
1.2	Cardinalities of types	5
1.3	Classes with at least 1 and 2	5
1.4	Numeral Types	6
1.5	Locale for modular arithmetic subtypes	6
1.6	Number ring instances	9
1.7	Syntax	11
1.8	Examples	11
2	Size: The len classes	11
3	Num-Lemmas: Useful Numerical Lemmas	12
4	BinGeneral: Basic Definitions for Binary Integers	19
4.1	Further properties of numerals	19
4.2	Destructors for binary integers	21
4.3	Recursion combinator for binary integers	24
4.4	Truncating binary integers	25
4.5	Simplifications for (s)bintrunc	26
4.6	Splitting and concatenation	34
4.7	Miscellaneous lemmas	34
5	BitSyntax: Syntactic classes for bitwise operations	35
5.1	Bitwise operations on <i>bit</i>	36

6	BinOperations: Bitwise Operations on Binary Integers	37
6.1	Logical operations	37
6.2	Setting and clearing bits	42
6.3	Operations on lists of booleans	44
6.4	Splitting and concatenation	44
6.5	Miscellaneous lemmas	47
7	BinBoolList: Bool lists and integers	47
7.1	Arithmetic in terms of bool lists	47
7.2	Repeated splitting or concatenation	60
8	TdThs: Type Definition Theorems	62
9	More lemmas about normal type definitions	62
9.1	Extended form of type definition predicate	64
10	WordDefinition: Definition of Word Type	66
10.1	Type definition	66
10.2	Type conversions and casting	66
10.3	Arithmetic operations	68
10.4	Bit-wise operations	69
10.5	Shift operations	71
10.6	Rotation	71
10.7	Split and cat operations	71
11	WordArith: Word Arithmetic	83
11.1	Transferring goals from words to ints	86
11.2	Order on fixed-length words	88
11.3	Conditions for the addition (etc) of two words to overflow	90
11.4	Definition of uint_arith	90
11.5	More on overflows and monotonicity	91
11.6	Arithmetic type class instantiations	95
11.7	Word and nat	96
11.8	Definition of unat_arith tactic	99
11.9	Cardinality, finiteness of set of words	101
12	WordBitwise: Bitwise Operations on Words	102
13	WordShift: Shifting, Rotating, and Splitting Words	109
13.1	Bit shifting	109
13.1.1	shift functions in terms of lists of bools	111
13.1.2	Mask	114
13.1.3	Revcast	116
13.1.4	Slices	118
13.2	Split and cat	120

13.2.1	Split and slice	121
13.3	Rotation	124
13.3.1	Rotation of list to right	125
13.3.2	map, map2, commuting with rotate(r)	126
13.3.3	Word rotation commutes with bit-wise operations	128
14	Boolean-Algebra: Boolean Algebras	130
14.1	Complement	130
14.2	Conjunction	131
14.3	Disjunction	131
14.4	De Morgan's Laws	132
14.5	Symmetric Difference	132
15	WordGenLib: Miscellaneous Library for Words	133
16	Word: Word Library interafce	139



1 Numeral-Type: Numeral Syntax for Types

```
theory Numeral-Type
imports Main
begin
```

1.1 Preliminary lemmas

```
lemma (in type-definition) univ:
  UNIV = Abs ‘ A
<proof>
```

```
lemma (in type-definition) card: card (UNIV :: 'b set) = card A
<proof>
```

1.2 Cardinalities of types

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

```
translations CARD(t) => CONST card (CONST UNIV :: t set)
```

```
<ML>
```

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

```
lemma card-bool [simp]: CARD(bool) = 2
<proof>
```

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

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

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

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

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

1.3 Classes with at least 1 and 2

Class finite already captures ”at least 1”

```
lemma zero-less-card-finite [simp]: 0 < CARD('a::finite)
<proof>
```

lemma *one-le-card-finite* [simp]: $\text{Suc } 0 \leq \text{CARD}('a::\text{finite})$
 ⟨proof⟩

Class for cardinality ”at least 2”

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

lemma *one-less-card*: $\text{Suc } 0 < \text{CARD}('a::\text{card2})$
 ⟨proof⟩

lemma *one-less-int-card*: $1 < \text{int } \text{CARD}('a::\text{card2})$
 ⟨proof⟩

1.4 Numeral Types

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

typedef (open) *'a bit0* = $\{0 \dots 2 * \text{int } \text{CARD}('a::\text{finite})\}$
 ⟨proof⟩

typedef (open) *'a bit1* = $\{0 \dots 1 + 2 * \text{int } \text{CARD}('a::\text{finite})\}$
 ⟨proof⟩

lemma *card-num0* [simp]: $\text{CARD } (\text{num0}) = 0$
 ⟨proof⟩

lemma *card-num1* [simp]: $\text{CARD}(\text{num1}) = 1$
 ⟨proof⟩

lemma *card-bit0* [simp]: $\text{CARD}('a \text{ bit0}) = 2 * \text{CARD}('a::\text{finite})$
 ⟨proof⟩

lemma *card-bit1* [simp]: $\text{CARD}('a \text{ bit1}) = \text{Suc } (2 * \text{CARD}('a::\text{finite}))$
 ⟨proof⟩

instance *num1* :: *finite*
 ⟨proof⟩

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

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

1.5 Locale for modular arithmetic subtypes

locale *mod-type* =

```

fixes  $n :: int$ 
and  $Rep :: 'a :: \{zero, one, plus, times, uminus, minus, power\} \Rightarrow int$ 
and  $Abs :: int \Rightarrow 'a :: \{zero, one, plus, times, uminus, minus, power\}$ 
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  $power-def: x ^ k = Abs\ (Rep\ x ^ k\ mod\ n)$ 
begin

```

```

lemma  $size0: 0 < n$ 
 $\langle proof \rangle$ 

```

```

lemmas  $definitions =$ 
 $zero-def\ one-def\ add-def\ mult-def\ minus-def\ diff-def\ power-def$ 

```

```

lemma  $Rep-less-n: Rep\ x < n$ 
 $\langle proof \rangle$ 

```

```

lemma  $Rep-le-n: Rep\ x \leq n$ 
 $\langle proof \rangle$ 

```

```

lemma  $Rep-inject-sym: x = y \longleftrightarrow Rep\ x = Rep\ y$ 
 $\langle proof \rangle$ 

```

```

lemma  $Rep-inverse: Abs\ (Rep\ x) = x$ 
 $\langle proof \rangle$ 

```

```

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

```

```

lemma  $Rep-Abs-mod: Rep\ (Abs\ (m\ mod\ n)) = m\ mod\ n$ 
 $\langle proof \rangle$ 

```

```

lemma  $Rep-Abs-0: Rep\ (Abs\ 0) = 0$ 
 $\langle proof \rangle$ 

```

```

lemma  $Rep-0: Rep\ 0 = 0$ 
 $\langle proof \rangle$ 

```

```

lemma  $Rep-Abs-1: Rep\ (Abs\ 1) = 1$ 
 $\langle proof \rangle$ 

```

```

lemma  $Rep-1: Rep\ 1 = 1$ 
 $\langle proof \rangle$ 

```

lemma *Rep-mod*: $\text{Rep } x \text{ mod } n = \text{Rep } x$

$\langle \text{proof} \rangle$

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*)

$\langle \text{proof} \rangle$

lemma *recpower*: *OFCLASS*('a, *recpower-class*)

$\langle \text{proof} \rangle$

end

locale *mod-ring* = *mod-type* +

constrains *n* :: *int*

and *Rep* :: 'a::{*number-ring*,*power*} \Rightarrow *int*

and *Abs* :: *int* \Rightarrow 'a::{*number-ring*,*power*}

begin

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *Rep-number-of*:

$\text{Rep } (\text{number-of } w) = \text{number-of } w \text{ mod } n$

$\langle \text{proof} \rangle$

lemma *iszero-number-of*:

$\text{iszero } (\text{number-of } w :: 'a) \longleftrightarrow \text{number-of } w \text{ mod } n = 0$

$\langle \text{proof} \rangle$

lemma *cases*:

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

shows *P*

$\langle \text{proof} \rangle$

lemma *induct*:

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

$\langle \text{proof} \rangle$

end

1.6 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, recpower*}
begin

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

instance ⟨*proof*⟩

end

instantiation

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

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

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

definition 0 = *Abs-bit0* 0

definition 1 = *Abs-bit0* 1

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

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

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

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

definition *x* ^ *k* = *Abs-bit0'* (*Rep-bit0* *x* ^ *k*)

definition 0 = *Abs-bit1* 0

definition 1 = *Abs-bit1* 1

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

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

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

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

definition *x* ^ *k* = *Abs-bit1'* (*Rep-bit1* *x* ^ *k*)

instance ⟨*proof*⟩

end

interpretation *bit0*:

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

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

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

⟨*proof*⟩

interpretation *bit1*:

mod-type int CARD('a::finite bit1)
Rep-bit1 :: 'a::finite bit1 \Rightarrow int
Abs-bit1 :: int \Rightarrow 'a::finite bit1

<proof>

instance *bit0* :: (*finite*) {*comm-ring-1,recpower*}

<proof>

instance *bit1* :: (*finite*) {*comm-ring-1,recpower*}

<proof>

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>*

end

interpretation *bit0*:

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

<proof>

interpretation *bit1*:

mod-ring int CARD('a::finite bit1)
Rep-bit1 :: 'a::finite bit1 \Rightarrow int
Abs-bit1 :: int \Rightarrow 'a::finite bit1

<proof>

Set up cases, induction, and arithmetic

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

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

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

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

lemmas *bit0-iszero-number-of* [*simp*] = *bit0.iszero-number-of*

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

declare *power-Suc* [**where** ?'a='a::finite *bit0*, *standard*, *simp*]

declare *power-Suc* [**where** ?'a='a::finite *bit1*, *standard*, *simp*]

1.7 Syntax

syntax

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

translations

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

⟨*ML*⟩

1.8 Examples

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

end

2 Size: The len classes

theory *Size*

imports *Numeral-Type*

begin

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

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 ⟨*proof*⟩

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 *<proof>*

end

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

instance *num1* :: *len* *<proof>*

instance *bit0* :: (*len*) *len* *<proof>*

instance *bit1* :: (*len0*) *len* *<proof>*

lemma *len-of TYPE*(17) = 17 *<proof>*

lemma *len-of TYPE*(0) = 0 *<proof>*

lemma *len-of TYPE*('*a*::*len0*) = *x*
<proof>

end

3 Num-Lemmas: Useful Numerical Lemmas

theory *Num-Lemmas*

imports *Main Parity*

begin

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

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

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

lemmas *split-splits* = *split-split split-split-asm*

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

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

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

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

declare *iszero-0* [iff]

lemmas *xtr1* = *xtrans*(1)
lemmas *xtr2* = *xtrans*(2)
lemmas *xtr3* = *xtrans*(3)
lemmas *xtr4* = *xtrans*(4)
lemmas *xtr5* = *xtrans*(5)
lemmas *xtr6* = *xtrans*(6)
lemmas *xtr7* = *xtrans*(7)
lemmas *xtr8* = *xtrans*(8)

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

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

lemma *nobm1*:

$0 < (\text{number-of } w :: \text{nat}) \implies$
 $\text{number-of } w - (1 :: \text{nat}) = \text{number-of } (\text{Int.pred } w)$
<proof>

lemma *of-int-power*:

$\text{of-int } (a \wedge n) = (\text{of-int } a \wedge n :: 'a :: \{\text{recpower}, \text{comm-ring-1}\})$
<proof>

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

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

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

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

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

— the inverse(s) of *number-of*

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

lemma *emep1*:

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

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

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

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

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

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

lemmas *m1mod2k* = *zless2p* [THEN *zmod-minus1*]

lemmas *m1mod22k* = *mult-pos-pos* [OF *zless2 zless2p*, THEN *zmod-minus1*]

lemmas *p1mod22k'* = *zless2p* [THEN *order-less-imp-le*, THEN *pos-zmod-mult-2*]

lemmas *z1pmod2'* = *zero-le-one* [THEN *pos-zmod-mult-2*, *simplified*]

lemmas *z1pdiv2'* = *zero-le-one* [THEN *pos-zdiv-mult-2*, *simplified*]

lemma *p1mod22k*:

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

<proof>

lemma *z1pmod2*:

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

lemma *z1pdiv2*:

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

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

lemma *axbbyy*:

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

lemma *axxmod2*:

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

lemma *axxdiv2*:

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

lemmas *iszero-minus* = *trans* [THEN *trans*,

OF *iszero-def neg-equal-0-iff-equal iszero-def* [*symmetric*], *standard*]

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

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

lemma *zmod-uminus*: $-(a::int) \bmod b \bmod b = -a \bmod b$
<proof>

lemma *zmod-zsub-distrib*: $((a::int) - b) \bmod c = (a \bmod c - b \bmod c) \bmod c$
<proof>

lemma *zmod-zsub-right-eq*: $((a::int) - b) \bmod c = (a - b \bmod c) \bmod c$
<proof>

lemma *zmod-zsub-left-eq*: $((a::int) - b) \bmod c = (a \bmod c - b) \bmod c$
 $\langle proof \rangle$

lemma *zmod-zsub-self* [simp]:
 $((b::int) - a) \bmod a = b \bmod a$
 $\langle proof \rangle$

lemma *zmod-zmult1-eq-rev*:
 $b * a \bmod c = b \bmod c * a \bmod (c::int)$
 $\langle proof \rangle$

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*:
 $((a + x) \bmod m = (b + x) \bmod m) = (a \bmod m = b \bmod (m::nat))$
 $\langle proof \rangle$

lemma *nat-minus-mod*: $(n - n \bmod m) \bmod m = (0::nat)$
 $\langle proof \rangle$

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*:
 $!!b. b < n ==> a \bmod n = b \bmod n ==> a \bmod n = (b::nat)$
 $\langle proof \rangle$

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

lemma *nat-mod-lem*:
 $(0::nat) < n ==> b < n = (b \bmod n = b)$
 $\langle proof \rangle$

lemma *mod-nat-add*:
 $(x::nat) < z ==> y < z ==>$

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

lemma *mod-nat-sub*:

$(x :: \text{nat}) < z \implies (x - y) \bmod z = x - y$
 $\langle \text{proof} \rangle$

lemma *int-mod-lem*:

$(0 :: \text{int}) < n \implies (0 \leq b \ \& \ b < n) = (b \bmod n = b)$
 $\langle \text{proof} \rangle$

lemma *int-mod-eq*:

$(0 :: \text{int}) \leq b \implies b < n \implies a \bmod n = b \bmod n \implies a \bmod n = b$
 $\langle \text{proof} \rangle$

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

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

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

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

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

lemma *mod-add-if-z*:

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

lemma *mod-sub-if-z*:

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

lemmas *zmde* = *zmod-zdiv-equality* [THEN *diff-eq-eq* [THEN *iffD2*], *symmetric*]

lemmas *mcl* = *mult-cancel-left* [THEN *iffD1*, THEN *make-pos-rule*]

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

lemma *eqne*: *equiv* $A \ r \implies X : A // r \implies X \sim = \{\}$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

lemma *mod-power-lem*:
 $a > 1 \implies a \wedge n \bmod a \wedge m = (\text{if } m \leq n \text{ then } 0 \text{ else } (a :: \text{int}) \wedge n)$
 ⟨*proof*⟩

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

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

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

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

lemma *pl-pl-rels*:
 $a + b = c + d \implies$

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

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

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

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

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

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

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

lemma *nat-no-eq-iff*:

$$\begin{aligned} (\text{number-of } b :: \text{int}) \geq 0 \implies (\text{number-of } c :: \text{int}) \geq 0 \implies \\ (\text{number-of } b = (\text{number-of } c :: \text{nat})) = (b = c) \\ \langle \text{proof} \rangle \end{aligned}$$

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*:

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

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

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

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

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

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

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

lemma *less-le-mult'*:

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

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* *<proof>*

lemma *lrlem*: $(0::nat) < sc \implies$
 $(sc - n + (n + lb * n) \leq m * n) = (sc + lb * n \leq m * n)$
<proof>

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

lemma *mpl-lem*: $j \leq (i :: nat) \implies k < j \implies i - j + k < i$ *<proof>*

lemma *nonneg-mod-div*:
 $0 \leq a \implies 0 \leq b \implies 0 \leq (a \bmod b :: int) \ \& \ 0 \leq a \div b$
<proof>

end

4 BinGeneral: Basic Definitions for Binary Integers

theory *BinGeneral*
imports *Num-Lemmas*
begin

4.1 Further properties of numerals

datatype *bit* = *B0* | *B1*

definition
 $Bit :: int \Rightarrow bit \Rightarrow int$ (**infixl** *BIT* 90) **where**
 $k\ BIT\ b = (case\ b\ of\ B0 \Rightarrow 0 \mid B1 \Rightarrow 1) + k + k$

lemma *BIT-B0-eq-Bit0* [*simp*]: $w\ BIT\ B0 = Int.Bit0\ w$
<proof>

lemma *BIT-B1-eq-Bit1* [*simp*]: $w\ BIT\ B1 = Int.Bit1\ w$
<proof>

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

hide (**open**) *const B0 B1*

lemma *Min-ne-Pls* [iff]:
 $Int.Min \sim = Int.Pl$
 ⟨proof⟩

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

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

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

lemma *number-of-False-cong*:
 $False \implies number-of \ x = number-of \ y$
 ⟨proof⟩

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

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

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

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

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

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

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

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

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

lemma *Bit-B0-2t*: $k \text{ BIT } \text{bit}.B0 = 2 * k$
 $\langle \text{proof} \rangle$

lemma *Bit-B1-2t*: $k \text{ BIT } \text{bit}.B1 = 2 * k + 1$
 $\langle \text{proof} \rangle$

lemma *B-mod-2'*:
 $X = 2 \implies (w \text{ BIT } \text{bit}.B1) \text{ mod } X = 1 \ \& \ (w \text{ BIT } \text{bit}.B0) \text{ mod } X = 0$
 $\langle \text{proof} \rangle$

lemma *B1-mod-2 [simp]*: $(\text{Int}.Bit1 \ w) \text{ mod } 2 = 1$
 $\langle \text{proof} \rangle$

lemma *B0-mod-2 [simp]*: $(\text{Int}.Bit0 \ w) \text{ mod } 2 = 0$
 $\langle \text{proof} \rangle$

lemma *neB1E [elim!]*:
assumes *ne*: $y \neq \text{bit}.B1$
assumes *y*: $y = \text{bit}.B0 \implies P$
shows *P*
 $\langle \text{proof} \rangle$

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

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

4.2 Destructors for binary integers

definition *bin-rl* :: $\text{int} \Rightarrow \text{int} \times \text{bit}$ **where**
 $\text{[code del]: } \text{bin-rl } w = (\text{THE } (r, l). \ w = r \text{ BIT } l)$

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

definition
 $\text{bin-rest-def [code del]: } \text{bin-rest } w = \text{fst } (\text{bin-rl } w)$

definition
 $\text{bin-last-def [code del]: } \text{bin-last } w = \text{snd } (\text{bin-rl } w)$

primrec *bin-nth* **where**
 $Z: \text{bin-nth } w \ 0 = (\text{bin-last } w = \text{bit}.B1)$
 $| \text{Suc}: \text{bin-nth } w \ (\text{Suc } n) = \text{bin-nth } (\text{bin-rest } w) \ n$

lemma *bin-rl*: $\text{bin-rl } w = (\text{bin-rest } w, \text{bin-last } w)$

$\langle \text{proof} \rangle$

lemma *bin-rl-simps* [*simp*]:

bin-rl *Int.Pls* = (*Int.Pls*, *bit.B0*)
bin-rl *Int.Min* = (*Int.Min*, *bit.B1*)
bin-rl (*Int.Bit0* *r*) = (*r*, *bit.B0*)
bin-rl (*Int.Bit1* *r*) = (*r*, *bit.B1*)
bin-rl (*r BIT b*) = (*r*, *b*)
 $\langle \text{proof} \rangle$

declare *bin-rl-simps*(1–4) [*code*]

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

lemma *bin-abs-lem*:

bin = (*w BIT b*) $\implies \sim \text{bin} = \text{Int.Min} \dashrightarrow \sim \text{bin} = \text{Int.Pls} \dashrightarrow$
 $\text{nat}(\text{abs } w) < \text{nat}(\text{abs } \text{bin})$
 $\langle \text{proof} \rangle$

lemma *bin-induct*:

assumes *PPls*: *P Int.Pls*
and *PMin*: *P Int.Min*
and *PBit*: $\forall \text{bin bit. } P \text{ bin} \implies P (\text{bin BIT bit})$
shows *P bin*
 $\langle \text{proof} \rangle$

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

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

declare *bin-rest-simps*(1–4) [*code*]

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

bin-last *Int.Pls* = *bit.B0*
bin-last *Int.Min* = *bit.B1*
bin-last (*Int.Bit0* *w*) = *bit.B0*
bin-last (*Int.Bit1* *w*) = *bit.B1*

bin-last (*w BIT b*) = *b*
 ⟨*proof*⟩

declare *bin-last-simps*(1-4) [code]

lemma *bin-r-l-extras* [simp]:

bin-last 0 = *bit.B0*
bin-last (− 1) = *bit.B1*
bin-last −1 = *bit.B1*
bin-last 1 = *bit.B1*
bin-rest 1 = 0
bin-rest 0 = 0
bin-rest (− 1) = − 1
bin-rest −1 = −1
 ⟨*proof*⟩

lemma *bin-last-mod*:

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

lemma *bin-rest-div*:

bin-rest *w* = *w div* 2
 ⟨*proof*⟩

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

lemma *Bit0-div2* [simp]: (*Int.Bit0* *w*) *div* 2 = *w*
 ⟨*proof*⟩

lemma *Bit1-div2* [simp]: (*Int.Bit1* *w*) *div* 2 = *w*
 ⟨*proof*⟩

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

ALL *y*. *bin-nth* *x* = *bin-nth* *y* \longrightarrow *x* = *y*
 ⟨*proof*⟩

lemma *bin-nth-eq-iff*: (*bin-nth* *x* = *bin-nth* *y*) = (*x* = *y*)
 ⟨*proof*⟩

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

lemma *bin-nth-Pls* [simp]: \sim *bin-nth* *Int.Pl*s *n*
 ⟨*proof*⟩

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

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

$\langle \text{proof} \rangle$

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

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

lemma *bin-nth-minus-Bit0* [simp]:
 $0 < n \implies \text{bin-nth } (\text{Int.Bit0 } w) \ n = \text{bin-nth } w \ (n - 1)$
 $\langle \text{proof} \rangle$

lemma *bin-nth-minus-Bit1* [simp]:
 $0 < n \implies \text{bin-nth } (\text{Int.Bit1 } w) \ n = \text{bin-nth } w \ (n - 1)$
 $\langle \text{proof} \rangle$

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

4.3 Recursion combinator for binary integers

lemma *brlem*: $(\text{bin} = \text{Int.Min}) = (- \text{bin} + \text{Int.pred } 0 = 0)$
 $\langle \text{proof} \rangle$

function

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

where

bin-rec *f1 f2 f3 bin* = (if *bin* = *Int.Plus* then *f1*
 else if *bin* = *Int.Min* then *f2*
 else case *bin-rl bin of* (*w, b*) => *f3 w b (bin-rec f1 f2 f3 w)*)
 $\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

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

lemma *bin-rec-PM*:

$f = \text{bin-rec } f1 \ f2 \ f3 \implies f \ \text{Int.Plus} = f1 \ \& \ f \ \text{Int.Min} = f2$
 $\langle \text{proof} \rangle$

lemma *bin-rec-Pls*: $\text{bin-rec } f1 \ f2 \ f3 \ \text{Int.Plus} = f1$
 $\langle \text{proof} \rangle$

lemma *bin-rec-Min*: $\text{bin-rec } f1 \ f2 \ f3 \ \text{Int.Min} = f2$
 $\langle \text{proof} \rangle$

lemma *bin-rec-Bit0*:
 $f3 \ \text{Int.Pl} \ \text{bit.B0} \ f1 = f1 \implies$
 $\text{bin-rec } f1 \ f2 \ f3 \ (\text{Int.Bit0 } w) = f3 \ w \ \text{bit.B0} \ (\text{bin-rec } f1 \ f2 \ f3 \ w)$
 $\langle \text{proof} \rangle$

lemma *bin-rec-Bit1*:
 $f3 \ \text{Int.Min} \ \text{bit.B1} \ f2 = f2 \implies$
 $\text{bin-rec } f1 \ f2 \ f3 \ (\text{Int.Bit1 } w) = f3 \ w \ \text{bit.B1} \ (\text{bin-rec } f1 \ f2 \ f3 \ w)$
 $\langle \text{proof} \rangle$

lemma *bin-rec-Bit*:
 $f = \text{bin-rec } f1 \ f2 \ f3 \implies f3 \ \text{Int.Pl} \ \text{bit.B0} \ f1 = f1 \implies$
 $f3 \ \text{Int.Min} \ \text{bit.B1} \ f2 = f2 \implies f \ (w \ \text{BIT } b) = f3 \ w \ b \ (f \ w)$
 $\langle \text{proof} \rangle$

lemmas *bin-rec-simps* = $\text{refl } [\text{THEN } \text{bin-rec-Bit}] \ \text{bin-rec-Pls} \ \text{bin-rec-Min}$
 $\text{bin-rec-Bit0} \ \text{bin-rec-Bit1}$

4.4 Truncating binary integers

definition

bin-sign-def $[\text{code del}] : \text{bin-sign} = \text{bin-rec } \text{Int.Pl} \ \text{Int.Min} \ (\%w \ b \ s. \ s)$

lemma *bin-sign-simps* $[\text{simp}]$:
 $\text{bin-sign } \text{Int.Pl} = \text{Int.Pl}$
 $\text{bin-sign } \text{Int.Min} = \text{Int.Min}$
 $\text{bin-sign } (\text{Int.Bit0 } w) = \text{bin-sign } w$
 $\text{bin-sign } (\text{Int.Bit1 } w) = \text{bin-sign } w$
 $\text{bin-sign } (w \ \text{BIT } b) = \text{bin-sign } w$
 $\langle \text{proof} \rangle$

declare *bin-sign-simps*(1–4) $[\text{code}]$

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

consts

bintrunc :: $\text{nat} \Rightarrow \text{int} \Rightarrow \text{int}$

primrec

$Z : \text{bintrunc } 0 \ \text{bin} = \text{Int.Pl}$

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

consts

sbintrunc :: $\text{nat} \Rightarrow \text{int} \Rightarrow \text{int}$

primrec

$Z : \text{sbintrunc } 0 \text{ bin} =$
 $(\text{case bin-last bin of bit.B1} \Rightarrow \text{Int.Min} \mid \text{bit.B0} \Rightarrow \text{Int.Pl})$
 $\text{Suc} : \text{sbintrunc } (\text{Suc } n) \text{ bin} = \text{sbintrunc } n \text{ (bin-rest bin) BIT (bin-last bin)}$

lemma *sign-bintr*:

$!!w. \text{bin-sign } (\text{bintrunc } n \ w) = \text{Int.Pl}$
 $\langle \text{proof} \rangle$

lemma *bintrunc-mod2p*:

$!!w. \text{bintrunc } n \ w = (w \bmod 2^n :: \text{int})$
 $\langle \text{proof} \rangle$

lemma *sbintrunc-mod2p*:

$!!w. \text{sbintrunc } n \ w = ((w + 2^n) \bmod 2^{(\text{Suc } n)} - 2^n :: \text{int})$
 $\langle \text{proof} \rangle$

4.5 Simplifications for (s)bintrunc

lemma *bit-bool*:

$(b = (b' = \text{bit.B1})) = (b' = (\text{if } b \text{ then bit.B1 else bit.B0}))$
 $\langle \text{proof} \rangle$

lemmas *bit-bool1* $[\text{simp}] = \text{refl } [\text{THEN bit-bool } [\text{THEN iffD1}], \text{symmetric}]$

lemma *bin-sign-lem*:

$!!\text{bin. } (\text{bin-sign } (\text{sbintrunc } n \ \text{bin}) = \text{Int.Min}) = \text{bin-nth bin } n$
 $\langle \text{proof} \rangle$

lemma *nth-bintr*:

$!!w \ m. \text{bin-nth } (\text{bintrunc } m \ w) \ n = (n < m \ \& \ \text{bin-nth } w \ n)$
 $\langle \text{proof} \rangle$

lemma *nth-sbintr*:

$!!w \ m. \text{bin-nth } (\text{sbintrunc } m \ w) \ n =$
 $(\text{if } n < m \text{ then bin-nth } w \ n \text{ else bin-nth } w \ m)$
 $\langle \text{proof} \rangle$

lemma *bin-nth-Bit*:

$\text{bin-nth } (w \ \text{BIT } b) \ n = (n = 0 \ \& \ b = \text{bit.B1} \mid (\text{EX } m. n = \text{Suc } m \ \& \ \text{bin-nth } w \ m))$
 $\langle \text{proof} \rangle$

lemma *bin-nth-Bit0*:

$\text{bin-nth } (\text{Int.Bit0 } w) \ n = (\text{EX } m. n = \text{Suc } m \ \& \ \text{bin-nth } w \ m)$
 $\langle \text{proof} \rangle$

lemma *bin-nth-Bit1*:

$\text{bin-nth } (\text{Int.Bit1 } w) \ n = (n = 0 \mid (\text{EX } m. n = \text{Suc } m \ \& \ \text{bin-nth } w \ m))$
 $\langle \text{proof} \rangle$

lemma *bintrunc-bintrunc-l*:

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

lemma *sbintrunc-sbintrunc-l*:

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

lemma *bintrunc-bintrunc-ge*:

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

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

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

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

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

lemmas *bintrunc-Pls* =

bintrunc.Suc [where $\text{bin} = \text{Int.Pls}$, simplified bin-last-simps bin-rest-simps, standard]

lemmas *bintrunc-Min* [simp] =

bintrunc.Suc [where $\text{bin} = \text{Int.Min}$, simplified bin-last-simps bin-rest-simps, standard]

lemmas *bintrunc-BIT* [simp] =

bintrunc.Suc [where $\text{bin} = w \text{ BIT } b$, simplified bin-last-simps bin-rest-simps, standard]

lemma *bintrunc-Bit0* [simp]:

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

lemma *bintrunc-Bit1* [simp]:

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

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

bintrunc-Bit0 bintrunc-Bit1

lemmas *sbintrunc-Suc-Pls* =

sbintrunc.Suc [where $\text{bin} = \text{Int.Pls}$, simplified bin-last-simps bin-rest-simps, standard]

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)
 ⟨proof⟩

lemma *sbintrunc-Suc-Bit1* [simp]:
sbintrunc (Suc n) (Int.Bit1 w) = Int.Bit1 (sbintrunc n w)
 ⟨proof⟩

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 bit.B0*,
simplified bin-last-simps bin-rest-simps bit.simps, *standard*]

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

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

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

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

$\langle \text{proof} \rangle$

lemma *sbintrunc-minus*:

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

$\langle \text{proof} \rangle$

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]:

$\text{bintrunc } n \ \text{Int.Pls} = \text{Int.Pls}$

$\langle \text{proof} \rangle$

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

$\text{sbintrunc } n \ \text{Int.Pls} = \text{Int.Pls}$

$\text{sbintrunc } n \ \text{Int.Min} = \text{Int.Min}$

$\langle \text{proof} \rangle$

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*: $\text{bintrunc } 0 \ \text{Int.Min} = \text{Int.Pls}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *bintrunc-Suc-lem*:

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

$\langle \text{proof} \rangle$

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*:
 $\text{sbintrunc } (\text{Suc } n) \ x = y \implies m = \text{Suc } n \implies \text{sbintrunc } m \ x = y$
 ⟨proof⟩

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

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

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

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

lemma *bintrunc-sbintrunc' [simp]*:
 $0 < n \implies \text{bintrunc } n \ (\text{sbintrunc } (n - 1) \ w) = \text{bintrunc } n \ w$
 ⟨proof⟩

lemma *sbintrunc-bintrunc' [simp]*:
 $0 < n \implies \text{sbintrunc } (n - 1) \ (\text{bintrunc } n \ w) = \text{sbintrunc } (n - 1) \ w$
 ⟨proof⟩

lemma *bin-sbin-eq-iff*:
 $\text{bintrunc } (\text{Suc } n) \ x = \text{bintrunc } (\text{Suc } n) \ y \iff$
 $\text{sbintrunc } n \ x = \text{sbintrunc } n \ y$
 ⟨proof⟩

lemma *bin-sbin-eq-iff'*:
 $0 < n \implies \text{bintrunc } n \ x = \text{bintrunc } n \ y \iff$
 $\text{sbintrunc } (n - 1) \ x = \text{sbintrunc } (n - 1) \ y$
 ⟨proof⟩

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

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

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

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

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

lemma *no-bintr*:
number-of (bintrunc n w) = (number-of w mod 2 ^ n :: int)
<proof>

lemma *no-sbintr-alt2*:
sbintrunc n = (%w. (w + 2 ^ n) mod 2 ^ Suc n - 2 ^ n :: int)
<proof>

lemma *no-sbintr*:
number-of (sbintrunc n w) =
((number-of w + 2 ^ n) mod 2 ^ Suc n - 2 ^ n :: int)
<proof>

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

lemma *sb-inc-lem*:
(a::int) + 2 ^ k < 0 ==> a + 2 ^ k + 2 ^ (Suc k) <= (a + 2 ^ k) mod 2 ^ (Suc k)
<proof>

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

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

lemma *sb-dec-lem*:

$(0::int) \leq - (2^k) + a \implies (a + 2^k) \bmod (2 * 2^k) \leq - (2^k) + a$
 ⟨proof⟩

lemma *sb-dec-lem'*:

$(2::int) ^ k \leq a \implies (a + 2^k) \bmod (2 * 2^k) \leq - (2^k) + a$
 ⟨proof⟩

lemma *sbintrunc-dec*:

$x \geq (2^n) \implies x - 2^n (Suc\ n) \geq sbintrunc\ n\ x$
 ⟨proof⟩

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

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

lemmas *brdmod1s'* [symmetric] =
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*: $(0 :: int) \leq \text{number-of } (bintrunc\ n\ w)$
 ⟨proof⟩

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

lemma *bintr-Min*:

$\text{number-of } (bintrunc\ n\ Int.Min) = (2^n :: int) - 1$
 ⟨proof⟩

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

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

$\langle \text{proof} \rangle$

lemma *bintrunc-Suc*:

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

$\langle \text{proof} \rangle$

lemma *sign-Pls-ge-0*:

$(\text{bin-sign bin} = \text{Int.Pls}) = (\text{number-of bin} \geq (0 :: \text{int}))$

$\langle \text{proof} \rangle$

lemma *sign-Min-lt-0*:

$(\text{bin-sign bin} = \text{Int.Min}) = (\text{number-of bin} < (0 :: \text{int}))$

$\langle \text{proof} \rangle$

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

lemma *bin-rest-trunc*:

$!!\text{bin. } (\text{bin-rest } (\text{bintrunc } n \text{ bin})) = \text{bintrunc } (n - 1) \text{ (bin-rest bin)}$

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

lemma *bin-rest-trunc-i*:

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

$\langle \text{proof} \rangle$

lemma *bin-rest-strunc*:

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

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

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

$\langle \text{proof} \rangle$

lemma *bintrunc-rest'*:

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

$\langle \text{proof} \rangle$

lemma *sbintrunc-rest'* :

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

$\langle \text{proof} \rangle$

lemma *rco-lem*:

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

<proof>

lemma *rco-alt*: $(f \circ g) \wedge n \circ f = f \circ (g \circ f) \wedge n$
<proof>

lemmas *rco-bintr* = *bintrunc-rest'*

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

lemmas *rco-sbintr* = *sbintrunc-rest'*

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

4.6 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*))

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*

4.7 Miscellaneous lemmas

lemmas *funpow-minus-simp* =

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

lemmas *funpow-pred-simp* [*simp*] =

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

lemmas *replicate-minus-simp* =

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

lemmas *replicate-pred-simp* [*simp*] =

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

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

lemmas *power-minus-simp* =

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

lemmas *power-pred-simp* =

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

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

lemma *list-exhaust-size-gt0*:

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

```

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

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

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

lemma size-Cons-lem-eq-bin:
 $y = xa \# \text{list} \implies \text{size } y = \text{number-of } (\text{Int.succ } k) \implies$ 
 $\text{size list} = \text{number-of } k$ 
 $\langle \text{proof} \rangle$ 

lemmas ls-splits =
 $\text{prod.split split-split prod.split-asm split-split-asm split-if-asm}$ 

lemma not-B1-is-B0:  $y \neq \text{bit.B1} \implies y = \text{bit.B0}$ 
 $\langle \text{proof} \rangle$ 

lemma B1-ass-B0:
assumes  $y: y = \text{bit.B0} \implies y = \text{bit.B1}$ 
shows  $y = \text{bit.B1}$ 
 $\langle \text{proof} \rangle$ 

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

lemmas s2n-ths = n2s-ths [symmetric]

end

```

5 BitSyntax: Syntactic classes for bitwise operations

```

theory BitSyntax
imports BinGeneral
begin

```

```

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

```

We want the bitwise operations to bind slightly weaker than $+$ and $-$, but

$\sim\sim$ to bind slightly stronger than $*$.

Testing and shifting operations.

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

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

5.1 Bitwise operations on *bit*

```
instantiation bit :: bit
begin
```

```
primrec bitNOT-bit where
  NOT bit.B0 = bit.B1
| NOT bit.B1 = bit.B0
```

```
primrec bitAND-bit where
  bit.B0 AND y = bit.B0
| bit.B1 AND y = y
```

```
primrec bitOR-bit where
  bit.B0 OR y = y
| bit.B1 OR y = bit.B1
```

```
primrec bitXOR-bit where
  bit.B0 XOR y = y
| bit.B1 XOR y = NOT y
```

```
instance <proof>
```

```
end
```

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

```
lemma bit-extra-simps [simp]:
  x AND bit.B0 = bit.B0
  x AND bit.B1 = x
  x OR bit.B1 = bit.B1
  x OR bit.B0 = x
  x XOR bit.B1 = NOT x
  x XOR bit.B0 = x
```

$\langle proof \rangle$

lemma *bit-ops-comm*:

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

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

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

$\langle proof \rangle$

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

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

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

$(x::bit) \text{ XOR } x = bit.B0$

$\langle proof \rangle$

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

$\langle proof \rangle$

end

6 BinOperations: Bitwise Operations on Binary Integers

theory *BinOperations*

imports *BinGeneral BitSyntax*

begin

6.1 Logical operations

bit-wise logical operations on the int type

instantiation *int* :: *bit*

begin

definition

int-not-def [code del]: $bitNOT = bin-rec \ Int.Min \ Int.Plts$
 $(\lambda w \ b \ s. \ s \ BIT \ (NOT \ b))$

definition

int-and-def [code del]: $bitAND = bin-rec \ (\lambda x. \ Int.Plts) \ (\lambda y. \ y)$
 $(\lambda w \ b \ s \ y. \ s \ (bin-rest \ y) \ BIT \ (b \ AND \ bin-last \ y))$

definition

int-or-def [code del]: $bitOR = bin-rec \ (\lambda x. \ x) \ (\lambda y. \ Int.Min)$
 $(\lambda w \ b \ s \ y. \ s \ (bin-rest \ y) \ BIT \ (b \ OR \ bin-last \ y))$

definition

int-xor-def [code del]: $bitXOR = bin-rec \ (\lambda x. \ x) \ bitNOT$
 $(\lambda w \ b \ s \ y. \ s \ (bin-rest \ y) \ BIT \ (b \ XOR \ bin-last \ y))$

instance $\langle proof \rangle$

end

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

$NOT\ Int.Pls = Int.Min$
 $NOT\ Int.Min = Int.Pls$
 $NOT\ (Int.Bit0\ w) = Int.Bit1\ (NOT\ w)$
 $NOT\ (Int.Bit1\ w) = Int.Bit0\ (NOT\ w)$
 $NOT\ (w\ BIT\ b) = (NOT\ w)\ BIT\ (NOT\ b)$
 $\langle proof \rangle$

declare *int-not-simps*(1–4) [code]

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

$Int.Pls\ XOR\ x = x$
 $\langle proof \rangle$

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

$Int.Min\ XOR\ x = NOT\ x$
 $\langle proof \rangle$

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

$(x\ BIT\ b)\ XOR\ (y\ BIT\ c) = (x\ XOR\ y)\ BIT\ (b\ XOR\ c)$
 $\langle proof \rangle$

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

$(Int.Bit0\ x)\ XOR\ (Int.Bit0\ y) = Int.Bit0\ (x\ XOR\ y)$
 $(Int.Bit0\ x)\ XOR\ (Int.Bit1\ y) = Int.Bit1\ (x\ XOR\ y)$
 $(Int.Bit1\ x)\ XOR\ (Int.Bit0\ y) = Int.Bit1\ (x\ XOR\ y)$
 $(Int.Bit1\ x)\ XOR\ (Int.Bit1\ y) = Int.Bit0\ (x\ XOR\ y)$
 $\langle proof \rangle$

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

$w\ XOR\ (Int.Pls\ BIT\ bit.B0) = w$
 $w\ XOR\ (Int.Min\ BIT\ bit.B1) = NOT\ w$
 $\langle proof \rangle$

lemma *int-xor-extra-simps* [simp, code]:

$w\ XOR\ Int.Pls = w$
 $w\ XOR\ Int.Min = NOT\ w$
 $\langle proof \rangle$

lemma *int-or-Pls* [simp, code]:

$Int.Pls\ OR\ x = x$
 $\langle proof \rangle$

lemma *int-or-Min* [simp, code]:

$Int.Min \ OR \ x = Int.Min$
 $\langle proof \rangle$

lemma *int-or-Bits* [simp]:
 $(x \ BIT \ b) \ OR \ (y \ BIT \ c) = (x \ OR \ y) \ BIT \ (b \ OR \ c)$
 $\langle proof \rangle$

lemma *int-or-Bits2* [simp, code]:
 $(Int.Bit0 \ x) \ OR \ (Int.Bit0 \ y) = Int.Bit0 \ (x \ OR \ y)$
 $(Int.Bit0 \ x) \ OR \ (Int.Bit1 \ y) = Int.Bit1 \ (x \ OR \ y)$
 $(Int.Bit1 \ x) \ OR \ (Int.Bit0 \ y) = Int.Bit1 \ (x \ OR \ y)$
 $(Int.Bit1 \ x) \ OR \ (Int.Bit1 \ y) = Int.Bit1 \ (x \ OR \ y)$
 $\langle proof \rangle$

lemma *int-or-x-simps'*:
 $w \ OR \ (Int.Pls \ BIT \ bit.B0) = w$
 $w \ OR \ (Int.Min \ BIT \ bit.B1) = Int.Min$
 $\langle proof \rangle$

lemma *int-or-extra-simps* [simp, code]:
 $w \ OR \ Int.Pls = w$
 $w \ OR \ Int.Min = Int.Min$
 $\langle proof \rangle$

lemma *int-and-Pls* [simp, code]:
 $Int.Pls \ AND \ x = Int.Pls$
 $\langle proof \rangle$

lemma *int-and-Min* [simp, code]:
 $Int.Min \ AND \ x = x$
 $\langle proof \rangle$

lemma *int-and-Bits* [simp]:
 $(x \ BIT \ b) \ AND \ (y \ BIT \ c) = (x \ AND \ y) \ BIT \ (b \ AND \ c)$
 $\langle proof \rangle$

lemma *int-and-Bits2* [simp, code]:
 $(Int.Bit0 \ x) \ AND \ (Int.Bit0 \ y) = Int.Bit0 \ (x \ AND \ y)$
 $(Int.Bit0 \ x) \ AND \ (Int.Bit1 \ y) = Int.Bit0 \ (x \ AND \ y)$
 $(Int.Bit1 \ x) \ AND \ (Int.Bit0 \ y) = Int.Bit0 \ (x \ AND \ y)$
 $(Int.Bit1 \ x) \ AND \ (Int.Bit1 \ y) = Int.Bit1 \ (x \ AND \ y)$
 $\langle proof \rangle$

lemma *int-and-x-simps'*:
 $w \ AND \ (Int.Pls \ BIT \ bit.B0) = Int.Pls$
 $w \ AND \ (Int.Min \ BIT \ bit.B1) = w$
 $\langle proof \rangle$

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

$w \text{ AND } \text{Int.Pls} = \text{Int.Pls}$
 $w \text{ AND } \text{Int.Min} = w$
 $\langle \text{proof} \rangle$

lemma *bin-ops-comm*:

shows

$\text{int-and-comm}: !!y::\text{int}. x \text{ AND } y = y \text{ AND } x$ **and**
 $\text{int-or-comm}: !!y::\text{int}. x \text{ OR } y = y \text{ OR } x$ **and**
 $\text{int-xor-comm}: !!y::\text{int}. x \text{ XOR } y = y \text{ XOR } x$
 $\langle \text{proof} \rangle$

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

$(x::\text{int}) \text{ AND } x = x$
 $(x::\text{int}) \text{ OR } x = x$
 $(x::\text{int}) \text{ XOR } x = \text{Int.Pls}$
 $\langle \text{proof} \rangle$

lemma *int-not-not* [simp]: $\text{NOT } (\text{NOT } (x::\text{int})) = x$
 $\langle \text{proof} \rangle$

lemmas *bin-log-esimps* =

$\text{int-and-extra-simps}$ $\text{int-or-extra-simps}$ $\text{int-xor-extra-simps}$
 int-and-Pls int-and-Min int-or-Pls int-or-Min int-xor-Pls int-xor-Min

lemma *bbw-ao-absorb*:

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

lemma *bbw-ao-absorbs-other*:

$x \text{ AND } (x \text{ OR } y) = x \wedge (y \text{ AND } x) \text{ OR } x = (x::\text{int})$
 $(y \text{ OR } x) \text{ AND } x = x \wedge x \text{ OR } (x \text{ AND } y) = (x::\text{int})$
 $(x \text{ OR } y) \text{ AND } x = x \wedge (x \text{ AND } y) \text{ OR } x = (x::\text{int})$
 $\langle \text{proof} \rangle$

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

lemma *int-xor-not*:

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

lemma *bbw-assocs'*:

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

lemma *int-and-assoc*:

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

<proof>

lemma *int-or-assoc*:

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

<proof>

lemma *int-xor-assoc*:

$$(x \text{ XOR } y) \text{ XOR } (z::\text{int}) = x \text{ XOR } (y \text{ XOR } z)$$

<proof>

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

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

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

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

$$(y::\text{int}) \text{ XOR } (x \text{ XOR } z) = x \text{ XOR } (y \text{ XOR } z)$$

<proof>

lemma *bbw-not-dist*:

$$!!y::\text{int}. \text{ NOT } (x \text{ OR } y) = (\text{ NOT } x) \text{ AND } (\text{ NOT } y)$$

$$!!y::\text{int}. \text{ NOT } (x \text{ AND } y) = (\text{ NOT } x) \text{ OR } (\text{ NOT } y)$$

<proof>

lemma *bbw-oa-dist*:

$$!!y \ z::\text{int}. (x \text{ AND } y) \text{ OR } z =$$

$$(x \text{ OR } z) \text{ AND } (y \text{ OR } z)$$

<proof>

lemma *bbw-ao-dist*:

$$!!y \ z::\text{int}. (x \text{ OR } y) \text{ AND } z =$$

$$(x \text{ AND } z) \text{ OR } (y \text{ AND } z)$$

<proof>

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

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

<proof>

lemma *le-int-or*:

$$!!x. \text{ bin-sign } y = \text{ Int.Pls } ==> x \leq x \text{ OR } y$$

<proof>

lemmas *int-and-le* =

$$\text{ xtr3 } [\text{ OF } \text{ bbw-ao-absorbs } (2) \text{ [THEN } \text{ conjunct2, symmetric }] \text{ le-int-or }]$$

lemma *bin-nth-ops*:

!! $x\ y$. $\text{bin-nth } (x \text{ AND } y) \ n = (\text{bin-nth } x \ n \ \& \ \text{bin-nth } y \ n)$
 !! $x\ y$. $\text{bin-nth } (x \text{ OR } y) \ n = (\text{bin-nth } x \ n \ | \ \text{bin-nth } y \ n)$
 !! $x\ y$. $\text{bin-nth } (x \text{ XOR } y) \ n = (\text{bin-nth } x \ n \ \sim = \text{bin-nth } y \ n)$
 !! x . $\text{bin-nth } (\text{NOT } x) \ n = (\sim \text{bin-nth } x \ n)$
 $\langle \text{proof} \rangle$

lemma *bin-add-not*: $x + \text{NOT } x = \text{Int.Min}$

$\langle \text{proof} \rangle$

lemma *bin-trunc-ao*:

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

lemma *bin-trunc-xor*:

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

lemma *bin-trunc-not*:

!! x . $\text{bintrunc } n \ (\text{NOT } (\text{bintrunc } n \ x)) = \text{bintrunc } n \ (\text{NOT } x)$
 $\langle \text{proof} \rangle$

lemma *bintr-bintr-i*:

$x = \text{bintrunc } n \ y \implies \text{bintrunc } n \ x = \text{bintrunc } n \ y$
 $\langle \text{proof} \rangle$

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

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

6.2 Setting and clearing bits

primrec

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

where

Z : $\text{bin-sc } 0 \ b \ w = \text{bin-rest } w \ \text{BIT } b$
 $| \text{Suc}$: $\text{bin-sc } (\text{Suc } n) \ b \ w = \text{bin-sc } n \ b \ (\text{bin-rest } w) \ \text{BIT } \text{bin-last } w$

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

!! w . $\text{bin-nth } (\text{bin-sc } n \ b \ w) \ n = (b = \text{bit.B1})$
 $\langle \text{proof} \rangle$

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

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

⟨proof⟩

lemma *bin-sc-sc-diff*:

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

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

⟨proof⟩

lemma *bin-nth-sc-gen*:

!!w m. *bin-nth* (*bin-sc* n b w) m = (if m = n then b = *bit.B1* else *bin-nth* w m)

⟨proof⟩

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

!!w. (*bin-sc* n (If (*bin-nth* w n) *bit.B1* *bit.B0*) w) = w

⟨proof⟩

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

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

⟨proof⟩

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

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

⟨proof⟩

lemma *bin-clr-le*:

!!w. *bin-sc* n *bit.B0* w <= w

⟨proof⟩

lemma *bin-set-ge*:

!!w. *bin-sc* n *bit.B1* w >= w

⟨proof⟩

lemma *bintr-bin-clr-le*:

!!w m. *bintrunc* n (*bin-sc* m *bit.B0* w) <= *bintrunc* n w

⟨proof⟩

lemma *bintr-bin-set-ge*:

!!w m. *bintrunc* n (*bin-sc* m *bit.B1* w) >= *bintrunc* n w

⟨proof⟩

lemma *bin-sc-FP* [simp]: *bin-sc* n *bit.B0* *Int.Pls* = *Int.Pls*

⟨proof⟩

lemma *bin-sc-TM* [simp]: *bin-sc* n *bit.B1* *Int.Min* = *Int.Min*

⟨proof⟩

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

lemma *bin-sc-minus*:

$0 < n \implies \text{bin-sc } (\text{Suc } (n - 1)) \text{ } b \text{ } w = \text{bin-sc } n \text{ } b \text{ } w$
 $\langle \text{proof} \rangle$

lemmas *bin-sc-Suc-minus* =

trans [OF bin-sc-minus [symmetric] bin-sc.Suc, standard]

lemmas *bin-sc-Suc-pred [simp]* =

bin-sc-Suc-minus [of number-of bin, simplified nobm1, standard]

6.3 Operations on lists of booleans

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 bit.B1 else bit.B0))

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

bl-to-bin-def : *bl-to-bin* *bs* = *bl-to-bin-aux* *bs* *Int.Pls*

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* = bit.B1) # *bl*)

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

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

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

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

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

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

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

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

6.4 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*

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-rsplittl-aux* :: *nat* \Rightarrow *nat* \Rightarrow *int* \Rightarrow *int list* \Rightarrow *int list* **where**
bin-rsplittl-aux n m c bs =
(if m = 0 | n = 0 then bs else
let (a, b) = bin-split (min m n) c
in bin-rsplittl-aux n (m - n) a (b # bs))

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

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

lemma *bin-sign-cat*:
 !!y. *bin-sign (bin-cat x n y) = bin-sign x*
 <proof>

lemma *bin-cat-Suc-Bit*:
bin-cat w (Suc n) (v BIT b) = bin-cat w n v BIT b
 <proof>

lemma *bin-nth-cat*:
 !!n y. *bin-nth (bin-cat x k y) n =*
(if n < k then bin-nth y n else bin-nth x (n - k))
 <proof>

lemma *bin-nth-split*:
 !!b c. *bin-split n c = (a, b) ==>*
(ALL k. bin-nth a k = bin-nth c (n + k)) &
(ALL k. bin-nth b k = (k < n & bin-nth c k))
 <proof>

lemma *bin-cat-assoc*:
 !!z. *bin-cat (bin-cat x m y) n z = bin-cat x (m + n) (bin-cat y n z)*
 <proof>

lemma *bin-cat-assoc-sym*: !!z m.
bin-cat x m (bin-cat y n z) = bin-cat (bin-cat x (m - n) y) (min m n) z
 <proof>

lemma *bin-cat-Pls* [*simp*]:
 !!w. *bin-cat Int.Pls n w = bintrunc n w*
 <proof>

lemma *bintr-cat1*:

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

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

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

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

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

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

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

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

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

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

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

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

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

6.5 Miscellaneous lemmas

lemma *nth-2p-bin*:
 $!!m. \text{bin-nth } (2 \wedge n) \ m = (m = n)$
 $\langle \text{proof} \rangle$

lemma *ex-eq-or*:
 $(EX \ m. n = \text{Suc } m \ \& \ (m = k \mid P \ m)) = (n = \text{Suc } k \mid (EX \ m. n = \text{Suc } m \ \& \ P \ m))$
 $\langle \text{proof} \rangle$

end

7 BinBoolList: Bool lists and integers

theory *BinBoolList*
imports *BinOperations*
begin

7.1 Arithmetic in terms of bool lists

primrec *rbl-succ* :: *bool list* => *bool list* **where**
 $Nil: \text{rbl-succ } Nil = Nil$
 $\mid \text{Cons: } \text{rbl-succ } (x \# xs) = (\text{if } x \text{ then } False \# \text{rbl-succ } xs \text{ else } True \# xs)$

primrec *rbl-pred* :: *bool list* => *bool list* **where**
 $Nil: \text{rbl-pred } Nil = Nil$
 $\mid \text{Cons: } \text{rbl-pred } (x \# xs) = (\text{if } x \text{ then } False \# xs \text{ else } True \# \text{rbl-pred } xs)$

primrec *rbl-add* :: *bool list* => *bool list* => *bool list* **where**

$Nil: \text{rbl-add } Nil \ x = Nil$
 $\mid \text{Cons: } \text{rbl-add } (y \# ys) \ x = (\text{let } ws = \text{rbl-add } ys \ (\text{tl } x) \text{ in } (y \sim = \text{hd } x) \# (\text{if } \text{hd } x \ \& \ y \text{ then } \text{rbl-succ } ws \text{ else } ws))$

primrec *rbl-mult* :: *bool list* => *bool list* => *bool list* **where**

$Nil: \text{rbl-mult } Nil \ x = Nil$
 $\mid \text{Cons: } \text{rbl-mult } (y \# ys) \ x = (\text{let } ws = False \# \text{rbl-mult } ys \ x \text{ in } \text{if } y \text{ then } \text{rbl-add } ws \ x \text{ else } ws)$

lemma *butlast-power*:

$(\text{butlast } \wedge n) \text{ bl} = \text{take } (\text{length bl} - n) \text{ bl}$
 $\langle \text{proof} \rangle$

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

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

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

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

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

$0 < n \implies \text{bin-to-bl-aux } n \text{ (w BIT b) bl} =$
 $\text{bin-to-bl-aux } (n - 1) \text{ w } ((b = \text{bit.B1}) \# \text{bl})$
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

lemma *bl-to-bin-append*:

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

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

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

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

$\langle \text{proof} \rangle$

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

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

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

$\langle \text{proof} \rangle$

lemma *bin-bl-bin'*:

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

lemma *bin-bl-bin*: $\text{bl-to-bin } (\text{bin-to-bl } n \ w) = \text{bintrunc } n \ w$

$\langle \text{proof} \rangle$

lemma *bl-bin-bl'*:

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

lemma *bl-bin-bl*: $\text{bin-to-bl } (\text{length } bs) \ (\text{bl-to-bin } bs) = bs$

$\langle \text{proof} \rangle$

declare

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

lemma *bl-to-bin-inj*:

$\text{bl-to-bin } bs = \text{bl-to-bin } cs \implies \text{length } bs = \text{length } cs \implies bs = cs$
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

lemma *bl-to-bin-Nil*: $\text{bl-to-bin } [] = \text{Int.Pls}$

$\langle \text{proof} \rangle$

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

$\text{bin-to-bl-aux } n \ \text{Int.Pls } \text{bl} = \text{replicate } n \ \text{False} \ @ \ \text{bl}$
 $\langle \text{proof} \rangle$

lemma *bin-to-bl-Pls*: $\text{bin-to-bl } n \ \text{Int.Pls} = \text{replicate } n \ \text{False}$

$\langle \text{proof} \rangle$

lemma *bin-to-bl-Min-aux* [rule-format] :

ALL *bl*. $\text{bin-to-bl-aux } n \ \text{Int.Min } \text{bl} = \text{replicate } n \ \text{True} \ @ \ \text{bl}$

$\langle \text{proof} \rangle$

lemma *bin-to-bl-Min*: $\text{bin-to-bl } n \text{ Int.Min} = \text{replicate } n \text{ True}$
 $\langle \text{proof} \rangle$

lemma *bl-to-bin-rep-F*:
 $\text{bl-to-bin } (\text{replicate } n \text{ False } @ \text{ bl}) = \text{bl-to-bin bl}$
 $\langle \text{proof} \rangle$

lemma *bin-to-bl-trunc*:
 $n \leq m \implies \text{bin-to-bl } n \text{ (bintrunc } m \text{ w)} = \text{bin-to-bl } n \text{ w}$
 $\langle \text{proof} \rangle$

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

lemma *bin-to-bl-aux-bintr* $[\text{rule-format}]$:
 $\text{ALL } m \text{ bin bl. } \text{bin-to-bl-aux } n \text{ (bintrunc } m \text{ bin)} \text{ bl} =$
 $\text{replicate } (n - m) \text{ False } @ \text{ bin-to-bl-aux } (\text{min } n \text{ m)} \text{ bin bl}$
 $\langle \text{proof} \rangle$

lemmas *bin-to-bl-bintr* =
 $\text{bin-to-bl-aux-bintr } [\text{where bl} = [], \text{ folded bin-to-bl-def}]$

lemma *bl-to-bin-rep-False*: $\text{bl-to-bin } (\text{replicate } n \text{ False}) = \text{Int.Pl}$
 $\langle \text{proof} \rangle$

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

lemma *len-bin-to-bl* $[\text{simp}]$: $\text{length } (\text{bin-to-bl } n \text{ w}) = n$
 $\langle \text{proof} \rangle$

lemma *sign-bl-bin'*:
 $\text{bin-sign } (\text{bl-to-bin-aux bs w}) = \text{bin-sign w}$
 $\langle \text{proof} \rangle$

lemma *sign-bl-bin*: $\text{bin-sign } (\text{bl-to-bin bs}) = \text{Int.Pl}$
 $\langle \text{proof} \rangle$

lemma *bl-sbin-sign-aux*:
 $\text{hd } (\text{bin-to-bl-aux } (\text{Suc } n) \text{ w bs}) =$
 $(\text{bin-sign } (\text{sbintrunc } n \text{ w}) = \text{Int.Min})$
 $\langle \text{proof} \rangle$

lemma *bl-sbin-sign*:

$hd\ (bin\text{-}to\text{-}bl\ (Suc\ n)\ w) = (bin\text{-}sign\ (sbintrunc\ n\ w) = Int.Min)$
 $\langle proof \rangle$

lemma *bin-nth-of-bl-aux* [rule-format]:
 $\forall w. bin\text{-}nth\ (bl\text{-}to\text{-}bin\text{-}aux\ bl\ w)\ n =$
 $(n < size\ bl \ \&\ rev\ bl\ !\ n \mid n >= length\ bl \ \&\ bin\text{-}nth\ w\ (n - size\ bl))$
 $\langle proof \rangle$

lemma *bin-nth-of-bl*: $bin\text{-}nth\ (bl\text{-}to\text{-}bin\ bl)\ n = (n < length\ bl \ \&\ rev\ bl\ !\ n)$
 $\langle proof \rangle$

lemma *bin-nth-bl* [rule-format] : $ALL\ m\ w. n < m \longrightarrow$
 $bin\text{-}nth\ w\ n = nth\ (rev\ (bin\text{-}to\text{-}bl\ m\ w))\ n$
 $\langle proof \rangle$

lemma *nth-rev* [rule-format] :
 $n < length\ xs \longrightarrow rev\ xs\ !\ n = xs\ !\ (length\ xs - 1 - n)$
 $\langle proof \rangle$

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

lemma *nth-bin-to-bl-aux* [rule-format] :
 $ALL\ w\ n\ bl. n < m + length\ bl \longrightarrow (bin\text{-}to\text{-}bl\text{-}aux\ m\ w\ bl)\ !\ n =$
 $(if\ n < m\ then\ bin\text{-}nth\ w\ (m - 1 - n)\ else\ bl\ !\ (n - m))$
 $\langle proof \rangle$

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

lemma *bl-to-bin-lt2p-aux* [rule-format]:
 $\forall w. bl\text{-}to\text{-}bin\text{-}aux\ bs\ w < (w + 1) * (2 ^ length\ bs)$
 $\langle proof \rangle$

lemma *bl-to-bin-lt2p*: $bl\text{-}to\text{-}bin\ bs < (2 ^ length\ bs)$
 $\langle proof \rangle$

lemma *bl-to-bin-ge2p-aux* [rule-format] :
 $\forall w. bl\text{-}to\text{-}bin\text{-}aux\ bs\ w >= w * (2 ^ length\ bs)$
 $\langle proof \rangle$

lemma *bl-to-bin-ge0*: $bl\text{-}to\text{-}bin\ bs >= 0$
 $\langle proof \rangle$

lemma *butlast-rest-bin*:
 $butlast\ (bin\text{-}to\text{-}bl\ n\ w) = bin\text{-}to\text{-}bl\ (n - 1)\ (bin\text{-}rest\ w)$
 $\langle proof \rangle$

lemmas *butlast-bin-rest* = *butlast-rest-bin*
[where $w = bl\text{-}to\text{-}bin\ bl$ and $n = length\ bl$, simplified, standard]

lemma *butlast-rest-bl2bin-aux*:

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

lemma *butlast-rest-bl2bin*:

$bl\text{-}to\text{-}bin\ (butlast\ bl) = bin\text{-}rest\ (bl\text{-}to\text{-}bin\ bl)$
 $\langle proof \rangle$

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

$ALL\ w.\ bintrunc\ m\ (bl\text{-}to\text{-}bin\text{-}aux\ bl\ w) =$
 $bl\text{-}to\text{-}bin\text{-}aux\ (drop\ (length\ bl - m)\ bl)\ (bintrunc\ (m - length\ bl)\ w)$
 $\langle proof \rangle$

lemma *trunc-bl2bin*:

$bintrunc\ m\ (bl\text{-}to\text{-}bin\ bl) = bl\text{-}to\text{-}bin\ (drop\ (length\ bl - m)\ bl)$
 $\langle proof \rangle$

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

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

lemma *bl2bin-drop*:

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

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

$ALL\ n.\ bin\text{-}nth\ ((bin\text{-}rest\ ^ k)\ w)\ n = bin\text{-}nth\ w\ (n + k)$
 $\langle proof \rangle$

lemma *take-rest-power-bin*:

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

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

$\langle proof \rangle$

lemma *last-bin-last'*:

$size\ xs > 0 \implies last\ xs = (bin\text{-}last\ (bl\text{-}to\text{-}bin\text{-}aux\ xs\ w) = bit.B1)$
 $\langle proof \rangle$

lemma *last-bin-last*:

$size\ xs > 0 \implies last\ xs = (bin\text{-}last\ (bl\text{-}to\text{-}bin\ xs) = bit.B1)$
 $\langle proof \rangle$

lemma *bin-last-last*:

$bin\text{-}last\ w = (if\ last\ (bin\text{-}to\text{-}bl\ (Suc\ n)\ w)\ then\ bit.B1\ else\ bit.B0)$
 $\langle proof \rangle$

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

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

lemma *bl-xor-aux-bin* [*rule-format*] : *ALL* $v \ w \ bs \ cs$.
 $\text{map2 } (\%x \ y. \ x \ \sim = \ y) \ (\text{bin-to-bl-aux } n \ v \ bs) \ (\text{bin-to-bl-aux } n \ w \ cs) =$
 $\text{bin-to-bl-aux } n \ (v \ XOR \ w) \ (\text{map2 } (\%x \ y. \ x \ \sim = \ y) \ bs \ cs)$
 ⟨*proof*⟩

lemma *bl-or-aux-bin* [*rule-format*] : *ALL* $v \ w \ bs \ cs$.
 $\text{map2 } (op \ | \) \ (\text{bin-to-bl-aux } n \ v \ bs) \ (\text{bin-to-bl-aux } n \ w \ cs) =$
 $\text{bin-to-bl-aux } n \ (v \ OR \ w) \ (\text{map2 } (op \ | \) \ bs \ cs)$
 ⟨*proof*⟩

lemma *bl-and-aux-bin* [*rule-format*] : *ALL* $v \ w \ bs \ cs$.
 $\text{map2 } (op \ \& \) \ (\text{bin-to-bl-aux } n \ v \ bs) \ (\text{bin-to-bl-aux } n \ w \ cs) =$
 $\text{bin-to-bl-aux } n \ (v \ AND \ w) \ (\text{map2 } (op \ \& \) \ bs \ cs)$
 ⟨*proof*⟩

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

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

lemmas *bl-and-bin* = *bl-and-aux-bin* [where $bs=[]$ and $cs=[]$,
unfolded 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 $m \ bin \ bs$. $\text{drop } m \ (\text{bin-to-bl-aux } n \ bin \ bs) =$
 $\text{bin-to-bl-aux } (n - m) \ bin \ (\text{drop } (m - n) \ bs)$
 ⟨*proof*⟩

lemma *drop-bin2bl*: $\text{drop } m \ (\text{bin-to-bl } n \ bin) = \text{bin-to-bl } (n - m) \ bin$
 ⟨*proof*⟩

lemma *take-bin2bl-lem1* [rule-format] :
 $ALL\ w\ bs.\ take\ m\ (bin\text{-}to\text{-}bl\text{-}aux\ m\ w\ bs) = bin\text{-}to\text{-}bl\ m\ w$
 ⟨proof⟩

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

lemma *bin-split-take* [rule-format] :
 $ALL\ b\ c.\ bin\text{-}split\ n\ c = (a, b) \dashrightarrow$
 $bin\text{-}to\text{-}bl\ m\ a = take\ m\ (bin\text{-}to\text{-}bl\ (m + n)\ c)$
 ⟨proof⟩

lemma *bin-split-take1*:
 $k = m + n \implies bin\text{-}split\ n\ c = (a, b) \implies$
 $bin\text{-}to\text{-}bl\ m\ a = take\ m\ (bin\text{-}to\text{-}bl\ k\ c)$
 ⟨proof⟩

lemma *nth-takefill* [rule-format] : $ALL\ m\ l.\ m < n \dashrightarrow$
 $takefill\ fill\ n\ l\ !\ m = (if\ m < length\ l\ then\ l\ !\ m\ else\ fill)$
 ⟨proof⟩

lemma *takefill-alt* [rule-format] :
 $ALL\ l.\ takefill\ fill\ n\ l = take\ n\ l\ @\ replicate\ (n - length\ l)\ fill$
 ⟨proof⟩

lemma *takefill-replicate* [simp]:
 $takefill\ fill\ n\ (replicate\ m\ fill) = replicate\ n\ fill$
 ⟨proof⟩

lemma *takefill-le'* [rule-format] :
 $ALL\ l\ n.\ n = m + k \dashrightarrow takefill\ x\ m\ (takefill\ x\ n\ l) = takefill\ x\ m\ l$
 ⟨proof⟩

lemma *length-takefill* [simp]: $length\ (takefill\ fill\ n\ l) = n$
 ⟨proof⟩

lemma *take-takefill'*:
 $!!w\ n.\ n = k + m \implies take\ k\ (takefill\ fill\ n\ w) = takefill\ fill\ k\ w$
 ⟨proof⟩

lemma *drop-takefill*:
 $!!w.\ drop\ k\ (takefill\ fill\ (m + k)\ w) = takefill\ fill\ m\ (drop\ k\ w)$
 ⟨proof⟩

lemma *takefill-le* [simp]:
 $m \leq n \implies takefill\ x\ m\ (takefill\ x\ n\ l) = takefill\ x\ m\ l$
 ⟨proof⟩

lemma *take-takefill [simp]*:

$$m \leq n \implies \text{take } m \ (\text{takefill fill } n \ w) = \text{takefill fill } m \ w$$

<proof>

lemma *takefill-append*:

$$\text{takefill fill } (m + \text{length } xs) \ (xs \ @ \ w) = xs \ @ \ (\text{takefill fill } m \ w)$$

<proof>

lemma *takefill-same'*:

$$l = \text{length } xs \implies \text{takefill fill } l \ xs = xs$$

<proof>

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

lemma *takefill-bintrunc*:

$$\text{takefill False } n \ bl = \text{rev } (\text{bin-to-bl } n \ (\text{bl-to-bin } (\text{rev } bl)))$$

<proof>

lemma *bl-bin-bl-rtf*:

$$\text{bin-to-bl } n \ (\text{bl-to-bin } bl) = \text{rev } (\text{takefill False } n \ (\text{rev } bl))$$

<proof>

lemmas *bl-bin-bl-rep-drop =*

$$\text{bl-bin-bl-rtf } [\text{simplified takefill-alt}, \\ \text{simplified, simplified rev-take, simplified}]$$

lemma *tf-rev*:

$$n + k = m + \text{length } bl \implies \text{takefill } x \ m \ (\text{rev } (\text{takefill } y \ n \ bl)) = \\ \text{rev } (\text{takefill } y \ m \ (\text{rev } (\text{takefill } x \ k \ (\text{rev } bl))))$$

<proof>

lemma *takefill-minus*:

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

<proof>

lemmas *takefill-Suc-cases =*

$$\text{list.cases } [\text{THEN takefill.Suc } [\text{THEN trans}], \text{ standard}]$$

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]*

$$\text{takefill-minus } [\text{symmetric, THEN trans}], \text{ standard}]$$

lemmas *takefill-pred-simps [simp] =*

$$\text{takefill-minus-simps } [\text{where } n = \text{number-of bin, simplified nobm1, standard}]$$

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

!!*nv v. bl-to-bin-aux bs (bin-cat w nv v) =
 bin-cat w (nv + length bs) (bl-to-bin-aux bs v)*
<proof>

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

!!*w bs. bin-to-bl-aux (nv + nw) (bin-cat v nw w) bs =
 bin-to-bl-aux nv v (bin-to-bl-aux nw w bs)*
<proof>

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.Pls, folded bl-to-bin-def]

lemmas *bin-to-bl-cat-app* = *bin-to-bl-aux-cat-app*

[where bs = [], folded bin-to-bl-def]

lemma *bl-to-bin-app-cat-alt*:

bin-cat (bl-to-bin cs) n w = bl-to-bin (cs @ bin-to-bl n w)
<proof>

lemma *mask-lem*: *(bl-to-bin (True # replicate n False)) =*

Int.succ (bl-to-bin (replicate n True))
<proof>

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

<proof>

lemma *nth-bl-of-nth [simp]*:

m < n ==> rev (bl-of-nth n f) ! m = f m
<proof>

lemma *bl-of-nth-inj*:

(!!k. k < n ==> f k = g k) ==> bl-of-nth n f = bl-of-nth n g

$\langle \text{proof} \rangle$

lemma *bl-of-nth-nth-le* [rule-format] : *ALL xs.*

length xs >= n --> bl-of-nth n (nth (rev xs)) = drop (length xs - n) xs

$\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

lemma *size-rbl-succ*: *length (rbl-succ bl) = length bl*

$\langle \text{proof} \rangle$

lemma *size-rbl-add*:

!!cl. length (rbl-add bl cl) = length bl

$\langle \text{proof} \rangle$

lemma *size-rbl-mult*:

!!cl. length (rbl-mult bl cl) = length bl

$\langle \text{proof} \rangle$

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))

$\langle \text{proof} \rangle$

lemma *rbl-succ*:

!!bin. rbl-succ (rev (bin-to-bl n bin)) = rev (bin-to-bl n (Int.succ bin))

$\langle \text{proof} \rangle$

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))

$\langle \text{proof} \rangle$

lemma *rbl-add-app2*:

!!blb. length blb >= length bla ==>

rbl-add bla (blb @ blc) = rbl-add bla blb

$\langle \text{proof} \rangle$

lemma *rbl-add-take2*:

!!blb. length blb >= length bla ==>

rbl-add bla (take (length bla) blb) = rbl-add bla blb

$\langle \text{proof} \rangle$

lemma *rbl-add-long*:

$m \geq n \implies \text{rbl-add } (\text{rev } (\text{bin-to-bl } n \text{ bina})) (\text{rev } (\text{bin-to-bl } m \text{ binb})) =$
 $\text{rev } (\text{bin-to-bl } n \text{ (bina + binb)})$
 $\langle \text{proof} \rangle$

lemma *rbl-mult-app2*:

$!!\text{blb}. \text{length blb} \geq \text{length bla} \implies$
 $\text{rbl-mult bla (blb @ blc)} = \text{rbl-mult bla blb}$
 $\langle \text{proof} \rangle$

lemma *rbl-mult-take2*:

$\text{length blb} \geq \text{length bla} \implies$
 $\text{rbl-mult bla (take (length bla) blb)} = \text{rbl-mult bla blb}$
 $\langle \text{proof} \rangle$

lemma *rbl-mult-gt1*:

$m \geq \text{length bl} \implies \text{rbl-mult bl (rev (bin-to-bl m binb))} =$
 $\text{rbl-mult bl (rev (bin-to-bl (length bl) binb))}$
 $\langle \text{proof} \rangle$

lemma *rbl-mult-gt*:

$m > n \implies \text{rbl-mult } (\text{rev } (\text{bin-to-bl } n \text{ bina})) (\text{rev } (\text{bin-to-bl } m \text{ binb})) =$
 $\text{rbl-mult } (\text{rev } (\text{bin-to-bl } n \text{ bina})) (\text{rev } (\text{bin-to-bl } n \text{ binb}))$
 $\langle \text{proof} \rangle$

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

lemma *rbbl-Cons*:

$b \# \text{rev } (\text{bin-to-bl } n \text{ x}) = \text{rev } (\text{bin-to-bl } (\text{Suc } n) \text{ (x BIT If b bit.B1 bit.B0)})$
 $\langle \text{proof} \rangle$

lemma *rbl-mult*: $!!\text{bina binb}.$

$\text{rbl-mult } (\text{rev } (\text{bin-to-bl } n \text{ bina})) (\text{rev } (\text{bin-to-bl } n \text{ binb})) =$
 $\text{rev } (\text{bin-to-bl } n \text{ (bina * binb)})$
 $\langle \text{proof} \rangle$

lemma *rbl-add-split*:

$P (\text{rbl-add } (y \# ys) (x \# xs)) =$
 $(\text{ALL } ws. \text{length } ws = \text{length } ys \implies ws = \text{rbl-add } ys xs \implies$
 $(y \implies ((x \implies P (\text{False} \# \text{rbl-succ } ws)) \ \& \ (\sim x \implies P (\text{True} \# ws)))) \ \&$
 $(\sim y \implies P (x \# ws)))$
 $\langle \text{proof} \rangle$

lemma *rbl-mult-split*:

$P (\text{rbl-mult } (y \# ys) xs) =$
 $(\text{ALL } ws. \text{length } ws = \text{Suc } (\text{length } ys) \implies ws = \text{False} \# \text{rbl-mult } ys xs \implies$

$(y \dashrightarrow P \text{ (rbl-add ws xs)}) \ \& \ (\sim y \dashrightarrow P \text{ ws}))$
 $\langle \text{proof} \rangle$

lemma *and-len*: $xs = ys \implies xs = ys \ \& \ \text{length } xs = \text{length } ys$
 $\langle \text{proof} \rangle$

lemma *size-if*: $\text{size } (\text{if } p \text{ then } xs \text{ else } ys) = (\text{if } p \text{ then } \text{size } xs \text{ else } \text{size } ys)$
 $\langle \text{proof} \rangle$

lemma *tl-if*: $\text{tl } (\text{if } p \text{ then } xs \text{ else } ys) = (\text{if } p \text{ then } \text{tl } xs \text{ else } \text{tl } ys)$
 $\langle \text{proof} \rangle$

lemma *hd-if*: $\text{hd } (\text{if } p \text{ then } xs \text{ else } ys) = (\text{if } p \text{ then } \text{hd } xs \text{ else } \text{hd } ys)$
 $\langle \text{proof} \rangle$

lemma *if-Not-x*: $(\text{if } p \text{ then } \sim x \text{ else } x) = (p = (\sim x))$
 $\langle \text{proof} \rangle$

lemma *if-x-Not*: $(\text{if } p \text{ then } x \text{ else } \sim x) = (p = x)$
 $\langle \text{proof} \rangle$

lemma *if-same-and*: $(\text{If } p \ x \ y \ \& \ \text{If } p \ u \ v) = (\text{if } p \text{ then } x \ \& \ u \text{ else } y \ \& \ v)$
 $\langle \text{proof} \rangle$

lemma *if-same-eq*: $(\text{If } p \ x \ y = (\text{If } p \ u \ v)) = (\text{if } p \text{ then } x = (u) \text{ else } y = (v))$
 $\langle \text{proof} \rangle$

lemma *if-same-eq-not*:
 $(\text{If } p \ x \ y = (\sim \text{If } p \ u \ v)) = (\text{if } p \text{ then } x = (\sim u) \text{ else } y = (\sim v))$
 $\langle \text{proof} \rangle$

lemma *if-Cons*: $(\text{if } p \text{ then } x \ \# \ xs \text{ else } y \ \# \ ys) = \text{If } p \ x \ y \ \# \ \text{If } p \ xs \ ys$
 $\langle \text{proof} \rangle$

lemma *if-single*:
 $(\text{if } xc \text{ then } [xab] \text{ else } [an]) = [\text{if } xc \text{ then } xab \text{ else } an]$
 $\langle \text{proof} \rangle$

lemma *if-bool-simps*:
 $\text{If } p \ \text{True} \ y = (p \mid y) \ \& \ \text{If } p \ \text{False} \ y = (\sim p \ \& \ y) \ \& \$
 $\text{If } p \ y \ \text{True} = (p \dashrightarrow y) \ \& \ \text{If } p \ y \ \text{False} = (p \ \& \ y)$
 $\langle \text{proof} \rangle$

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

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

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

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

7.2 Repeated splitting or concatenation

lemma *sclem*:

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

lemma *bin-cat-foldl-lem* [*rule-format*] :

ALL *x*. *foldl* (%*u*. *bin-cat* *u* *n*) *x* *xs* =
bin-cat *x* (*size* *xs* * *n*) (*foldl* (%*u*. *bin-cat* *u* *n*) *y* *xs*)
 ⟨*proof*⟩

lemma *bin-rcat-bl*:

(*bin-rcat* *n* *wl*) = *bl-to-bin* (*concat* (*map* (*bin-to-bl* *n*) *wl*))
 ⟨*proof*⟩

lemmas *bin-rsplit-aux-simps* = *bin-rsplit-aux.simps* *bin-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*
 ⟨*proof*⟩

lemma *bin-rsplittl-aux-append*:

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

lemmas *rsplit-aux-apps* [**where** *bs* = []] =

bin-rsplit-aux-append *bin-rsplittl-aux-append*

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

lemmas *rsplit-aux-alt*s = *rsplit-aux-apps*

[*unfolded append-Nil rsplit-def-auxs [symmetric]*]

lemma *bin-split-minus*: $0 < n \implies \text{bin-split } (\text{Suc } (n - 1)) \ w = \text{bin-split } n \ w$
 ⟨*proof*⟩

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

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

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

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

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*] :
 $n > 0 \implies (\text{ALL } nw \ w. \text{rev } sw = \text{bin-rsplit } n \ (nw, w) \dashrightarrow$
 $(\text{ALL } v: \text{set } sw. \text{bintrunc } n \ v = v))$
 ⟨*proof*⟩

lemmas *bin-rsplit-size-sign* = *bin-rsplit-size-sign' [OF asm-rl*
rev-rev-ident [THEN trans] set-rev [THEN equalityD2 [THEN subsetD]],
standard]

lemma *bin-nth-rsplit* [*rule-format*] :
 $n > 0 \implies m < n \implies (\text{ALL } w \ k \ nw. \text{rev } sw = \text{bin-rsplit } n \ (nw, w) \dashrightarrow$
 $k < \text{size } sw \dashrightarrow \text{bin-nth } (sw ! k) \ m = \text{bin-nth } w \ (k * n + m))$
 ⟨*proof*⟩

lemma *bin-rsplit-all*:
 $0 < nw \implies nw \leq n \implies \text{bin-rsplit } n \ (nw, w) = [\text{bintrunc } n \ w]$
 ⟨*proof*⟩

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

ALL bin. $\text{bin-rsplitl } n \ (m, \text{bin}) = \text{bin-rsplit } n \ (m, \text{bintrunc } m \ \text{bin})$
 ⟨proof⟩

lemma *bin-rsplit-rcat* [rule-format] :
 $n > 0 \dashrightarrow \text{bin-rsplit } n \ (n * \text{size } ws, \text{bin-rcat } n \ ws) = \text{map } (\text{bintrunc } n) \ ws$
 ⟨proof⟩

lemma *bin-rsplit-aux-len-le* [rule-format] :
 $\forall ws \ m. \ n \neq 0 \longrightarrow ws = \text{bin-rsplit-aux } n \ nw \ w \ bs \longrightarrow$
 $\text{length } ws \leq m \iff nw + \text{length } bs * n \leq m * n$
 ⟨proof⟩

lemma *bin-rsplit-len-le*:
 $n \neq 0 \dashrightarrow ws = \text{bin-rsplit } n \ (nw, w) \dashrightarrow (\text{length } ws \leq m) = (nw \leq m * n)$
 ⟨proof⟩

lemma *bin-rsplit-aux-len* [rule-format] :
 $n \neq 0 \dashrightarrow \text{length } (\text{bin-rsplit-aux } n \ nw \ w \ cs) =$
 $(nw + n - 1) \text{ div } n + \text{length } cs$
 ⟨proof⟩

lemma *bin-rsplit-len*:
 $n \neq 0 \implies \text{length } (\text{bin-rsplit } n \ (nw, w)) = (nw + n - 1) \text{ div } n$
 ⟨proof⟩

lemma *bin-rsplit-aux-len-indep*:
 $n \neq 0 \implies \text{length } bs = \text{length } cs \implies$
 $\text{length } (\text{bin-rsplit-aux } n \ nw \ v \ bs) =$
 $\text{length } (\text{bin-rsplit-aux } n \ nw \ w \ cs)$
 ⟨proof⟩

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

end

8 TdThs: Type Definition Theorems

theory *TdThs*
imports *Main*
begin

9 More lemmas about normal type definitions

lemma

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

lemma *td-nat-int:*

type-definition int nat (Collect (op <= 0))
 $\langle \text{proof} \rangle$

context *type-definition*

begin

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

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

lemma *Abs-eqD: Abs x = Abs y $\implies x \in A \implies y \in A \implies x = y$*
 $\langle \text{proof} \rangle$

lemma *Abs-inverse':*

r : A $\implies \text{Abs } r = a \implies \text{Rep } a = r$
 $\langle \text{proof} \rangle$

lemma *Rep-comp-inverse:*

Rep o f = g $\implies \text{Abs } o g = f$
 $\langle \text{proof} \rangle$

lemma *Rep-eqD [elim!]: Rep x = Rep y $\implies x = y$*
 $\langle \text{proof} \rangle$

lemma *Rep-inverse': Rep a = r $\implies \text{Abs } r = a$*
 $\langle \text{proof} \rangle$

lemma *comp-Abs-inverse:*

f o Abs = g $\implies g o \text{Rep} = f$
 $\langle \text{proof} \rangle$

lemma *set-Rep:*

A = range Rep
 $\langle \text{proof} \rangle$

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

lemma *Abs-inj-on: inj-on Abs A*
 $\langle \text{proof} \rangle$

lemma *image: Abs ' A = UNIV*
 $\langle \text{proof} \rangle$

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

lemma *fns1*:

$Rep \circ fa = fr \circ Rep \mid fa \circ Abs = Abs \circ fr \implies Abs \circ fr \circ Rep = fa$
 $\langle proof \rangle$

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

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

lemma *fns4*:

$Rep \circ fa \circ Abs = fr \implies$
 $Rep \circ fa = fr \circ Rep \ \& \ fa \circ Abs = Abs \circ fr$
 $\langle proof \rangle$

end

interpretation *nat-int*: *type-definition int nat Collect* (*op* ≤ 0)
 $\langle proof \rangle$

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*]

9.1 Extended form of type definition predicate

lemma *td-conds*:

$norm \circ norm = norm \implies (fr \circ norm = norm \circ fr) =$
 $(norm \circ fr \circ norm = fr \circ norm \ \& \ norm \circ fr \circ norm = norm \circ fr)$
 $\langle proof \rangle$

lemma *fn-comm-power*:

$fa \circ tr = tr \circ fr \implies fa \wedge n \circ tr = tr \circ fr \wedge n$
 $\langle proof \rangle$

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*]:

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

lemma *td-th*:

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

<proof>

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

<proof>

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

<proof>

lemmas *td = td-thm*

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

<proof>

lemma *inverse-norm*:

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

<proof>

lemma *norm-eq-iff*:

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

<proof>

lemma *norm-comps*:

$$Abs \circ norm = Abs$$

$$norm \circ Rep = Rep$$

$$norm \circ norm = norm$$

<proof>

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

lemma *fns5*:

$$Rep \circ fa \circ Abs = fr \implies$$

$$fr \circ norm = fr \ \& \ norm \circ fr = fr$$

<proof>

lemma *fns2*:

$$Abs \circ fr \circ Rep = fa \implies$$

$$(norm \circ fr \circ norm = fr \circ norm) = (Rep \circ fa = fr \circ Rep)$$

<proof>

lemma *fns3*:

$$Abs \circ fr \circ Rep = fa \implies$$

$$(norm \circ fr \circ norm = norm \circ fr) = (fa \circ Abs = Abs \circ fr)$$

<proof>

lemma *fns*:

```

fr o norm = norm o fr ==>
  (fa o Abs = Abs o fr) = (Rep o fa = fr o Rep)
⟨proof⟩

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

end

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

end

```

10 WordDefinition: Definition of Word Type

```

theory WordDefinition
imports Size BinBoolList TdThs
begin

```

10.1 Type definition

```

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

```

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

lemma *uint-word-of-int* [*code*]: *uint* (*word-of-int* *w* :: *'a::len0 word*) = *w mod 2 ^ len-of TYPE('a)*
 ⟨proof⟩

```
code-datatype word-of-int
```

10.2 Type conversions and casting

definition *sint* :: *'a :: len word* \Rightarrow *int* **where**
 — treats the most-significant-bit as a sign bit
 $sint-uint: sint\ w = sbintrunc\ (len-of\ TYPE\ ('a) - 1)\ (uint\ w)$

definition *unat* :: *'a :: len0 word* \Rightarrow *nat* **where**
 $unat\ w = nat\ (uint\ w)$

definition *uints* :: *nat* \Rightarrow *int set* **where**

— the sets of integers representing the words
 $uints\ n = range\ (bintrunc\ n)$

definition $sints :: nat \Rightarrow int\ set$ **where**
 $sints\ n = range\ (sbintrunc\ (n - 1))$

definition $unats :: nat \Rightarrow nat\ set$ **where**
 $unats\ n = \{i. i < 2^{\wedge} n\}$

definition $norm-sint :: nat \Rightarrow int \Rightarrow int$ **where**
 $norm-sint\ n\ w = (w + 2^{\wedge} (n - 1)) \bmod 2^{\wedge} n - 2^{\wedge} (n - 1)$

definition $scast :: 'a :: len\ word \Rightarrow 'b :: len\ word$ **where**
 — cast a word to a different length
 $scast\ w = word-of-int\ (sint\ w)$

definition $ucast :: 'a :: len0\ word \Rightarrow 'b :: len0\ word$ **where**
 $ucast\ w = word-of-int\ (uint\ w)$

instantiation $word :: (len0)\ size$
begin

definition
 $word-size: size\ (w :: 'a\ word) = len-of\ TYPE('a)$

instance $\langle proof \rangle$

end

definition $source-size :: ('a :: len0\ word \Rightarrow 'b) \Rightarrow nat$ **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 $target-size :: ('a \Rightarrow 'b :: len0\ word) \Rightarrow nat$ **where**
 $target-size\ c = size\ (c\ undefined)$

definition $is-up :: ('a :: len0\ word \Rightarrow 'b :: len0\ word) \Rightarrow bool$ **where**
 $is-up\ c \longleftrightarrow source-size\ c \leq target-size\ c$

definition $is-down :: ('a :: len0\ word \Rightarrow 'b :: len0\ word) \Rightarrow bool$ **where**
 $is-down\ c \longleftrightarrow target-size\ c \leq source-size\ c$

definition $of-bl :: bool\ list \Rightarrow 'a :: len0\ word$ **where**
 $of-bl\ bl = word-of-int\ (bl-to-bin\ bl)$

definition $to-bl :: 'a :: len0\ word \Rightarrow bool\ list$ **where**
 $to-bl\ w = bin-to-bl\ (len-of\ TYPE\ ('a))\ (uint\ w)$

definition $word-reverse :: 'a :: len0\ word \Rightarrow 'a\ word$ **where**

word-reverse $w = \text{of-bl } (\text{rev } (\text{to-bl } w))$

definition *word-int-case* $:: (int \Rightarrow 'b) \Rightarrow ('a :: \text{len0 } word) \Rightarrow 'b$ **where**
word-int-case $f \ w = f \ (\text{uint } w)$

syntax

of-int $:: int \Rightarrow 'a$

translations

case $x \text{ of } \text{of-int } y \Rightarrow b == \text{CONST } \text{word-int-case } (\%y. b) \ x$

10.3 Arithmetic operations

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

begin

definition

word-0-wi: $0 = \text{word-of-int } 0$

definition

word-1-wi: $1 = \text{word-of-int } 1$

definition

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

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

primrec *power-word* **where**

$(a :: 'a \text{ word}) \ ^ 0 = 1$

$| (a :: 'a \text{ word}) \ ^ \text{Suc } n = a * a \ ^ n$

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: $x < y \iff x \leq y \wedge x \neq (y :: 'a \text{ word})$

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 $\langle \text{proof} \rangle$

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

where

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

constdefs

udvd :: $'a :: \text{len word} \Rightarrow 'a :: \text{len word} \Rightarrow \text{bool}$ (**infixl** *udvd* 50)

$a \text{ udvd } b == \text{EX } n \geq 0. \text{uint } b = n * \text{uint } a$

word-sle :: $'a :: \text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool}$ $((-/ \leq s -) [50, 51] 50)$

$a \leq s b == \text{sint } a \leq \text{sint } b$

word-sless :: $'a :: \text{len word} \Rightarrow 'a \text{ word} \Rightarrow \text{bool}$ $((-/ < s -) [50, 51] 50)$

$(x < s y) == (x \leq s y \ \& \ x \neq y)$

10.4 Bit-wise operations

instantiation *word* :: $(\text{len0}) \text{ bits}$

begin

definition

word-test-bit-def: $\text{test-bit } a = \text{bin-nth } (\text{uint } a)$

definition

word-set-bit-def: $\text{set-bit } a \ n \ x =$
word-of-int ($\text{bin-sc } n \ (\text{If } x \ \text{bit.B1} \ \text{bit.B0}) \ (\text{uint } a)$)

definition

word-set-bits-def: $(\text{BITS } n. \ f \ n) = \text{of-bl } (\text{bl-of-nth } (\text{len-of TYPE } ('a)) \ f)$

definition

word-lsb-def: $\text{lsb } a \longleftrightarrow \text{bin-last } (\text{uint } a) = \text{bit.B1}$

definition *shiftrl1* :: 'a word \Rightarrow 'a word **where**

shiftrl1 $w = \text{word-of-int } (\text{uint } w \ \text{BIT} \ \text{bit.B0})$

definition *shiftr1* :: 'a word \Rightarrow 'a word **where**

— shift right as unsigned or as signed, ie logical or arithmetic
shiftr1 $w = \text{word-of-int } (\text{bin-rest } (\text{uint } w))$

definition

shiftrl-def: $w << n = (\text{shiftrl1 } ^n) \ w$

definition

shiftr-def: $w >> n = (\text{shiftr1 } ^n) \ w$

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

end

instantiation *word* :: (len) bitss

begin

definition

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

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

end

constdefs

setBit :: 'a :: len0 word \Rightarrow nat \Rightarrow 'a word
setBit $w \ n == \text{set-bit } w \ n \ \text{True}$

clearBit :: 'a :: len0 word \Rightarrow nat \Rightarrow 'a word
clearBit $w \ n == \text{set-bit } w \ n \ \text{False}$

10.5 Shift operations

constdefs

```

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

bsshiftr1 :: bool => 'a :: len word => 'a word
bsshiftr1 b w == of-bl (b # butlast (to-bl w))

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

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

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

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

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

```

10.6 Rotation

constdefs

```

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

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

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

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

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

```

10.7 Split and cat operations

constdefs

```

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

```

```

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

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

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

```

constdefs

— Largest representable machine integer.

```

max-word :: 'a::len word
max-word ≡ word-of-int (2len-of TYPE('a) - 1)

```

consts

```

of-bool :: bool ⇒ 'a::len word

```

primrec

```

of-bool False = 0
of-bool True = 1

```

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

lemmas *word-size-gt-0* [iff] =

xtr1 [OF *word-size 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*: $\text{uints } n = \{i. 0 \leq i \wedge i < 2^n\}$
 ⟨*proof*⟩

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

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

lemma *mod-in-reps*: $m > 0 \implies y \bmod m : \{0::\text{int} ..< m\}$
 ⟨*proof*⟩

lemma

uint-0:0 ≤ *uint x* **and**
uint-lt: *uint* (*x::'a::len0 word*) < 2^{len-of TYPE('a)}
 ⟨*proof*⟩

lemma *uint-mod-same*:

uint $x \bmod 2^{\text{len-of TYPE('a)}}$ = *uint* ($x :: 'a :: \text{len0 word}$)
 ⟨proof⟩

lemma *td-ext-uint*:

td-ext (*uint* :: $'a \text{ word} \Rightarrow \text{int}$) *word-of-int* (*uints* ($\text{len-of TYPE('a :: len0)}$))
 ($\%w :: \text{int}. w \bmod 2^{\text{len-of TYPE('a)}}$)
 ⟨proof⟩

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

interpretation *word-uint*:

td-ext uint :: 'a :: len0 word \Rightarrow int
word-of-int
uints ($\text{len-of TYPE('a :: len0)}$)
 $\lambda w. w \bmod 2^{\text{len-of TYPE('a :: len0)}}$
 ⟨proof⟩

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

lemmas *td-ext-ubin* = *td-ext-uint*

[*simplified len-gt-0 no-bintr-alt1 [symmetric]*]

interpretation *word-ubin*:

td-ext uint :: 'a :: len0 word \Rightarrow int
word-of-int
uints ($\text{len-of TYPE('a :: len0)}$)
bintrunc ($\text{len-of TYPE('a :: len0)}$)
 ⟨proof⟩

lemma *sint-sbintrunc'*:

sint (*word-of-int bin* :: $'a \text{ word}$) =
 (*sbintrunc* ($\text{len-of TYPE('a :: len)} - 1$) *bin*)
 ⟨proof⟩

lemma *uint-sint*:

uint $w = \text{bintrunc} (\text{len-of TYPE('a)}) (\text{sint } (w :: 'a :: \text{len word}))$
 ⟨proof⟩

lemma *bintr-uint'*:

$n \geq \text{size } w \Rightarrow \text{bintrunc } n (\text{uint } w) = \text{uint } w$
 ⟨proof⟩

lemma *wi-bintr'*:

$wb = \text{word-of-int bin} \Rightarrow n \geq \text{size } wb \Rightarrow$
 $\text{word-of-int } (\text{bintrunc } n \text{ bin}) = wb$
 ⟨proof⟩

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

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

lemma *td-ext-sbin*:

td-ext (*sint* :: 'a word => int) *word-of-int* (*sints* (*len-of* TYPE('a::len)))
 (*sbintrunc* (*len-of* TYPE('a) - 1))
 <proof>

lemmas *td-ext-sint* = *td-ext-sbin*

[*simplified len-gt-0 no-sbintr-alt2 Suc-pred'* [*symmetric*]]

interpretation *word-sint*:

td-ext sint :: 'a::len word => int
word-of-int
sints (*len-of* TYPE('a::len))
 $\%w. (w + 2^{(\text{len-of TYPE('a::len)} - 1)}) \bmod 2^{\text{len-of TYPE('a::len)} - 1}$
 <proof>

interpretation *word-sbin*:

td-ext sint :: 'a::len word => int
word-of-int
sints (*len-of* TYPE('a::len))
sbintrunc (*len-of* TYPE('a::len) - 1)
 <proof>

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

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

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

<proof>

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

<proof>

lemma *to-bl-def'*:

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

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

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

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

number-of (*bintrunc* (*len-of* TYPE('a :: len0)) *bin*)
 <proof>

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

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

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

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

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

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

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

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

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

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

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

lemma *unat-number-of*:
bin-sign *b* = *Int.Pl* ==>

$\text{unat } (\text{number-of } b :: 'a :: \text{len0 word}) = \text{number-of } b \bmod 2^{\text{len-of TYPE } ('a)}$
 ⟨proof⟩

lemma *sint-number-of*: $\text{sint } (\text{number-of } b :: 'a :: \text{len word}) = (\text{number-of } b + 2^{\text{len-of TYPE } ('a) - 1}) \bmod 2^{\text{len-of TYPE } ('a)}$
 ⟨proof⟩

lemma *word-of-int-bin* [simp] :
 $(\text{word-of-int } (\text{number-of bin}) :: 'a :: \text{len0 word}) = (\text{number-of bin})$
 ⟨proof⟩

lemma *word-int-case-wi*:
 $\text{word-int-case } f (\text{word-of-int } i :: 'b \text{ word}) = f (i \bmod 2^{\text{len-of TYPE } ('b :: \text{len0})})$
 ⟨proof⟩

lemma *word-int-split*:
 $P (\text{word-int-case } f x) = (\text{ALL } i. x = (\text{word-of-int } i :: 'b :: \text{len0 word}) \ \& \ 0 \leq i \ \& \ i < 2^{\text{len-of TYPE } ('b)}) \longrightarrow P (f i)$
 ⟨proof⟩

lemma *word-int-split-asm*:
 $P (\text{word-int-case } f x) = (\sim (EX \ n. x = (\text{word-of-int } n :: 'b :: \text{len0 word}) \ \& \ 0 \leq n \ \& \ n < 2^{\text{len-of TYPE } ('b :: \text{len0})} \ \& \ \sim P (f n)))$
 ⟨proof⟩

lemmas *uint-range'* =
 $\text{word-uint.Rep [unfolded uints-num mem-Collect-eq, standard]}$
lemmas *sint-range'* = $\text{word-sint.Rep [unfolded One-nat-def sints-num mem-Collect-eq, standard]}$

lemma *uint-range-size*: $0 \leq \text{uint } w \ \& \ \text{uint } w < 2^{\text{size } w}$
 ⟨proof⟩

lemma *sint-range-size*:
 $-(2^{\text{size } w - \text{Suc } 0}) \leq \text{sint } w \ \& \ \text{sint } w < 2^{\text{size } w - \text{Suc } 0}$
 ⟨proof⟩

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

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

lemma *test-bit-eq-iff*: $(\text{test-bit } (u :: 'a :: \text{len0 word}) = \text{test-bit } v) = (u = v)$
 ⟨proof⟩

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

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

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

lemma *test-bit-bin'*: w !! n = (n < size w & bin-nth (uint w) n)
 <proof>

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

lemma *bin-nth-uint-imp'*: bin-nth (uint w) n --> n < size w
 <proof>

lemma *bin-nth-sint'*:
 n >= size w --> bin-nth (sint w) n = bin-nth (sint w) (size w - 1)
 <proof>

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

lemmas *bin-nth-sint* = *bin-nth-sint'* [rule-format, unfolded word-size]

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

interpretation *word-bl*:
 type-definition to-bl :: 'a::len0 word => bool list
 of-bl
 {bl. length bl = len-of TYPE('a::len0)}
 <proof>

lemma *word-size-bl*: size w == size (to-bl w)
 <proof>

lemma *to-bl-use-of-bl*:
 (to-bl w = bl) = (w = of-bl bl & length bl = length (to-bl w))
 <proof>

lemma *to-bl-word-rev*: to-bl (word-reverse w) = rev (to-bl w)
 <proof>

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

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

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

lemmas *length-bl-gt-0* [*iff*] = *xtr1* [*OF word-bl.Rep' len-gt-0, standard*]

lemmas *bl-not-Nil* [*iff*] =
length-bl-gt-0 [*THEN length-greater-0-conv* [*THEN iffD1*], *standard*]

lemmas *length-bl-neq-0* [*iff*] = *length-bl-gt-0* [*THEN gr-implies-not0*]

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

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

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

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

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

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

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

lemma *to-bl-of-bin*:
to-bl (*word-of-int* *bin*::'*a*::*len0* *word*) = *bin-to-bl* (*len-of TYPE*('*a*)) *bin*
 ⟨*proof*⟩

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

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

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

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

lemma *uints-unats*: *uints n = int ‘ unats n*
<proof>

lemma *unats-uints*: *unats n = nat ‘ uints n*
<proof>

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

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

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

lemma *num-of-sbintr'*:
sbintrunc (len-of TYPE('a :: len) - 1) a = b ==>
number-of a = (number-of b :: 'a word)
<proof>

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

lemma *ucast-id*: *ucast w = w*
<proof>

lemma *scast-id*: *scast w = w*
<proof>

lemma *ucast-bl*: *ucast w == of-bl (to-bl w)*
<proof>

lemma *nth-ucast*:

$(ucast\ w :: 'a :: len0\ word) !!\ n = (w !!\ n \ \&\ n < len-of\ TYPE('a))$
 $\langle proof \rangle$

lemma *ucast-bintr* [*simp*]:

$ucast\ (number-of\ w :: 'a :: len0\ word) =$
 $number-of\ (bintrunc\ (len-of\ TYPE('a))\ w)$
 $\langle proof \rangle$

lemma *scast-sbintr* [*simp*]:

$scast\ (number-of\ w :: 'a :: len\ word) =$
 $number-of\ (sbintrunc\ (len-of\ TYPE('a) - Suc\ 0)\ w)$
 $\langle proof \rangle$

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'*: $uc = ucast ==> is-down\ uc ==> uc = scast$
 $\langle proof \rangle$

lemma *word-rev-tf'*:

$r = to-bl\ (of-bl\ bl) ==> r = rev\ (takefill\ False\ (length\ r)\ (rev\ bl))$
 $\langle proof \rangle$

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*:

$to-bl\ (ucast\ (w :: 'b :: len0\ word) :: 'a :: len0\ word) =$
 $replicate\ (len-of\ TYPE('a) - len-of\ TYPE('b))\ False\ @$
 $drop\ (len-of\ TYPE('b) - len-of\ TYPE('a))\ (to-bl\ w)$
 $\langle proof \rangle$

lemma *ucast-up-app'*:

$uc = ucast ==> source-size\ uc + n = target-size\ uc ==>$
 $to-bl\ (uc\ w) = replicate\ n\ False\ @\ (to-bl\ w)$
 $\langle proof \rangle$

lemma *ucast-down-drop'*:

$uc = ucast ==> source-size\ uc = target-size\ uc + n ==>$

$to\text{-}bl\ (uc\ w) = drop\ n\ (to\text{-}bl\ w)$
 $\langle proof \rangle$

lemma *scast-down-drop'*:

$sc = scast ==> source\text{-}size\ sc = target\text{-}size\ sc + n ==>$
 $to\text{-}bl\ (sc\ w) = drop\ n\ (to\text{-}bl\ w)$
 $\langle proof \rangle$

lemma *sint-up-scast'*:

$sc = scast ==> is\text{-}up\ sc ==> sint\ (sc\ w) = sint\ w$
 $\langle proof \rangle$

lemma *uint-up-ucast'*:

$uc = ucast ==> is\text{-}up\ uc ==> uint\ (uc\ w) = uint\ w$
 $\langle proof \rangle$

lemmas *down-cast-same* = refl [THEN down-cast-same']

lemmas *ucast-up-app* = refl [THEN ucast-up-app']

lemmas *ucast-down-drop* = refl [THEN ucast-down-drop']

lemmas *scast-down-drop* = refl [THEN scast-down-drop']

lemmas *uint-up-ucast* = refl [THEN uint-up-ucast']

lemmas *sint-up-scast* = refl [THEN sint-up-scast']

lemma *ucast-up-ucast'*: $uc = ucast ==> is\text{-}up\ uc ==> ucast\ (uc\ w) = ucast\ w$
 $\langle proof \rangle$

lemma *scast-up-scast'*: $sc = scast ==> is\text{-}up\ sc ==> scast\ (sc\ w) = scast\ w$
 $\langle proof \rangle$

lemma *ucast-of-bl-up'*:

$w = of\text{-}bl\ bl ==> size\ bl <= size\ w ==> ucast\ w = of\text{-}bl\ bl$
 $\langle proof \rangle$

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*:

$is\text{-}up\ (ucast :: 'b::len0\ word ==> 'a::len0\ word) ==>$
 $surj\ (ucast :: 'a\ word ==> 'b\ word)$
 $\langle proof \rangle$

lemma *up-scast-surj*:

is-up (*scast* :: 'b::len word => 'a::len word) ==>
surj (*scast* :: 'a word => 'b word)
 <proof>

lemma *down-scast-inj*:

is-down (*scast* :: 'b::len word => 'a::len word) ==>
inj-on (*ucast* :: 'a word => 'b word) A
 <proof>

lemma *down-ucast-inj*:

is-down (*ucast* :: 'b::len0 word => 'a::len0 word) ==>
inj-on (*ucast* :: 'a word => 'b word) A
 <proof>

lemma *of-bl-append-same*: *of-bl* (X @ *to-bl* w) = w

<proof>

lemma *ucast-down-no'*:

uc = *ucast* ==> *is-down* *uc* ==> *uc* (*number-of bin*) = *number-of bin*
 <proof>

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

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

<proof>

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

lemmas *slice-def'* = *slice-def* [unfolded word-size]

lemmas *test-bit-def'* = *word-test-bit-def* [THEN 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}) eq

begin

definition *eq-word* :: 'a word ⇒ 'a word ⇒ bool **where**

eq-word k l ⇔ HOL.eq (uint k) (uint l)

instance <proof>

end

end

11 WordArith: Word Arithmetic

```
theory WordArith
imports WordDefinition
begin
```

```
lemma word-less-alt:  $(a < b) = (uint\ a < uint\ b)$ 
   $\langle proof \rangle$ 
```

```
lemma signed-linorder: linorder word-sle word-sless
   $\langle proof \rangle$ 
```

```
interpretation signed: linorder word-sle word-sless
   $\langle proof \rangle$ 
```

```
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:
   $0 \leq n \implies uint\ b = n * uint\ a \implies a\ udvd\ b$ 
   $\langle proof \rangle$ 
```

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

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

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

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

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

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

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

```
lemma int-one-bin:  $(1 :: int) == (Int.Pls\ BIT\ bit.B1)$ 
   $\langle proof \rangle$ 
```

```
lemma word-1-no:
```

$(1 :: 'a :: \text{len0 word}) == \text{number-of } (\text{Int.Pls BIT bit.B1})$
 $\langle \text{proof} \rangle$

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

lemma *word-m1-wi-Min*: $-1 = \text{word-of-int Int.Min}$
 $\langle \text{proof} \rangle$

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

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

lemma *uint-0 [simp]*: $(\text{uint } 0 = 0)$
 $\langle \text{proof} \rangle$

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

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

lemma *uint-0-iff*: $(\text{uint } x = 0) = (x = 0)$
 $\langle \text{proof} \rangle$

lemma *unat-0-iff*: $(\text{unat } x = 0) = (x = 0)$
 $\langle \text{proof} \rangle$

lemma *unat-0 [simp]*: $\text{unat } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *size-0-same'*: $\text{size } w = 0 ==> w = (v :: 'a :: \text{len0 word})$
 $\langle \text{proof} \rangle$

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*: $(0 < \text{unat } x) = (x \sim= 0)$
 $\langle \text{proof} \rangle$

lemma *ucast-0 [simp]*: $\text{ucast } 0 = 0$
 $\langle \text{proof} \rangle$

lemma *sint-0 [simp]*: $\text{sint } 0 = 0$

$\langle proof \rangle$

lemma *scast-0* [*simp*] : *scast* 0 = 0
 $\langle proof \rangle$

lemma *sint-n1* [*simp*] : *sint* -1 = -1
 $\langle proof \rangle$

lemma *scast-n1* [*simp*] : *scast* -1 = -1
 $\langle proof \rangle$

lemma *uint-1* [*simp*] : *uint* (1 :: 'a :: len word) = 1
 $\langle proof \rangle$

lemma *unat-1* [*simp*] : *unat* (1 :: 'a :: len word) = 1
 $\langle proof \rangle$

lemma *ucast-1* [*simp*] : *ucast* (1 :: 'a :: len word) = 1
 $\langle proof \rangle$

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: *word-of-int* a + *word-of-int* b = *word-of-int* (a + b) **and**
wi-hom-mult: *word-of-int* a * *word-of-int* b = *word-of-int* (a * b) **and**
wi-hom-neg: - *word-of-int* a = *word-of-int* (- a) **and**
wi-hom-succ: *word-succ* (*word-of-int* a) = *word-of-int* (*Int.succ* a) **and**
wi-hom-pred: *word-pred* (*word-of-int* a) = *word-of-int* (*Int.pred* a)
 $\langle proof \rangle$

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

lemma *word-sub-def*: a - b == a + - (b :: 'a :: len0 word)
 $\langle proof \rangle$

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

lemma *word-of-int-sub-hom*:

(*word-of-int* a) - *word-of-int* b = *word-of-int* (a - b)
 $\langle proof \rangle$

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-alts* (6)
lemmas *word-0-alt* = *word-arith-alts* (7)
lemmas *word-1-alt* = *word-arith-alts* (8)

11.1 Transferring goals from words to ints

lemma *word-ths*:
shows
word-succ-p1: *word-succ a* = *a + 1* **and**
word-pred-m1: *word-pred a* = *a - 1* **and**
word-pred-succ: *word-pred (word-succ a)* = *a* **and**
word-succ-pred: *word-succ (word-pred a)* = *a* **and**
word-mult-succ: *word-succ a * b* = *b + a * b*
 ⟨*proof*⟩

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

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

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

lemmas *sint-word-ariths* = *uint-word-arith-bintr*
 [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 = word-of-int -1*
 ⟨*proof*⟩

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

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

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)
 ⟨*proof*⟩

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

lemma *word-of-int-Ex*:
 $\exists y. x = \text{word-of-int } y$
 ⟨*proof*⟩

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

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

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

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

11.2 Order on fixed-length words

lemma *word-order-trans*: $x \leq y \implies y \leq z \implies x \leq (z :: 'a :: \text{len0 word})$
 ⟨*proof*⟩

lemma *word-order-refl*: $z \leq (z :: 'a :: \text{len0 word})$
 ⟨*proof*⟩

lemma *word-order-antisym*: $x \leq y \implies y \leq x \implies x = (y :: 'a :: \text{len0 word})$
 ⟨*proof*⟩

lemma *word-order-linear*:
 $y \leq x \mid x \leq (y :: 'a :: \text{len0 word})$
 ⟨*proof*⟩

lemma *word-zero-le* [*simp*] :
 $0 \leq (y :: 'a :: \text{len0 word})$
 ⟨*proof*⟩

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

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

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

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

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

lemmas *word-not-simps* [*simp*] =
word-zero-le [*THEN leD*] *word-m1-ge* [*THEN leD*] *word-n1-ge* [*THEN leD*]

lemma *word-gt-0*: $0 < y = (0 \sim (y :: 'a :: \text{len0 word}))$

$\langle proof \rangle$

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

lemma *word-sless-alt*: $(a <_s b) == (sint\ a < sint\ b)$
 $\langle proof \rangle$

lemma *word-le-nat-alt*: $(a \leq b) = (unat\ a \leq unat\ b)$
 $\langle proof \rangle$

lemma *word-less-nat-alt*: $(a < b) = (unat\ a < unat\ b)$
 $\langle proof \rangle$

lemma *wi-less*:
 $(word-of-int\ n < (word-of-int\ m :: 'a :: len0\ word)) =$
 $(n \bmod 2 \wedge len-of\ TYPE('a) < m \bmod 2 \wedge len-of\ TYPE('a))$
 $\langle proof \rangle$

lemma *wi-le*:
 $(word-of-int\ n \leq (word-of-int\ m :: 'a :: len0\ word)) =$
 $(n \bmod 2 \wedge len-of\ TYPE('a) \leq m \bmod 2 \wedge len-of\ TYPE('a))$
 $\langle proof \rangle$

lemma *udvd-nat-alt*: $a\ udvd\ b = (EX\ n \geq 0. unat\ b = n * unat\ a)$
 $\langle proof \rangle$

lemma *udvd-iff-dvd*: $x\ udvd\ y \iff unat\ x\ dvd\ unat\ y$
 $\langle proof \rangle$

lemmas *unat-mono* = *word-less-nat-alt* [*THEN iffD1, standard*]

lemma *word-zero-neq-one*: $0 < len-of\ TYPE\ ('a :: len0) ==> (0 :: 'a\ word) \sim=$
 1
 $\langle proof \rangle$

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

lemma *no-no* [*simp*] : *number-of* (*number-of* *b*) = *number-of* *b*
 $\langle proof \rangle$

lemma *unat-minus-one*: $x \sim= 0 ==> unat\ (x - 1) = unat\ x - 1$
 $\langle proof \rangle$

lemma *measure-unat*: $p \sim= 0 ==> unat\ (p - 1) < unat\ p$
 $\langle proof \rangle$

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]:*
 $\text{uint } (x :: 'a :: \text{len0 word}) - \text{uint } (y :: 'b :: \text{len0 word}) <$
 $2^{\wedge \text{len-of TYPE('a)}}$
 $\langle \text{proof} \rangle$

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

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

lemma *uint-mult-lem:*
 $(\text{uint } x * \text{uint } y < 2^{\wedge \text{len-of TYPE('a)}} =$
 $(\text{uint } (x * y :: 'a :: \text{len0 word}) = \text{uint } x * \text{uint } y)$
 $\langle \text{proof} \rangle$

lemma *uint-sub-lem:*
 $(\text{uint } x \geq \text{uint } y) = (\text{uint } (x - y) = \text{uint } x - \text{uint } y)$
 $\langle \text{proof} \rangle$

lemma *uint-add-le:* $\text{uint } (x + y) \leq \text{uint } x + \text{uint } y$
 $\langle \text{proof} \rangle$

lemma *uint-sub-ge:* $\text{uint } (x - y) \geq \text{uint } x - \text{uint } y$
 $\langle \text{proof} \rangle$

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

11.4 Definition of uint_arith

lemma *word-of-int-inverse:*
 $\text{word-of-int } r = a \implies 0 \leq r \implies r < 2^{\wedge \text{len-of TYPE('a)}} \implies$
 $\text{uint } (a :: 'a :: \text{len0 word}) = r$
 $\langle \text{proof} \rangle$

lemma *uint-split:*
fixes $x :: 'a :: \text{len0 word}$
shows $P (\text{uint } x) =$
 $(\text{ALL } i. \text{word-of-int } i = x \ \& \ 0 \leq i \ \& \ i < 2^{\wedge \text{len-of TYPE('a)}} \longrightarrow P \ i)$
 $\langle \text{proof} \rangle$

lemma *uint-split-asm:*

fixes $x :: 'a :: \text{len0 word}$
shows $P (\text{uint } x) =$
 $(\sim (EX i. \text{word-of-int } i = x \ \& \ 0 \leq i \ \& \ i < 2^{\text{len-of TYPE('a)}} \ \& \ \sim P i))$
 $\langle \text{proof} \rangle$

lemmas $\text{uint-splits} = \text{uint-split uint-split-asm}$

lemmas $\text{uint-arith-simps} =$
 $\text{word-le-def word-less-alt}$
 $\text{word-uint.Rep-inject [symmetric]}$
 $\text{uint-sub-if' uint-plus-if'}$

lemma $\text{power-False-cong: False} ==> a \wedge b = c \wedge d$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

11.5 More on overflows and monotonicity

lemma $\text{no-plus-overflow-uint-size:}$
 $((x :: 'a :: \text{len0 word}) \leq x + y) = (\text{uint } x + \text{uint } y < 2^{\text{size } x})$
 $\langle \text{proof} \rangle$

lemmas $\text{no-olen-add} = \text{no-plus-overflow-uint-size [unfolded word-size]}$

lemma $\text{no-ulen-sub: } ((x :: 'a :: \text{len0 word}) \geq x - y) = (\text{uint } y \leq \text{uint } x)$
 $\langle \text{proof} \rangle$

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

lemmas $\text{olen-add-eqv} = \text{trans [OF no-olen-add no-olen-add' [symmetric], standard]}$

lemmas $\text{uint-plus-simple-iff} = \text{trans [OF no-olen-add uint-add-lem, standard]}$

lemmas $\text{uint-plus-simple} = \text{uint-plus-simple-iff [THEN iffD1, standard]}$

lemmas $\text{uint-minus-simple-iff} = \text{trans [OF no-ulen-sub uint-sub-lem, standard]}$

lemmas $\text{uint-minus-simple-alt} = \text{uint-sub-lem [folded word-le-def]}$

lemmas $\text{word-sub-le-iff} = \text{no-ulen-sub [folded word-le-def]}$

lemmas $\text{word-sub-le} = \text{word-sub-le-iff [THEN iffD2, standard]}$

lemma word-less-sub1:
 $(x :: 'a :: \text{len word}) \sim 0 ==> (1 < x) = (0 < x - 1)$
 $\langle \text{proof} \rangle$

lemma *word-le-sub1*:

$$(x :: 'a :: \text{len0 word}) \sim = 0 \implies (1 \leq x) = (0 \leq x - 1)$$

<proof>

lemma *sub-wrap-lt*:

$$((x :: 'a :: \text{len0 word}) < x - z) = (x < z)$$

<proof>

lemma *sub-wrap*:

$$((x :: 'a :: \text{len0 word}) \leq x - z) = (z = 0 \mid x < z)$$

<proof>

lemma *plus-minus-not-NULL-ab*:

$$(x :: 'a :: \text{len0 word}) \leq ab - c \implies c \leq ab \implies c \sim = 0 \implies x + c \sim = 0$$

<proof>

lemma *plus-minus-no-overflow-ab*:

$$(x :: 'a :: \text{len0 word}) \leq ab - c \implies c \leq ab \implies x \leq x + c$$

<proof>

lemma *le-minus'*:

$$(a :: 'a :: \text{len0 word}) + c \leq b \implies a \leq a + c \implies c \leq b - a$$

<proof>

lemma *le-plus'*:

$$(a :: 'a :: \text{len0 word}) \leq b \implies c \leq b - a \implies a + c \leq b$$

<proof>

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

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

lemma *word-plus-mono-right*:

$$(y :: 'a :: \text{len0 word}) \leq z \implies x \leq x + z \implies x + y \leq x + z$$

<proof>

lemma *word-less-minus-cancel*:

$$y - x < z - x \implies x \leq z \implies (y :: 'a :: \text{len0 word}) < z$$

<proof>

lemma *word-less-minus-mono-left*:

$$(y :: 'a :: \text{len0 word}) < z \implies x \leq y \implies y - x < z - x$$

<proof>

lemma *word-less-minus-mono*:

$$a < c \implies d < b \implies a - b < a \implies c - d < c$$

$$\implies a - b < c - (d :: 'a :: \text{len word})$$

<proof>

lemma *word-le-minus-cancel*:

$y - x \leq z - x \implies x \leq z \implies (y :: 'a :: \text{len0 word}) \leq z$
 $\langle \text{proof} \rangle$

lemma *word-le-minus-mono-left*:

$(y :: 'a :: \text{len0 word}) \leq z \implies x \leq y \implies y - x \leq z - x$
 $\langle \text{proof} \rangle$

lemma *word-le-minus-mono*:

$a \leq c \implies d \leq b \implies a - b \leq a \implies c - d \leq c$
 $\implies a - b \leq c - (d :: 'a :: \text{len word})$
 $\langle \text{proof} \rangle$

lemma *plus-le-left-cancel-wrap*:

$(x :: 'a :: \text{len0 word}) + y' < x \implies x + y < x \implies (x + y' < x + y) = (y' < y)$
 $\langle \text{proof} \rangle$

lemma *plus-le-left-cancel-nowrap*:

$(x :: 'a :: \text{len0 word}) \leq x + y' \implies x \leq x + y \implies$
 $(x + y' < x + y) = (y' < y)$
 $\langle \text{proof} \rangle$

lemma *word-plus-mono-right2*:

$(a :: 'a :: \text{len0 word}) \leq a + b \implies c \leq b \implies a \leq a + c$
 $\langle \text{proof} \rangle$

lemma *word-less-add-right*:

$(x :: 'a :: \text{len0 word}) < y - z \implies z \leq y \implies x + z < y$
 $\langle \text{proof} \rangle$

lemma *word-less-sub-right*:

$(x :: 'a :: \text{len0 word}) < y + z \implies y \leq x \implies x - y < z$
 $\langle \text{proof} \rangle$

lemma *word-le-plus-either*:

$(x :: 'a :: \text{len0 word}) \leq y \mid x \leq z \implies y \leq y + z \implies x \leq y + z$
 $\langle \text{proof} \rangle$

lemma *word-less-nowrapI*:

$(x :: 'a :: \text{len0 word}) < z - k \implies k \leq z \implies 0 < k \implies x < x + k$
 $\langle \text{proof} \rangle$

lemma *inc-le*: $(i :: 'a :: \text{len word}) < m \implies i + 1 \leq m$

$\langle \text{proof} \rangle$

lemma *inc-i*:

$(1 :: 'a :: \text{len word}) \leq i \implies i < m \implies 1 \leq (i + 1) \ \& \ i + 1 \leq m$
 $\langle \text{proof} \rangle$

lemma *udvd-incr-lem*:

$up < uq \implies up = ua + n * uint\ K \implies$
 $uq = ua + n' * uint\ K \implies up + uint\ K \leq uq$
 $\langle proof \rangle$

lemma *udvd-incr'*:

$p < q \implies uint\ p = ua + n * uint\ K \implies$
 $uint\ q = ua + n' * uint\ K \implies p + K \leq q$
 $\langle proof \rangle$

lemma *udvd-decr'*:

$p < q \implies uint\ p = ua + n * uint\ K \implies$
 $uint\ q = ua + n' * uint\ K \implies p \leq q - K$
 $\langle proof \rangle$

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'*:

$xy < k \implies z\ udvd\ xy \implies z\ udvd\ k \implies xy \leq k - z$
 $\langle proof \rangle$

$\langle ML \rangle$

lemma *udvd-incr2-K*:

$p < a + s \implies a \leq a + s \implies K\ udvd\ s \implies K\ udvd\ p - a \implies a \leq$
 $p \implies$
 $0 < K \implies p \leq p + K \ \& \ p + K \leq a + s$
 $\langle proof \rangle$
 $\langle ML \rangle$

lemma *word-succ-rbl*:

$to-bl\ w = bl \implies to-bl\ (word-succ\ w) = (rev\ (rbl-succ\ (rev\ bl)))$
 $\langle proof \rangle$

lemma *word-pred-rbl*:

$to-bl\ w = bl \implies to-bl\ (word-pred\ w) = (rev\ (rbl-pred\ (rev\ bl)))$
 $\langle proof \rangle$

lemma *word-add-rbl*:

$to-bl\ v = vbl \implies to-bl\ w = wbl \implies$
 $to-bl\ (v + w) = (rev\ (rbl-add\ (rev\ vbl)\ (rev\ wbl)))$
 $\langle proof \rangle$

lemma *word-mult-rbl*:

$to-bl\ v = vbl \implies to-bl\ w = wbl \implies$
 $to-bl\ (v * w) = (rev\ (rbl-mult\ (rev\ vbl)\ (rev\ wbl)))$

$\langle proof \rangle$

lemma *rtb-rbl-ariths*:

$rev\ (to\text{-}bl\ w) = ys \implies rev\ (to\text{-}bl\ (word\text{-}succ\ w)) = rbl\text{-}succ\ ys$

$rev\ (to\text{-}bl\ w) = ys \implies rev\ (to\text{-}bl\ (word\text{-}pred\ w)) = rbl\text{-}pred\ ys$

$[| rev\ (to\text{-}bl\ v) = ys; rev\ (to\text{-}bl\ w) = xs |]$
 $\implies rev\ (to\text{-}bl\ (v * w)) = rbl\text{-}mult\ ys\ xs$

$[| rev\ (to\text{-}bl\ v) = ys; rev\ (to\text{-}bl\ w) = xs |]$
 $\implies rev\ (to\text{-}bl\ (v + w)) = rbl\text{-}add\ ys\ xs$
 $\langle proof \rangle$

11.6 Arithmetic type class instantiations

instance *word* :: (len0) comm-monoid-add $\langle proof \rangle$

instance *word* :: (len0) comm-monoid-mult
 $\langle proof \rangle$

instance *word* :: (len0) comm-semiring
 $\langle proof \rangle$

instance *word* :: (len0) ab-group-add $\langle proof \rangle$

instance *word* :: (len0) comm-ring $\langle proof \rangle$

instance *word* :: (len) comm-semiring-1
 $\langle proof \rangle$

instance *word* :: (len) comm-ring-1 $\langle proof \rangle$

instance *word* :: (len0) comm-semiring-0 $\langle proof \rangle$

instance *word* :: (len0) order $\langle proof \rangle$

instance *word* :: (len) recpower
 $\langle proof \rangle$

lemma *zero-bintrunc*:

$iszero\ (number\text{-}of\ x :: 'a :: len\ word) =$
 $(bintrunc\ (len\text{-}of\ TYPE('a))\ x = Int.Pls)$
 $\langle proof \rangle$

lemmas *word-le-0-iff* [simp] =
 $word\text{-}zero\text{-}le\ [THEN\ leD,\ THEN\ linorder\text{-}antisym\text{-}conv1]$

lemma *word-of-nat*: $of\text{-}nat\ n = word\text{-}of\text{-}int\ (int\ n)$
 $\langle proof \rangle$

lemma *word-of-int*: $of\text{-}int = word\text{-}of\text{-}int$
 $\langle proof \rangle$

lemma *word-of-int-nat*:
 $0 \leq x \implies word\text{-}of\text{-}int\ x = of\text{-}nat\ (nat\ x)$
 $\langle proof \rangle$

lemma *word-number-of-eq*:
 $number\text{-}of\ w = (of\text{-}int\ w :: 'a :: len\ word)$
 $\langle proof \rangle$

instance *word* :: $(len)\ number\text{-}ring$
 $\langle proof \rangle$

lemma *iszero-word-no* [simp] :
 $iszero\ (number\text{-}of\ bin :: 'a :: len\ word) =$
 $iszero\ (number\text{-}of\ (bintrunc\ (len\text{-}of\ TYPE('a))\ bin) :: int)$
 $\langle proof \rangle$

11.7 Word and nat

lemma *td-ext-unat'*:
 $n = len\text{-}of\ TYPE\ ('a :: len) \implies$
 $td\text{-}ext\ (unat :: 'a\ word \Rightarrow nat)\ of\text{-}nat$
 $(unats\ n)\ (\%i. i\ mod\ 2 \wedge n)$
 $\langle proof \rangle$

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*:
 $td\text{-}ext\ unat :: 'a :: len\ word \Rightarrow nat$
 $of\text{-}nat$
 $unats\ (len\text{-}of\ TYPE('a :: len))$
 $\%i. i\ mod\ 2 \wedge len\text{-}of\ TYPE('a :: len)$
 $\langle proof \rangle$

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

lemmas *unat-lt2p* [iff] = *word-unat.Rep* [unfolded *unats-def* *mem-Collect-eq*]

lemma *unat-le*: $y \leq unat\ (z :: 'a :: len\ word) \implies y : unats\ (len\text{-}of\ TYPE('a))$
 $\langle proof \rangle$

lemma *word-nchotomy*:

ALL w . *EX* n . $(w :: 'a :: \text{len word}) = \text{of-nat } n \ \& \ n < 2 \wedge \text{len-of TYPE } ('a)$
 $\langle \text{proof} \rangle$

lemma *of-nat-eq*:

fixes $w :: 'a :: \text{len word}$

shows $(\text{of-nat } n = w) = (\exists q. n = \text{unat } w + q * 2 \wedge \text{len-of TYPE } ('a))$

$\langle \text{proof} \rangle$

lemma *of-nat-eq-size*:

$(\text{of-nat } n = w) = (\text{EX } q. n = \text{unat } w + q * 2 \wedge \text{size } w)$

$\langle \text{proof} \rangle$

lemma *of-nat-0*:

$(\text{of-nat } m = (0 :: 'a :: \text{len word})) = (\exists q. m = q * 2 \wedge \text{len-of TYPE } ('a))$

$\langle \text{proof} \rangle$

lemmas *of-nat-2p = mult-1 [symmetric, THEN iffD2 [OF of-nat-0 exI]]*

lemma *of-nat-gt-0*: $\text{of-nat } k \sim = 0 \implies 0 < k$

$\langle \text{proof} \rangle$

lemma *of-nat-neq-0*:

$0 < k \implies k < 2 \wedge \text{len-of TYPE } ('a :: \text{len}) \implies \text{of-nat } k \sim = (0 :: 'a \text{ word})$

$\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-add*:

$\text{of-nat } a + \text{of-nat } b = \text{of-nat } (a + b)$

$\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-mult*:

$\text{of-nat } a * \text{of-nat } b = (\text{of-nat } (a * b) :: 'a :: \text{len word})$

$\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-Suc*:

$\text{word-succ } (\text{of-nat } a) = \text{of-nat } (\text{Suc } a)$

$\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-0*: $(0 :: 'a :: \text{len word}) = \text{of-nat } 0$

$\langle \text{proof} \rangle$

lemma *Abs-fnat-hom-1*: $(1 :: 'a :: \text{len word}) = \text{of-nat } (\text{Suc } 0)$

$\langle \text{proof} \rangle$

lemmas *Abs-fnat-homs =*

Abs-fnat-hom-add Abs-fnat-hom-mult Abs-fnat-hom-Suc

Abs-fnat-hom-0 Abs-fnat-hom-1

lemma *word-arith-nat-add*:

$a + b = \text{of-nat } (\text{unat } a + \text{unat } b)$

$\langle \text{proof} \rangle$

lemma *word-arith-nat-mult*:

$a * b = \text{of-nat } (\text{unat } a * \text{unat } b)$

$\langle \text{proof} \rangle$

lemma *word-arith-nat-Suc*:

$\text{word-succ } a = \text{of-nat } (\text{Suc } (\text{unat } a))$

$\langle \text{proof} \rangle$

lemma *word-arith-nat-div*:

$a \text{ div } b = \text{of-nat } (\text{unat } a \text{ div } \text{unat } b)$

$\langle \text{proof} \rangle$

lemma *word-arith-nat-mod*:

$a \text{ mod } b = \text{of-nat } (\text{unat } a \text{ mod } \text{unat } b)$

$\langle \text{proof} \rangle$

lemmas *word-arith-nat-defs* =

word-arith-nat-add word-arith-nat-mult

word-arith-nat-Suc Abs-fnat-hom-0

Abs-fnat-hom-1 word-arith-nat-div

word-arith-nat-mod

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*:

$(\text{unat } x + \text{unat } y < 2 \wedge \text{len-of TYPE('a)}) =$

$(\text{unat } (x + y :: 'a :: \text{len word}) = \text{unat } x + \text{unat } y)$

$\langle \text{proof} \rangle$

lemma *unat-mult-lem*:

$(\text{unat } x * \text{unat } y < 2 \wedge \text{len-of TYPE('a)}) =$

$(\text{unat } (x * y :: 'a :: \text{len word}) = \text{unat } x * \text{unat } y)$

$\langle \text{proof} \rangle$

lemmas *unat-plus-if'* =

trans [*OF unat-word-ariths(1) mod-nat-add, simplified, standard*]

lemma *le-no-overflow*:

$x \leq b \implies a \leq a + b \implies x \leq a + (b :: 'a :: \text{len0 word})$

$\langle \text{proof} \rangle$

lemmas *un-ui-le = trans*
[OF word-le-nat-alt [symmetric]
word-le-def,
standard]

lemma *unat-sub-if-size:*
 $\text{unat } (x - y) = (\text{if } \text{unat } y \leq \text{unat } x$
 $\text{then } \text{unat } x - \text{unat } y$
 $\text{else } \text{unat } x + 2^{\text{size } x} - \text{unat } y)$
 $\langle \text{proof} \rangle$

lemmas *unat-sub-if' = unat-sub-if-size [unfolded word-size]*

lemma *unat-div:* $\text{unat } ((x :: 'a :: \text{len word}) \text{ div } y) = \text{unat } x \text{ div } \text{unat } y$
 $\langle \text{proof} \rangle$

lemma *unat-mod:* $\text{unat } ((x :: 'a :: \text{len word}) \text{ mod } y) = \text{unat } x \text{ mod } \text{unat } y$
 $\langle \text{proof} \rangle$

lemma *uint-div:* $\text{uint } ((x :: 'a :: \text{len word}) \text{ div } y) = \text{uint } x \text{ div } \text{uint } y$
 $\langle \text{proof} \rangle$

lemma *uint-mod:* $\text{uint } ((x :: 'a :: \text{len word}) \text{ mod } y) = \text{uint } x \text{ mod } \text{uint } y$
 $\langle \text{proof} \rangle$

11.8 Definition of unat_arith tactic

lemma *unat-split:*
fixes $x :: 'a :: \text{len word}$
shows $P (\text{unat } x) =$
 $(\text{ALL } n. \text{ of-nat } n = x \ \& \ n < 2^{\text{len-of TYPE('a)}} \longrightarrow P \ n)$
 $\langle \text{proof} \rangle$

lemma *unat-split-asm:*
fixes $x :: 'a :: \text{len word}$
shows $P (\text{unat } x) =$
 $(\sim (\text{EX } n. \text{ of-nat } n = x \ \& \ n < 2^{\text{len-of TYPE('a)}} \ \& \ \sim P \ n))$
 $\langle \text{proof} \rangle$

lemmas *of-nat-inverse =*
 $\text{word-unat.Abs-inverse' [rotated, unfolded unats-def, simplified]}$

lemmas *unat-splits = unat-split unat-split-asm*

lemmas *unat-arith-simps =*
 $\text{word-le-nat-alt word-less-nat-alt}$
 $\text{word-unat.Rep-inject [symmetric]}$
 $\text{unat-sub-if' unat-plus-if' unat-div unat-mod}$

$\langle ML \rangle$

lemma *no-plus-overflow-unat-size*:

$((x :: 'a :: \text{len word}) \leq x + y) = (\text{unat } x + \text{unat } y < 2^{\text{size } x})$
 $\langle \text{proof} \rangle$

lemma *unat-sub*: $b \leq a \implies \text{unat } (a - b) = \text{unat } a - \text{unat } (b :: 'a :: \text{len word})$
 $\langle \text{proof} \rangle$

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*:

$(0 :: 'a :: \text{len word}) < y \implies \text{unat } x * \text{unat } y < 2^{\text{len-of TYPE('a)}} \implies$
 $x * y \text{ div } y = x$
 $\langle \text{proof} \rangle$

lemma *div-lt'*: $(i :: 'a :: \text{len word}) \leq k \text{ div } x \implies$
 $\text{unat } i * \text{unat } x < 2^{\text{len-of TYPE('a)}}$
 $\langle \text{proof} \rangle$

lemmas *div-lt''* = *order-less-imp-le* [THEN div-lt']

lemma *div-lt-mult*: $(i :: 'a :: \text{len word}) < k \text{ div } x \implies 0 < x \implies i * x < k$
 $\langle \text{proof} \rangle$

lemma *div-le-mult*:

$(i :: 'a :: \text{len word}) \leq k \text{ div } x \implies 0 < x \implies i * x \leq k$
 $\langle \text{proof} \rangle$

lemma *div-lt-uint'*:

$(i :: 'a :: \text{len word}) \leq k \text{ div } x \implies \text{uint } i * \text{uint } x < 2^{\text{len-of TYPE('a)}}$
 $\langle \text{proof} \rangle$

lemmas *div-lt-uint''* = *order-less-imp-le* [THEN div-lt-uint']

lemma *word-le-exists'*:

$(x :: 'a :: \text{len0 word}) \leq y \implies$
 $(\exists z. y = x + z \ \& \ \text{uint } x + \text{uint } z < 2^{\text{len-of TYPE('a)}})$
 $\langle \text{proof} \rangle$

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-uo* = *unat-le* [*THEN word-unat.Abs-inverse*]

lemmas *thd* = *refl* [*THEN* [2] *split-div-lemma* [*THEN iffD2*], *THEN conjunct1*]

lemma *thd1*:
 $a \text{ div } b * b \leq (a :: \text{nat})$
<proof>

lemmas *uno-simps* [*THEN le-unat-uo*, *standard*] =
mod-le-divisor div-le-dividend thd1

lemma *word-mod-div-equality*:
 $(n \text{ div } b) * b + (n \text{ mod } b) = (n :: 'a :: \text{len word})$
<proof>

lemma *word-div-mult-le*: $a \text{ div } b * b \leq (a :: 'a :: \text{len word})$
<proof>

lemma *word-mod-less-divisor*: $0 < n \implies m \text{ mod } n < (n :: 'a :: \text{len word})$
<proof>

lemma *word-of-int-power-hom*:
 $\text{word-of-int } a ^ n = (\text{word-of-int } (a ^ n) :: 'a :: \text{len word})$
<proof>

lemma *word-arith-power-alt*:
 $a ^ n = (\text{word-of-int } (\text{uint } a ^ n) :: 'a :: \text{len word})$
<proof>

lemma *of-bl-length-less*:
 $\text{length } x = k \implies k < \text{len-of TYPE('a)} \implies (\text{of-bl } x :: 'a :: \text{len word}) < 2 ^ k$
<proof>

11.9 Cardinality, finiteness of set of words

lemmas *card-lessThan'* = *card-lessThan* [*unfolded lessThan-def*]

lemmas *card-eq* = *word-unat.Abs-inj-on* [*THEN card-image*,
unfolded word-unat.image, unfolded unats-def, standard]

lemmas *card-word* = *trans* [*OF card-eq card-lessThan'*, *standard*]

lemma *finite-word-UNIV*: *finite* (*UNIV* :: 'a :: len word set)
 <proof>

lemma *card-word-size*:
 $\text{card } (\text{UNIV} :: 'a :: \text{len word set}) = (2 \wedge \text{size } (x :: 'a \text{ word}))$
 <proof>

end

12 WordBitwise: Bitwise Operations on Words

theory *WordBitwise*
imports *WordArith*
begin

lemmas *bin-log-bintrs* = *bin-trunc-not bin-trunc-xor bin-trunc-and bin-trunc-or*

lemmas *wils1* = *bin-log-bintrs* [*THEN word-ubin.norm-eq-iff* [*THEN iffD1*],
folded word-ubin.eq-norm, THEN eq-reflection, standard]

lemmas *word-log-binary-defs* =
word-and-def word-or-def word-xor-def

lemmas *word-no-log-defs* [*simp*] =
word-not-def [**where** *a=number-of a*,
unfolded word-no-wi wils1, folded word-no-wi, standard]
word-log-binary-defs [**where** *a=number-of a and b=number-of b*,
unfolded word-no-wi wils1, folded word-no-wi, standard]

lemmas *word-wi-log-defs* = *word-no-log-defs* [*unfolded word-no-wi*]

lemma *uint-or*: *uint* (*x OR y*) = (*uint x*) *OR* (*uint y*)
 <proof>

lemma *uint-and*: *uint* (*x AND y*) = (*uint x*) *AND* (*uint y*)
 <proof>

lemma *word-ops-nth-size*:
 $n < \text{size } (x :: 'a :: \text{len0 word}) \implies$
 $(x \text{ OR } y) !! n = (x !! n \mid y !! n) \ \&$
 $(x \text{ AND } y) !! n = (x !! n \ \& y !! n) \ \&$
 $(x \text{ XOR } y) !! n = (x !! n \ \sim = y !! n) \ \&$
 $(\text{NOT } x) !! n = (\sim x !! n)$
 <proof>

lemma *word-ao-nth*:
fixes $x :: 'a::len0$ *word*
shows $(x \text{ OR } y) !! n = (x !! n \mid y !! n) \ \&$
 $(x \text{ AND } y) !! n = (x !! n \ \& \ y !! n)$
 $\langle proof \rangle$

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
 $(x \text{ AND } y) \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $(x \text{ OR } y) \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $(x \text{ XOR } y) \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
 $\langle proof \rangle$

lemma *word-bw-comms*:
fixes $x :: 'a::len0$ *word*
shows
 $x \text{ AND } y = y \text{ AND } x$
 $x \text{ OR } y = y \text{ OR } x$
 $x \text{ XOR } y = y \text{ XOR } x$
 $\langle proof \rangle$

lemma *word-bw-lcs*:
fixes $x :: 'a::len0$ *word*
shows
 $y \text{ AND } x \text{ AND } z = x \text{ AND } y \text{ AND } z$
 $y \text{ OR } x \text{ OR } z = x \text{ OR } y \text{ OR } z$
 $y \text{ XOR } x \text{ XOR } z = x \text{ XOR } y \text{ XOR } z$
 $\langle proof \rangle$

lemma *word-log-esimps* [*simp*]:
fixes $x :: 'a::len0$ *word*
shows
 $x \text{ AND } 0 = 0$
 $x \text{ AND } -1 = x$
 $x \text{ OR } 0 = x$
 $x \text{ OR } -1 = -1$
 $x \text{ XOR } 0 = x$
 $x \text{ XOR } -1 = \text{NOT } x$
 $0 \text{ AND } x = 0$

$-1 \text{ AND } x = x$
 $0 \text{ OR } x = x$
 $-1 \text{ OR } x = -1$
 $0 \text{ XOR } x = x$
 $-1 \text{ XOR } x = \text{NOT } x$
 $\langle \text{proof} \rangle$

lemma *word-not-dist*:
fixes $x :: 'a::\text{len}0 \text{ word}$
shows
 $\text{NOT } (x \text{ OR } y) = \text{NOT } x \text{ AND } \text{NOT } y$
 $\text{NOT } (x \text{ AND } y) = \text{NOT } x \text{ OR } \text{NOT } y$
 $\langle \text{proof} \rangle$

lemma *word-bw-same*:
fixes $x :: 'a::\text{len}0 \text{ word}$
shows
 $x \text{ AND } x = x$
 $x \text{ OR } x = x$
 $x \text{ XOR } x = 0$
 $\langle \text{proof} \rangle$

lemma *word-ao-absorbs* [simp]:
fixes $x :: 'a::\text{len}0 \text{ word}$
shows
 $x \text{ AND } (y \text{ OR } x) = x$
 $x \text{ OR } y \text{ AND } x = x$
 $x \text{ AND } (x \text{ OR } y) = x$
 $y \text{ AND } x \text{ OR } x = x$
 $(y \text{ OR } x) \text{ AND } x = x$
 $x \text{ OR } x \text{ AND } y = x$
 $(x \text{ OR } y) \text{ AND } x = x$
 $x \text{ AND } y \text{ OR } x = x$
 $\langle \text{proof} \rangle$

lemma *word-not-not* [simp]:
 $\text{NOT } \text{NOT } (x :: 'a::\text{len}0 \text{ word}) = x$
 $\langle \text{proof} \rangle$

lemma *word-ao-dist*:
fixes $x :: 'a::\text{len}0 \text{ word}$
shows $(x \text{ OR } y) \text{ AND } z = x \text{ AND } z \text{ OR } y \text{ AND } z$
 $\langle \text{proof} \rangle$

lemma *word-oa-dist*:
fixes $x :: 'a::\text{len}0 \text{ word}$
shows $x \text{ AND } y \text{ OR } z = (x \text{ OR } z) \text{ AND } (y \text{ OR } z)$
 $\langle \text{proof} \rangle$

lemma *word-add-not* [simp]:

fixes $x :: 'a::len0 \text{ word}$

shows $x + NOT\ x = -1$

$\langle proof \rangle$

lemma *word-plus-and-or* [simp]:

fixes $x :: 'a::len0 \text{ word}$

shows $(x\ AND\ y) + (x\ OR\ y) = x + y$

$\langle proof \rangle$

lemma *leoa*:

fixes $x :: 'a::len0 \text{ word}$

shows $(w = (x\ OR\ y)) ==> (y = (w\ AND\ y))\ \langle proof \rangle$

lemma *leao*:

fixes $x' :: 'a::len0 \text{ word}$

shows $(w' = (x'\ AND\ y')) ==> (x' = (x'\ OR\ w'))\ \langle proof \rangle$

lemmas *word-ao-equiv* = *leao* [COMP *leoa* [COMP *iffI*]]

lemma *le-word-or2*: $x \leq x\ OR\ (y :: 'a::len0 \text{ word})$

$\langle proof \rangle$

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*: $to_bl\ (NOT\ w) = map\ Not\ (to_bl\ w)$

$\langle proof \rangle$

lemma *bl-word-xor*: $to_bl\ (v\ XOR\ w) = map2\ op\ \sim\ (to_bl\ v)\ (to_bl\ w)$

$\langle proof \rangle$

lemma *bl-word-or*: $to_bl\ (v\ OR\ w) = map2\ op\ |\ (to_bl\ v)\ (to_bl\ w)$

$\langle proof \rangle$

lemma *bl-word-and*: $to_bl\ (v\ AND\ w) = map2\ op\ \&\ (to_bl\ v)\ (to_bl\ w)$

$\langle proof \rangle$

lemma *word-lsb-alt*: $lsb\ (w :: 'a::len0 \text{ word}) = test_bit\ w\ 0$

$\langle proof \rangle$

lemma *word-lsb-1-0*: $lsb\ (1 :: 'a::len \text{ word}) \&\ \sim\ lsb\ (0 :: 'b::len0 \text{ word})$

$\langle proof \rangle$

lemma *word-lsb-last*: $lsb\ (w :: 'a::len \text{ word}) = last\ (to_bl\ w)$

$\langle proof \rangle$

lemma *word-lsb-int*: $lsb\ w = (uint\ w\ mod\ 2 = 1)$
 ⟨proof⟩

lemma *word-msb-sint*: $msb\ w = (sint\ w < 0)$
 ⟨proof⟩

lemma *word-msb-no'*:
 $w = number-of\ bin ==> msb\ (w::'a::len\ word) = bin_nth\ bin\ (size\ w - 1)$
 ⟨proof⟩

lemmas *word-msb-no* = *refl* [THEN *word-msb-no'*, *unfolded word-size*]

lemma *word-msb-nth'*: $msb\ (w::'a::len\ word) = bin_nth\ (uint\ w)\ (size\ w - 1)$
 ⟨proof⟩

lemmas *word-msb-nth* = *word-msb-nth'* [unfolded *word-size*]

lemma *word-msb-alt*: $msb\ (w::'a::len\ word) = hd\ (to_bl\ w)$
 ⟨proof⟩

lemma *word-set-nth*:
 $set_bit\ w\ n\ (test_bit\ w\ n) = (w::'a::len0\ word)$
 ⟨proof⟩

lemma *bin-nth-uint'*:
 $bin_nth\ (uint\ w)\ n = (rev\ (bin_to_bl\ (size\ w)\ (uint\ w))\ !\ n \ \&\ n < size\ w)$
 ⟨proof⟩

lemmas *bin-nth-uint* = *bin-nth-uint'* [unfolded *word-size*]

lemma *test-bit-bl*: $w\ !!\ n = (rev\ (to_bl\ w)\ !\ n \ \&\ n < size\ w)$
 ⟨proof⟩

lemma *to-bl-nth*: $n < size\ w ==> to_bl\ w\ !\ n = w\ !!\ (size\ w - Suc\ n)$
 ⟨proof⟩

lemma *test-bit-set*:
fixes $w :: 'a::len0\ word$
shows $(set_bit\ w\ n\ x) !!\ n = (n < size\ w \ \&\ x)$
 ⟨proof⟩

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 $test_bit\ w\ m$)
 ⟨proof⟩

lemma *of-bl-rep-False*: $of_bl\ (replicate\ n\ False\ @\ bs) = of_bl\ bs$
 ⟨proof⟩

lemma *msb-nth'*:

fixes $w :: 'a::len\ word$

shows $msb\ w = w\ !!\ (size\ w - 1)$

<proof>

lemmas $msb-nth = msb-nth'\ [unfolded\ word-size]$

lemmas $msb0 = len-gt-0\ [THEN\ diff-Suc-less,\ THEN$

$word-ops-nth-size\ [unfolded\ word-size],\ standard]$

lemmas $msb1 = msb0\ [where\ i = 0]$

lemmas $word-ops-msb = msb1\ [unfolded\ msb-nth\ [symmetric,\ unfolded\ One-nat-def]]$

lemmas $lsb0 = len-gt-0\ [THEN\ word-ops-nth-size\ [unfolded\ word-size],\ standard]$

lemmas $word-ops-lsb = lsb0\ [unfolded\ word-lsb-alt]$

lemma *td-ext-nth'*:

$n = size\ (w::'a::len0\ word) ==> ofn = set-bits ==> [w,\ ofn\ g] = l ==>$

$td-ext\ test-bit\ ofn\ \{f.\ ALL\ i.\ f\ i \longrightarrow i < n\}\ (\%h\ i.\ h\ i \ \&\ i < n)$

<proof>

lemmas $td-ext-nth = td-ext-nth'\ [OF\ refl\ refl\ refl,\ unfolded\ word-size]$

interpretation *test-bit*:

$td-ext\ op\ !! :: 'a::len0\ word ==> nat ==> bool$

set-bits

$\{f.\ \forall i.\ f\ i \longrightarrow i < len-of\ TYPE('a::len0)\}$

$(\lambda h\ i.\ h\ i \wedge i < len-of\ TYPE('a::len0))$

<proof>

declare *test-bit.Rep'* [simp del]

declare *test-bit.Rep'* [rule del]

lemmas $td-nth = test-bit.td-thm$

lemma *word-set-set-same*:

fixes $w :: 'a::len0\ word$

shows $set-bit\ (set-bit\ w\ n\ x)\ n\ y = set-bit\ w\ n\ y$

<proof>

lemma *word-set-set-diff*:

fixes $w :: 'a::len0\ word$

assumes $m \sim n$

shows $set-bit\ (set-bit\ w\ m\ x)\ n\ y = set-bit\ (set-bit\ w\ n\ y)\ m\ x$

<proof>

lemma *test-bit-no'*:

fixes $w :: 'a::len0\ word$

shows $w = number-of\ bin ==> test-bit\ w\ n = (n < size\ w \ \&\ bin-nth\ bin\ n)$

<proof>

lemmas *test-bit-no* =
refl [THEN test-bit-no', unfolded word-size, THEN eq-reflection, standard]

lemma *nth-0*: $\sim (0::'a::len0 \text{ word}) !! n$
<proof>

lemma *nth-sint*:
fixes *w* :: *'a*::*len* *word*
defines *l* \equiv *len-of TYPE ('a)*
shows *bin-nth (sint w) n* = (*if n < l - 1 then w !! n else w !! (l - 1)*)
<proof>

lemma *word-lsb-no*:
lsb (number-of bin :: 'a :: len word) = (*bin-last bin = bit.B1*)
<proof>

lemma *word-set-no*:
set-bit (number-of bin::'a::len0 word) n b =
number-of (bin-sc n (if b then bit.B1 else bit.B0) bin)
<proof>

lemmas *setBit-no* = *setBit-def [THEN trans [OF meta-eq-to-obj-eq word-set-no],*
simplified if-simps, THEN eq-reflection, standard]

lemmas *clearBit-no* = *clearBit-def [THEN trans [OF meta-eq-to-obj-eq word-set-no],*
simplified if-simps, THEN eq-reflection, standard]

lemma *to-bl-n1*:
to-bl (-1::'a::len0 word) = *replicate (len-of TYPE ('a)) True*
<proof>

lemma *word-msb-n1*: *msb (-1::'a::len word)*
<proof>

declare *word-set-set-same* [*simp*] *word-set-nth* [*simp*]
test-bit-no [*simp*] *word-set-no* [*simp*] *nth-0* [*simp*]
setBit-no [*simp*] *clearBit-no* [*simp*]
word-lsb-no [*simp*] *word-msb-no* [*simp*] *word-msb-n1* [*simp*] *word-lsb-1-0* [*simp*]

lemma *word-set-nth-iff*:
(set-bit w n b = w) = (w !! n = b | n >= size (w::'a::len0 word))
<proof>

lemma *test-bit-2p'*:
w = word-of-int (2 ^ n) ==>
w !! m = (m = n & m < size (w :: 'a :: len word))
<proof>

lemmas *test-bit-2p* = refl [THEN *test-bit-2p'*, unfolded *word-size*]

lemmas *nth-w2p* = *test-bit-2p* [unfolded *of-int-number-of-eq*
word-of-int [symmetric] *Int.of-int-power*]

lemma *uint-2p*:
 $(0 :: 'a :: \text{len } \text{word}) < 2^{\wedge} n ==> \text{uint } (2^{\wedge} n :: 'a :: \text{len } \text{word}) = 2^{\wedge} n$
 <proof>

lemma *word-of-int-2p*: $(\text{word-of-int } (2^{\wedge} n) :: 'a :: \text{len } \text{word}) = 2^{\wedge} n$
 <proof>

lemma *bang-is-le*: $x !! m ==> 2^{\wedge} m \leq (x :: 'a :: \text{len } \text{word})$
 <proof>

lemma *word-clr-le*:
 fixes $w :: 'a :: \text{len } 0 \text{ word}$
 shows $w \geq \text{set-bit } w \ n \ \text{False}$
 <proof>

lemma *word-set-ge*:
 fixes $w :: 'a :: \text{len } \text{word}$
 shows $w \leq \text{set-bit } w \ n \ \text{True}$
 <proof>

end

13 WordShift: Shifting, Rotating, and Splitting Words

theory *WordShift*
imports *WordBitwise*
begin

13.1 Bit shifting

lemma *shiffl1-number* [simp] :
 $\text{shiffl1 } (\text{number-of } w) = \text{number-of } (w \ \text{BIT } \text{bit.B0})$
 <proof>

lemma *shiffl1-0* [simp] : $\text{shiffl1 } 0 = 0$
 <proof>

lemmas *shiffl1-def-u* = *shiffl1-def* [folded *word-number-of-def*]

lemma *shiffl1-def-s*: $\text{shiffl1 } w = \text{number-of } (\text{sint } w \ \text{BIT } \text{bit.B0})$

$\langle proof \rangle$

lemma *shiftr1-0* [*simp*] : *shiftr1* 0 = 0
 $\langle proof \rangle$

lemma *sshiftr1-0* [*simp*] : *sshiftr1* 0 = 0
 $\langle proof \rangle$

lemma *sshiftr1-n1* [*simp*] : *sshiftr1* -1 = -1
 $\langle proof \rangle$

lemma *shiftr1-0* [*simp*] : (0::'a::len0 word) << n = 0
 $\langle proof \rangle$

lemma *shiftr-0* [*simp*] : (0::'a::len0 word) >> n = 0
 $\langle proof \rangle$

lemma *sshiftr-0* [*simp*] : 0 >>> n = 0
 $\langle proof \rangle$

lemma *sshiftr-n1* [*simp*] : -1 >>> n = -1
 $\langle proof \rangle$

lemma *nth-shiftr1*: *shiftr1* w !! n = (n < size w & n > 0 & w !! (n - 1))
 $\langle proof \rangle$

lemma *nth-shiftr1'* [*rule-format*]:
 ALL n. ((w::'a::len0 word) << m) !! n = (n < size w & n >= m & w !! (n - m))
 $\langle proof \rangle$

lemmas *nth-shiftr1* = *nth-shiftr1'* [*unfolded word-size*]

lemma *nth-shiftr1*: *shiftr1* w !! n = w !! Suc n
 $\langle proof \rangle$

lemma *nth-shiftr*:
 $\bigwedge n. ((w::'a::len0 word) >> m) !! n = w !! (n + m)$
 $\langle proof \rangle$

lemma *uint-shiftr1*: *uint* (*shiftr1* w) = *bin-rest* (*uint* w)
 $\langle proof \rangle$

lemma *nth-sshiftr1*:
sshiftr1 w !! n = (if n = size w - 1 then w !! n else w !! Suc n)
 $\langle proof \rangle$

lemma *nth-sshiftr* [rule-format] :

*ALL n. sshiftr w m !! n = (n < size w &
 (if n + m >= size w then w !! (size w - 1) else w !! (n + m)))*
 ⟨proof⟩

lemma *shiftr1-div-2*: *uint (shiftr1 w) = uint w div 2*

⟨proof⟩

lemma *sshiftr1-div-2*: *sint (sshiftr1 w) = sint w div 2*

⟨proof⟩

lemma *shiftr-div-2n*: *uint (shiftr w n) = uint w div 2 ^ n*

⟨proof⟩

lemma *sshiftr-div-2n*: *sint (sshiftr w n) = sint w div 2 ^ n*

⟨proof⟩

13.1.1 shift functions in terms of lists of bools

lemmas *bshiftr1-no-bin* [simp] =

bshiftr1-def [where *w=number-of w, unfolded to-bl-no-bin, standard*]

lemma *bshiftr1-bl*: *to-bl (bshiftr1 b w) = b # butlast (to-bl w)*

⟨proof⟩

lemma *shiftr1-of-bl*: *shiftr1 (of-bl bl) = of-bl (bl @ [False])*

⟨proof⟩

lemma *shiftr1-bl*: *shiftr1 (w :: 'a :: len0 word) = of-bl (to-bl w @ [False])*

⟨proof⟩

lemma *bl-shiftr1*:

to-bl (shiftr1 (w :: 'a :: len word)) = tl (to-bl w) @ [False]

⟨proof⟩

lemma *shiftr1-bl*: *shiftr1 w = of-bl (butlast (to-bl w))*

⟨proof⟩

lemma *bl-shiftr1*:

to-bl (shiftr1 (w :: 'a :: len word)) = False # butlast (to-bl w)

⟨proof⟩

lemma *shiftr1-rev*:

shiftr1 (w :: 'a :: len word) = word-reverse (shiftr1 (word-reverse w))

⟨proof⟩

lemma *shiftr-rev*:

shiffl (*w* :: 'a :: len word) *n* = word-reverse (*shiftr* (word-reverse *w*) *n*)
 ⟨proof⟩

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*)
 ⟨proof⟩

lemma *drop-shiftr*:
drop *n* (*to-bl* ((*w* :: 'a :: len word) >> *n*)) = *take* (*size* *w* − *n*) (*to-bl* *w*)
 ⟨proof⟩

lemma *drop-sshiftr*:
drop *n* (*to-bl* ((*w* :: 'a :: len word) >>> *n*)) = *take* (*size* *w* − *n*) (*to-bl* *w*)
 ⟨proof⟩

lemma *take-shiftr* [rule-format] :
n <= *size* (*w* :: 'a :: len word) --> *take* *n* (*to-bl* (*w* >> *n*)) =
replicate *n* False
 ⟨proof⟩

lemma *take-sshiftr'* [rule-format] :
n <= *size* (*w* :: 'a :: len word) --> *hd* (*to-bl* (*w* >>> *n*)) = *hd* (*to-bl* *w*) &
take *n* (*to-bl* (*w* >>> *n*)) = *replicate* *n* (*hd* (*to-bl* *w*))
 ⟨proof⟩

lemmas *hd-sshiftr* = *take-sshiftr'* [THEN conjunct1, standard]

lemmas *take-sshiftr* = *take-sshiftr'* [THEN conjunct2, standard]

lemma *atd-lem*: *take* *n* *xs* = *t* ==> *drop* *n* *xs* = *d* ==> *xs* = *t* @ *d*
 ⟨proof⟩

lemmas *bl-shiftr* = *atd-lem* [OF *take-shiftr* *drop-shiftr*]

lemmas *bl-sshiftr* = *atd-lem* [OF *take-sshiftr* *drop-sshiftr*]

lemma *shiffl*-of-bl: *of-bl* *bl* << *n* = *of-bl* (*bl* @ *replicate* *n* False)
 ⟨proof⟩

lemma *shiffl*-bl:
 (*w*::'a::len0 word) << (*n*::nat) = *of-bl* (*to-bl* *w* @ *replicate* *n* False)
 ⟨proof⟩

lemmas *shiffl*-number [simp] = *shiffl*-def [where *w*=number-of *w*, standard]

lemma *bl-shiftl*:

to-bl (*w* << *n*) = *drop n* (*to-bl w*) @ *replicate* (*min* (*size w*) *n*) *False*
 ⟨*proof*⟩

lemma *shiftl-zero-size*:

fixes *x* :: '*a*::len0 word
shows *size x* <= *n* ==> *x* << *n* = 0
 ⟨*proof*⟩

lemma *shiftl1-2t*: *shiftl1* (*w* :: '*a* :: len word) = 2 * *w*
 ⟨*proof*⟩

lemma *shiftl1-p*: *shiftl1* (*w* :: '*a* :: len word) = *w* + *w*
 ⟨*proof*⟩

lemma *shiftl-t2n*: *shiftl* (*w* :: '*a* :: len word) *n* = 2 ^ *n* * *w*
 ⟨*proof*⟩

lemma *shiftr1-bintr* [*simp*]:

(*shiftr1* (*number-of w*) :: '*a* :: len0 word) =
number-of (*bin-rest* (*bintrunc* (*len-of TYPE* ('*a*)) *w*))
 ⟨*proof*⟩

lemma *sshiftr1-sbintr* [*simp*] :

(*sshiftr1* (*number-of w*) :: '*a* :: len word) =
number-of (*bin-rest* (*sbintrunc* (*len-of TYPE* ('*a*) - 1) *w*))
 ⟨*proof*⟩

lemma *shiftr-no'*:

w = *number-of bin* ==>
(*w*::'*a*::len0 word) >> *n* = *number-of* ((*bin-rest* ^ *n*) (*bintrunc* (*size w*) *bin*))
 ⟨*proof*⟩

lemma *sshiftr-no'*:

w = *number-of bin* ==> *w* >>> *n* = *number-of* ((*bin-rest* ^ *n*)
(*sbintrunc* (*size w* - 1) *bin*))
 ⟨*proof*⟩

lemmas *sshiftr-no* [*simp*] =

sshiftr-no' [**where** *w* = *number-of w*, *OF refl*, *unfolded word-size*, *standard*]

lemmas *shiftr-no* [*simp*] =

shiftr-no' [**where** *w* = *number-of w*, *OF refl*, *unfolded word-size*, *standard*]

lemma *shiftr1-bl-of'*:

us = *shiftr1* (*of-bl bl*) ==> *length bl* <= *size us* ==>
us = *of-bl* (*butlast bl*)

<proof>

lemmas *shiftr1-bl-of* = *refl* [*THEN shiftr1-bl-of'*, *unfolded word-size*]

lemma *shiftr-bl-of'* [*rule-format*]:

us = *of-bl bl >> n ==> length bl <= size us -->*

us = *of-bl (take (length bl - n) bl)*

<proof>

lemmas *shiftr-bl-of* = *refl* [*THEN shiftr-bl-of'*, *unfolded word-size*]

lemmas *shiftr-bl* = *word-bl.Rep'* [*THEN eq-imp-le*, *THEN shiftr-bl-of*,
simplified word-size, *simplified*, *THEN eq-reflection*, *standard*]

lemma *msb-shift'*: *msb (w::'a::len word) <-> (w >> (size w - 1)) ~ = 0*

<proof>

lemmas *msb-shift* = *msb-shift'* [*unfolded word-size*]

lemma *align-lem-or* [*rule-format*] :

ALL x m. length x = n + m --> length y = n + m -->

drop m x = replicate n False --> take m y = replicate m False -->

map2 op | x y = take m x @ drop m y

<proof>

lemma *align-lem-and* [*rule-format*] :

ALL x m. length x = n + m --> length y = n + m -->

drop m x = replicate n False --> take m y = replicate m False -->

map2 op & x y = replicate (n + m) False

<proof>

lemma *aligned-bl-add-size'*:

size x - n = m ==> n <= size x ==> drop m (to-bl x) = replicate n False
==>

take m (to-bl y) = replicate m False ==>

to-bl (x + y) = take m (to-bl x) @ drop m (to-bl y)

<proof>

lemmas *aligned-bl-add-size* = *refl* [*THEN aligned-bl-add-size'*]

13.1.2 Mask

lemma *nth-mask'*: *m = mask n ==> test-bit m i = (i < n & i < size m)*

<proof>

lemmas *nth-mask* [*simp*] = *refl* [*THEN nth-mask'*]

lemma *mask-bl*: *mask n = of-bl (replicate n True)*

<proof>

lemma *mask-bin*: $\text{mask } n = \text{number-of } (\text{bintrunc } n \text{ Int.Min})$
 ⟨proof⟩

lemma *and-mask-bintr*: $w \text{ AND } \text{mask } n = \text{number-of } (\text{bintrunc } n \text{ (uint } w))$
 ⟨proof⟩

lemma *and-mask-no*: $\text{number-of } i \text{ AND } \text{mask } n = \text{number-of } (\text{bintrunc } n \text{ } i)$
 ⟨proof⟩

lemmas *and-mask-wi* = *and-mask-no* [unfolded word-number-of-def]

lemma *bl-and-mask*:
 $\text{to-bl } (w \text{ AND } \text{mask } n :: 'a :: \text{len word}) =$
 $\text{replicate } (\text{len-of TYPE('a)} - n) \text{ False } @$
 $\text{drop } (\text{len-of TYPE('a)} - n) (\text{to-bl } w)$
 ⟨proof⟩

lemmas *and-mask-mod-2p* =
and-mask-bintr [unfolded word-number-of-alt no-bintr-alt]

lemma *and-mask-lt-2p*: $\text{uint } (w \text{ AND } \text{mask } n) < 2 ^ n$
 ⟨proof⟩

lemmas *eq-mod-iff* = *trans* [symmetric, OF int-mod-lem eq-sym-conv]

lemma *mask-eq-iff*: $(w \text{ AND } \text{mask } n) = w <-> \text{uint } w < 2 ^ n$
 ⟨proof⟩

lemma *and-mask-dvd*: $2 ^ n \text{ dvd uint } w = (w \text{ AND } \text{mask } n = 0)$
 ⟨proof⟩

lemma *and-mask-dvd-nat*: $2 ^ n \text{ dvd unat } w = (w \text{ AND } \text{mask } n = 0)$
 ⟨proof⟩

lemma *word-2p-lem*:
 $n < \text{size } w ==> w < 2 ^ n = (\text{uint } (w :: 'a :: \text{len word}) < 2 ^ n)$
 ⟨proof⟩

lemma *less-mask-eq*: $x < 2 ^ n ==> x \text{ AND } \text{mask } n = (x :: 'a :: \text{len word})$
 ⟨proof⟩

lemmas *mask-eq-iff-w2p* =
trans [OF mask-eq-iff word-2p-lem [symmetric], standard]

lemmas *and-mask-less'* =
iffD2 [OF word-2p-lem and-mask-lt-2p, simplified word-size, standard]

lemma *and-mask-less-size*: $n < \text{size } x ==> x \text{ AND } \text{mask } n < 2 ^ n$

<proof>

lemma *word-mod-2p-is-mask'*:

$c = 2 \wedge n \implies c > 0 \implies x \bmod c = (x :: 'a :: \text{len word}) \text{ AND mask } n$
<proof>

lemmas *word-mod-2p-is-mask* = refl [THEN *word-mod-2p-is-mask'*]

lemma *mask-egs*:

$(a \text{ AND mask } n) + b \text{ AND mask } n = a + b \text{ AND mask } n$
 $a + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$
 $(a \text{ AND mask } n) - b \text{ AND mask } n = a - b \text{ AND mask } n$
 $a - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$
 $a * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$
 $(b \text{ AND mask } n) * a \text{ AND mask } n = b * a \text{ AND mask } n$
 $(a \text{ AND mask } n) + (b \text{ AND mask } n) \text{ AND mask } n = a + b \text{ AND mask } n$
 $(a \text{ AND mask } n) - (b \text{ AND mask } n) \text{ AND mask } n = a - b \text{ AND mask } n$
 $(a \text{ AND mask } n) * (b \text{ AND mask } n) \text{ AND mask } n = a * b \text{ AND mask } n$
 $-(a \text{ AND mask } n) \text{ AND mask } n = -a \text{ AND mask } n$
 $\text{word-succ } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-succ } a \text{ AND mask } n$
 $\text{word-pred } (a \text{ AND mask } n) \text{ AND mask } n = \text{word-pred } a \text{ AND mask } n$
<proof>

lemma *mask-power-eq*:

$(x \text{ AND mask } n) \wedge^k \text{ AND mask } n = x \wedge^k \text{ AND mask } n$
<proof>

13.1.3 Recast

lemmas *revcast-def'* = *revcast-def* [simplified]

lemmas *revcast-def''* = *revcast-def'* [simplified word-size]

lemmas *revcast-no-def* [simp] =

revcast-def' [where $w = \text{number-of } w$, unfolded word-size, standard]

lemma *to-bl-revcast*:

$\text{to-bl } (\text{revcast } w :: 'a :: \text{len0 word}) =$
 $\text{takefill False } (\text{len-of TYPE } ('a)) (\text{to-bl } w)$
<proof>

lemma *revcast-rev-ucast'*:

$cs = [rc, uc] \implies rc = \text{revcast } (\text{word-reverse } w) \implies uc = \text{ucast } w \implies$
 $rc = \text{word-reverse } uc$
<proof>

lemmas *revcast-rev-ucast* = *revcast-rev-ucast'* [OF refl refl refl]

lemmas *revcast-ucast* = *revcast-rev-ucast*

[where $w = \text{word-reverse } w$, simplified word-rev-rev, standard]

lemmas $ucast\text{-}revcast = revcast\text{-}rev\text{-}ucast$ [THEN word-rev-gal', standard]
lemmas $ucast\text{-}rev\text{-}revcast = revcast\text{-}ucast$ [THEN word-rev-gal', standard]

— linking revcast and cast via shift

lemmas $wsst\text{-}TYs = source\text{-}size\ target\text{-}size\ word\text{-}size$

lemma $revcast\text{-}down\text{-}uu'$:
 $rc = revcast ==> source\text{-}size\ rc = target\text{-}size\ rc + n ==>$
 $rc\ (w :: 'a :: len\ word) = ucast\ (w >> n)$
 ⟨proof⟩

lemma $revcast\text{-}down\text{-}us'$:
 $rc = revcast ==> source\text{-}size\ rc = target\text{-}size\ rc + n ==>$
 $rc\ (w :: 'a :: len\ word) = ucast\ (w >>> n)$
 ⟨proof⟩

lemma $revcast\text{-}down\text{-}su'$:
 $rc = revcast ==> source\text{-}size\ rc = target\text{-}size\ rc + n ==>$
 $rc\ (w :: 'a :: len\ word) = scast\ (w >> n)$
 ⟨proof⟩

lemma $revcast\text{-}down\text{-}ss'$:
 $rc = revcast ==> source\text{-}size\ rc = target\text{-}size\ rc + n ==>$
 $rc\ (w :: 'a :: len\ word) = scast\ (w >>> n)$
 ⟨proof⟩

lemmas $revcast\text{-}down\text{-}uu = refl$ [THEN revcast-down-uu']
lemmas $revcast\text{-}down\text{-}us = refl$ [THEN revcast-down-us']
lemmas $revcast\text{-}down\text{-}su = refl$ [THEN revcast-down-su']
lemmas $revcast\text{-}down\text{-}ss = refl$ [THEN revcast-down-ss']

lemma $cast\text{-}down\text{-}rev$:
 $uc = ucast ==> source\text{-}size\ uc = target\text{-}size\ uc + n ==>$
 $uc\ w = revcast\ ((w :: 'a :: len\ word) << n)$
 ⟨proof⟩

lemma $revcast\text{-}up'$:
 $rc = revcast ==> source\text{-}size\ rc + n = target\text{-}size\ rc ==>$
 $rc\ w = (ucast\ w :: 'a :: len\ word) << n$
 ⟨proof⟩

lemmas $revcast\text{-}up = refl$ [THEN revcast-up']

lemmas $rc1 = revcast\text{-}up$ [THEN
 $revcast\text{-}rev\text{-}ucast$ [symmetric, THEN trans, THEN word-rev-gal, symmetric]]
lemmas $rc2 = revcast\text{-}down\text{-}uu$ [THEN
 $revcast\text{-}rev\text{-}ucast$ [symmetric, THEN trans, THEN word-rev-gal, symmetric]]

lemmas *ucast-up* =
 rc1 [*simplified rev-shiftr* [*symmetric*] *revcast-ucast* [*symmetric*]]
lemmas *ucast-down* =
 rc2 [*simplified rev-shiftr revcast-ucast* [*symmetric*]]

13.1.4 Slices

lemmas *slice1-no-bin* [*simp*] =
 slice1-def [**where** *w=number-of w, unfolded to-bl-no-bin, standard*]

lemmas *slice-no-bin* [*simp*] =
 trans [*OF slice-def* [*THEN meta-eq-to-obj-eq*]
 slice1-no-bin [*THEN meta-eq-to-obj-eq*],
 unfolded word-size, standard]

lemma *slice1-0* [*simp*] : *slice1 n 0 = 0*
 ⟨*proof*⟩

lemma *slice-0* [*simp*] : *slice n 0 = 0*
 ⟨*proof*⟩

lemma *slice-take'*: *slice n w = of-bl (take (size w - n) (to-bl w))*
 ⟨*proof*⟩

lemmas *slice-take* = *slice-take'* [*unfolded word-size*]

— shiftr to a word of the same size is just slice, slice is just shiftr then ucast

lemmas *shiftr-slice* = *trans*
 [*OF shiftr-bl* [*THEN meta-eq-to-obj-eq*] *slice-take* [*symmetric*], *standard*]

lemma *slice-shiftr*: *slice n w = ucast (w >> n)*
 ⟨*proof*⟩

lemma *nth-slice*:
 (*slice n w :: 'a :: len0 word*) !! *m* =
 (*w* !! (*m* + *n*) & *m* < *len-of TYPE ('a)*)
 ⟨*proof*⟩

lemma *slice1-down-alt'*:
 sl = slice1 n w ==> fs = size sl ==> fs + k = n ==>
 to-bl sl = takefill False fs (drop k (to-bl w))
 ⟨*proof*⟩

lemma *slice1-up-alt'*:
 sl = slice1 n w ==> fs = size sl ==> fs = n + k ==>
 to-bl sl = takefill False fs (replicate k False @ (to-bl w))
 ⟨*proof*⟩

lemmas *sd1* = *slice1-down-alt'* [*OF* *refl refl*, *unfolded word-size*]
lemmas *su1* = *slice1-up-alt'* [*OF* *refl refl*, *unfolded word-size*]
lemmas *slice1-down-alt* = *le-add-diff-inverse* [*THEN* *sd1*]
lemmas *slice1-up-alt*s =
 le-add-diff-inverse [*symmetric*, *THEN* *su1*]
 le-add-diff-inverse2 [*symmetric*, *THEN* *su1*]

lemma *ucast-slice1*: *ucast w* = *slice1 (size w) w*
 ⟨*proof*⟩

lemma *ucast-slice*: *ucast w* = *slice 0 w*
 ⟨*proof*⟩

lemmas *slice-id* = *trans* [*OF* *ucast-slice* [*symmetric*] *ucast-id*]

lemma *revcast-slice1'*:
 rc = *revcast w* ==> *slice1 (size rc) w* = *rc*
 ⟨*proof*⟩

lemmas *revcast-slice1* = *refl* [*THEN* *revcast-slice1'*]

lemma *slice1-tf-tf'*:
 to-bl (slice1 n w :: 'a :: len0 word) =
 rev (takefill False (len-of TYPE('a)) (rev (takefill False n (to-bl w))))
 ⟨*proof*⟩

lemmas *slice1-tf-tf* = *slice1-tf-tf'*
 [*THEN* *word-bl.Rep-inverse'*, *symmetric*, *standard*]

lemma *rev-slice1*:
 n + k = *len-of TYPE('a) + len-of TYPE('b)* ==>
 slice1 n (word-reverse w :: 'b :: len0 word) =
 word-reverse (slice1 k w :: 'a :: len0 word)
 ⟨*proof*⟩

lemma *rev-slice'*:
 res = *slice n (word-reverse w)* ==> *n + k + size res* = *size w* ==>
 res = *word-reverse (slice k w)*
 ⟨*proof*⟩

lemmas *rev-slice* = *refl* [*THEN* *rev-slice'*, *unfolded word-size*]

lemmas *sym-notr* =
 not-iff [*THEN* *iffD2*, *THEN* *not-sym*, *THEN* *not-iff* [*THEN* *iffD1*]]

 — problem posed by TPHOLs referee: criterion for overflow of addition of signed integers

lemma *soft-test*:

$(\text{sint } (x :: 'a :: \text{len word}) + \text{sint } y = \text{sint } (x + y)) =$
 $((((x+y) \text{ XOR } x) \text{ AND } ((x+y) \text{ XOR } y)) >> (\text{size } x - 1) = 0)$
 $\langle \text{proof} \rangle$

13.2 Split and cat

lemmas *word-split-bin'* = *word-split-def* [THEN *meta-eq-to-obj-eq*, *standard*]

lemmas *word-cat-bin'* = *word-cat-def* [THEN *meta-eq-to-obj-eq*, *standard*]

lemma *word-rsplit-no*:

$(\text{word-rsplit } (\text{number-of } \text{bin} :: 'b :: \text{len0 word}) :: 'a \text{ word list}) =$
 $\text{map } \text{number-of } (\text{bin-rsplit } (\text{len-of TYPE } ('a :: \text{len})))$
 $(\text{len-of TYPE } ('b), \text{bintrunc } (\text{len-of TYPE } ('b)) \text{ bin}))$
 $\langle \text{proof} \rangle$

lemmas *word-rsplit-no-cl* [simp] = *word-rsplit-no*
 [unfolded *bin-rsplittl-def* *bin-rsplit-l* [symmetric]]

lemma *test-bit-cat*:

$\text{wc} = \text{word-cat } a \ b ==> \text{wc} !! n = (n < \text{size } \text{wc} \ \&$
 $(\text{if } n < \text{size } b \text{ then } b !! n \text{ else } a !! (n - \text{size } b)))$
 $\langle \text{proof} \rangle$

lemma *word-cat-bl*: $\text{word-cat } a \ b = \text{of-bl } (\text{to-bl } a \ @ \ \text{to-bl } b)$
 $\langle \text{proof} \rangle$

lemma *of-bl-append*:

$(\text{of-bl } (xs \ @ \ ys) :: 'a :: \text{len word}) = \text{of-bl } xs * 2^{(\text{length } ys)} + \text{of-bl } ys$
 $\langle \text{proof} \rangle$

lemma *of-bl-False* [simp]:

$\text{of-bl } (\text{False} \# xs) = \text{of-bl } xs$
 $\langle \text{proof} \rangle$

lemma *of-bl-True*:

$(\text{of-bl } (\text{True} \# xs) :: 'a :: \text{len word}) = 2^{\text{length } xs} + \text{of-bl } xs$
 $\langle \text{proof} \rangle$

lemma *of-bl-Cons*:

$\text{of-bl } (x \# xs) = \text{of-bool } x * 2^{\text{length } xs} + \text{of-bl } xs$
 $\langle \text{proof} \rangle$

lemma *split-uint-lem*: $\text{bin-split } n \ (\text{uint } (w :: 'a :: \text{len0 word})) = (a, b) ==>$
 $a = \text{bintrunc } (\text{len-of TYPE } ('a) - n) \ a \ \& \ b = \text{bintrunc } (\text{len-of TYPE } ('a)) \ b$
 $\langle \text{proof} \rangle$

lemma *word-split-bl'*:

$\text{std} = \text{size } c - \text{size } b ==> (\text{word-split } c = (a, b)) ==>$
 $(a = \text{of-bl } (\text{take } \text{std } (\text{to-bl } c)) \ \& \ b = \text{of-bl } (\text{drop } \text{std } (\text{to-bl } c)))$

$\langle \text{proof} \rangle$

lemma *word-split-bl*: $\text{std} = \text{size } c - \text{size } b \implies$
 $(a = \text{of-bl } (\text{take } \text{std } (\text{to-bl } c)) \ \& \ b = \text{of-bl } (\text{drop } \text{std } (\text{to-bl } c))) \iff$
 $\text{word-split } c = (a, b)$
 $\langle \text{proof} \rangle$

lemma *word-split-bl-eq*:
 $(\text{word-split } (c :: 'a :: \text{len } \text{word}) :: ('c :: \text{len0 } \text{word} * 'd :: \text{len0 } \text{word})) =$
 $(\text{of-bl } (\text{take } (\text{len-of TYPE('a :: \text{len})} - \text{len-of TYPE('d :: \text{len0})) } (\text{to-bl } c)),$
 $\text{of-bl } (\text{drop } (\text{len-of TYPE('a} - \text{len-of TYPE('d})) } (\text{to-bl } c)))$
 $\langle \text{proof} \rangle$

lemma *test-bit-split'*:
 $\text{word-split } c = (a, b) \iff (\text{ALL } n \ m. \ b \ \#\ n = (n < \text{size } b \ \& \ c \ \#\ n) \ \&$
 $a \ \#\ m = (m < \text{size } a \ \& \ c \ \#\ (m + \text{size } b)))$
 $\langle \text{proof} \rangle$

lemmas *test-bit-split* =
 $\text{test-bit-split' } [\text{THEN } \text{mp}, \text{ simplified all-simps, standard}]$

lemma *test-bit-split-eq*: $\text{word-split } c = (a, b) \iff$
 $((\text{ALL } n :: \text{nat}. \ b \ \#\ n = (n < \text{size } b \ \& \ c \ \#\ n)) \ \&$
 $(\text{ALL } m :: \text{nat}. \ a \ \#\ m = (m < \text{size } a \ \& \ c \ \#\ (m + \text{size } b))))$
 $\langle \text{proof} \rangle$

lemma *word-cat-id*: $\text{word-cat } a \ b = b$
 $\langle \text{proof} \rangle$

lemma *word-cat-hom*:
 $\text{len-of TYPE('a :: \text{len0})} \leq \text{len-of TYPE('b :: \text{len0})} + \text{len-of TYPE('c :: \text{len0})}$
 \implies
 $(\text{word-cat } (\text{word-of-int } w :: 'b \ \text{word}) \ (b :: 'c \ \text{word}) :: 'a \ \text{word}) =$
 $\text{word-of-int } (\text{bin-cat } w \ (\text{size } b) \ (\text{uint } b))$
 $\langle \text{proof} \rangle$

lemma *word-cat-split-alt*:
 $\text{size } w \leq \text{size } u + \text{size } v \implies \text{word-split } w = (u, v) \implies \text{word-cat } u \ v = w$
 $\langle \text{proof} \rangle$

lemmas *word-cat-split-size* =
 $\text{sym } [\text{THEN } [2] \ \text{word-cat-split-alt } [\text{symmetric}], \text{ standard}]$

13.2.1 Split and slice

lemma *split-slices*:
 $\text{word-split } w = (u, v) \implies u = \text{slice } (\text{size } v) \ w \ \& \ v = \text{slice } 0 \ w$
 $\langle \text{proof} \rangle$

lemma *slice-cat1'*:
 $wc = \text{word-cat } a \ b \implies \text{size } wc \geq \text{size } a + \text{size } b \implies \text{slice } (\text{size } b) \ wc = a$

$\langle \text{proof} \rangle$

lemmas *slice-cat1* = *refl* [*THEN slice-cat1*']

lemmas *slice-cat2* = *trans* [*OF slice-id word-cat-id*]

lemma *cat-slices*:

$a = \text{slice } n \ c ==> b = \text{slice } 0 \ c ==> n = \text{size } b ==>$

$\text{size } a + \text{size } b >= \text{size } c ==> \text{word-cat } a \ b = c$

$\langle \text{proof} \rangle$

lemma *word-split-cat-alt*:

$w = \text{word-cat } u \ v ==> \text{size } u + \text{size } v <= \text{size } w ==> \text{word-split } w = (u, v)$

$\langle \text{proof} \rangle$

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*]

$\langle \text{proof} \rangle$

lemma *word-rsplit-empty-iff-size*:

$(\text{word-rsplit } w = []) = (\text{size } w = 0)$

$\langle \text{proof} \rangle$

lemma *test-bit-rsplit*:

$sw = \text{word-rsplit } w ==> m < \text{size } (\text{hd } sw :: 'a :: \text{len } \text{word}) ==>$

$k < \text{length } sw ==> (\text{rev } sw ! k) !! m = (w !! (k * \text{size } (\text{hd } sw) + m))$

$\langle \text{proof} \rangle$

lemma *word-rcat-bl*: *word-rcat wl* == *of-bl (concat (map to-bl wl))*

$\langle \text{proof} \rangle$

lemma *size-rcat-lem'*:

$\text{size } (\text{concat } (\text{map } \text{to-bl } wl)) = \text{length } wl * \text{size } (\text{hd } wl)$

$\langle \text{proof} \rangle$

lemmas *size-rcat-lem* = *size-rcat-lem'* [*unfolded word-size*]

lemmas *td-gal-lt-len* = *len-gt-0* [*THEN td-gal-lt, standard*]

lemma *nth-rcat-lem'* [rule-format] :
 $sw = \text{size } (hd\ wl :: 'a :: \text{len word}) \implies (ALL\ n.\ n < \text{size } wl * sw \implies$
 $\text{rev } (\text{concat } (\text{map } \text{to-bl } wl))\ !\ n =$
 $\text{rev } (\text{to-bl } (\text{rev } wl\ !\ (n\ \text{div } sw)))\ !\ (n\ \text{mod } sw))$
 <proof>

lemmas *nth-rcat-lem* = refl [THEN *nth-rcat-lem'*, unfolded *word-size*]

lemma *test-bit-rcat*:
 $sw = \text{size } (hd\ wl :: 'a :: \text{len word}) \implies rc = \text{word-rcat } wl \implies rc\ !!\ n =$
 $(n < \text{size } rc \ \&\ n\ \text{div } sw < \text{size } wl \ \&\ (\text{rev } wl)\ !\ (n\ \text{div } sw)\ !!\ (n\ \text{mod } sw))$
 <proof>

lemma *foldl-eq-foldr* [rule-format] :
 $ALL\ x.\ \text{foldl } op\ +\ x\ xs = \text{foldr } op\ +\ (x\ \#\ xs)\ (0 :: 'a :: \text{comm-monoid-add})$
 <proof>

lemmas *test-bit-cong* = arg-cong [where $f = \text{test-bit}$, THEN *fun-cong*]

lemmas *test-bit-rsplit-alt* =
 trans [OF *nth-rev-alt* [THEN *test-bit-cong*]
test-bit-rsplit [OF refl *asm-rl diff-Suc-less*]]

— lazy way of expressing that *u* and *v*, and *su* and *sv*, have same types

lemma *word-rsplit-len-indep'*:
 $[u,v] = p \implies [su,sv] = q \implies \text{word-rsplit } u = su \implies$
 $\text{word-rsplit } v = sv \implies \text{length } su = \text{length } sv$
 <proof>

lemmas *word-rsplit-len-indep* = *word-rsplit-len-indep'* [OF refl refl refl refl]

lemma *length-word-rsplit-size*:
 $n = \text{len-of } TYPE\ ('a :: \text{len}) \implies$
 $(\text{length } (\text{word-rsplit } w :: 'a\ \text{word list}) \leq m) = (\text{size } w \leq m * n)$
 <proof>

lemmas *length-word-rsplit-lt-size* =
length-word-rsplit-size [unfolded *Not-eq-iff linorder-not-less* [symmetric]]

lemma *length-word-rsplit-exp-size*:
 $n = \text{len-of } TYPE\ ('a :: \text{len}) \implies$
 $\text{length } (\text{word-rsplit } w :: 'a\ \text{word list}) = (\text{size } w + n - 1)\ \text{div } n$
 <proof>

lemma *length-word-rsplit-even-size*:
 $n = \text{len-of } TYPE\ ('a :: \text{len}) \implies \text{size } w = m * n \implies$
 $\text{length } (\text{word-rsplit } w :: 'a\ \text{word list}) = m$
 <proof>

lemmas *length-word-rsplit-exp-size'* = refl [THEN *length-word-rsplit-exp-size*]

lemmas *tdle* = iffD2 [OF *split-div-lemma* refl, THEN *conjunct1*]

lemmas *dtle* = xtr4 [OF *tdle* *mult-commute*]

lemma *word-rcat-rsplit*: *word-rcat* (*word-rsplit* *w*) = *w*
 ⟨proof⟩

lemma *size-word-rsplit-rcat-size'*:
word-rcat (*ws* :: 'a :: len word list) = *frcw* ==>
size *frcw* = *length* *ws* * *len-of TYPE* ('a) ==>
size (*hd* [*word-rsplit* *frcw*, *ws*]) = *size* *ws*
 ⟨proof⟩

lemmas *size-word-rsplit-rcat-size* =
size-word-rsplit-rcat-size' [simplified]

lemma *msreus*:
fixes *n::nat*
shows $0 < n \implies (k * n + m) \text{ div } n = m \text{ div } n + k$
and $(k * n + m) \text{ mod } n = m \text{ mod } n$
 ⟨proof⟩

lemma *word-rsplit-rcat-size'*:
word-rcat (*ws* :: 'a :: len word list) = *frcw* ==>
size *frcw* = *length* *ws* * *len-of TYPE* ('a) ==> *word-rsplit* *frcw* = *ws*
 ⟨proof⟩

lemmas *word-rsplit-rcat-size* = refl [THEN *word-rsplit-rcat-size'*]

13.3 Rotation

lemmas *rotater-0'* [simp] = *rotater-def* [where *n* = 0, simplified]

lemmas *word-rot-defs* = *word-roti-def* *word-rotr-def* *word-rotl-def*

lemma *rotate-eq-mod*:
 $m \text{ mod } \text{length } xs = n \text{ mod } \text{length } xs \implies \text{rotate } m \text{ } xs = \text{rotate } n \text{ } xs$
 ⟨proof⟩

lemmas *rotate-eqs* [standard] =
trans [OF *rotate0* [THEN *fun-cong*] *id-apply*]
rotate-rotate [symmetric]
rotate-id
rotate-conv-mod
rotate-eq-mod

13.3.1 Rotation of list to right

lemma *rotate1-rl'*: $\text{rotater1 } (l @ [a]) = a \# l$
 $\langle \text{proof} \rangle$

lemma *rotate1-rl* [simp]: $\text{rotater1 } (\text{rotate1 } l) = l$
 $\langle \text{proof} \rangle$

lemma *rotate1-lr* [simp]: $\text{rotate1 } (\text{rotater1 } l) = l$
 $\langle \text{proof} \rangle$

lemma *rotater1-rev'*: $\text{rotater1 } (\text{rev } xs) = \text{rev } (\text{rotate1 } xs)$
 $\langle \text{proof} \rangle$

lemma *rotater-rev'*: $\text{rotater } n (\text{rev } xs) = \text{rev } (\text{rotate } n xs)$
 $\langle \text{proof} \rangle$

lemmas *rotater-rev* = *rotater-rev'* [where $xs = \text{rev } ys$, simplified, standard]

lemma *rotater-drop-take*:
 $\text{rotater } n xs =$
 $\text{drop } (\text{length } xs - n \bmod \text{length } xs) xs @$
 $\text{take } (\text{length } xs - n \bmod \text{length } xs) xs$
 $\langle \text{proof} \rangle$

lemma *rotater-Suc* [simp]:
 $\text{rotater } (\text{Suc } n) xs = \text{rotater1 } (\text{rotater } n xs)$
 $\langle \text{proof} \rangle$

lemma *rotate-inv-plus* [rule-format]:
 $ALL k. k = m + n \longrightarrow \text{rotater } k (\text{rotate } n xs) = \text{rotater } m xs \ \&$
 $\text{rotate } k (\text{rotater } n xs) = \text{rotate } m xs \ \&$
 $\text{rotater } n (\text{rotate } k xs) = \text{rotate } m xs \ \&$
 $\text{rotate } n (\text{rotater } k xs) = \text{rotater } m xs$
 $\langle \text{proof} \rangle$

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*: $(\text{rotater } n xs = ys) = (\text{rotate } n ys = xs)$
 $\langle \text{proof} \rangle$

lemma *rotate-gal'*: $(ys = \text{rotater } n xs) = (xs = \text{rotate } n ys)$
 $\langle \text{proof} \rangle$

lemma *length-rotater* [simp]:
 $\text{length } (\text{rotater } n \text{ } xs) = \text{length } xs$
 ⟨proof⟩

lemmas *rrs0* = *rotate-eqs* [THEN *restrict-to-left*,
simplified rotate-gal [symmetric] *rotate-gal'* [symmetric], *standard*]
lemmas *rrs1* = *rrs0* [THEN *reft* [THEN *rev-iffD1*]]
lemmas *rotater-eqs* = *rrs1* [simplified *length-rotater*, *standard*]
lemmas *rotater-0* = *rotater-eqs* (1)
lemmas *rotater-add* = *rotater-eqs* (2)

13.3.2 map, map2, commuting with rotate(r)

lemma *last-map*: $xs \sim [] ==> \text{last } (\text{map } f \text{ } xs) = f \text{ } (\text{last } xs)$
 ⟨proof⟩

lemma *butlast-map*:
 $xs \sim [] ==> \text{butlast } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{butlast } xs)$
 ⟨proof⟩

lemma *rotater1-map*: $\text{rotater1 } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{rotater1 } xs)$
 ⟨proof⟩

lemma *rotater-map*:
 $\text{rotater } n \text{ } (\text{map } f \text{ } xs) = \text{map } f \text{ } (\text{rotater } n \text{ } xs)$
 ⟨proof⟩

lemma *but-last-zip* [rule-format] :
 $ALL \text{ } ys. \text{length } xs = \text{length } ys \longrightarrow xs \sim [] \longrightarrow$
 $\text{last } (\text{zip } xs \text{ } ys) = (\text{last } xs, \text{last } ys) \ \&$
 $\text{butlast } (\text{zip } xs \text{ } ys) = \text{zip } (\text{butlast } xs) \text{ } (\text{butlast } ys)$
 ⟨proof⟩

lemma *but-last-map2* [rule-format] :
 $ALL \text{ } ys. \text{length } xs = \text{length } ys \longrightarrow xs \sim [] \longrightarrow$
 $\text{last } (\text{map2 } f \text{ } xs \text{ } ys) = f \text{ } (\text{last } xs) \text{ } (\text{last } ys) \ \&$
 $\text{butlast } (\text{map2 } f \text{ } xs \text{ } ys) = \text{map2 } f \text{ } (\text{butlast } xs) \text{ } (\text{butlast } ys)$
 ⟨proof⟩

lemma *rotater1-zip*:
 $\text{length } xs = \text{length } ys ==>$
 $\text{rotater1 } (\text{zip } xs \text{ } ys) = \text{zip } (\text{rotater1 } xs) \text{ } (\text{rotater1 } ys)$
 ⟨proof⟩

lemma *rotater1-map2*:
 $\text{length } xs = \text{length } ys ==>$
 $\text{rotater1 } (\text{map2 } f \text{ } xs \text{ } ys) = \text{map2 } f \text{ } (\text{rotater1 } xs) \text{ } (\text{rotater1 } ys)$
 ⟨proof⟩

lemmas *lrth* =
 box-equals [*OF asm-rl length-rotater* [*symmetric*]
 length-rotater [*symmetric*],
 THEN rotater1-map2]

lemma *rotater-map2*:
 length xs = length ys ==>
 rotater n (map2 f xs ys) = map2 f (rotater n xs) (rotater n ys)
 ⟨*proof*⟩

lemma *rotate1-map2*:
 length xs = length ys ==>
 rotate1 (map2 f xs ys) = map2 f (rotate1 xs) (rotate1 ys)
 ⟨*proof*⟩

lemmas *lth* = *box-equals* [*OF asm-rl length-rotate* [*symmetric*]
 length-rotate [*symmetric*], *THEN rotate1-map2*]

lemma *rotate-map2*:
 length xs = length ys ==>
 rotate n (map2 f xs ys) = map2 f (rotate n xs) (rotate n ys)
 ⟨*proof*⟩

lemma *to-bl-rotl*:
 to-bl (word-rotl n w) = rotate n (to-bl w)
 ⟨*proof*⟩

lemmas *blrs0* = *rotate-egs* [*THEN to-bl-rotl* [*THEN trans*]]

lemmas *word-rotl-egs* =
 blrs0 [*simplified word-bl.Rep' word-bl.Rep-inject to-bl-rotl* [*symmetric*]]

lemma *to-bl-rotr*:
 to-bl (word-rotr n w) = rotater n (to-bl w)
 ⟨*proof*⟩

lemmas *brrs0* = *rotater-egs* [*THEN to-bl-rotr* [*THEN trans*]]

lemmas *word-rotr-egs* =
 brrs0 [*simplified word-bl.Rep' word-bl.Rep-inject to-bl-rotr* [*symmetric*]]

declare *word-rotr-egs* (1) [*simp*]
declare *word-rotl-egs* (1) [*simp*]

lemma
 word-rot-rl [*simp*]:
 word-rotl k (word-rotr k v) = v **and**
 word-rot-lr [*simp*]:
 word-rotr k (word-rotl k v) = v

$\langle \text{proof} \rangle$

lemma

word-rot-gal:

$(\text{word-rot} n v = w) = (\text{word-rot} n w = v)$ **and**

word-rot-gal':

$(w = \text{word-rot} n v) = (v = \text{word-rot} n w)$

$\langle \text{proof} \rangle$

lemma *word-rot-rev:*

$\text{word-rot} n w = \text{word-reverse} (\text{word-rot} n (\text{word-reverse } w))$

$\langle \text{proof} \rangle$

lemma *word-roti-0 [simp]:* $\text{word-roti } 0 w = w$

$\langle \text{proof} \rangle$

lemmas *abl-cong = arg-cong [where f = of-bl]*

lemma *word-roti-add:*

$\text{word-roti } (m + n) w = \text{word-roti } m (\text{word-roti } n w)$

$\langle \text{proof} \rangle$

lemma *word-roti-conv-mod':* $\text{word-roti } n w = \text{word-roti } (n \bmod \text{int } (\text{size } w)) w$

$\langle \text{proof} \rangle$

lemmas *word-roti-conv-mod = word-roti-conv-mod' [unfolded word-size]*

13.3.3 Word rotation commutes with bit-wise operations

locale *word-rotate*

context *word-rotate*

begin

lemmas *word-rot-defs' = to-bl-rotl to-bl-rotr*

lemmas *blwl-syms [symmetric] = bl-word-not bl-word-and bl-word-or bl-word-xor*

lemmas *lbl-lbl = trans [OF word-bl.Rep' word-bl.Rep' [symmetric]]*

lemmas *ths-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:*

$\text{word-rotl } n (\text{NOT } v) = \text{NOT } \text{word-rotl } n v$

$\text{word-rotr } n (\text{NOT } v) = \text{NOT } \text{word-rotr } n v$


```

word-rotl n (x AND y) = word-rotl n x AND word-rotl n y
word-rotr n (x AND y) = word-rotr n x AND word-rotr n y
word-rotl n (x OR y) = word-rotl n x OR word-rotl n y
word-rotr n (x OR y) = word-rotr n x OR word-rotr n y
word-rotl n (x XOR y) = word-rotl n x XOR word-rotl n y
word-rotr n (x XOR y) = word-rotr n x XOR word-rotr n y
⟨proof⟩
end

```

```

lemmas word-rot-logs = word-rotate.word-rot-logs

```

```

lemmas bl-word-rotl-dt = trans [OF to-bl-rotl rotate-drop-take,
  simplified word-bl.Rep', standard]

```

```

lemmas bl-word-rotr-dt = trans [OF to-bl-rotr rotater-drop-take,
  simplified word-bl.Rep', standard]

```

```

lemma bl-word-roti-dt':
  n = nat ((- i) mod int (size (w :: 'a :: len word))) ==>
    to-bl (word-roti i w) = drop n (to-bl w) @ take n (to-bl w)
  ⟨proof⟩

```

```

lemmas bl-word-roti-dt = bl-word-roti-dt' [unfolded word-size]

```

```

lemmas word-rotl-dt = bl-word-rotl-dt
  [THEN word-bl.Rep-inverse' [symmetric], standard]

```

```

lemmas word-rotr-dt = bl-word-rotr-dt
  [THEN word-bl.Rep-inverse' [symmetric], standard]

```

```

lemmas word-roti-dt = bl-word-roti-dt
  [THEN word-bl.Rep-inverse' [symmetric], standard]

```

```

lemma word-rotx-0 [simp] : word-rotr i 0 = 0 & word-rotl i 0 = 0
  ⟨proof⟩

```

```

lemma word-roti-0' [simp] : word-roti n 0 = 0
  ⟨proof⟩

```

```

lemmas word-rotr-dt-no-bin' [simp] =
  word-rotr-dt [where w=number-of w, unfolded to-bl-no-bin, standard]

```

```

lemmas word-rotl-dt-no-bin' [simp] =
  word-rotl-dt [where w=number-of w, unfolded to-bl-no-bin, standard]

```

```

declare word-roti-def [simp]

```

```

end

```

14 Boolean-Algebra: Boolean Algebras

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

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

```
lemmas disj-ac =
  disj-assoc disj-commute
  mk-left-commute [where 'a = 'a, of disj, OF disj-assoc disj-commute]
```

```
lemmas conj-ac =
  conj-assoc conj-commute
  mk-left-commute [where 'a = 'a, of conj, OF conj-assoc conj-commute]
```

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

14.1 Complement

```
lemma complement-unique:
```

```
  assumes 1:  $a \sqcap x = \mathbf{0}$ 
  assumes 2:  $a \sqcup x = \mathbf{1}$ 
  assumes 3:  $a \sqcap y = \mathbf{0}$ 
  assumes 4:  $a \sqcup y = \mathbf{1}$ 
  shows  $x = y$ 
```

```
<proof>
```

```
lemma compl-unique:  $\llbracket x \sqcap y = \mathbf{0}; x \sqcup y = \mathbf{1} \rrbracket \Longrightarrow \sim x = y$ 
<proof>
```

```
lemma double-compl [simp]:  $\sim (\sim x) = x$ 
<proof>
```

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

14.2 Conjunction

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

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

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

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

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

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

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

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

lemmas *conj-disj-distrib* =
conj-disj-distrib conj-disj-distrib2

14.3 Disjunction

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

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

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

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

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

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

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

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

lemmas *disj-conj-distrib* =
disj-conj-distrib disj-conj-distrib2

14.4 De Morgan’s Laws

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

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

end

14.5 Symmetric Difference

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

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

lemma *xor-commute*: $x \oplus y = y \oplus x$
 $\langle proof \rangle$

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

lemmas *xor-ac* =
xor-assoc xor-commute
mk-left-commute [**where** $'a = 'a$, *of xor*, *OF xor-assoc xor-commute*]

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

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

lemma *xor-one-right* [*simp*]: $x \oplus \mathbf{1} = \sim x$
 $\langle proof \rangle$

lemma *xor-one-left* [*simp*]: $\mathbf{1} \oplus x = \sim x$
 $\langle proof \rangle$

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

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

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

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

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

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

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

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

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

end

end

15 WordGenLib: Miscellaneous Library for Words

theory *WordGenLib*
imports *WordShift Boolean-Algebra*
begin

declare *of-nat-2p* [*simp*]

lemma *word-int-cases*:

$$\llbracket \bigwedge n. \llbracket (x :: 'a::len0 \text{ word}) = \text{word-of-int } n; 0 \leq n; n < 2^{\text{len-of TYPE('a)}} \rrbracket \implies P \rrbracket$$

$$\implies P$$

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

lemma *word-nat-cases* [*cases type: word*]:

$$\llbracket \bigwedge n. \llbracket (x :: 'a::len \text{ word}) = \text{of-nat } n; n < 2^{\text{len-of TYPE('a)}} \rrbracket \implies P \rrbracket$$

$$\implies P$$

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

lemma *max-word-eq*:

$$(\text{max-word} :: 'a::len \text{ word}) = 2^{\text{len-of TYPE('a)}} - 1$$

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

lemma *max-word-max* [*simp,intro!*]:

$$n \leq \text{max-word}$$

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

lemma *word-of-int-2p-len*:

$$\text{word-of-int } (2^{\text{len-of TYPE('a)}}) = (0 :: 'a::len0 \text{ word})$$

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

lemma *word-pow-0*:

$$(2 :: 'a::len \text{ word})^{\text{len-of TYPE('a)}} = 0$$

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

lemma *max-word-wrap*: $x + 1 = 0 \implies x = \text{max-word}$

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

lemma *max-word-minus*:

$$\text{max-word} = (-1 :: 'a::len \text{ word})$$

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

lemma *max-word-bl* [*simp*]:

$$\text{to-bl } (\text{max-word} :: 'a::len \text{ word}) = \text{replicate } (\text{len-of TYPE('a)}) \text{ True}$$

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

lemma *max-test-bit* [*simp*]:

$$(\text{max-word} :: 'a::len \text{ word}) !! n = (n < \text{len-of TYPE('a)})$$

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

lemma *word-and-max* [*simp*]:

$$x \text{ AND } \text{max-word} = x$$

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

lemma *word-or-max* [*simp*]:

$$x \text{ OR } \text{max-word} = \text{max-word}$$

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

lemma *word-ao-dist2*:

$x \text{ AND } (y \text{ OR } z) = x \text{ AND } y \text{ OR } x \text{ AND } (z::'a::len0 \text{ word})$
 $\langle \text{proof} \rangle$

lemma *word-oa-dist2*:

$x \text{ OR } y \text{ AND } z = (x \text{ OR } y) \text{ AND } (x \text{ OR } (z::'a::len0 \text{ word}))$
 $\langle \text{proof} \rangle$

lemma *word-and-not [simp]*:

$x \text{ AND NOT } x = (0::'a::len0 \text{ word})$
 $\langle \text{proof} \rangle$

lemma *word-or-not [simp]*:

$x \text{ OR NOT } x = \text{max-word}$
 $\langle \text{proof} \rangle$

lemma *word-boolean*:

$\text{boolean } (op \text{ AND}) (op \text{ OR}) \text{ bitNOT } 0 \text{ max-word}$
 $\langle \text{proof} \rangle$

interpretation *word-bool-alg*:

$\text{boolean } op \text{ AND } op \text{ OR } \text{bitNOT } 0 \text{ max-word}$
 $\langle \text{proof} \rangle$

lemma *word-xor-and-or*:

$x \text{ XOR } y = x \text{ AND NOT } y \text{ OR NOT } x \text{ AND } (y::'a::len0 \text{ word})$
 $\langle \text{proof} \rangle$

interpretation *word-bool-alg*:

$\text{boolean-xor } op \text{ AND } op \text{ OR } \text{bitNOT } 0 \text{ max-word } op \text{ XOR}$
 $\langle \text{proof} \rangle$

lemma *shiftr-0 [iff]*:

$(x::'a::len0 \text{ word}) >> 0 = x$
 $\langle \text{proof} \rangle$

lemma *shiftr-0 [simp]*:

$(x :: 'a :: len \text{ word}) << 0 = x$
 $\langle \text{proof} \rangle$

lemma *shiftr-1 [simp]*:

$(1::'a::len \text{ word}) << n = 2^n$
 $\langle \text{proof} \rangle$

lemma *uint-lt-0 [simp]*:

$\text{uint } x < 0 = \text{False}$
 $\langle \text{proof} \rangle$

lemma *shiftr1-1 [simp]*:

shiftr1 (*1::'a::len word*) = 0
 ⟨proof⟩

lemma *shiftr-1* [*simp*]:
 (*1::'a::len word*) >> *n* = (if *n* = 0 then 1 else 0)
 ⟨proof⟩

lemma *word-less-1* [*simp*]:
 ((*x::'a::len word*) < 1) = (*x* = 0)
 ⟨proof⟩

lemma *to-bl-mask*:
to-bl (*mask n :: 'a::len word*) =
replicate (*len-of TYPE('a) - n*) *False* @
replicate (*min (len-of TYPE('a)) n*) *True*
 ⟨proof⟩

lemma *map-replicate-True*:
n = *length xs* ==>
map ($\lambda(x,y). x \& y$) (*zip xs (replicate n True)*) = *xs*
 ⟨proof⟩

lemma *map-replicate-False*:
n = *length xs* ==> *map* ($\lambda(x,y). x \& y$)
 (*zip xs (replicate n False)*) = *replicate n False*
 ⟨proof⟩

lemma *bl-and-mask*:
fixes *w :: 'a::len word*
fixes *n*
defines *n' ≡ len-of TYPE('a) - n*
shows *to-bl (w AND mask n) = replicate n' False @ drop n' (to-bl w)*
 ⟨proof⟩

lemma *drop-rev-takefill*:
length xs ≤ *n* ==>
drop (*n - length xs*) (*rev (takefill False n (rev xs))*) = *xs*
 ⟨proof⟩

lemma *map-nth-0* [*simp*]:
map (*op !! (0::'a::len0 word)*) *xs* = *replicate (length xs) False*
 ⟨proof⟩

lemma *uint-plus-if-size*:
uint (*x + y*) =
 (if *uint x + uint y* < 2^{size *x*} then
 uint x + uint y
 else
 uint x + uint y - 2^{size x})

$\langle \text{proof} \rangle$

lemma *unat-plus-if-size*:

$\text{unat } (x + (y :: 'a :: \text{len word})) =$
 $(\text{if } \text{unat } x + \text{unat } y < 2^{\text{size } x} \text{ then}$
 $\quad \text{unat } x + \text{unat } y$
 else
 $\quad \text{unat } x + \text{unat } y - 2^{\text{size } x})$
 $\langle \text{proof} \rangle$

lemma *word-neq-0-conv* [simp]:

fixes $w :: 'a :: \text{len word}$
shows $(w \neq 0) = (0 < w)$
 $\langle \text{proof} \rangle$

lemma *max-lt*:

$\text{unat } (\max a b \text{ div } c) = \text{unat } (\max a b) \text{ div } \text{unat } (c :: 'a :: \text{len word})$
 $\langle \text{proof} \rangle$

lemma *uint-sub-if-size*:

$\text{uint } (x - y) =$
 $(\text{if } \text{uint } y \leq \text{uint } x \text{ then}$
 $\quad \text{uint } x - \text{uint } y$
 else
 $\quad \text{uint } x - \text{uint } y + 2^{\text{size } x})$
 $\langle \text{proof} \rangle$

lemma *unat-sub-simple*:

$x \leq y \implies \text{unat } (y - x) = \text{unat } y - \text{unat } x$
 $\langle \text{proof} \rangle$

lemmas *unat-sub = unat-sub-simple*

lemma *word-less-sub1*:

fixes $x :: 'a :: \text{len word}$
shows $x \neq 0 \implies 1 < x = (0 < x - 1)$
 $\langle \text{proof} \rangle$

lemma *word-le-sub1*:

fixes $x :: 'a :: \text{len word}$
shows $x \neq 0 \implies 1 \leq x = (0 \leq x - 1)$
 $\langle \text{proof} \rangle$

lemmas *word-less-sub1-numberof* [simp] =
 $\text{word-less-sub1 } [\text{of number-of } w, \text{ standard}]$

lemmas *word-le-sub1-numberof* [simp] =
 $\text{word-le-sub1 } [\text{of number-of } w, \text{ standard}]$

lemma *word-of-int-minus*:

$\text{word-of-int } (2^{\text{len-of TYPE('a)} - i}) = (\text{word-of-int } (-i)::'a::\text{len word})$
 $\langle \text{proof} \rangle$

lemmas $\text{word-of-int-inj} =$
 $\text{word-uint.Abs-inject } [\text{unfolded uints-num, simplified}]$

lemma word-le-less-eq :
 $(x :: 'z::\text{len word}) \leq y = (x = y \vee x < y)$
 $\langle \text{proof} \rangle$

lemma mod-plus-cong :
assumes $1: (b::\text{int}) = b'$
and $2: x \bmod b' = x' \bmod b'$
and $3: y \bmod b' = y' \bmod b'$
and $4: x' + y' = z'$
shows $(x + y) \bmod b = z' \bmod b'$
 $\langle \text{proof} \rangle$

lemma mod-minus-cong :
assumes $1: (b::\text{int}) = b'$
and $2: x \bmod b' = x' \bmod b'$
and $3: y \bmod b' = y' \bmod b'$
and $4: x' - y' = z'$
shows $(x - y) \bmod b = z' \bmod b'$
 $\langle \text{proof} \rangle$

lemma word-induct-less :
 $\llbracket P (0::'a::\text{len word}); \bigwedge n. \llbracket n < m; P n \rrbracket \implies P (1 + n) \rrbracket \implies P m$
 $\langle \text{proof} \rangle$

lemma word-induct :
 $\llbracket P (0::'a::\text{len word}); \bigwedge n. P n \implies P (1 + n) \rrbracket \implies P m$
 $\langle \text{proof} \rangle$

lemma $\text{word-induct2 } [\text{induct type}]$:
 $\llbracket P 0; \bigwedge n. \llbracket 1 + n \neq 0; P n \rrbracket \implies P (1 + n) \rrbracket \implies P (n::'b::\text{len word})$
 $\langle \text{proof} \rangle$

constdefs
 $\text{word-rec} :: 'a \Rightarrow ('b::\text{len word} \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b \text{ word} \Rightarrow 'a$
 $\text{word-rec forZero forSuc } n \equiv \text{nat-rec forZero (forSuc } \circ \text{ of-nat)} (\text{unat } n)$

lemma word-rec-0 : $\text{word-rec } z \text{ } s \text{ } 0 = z$
 $\langle \text{proof} \rangle$

lemma word-rec-Suc :
 $1 + n \neq (0::'a::\text{len word}) \implies \text{word-rec } z \text{ } s \text{ } (1 + n) = s \text{ } n (\text{word-rec } z \text{ } s \text{ } n)$
 $\langle \text{proof} \rangle$

lemma *word-rec-Pred*:

$n \neq 0 \implies \text{word-rec } z \ s \ n = s \ (n - 1) \ (\text{word-rec } z \ s \ (n - 1))$
 $\langle \text{proof} \rangle$

lemma *word-rec-in*:

$f \ (\text{word-rec } z \ (\lambda -. f) \ n) = \text{word-rec } (f \ z) \ (\lambda -. f) \ n$
 $\langle \text{proof} \rangle$

lemma *word-rec-in2*:

$f \ n \ (\text{word-rec } z \ f \ n) = \text{word-rec } (f \ 0 \ z) \ (f \circ \text{op} + 1) \ n$
 $\langle \text{proof} \rangle$

lemma *word-rec-twice*:

$m \leq n \implies \text{word-rec } z \ f \ n = \text{word-rec } (\text{word-rec } z \ f \ (n - m)) \ (f \circ \text{op} + (n - m)) \ m$
 $\langle \text{proof} \rangle$

lemma *word-rec-id*: $\text{word-rec } z \ (\lambda -. \text{id}) \ n = z$

$\langle \text{proof} \rangle$

lemma *word-rec-id-eq*: $\forall m < n. f \ m = \text{id} \implies \text{word-rec } z \ f \ n = z$

$\langle \text{proof} \rangle$

lemma *word-rec-max*:

$\forall m \geq n. m \neq -1 \longrightarrow f \ m = \text{id} \implies \text{word-rec } z \ f \ -1 = \text{word-rec } z \ f \ n$
 $\langle \text{proof} \rangle$

lemma *unatSuc*:

$1 + n \neq (0::'a::\text{len } \text{word}) \implies \text{unat } (1 + n) = \text{Suc } (\text{unat } n)$
 $\langle \text{proof} \rangle$

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]

end

16 Word: Word Library interafce

theory *Word*

imports *WordGenLib*

begin

see *Examples/WordExamples.thy* for examples
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.