Machine Words in Isabelle/HOL

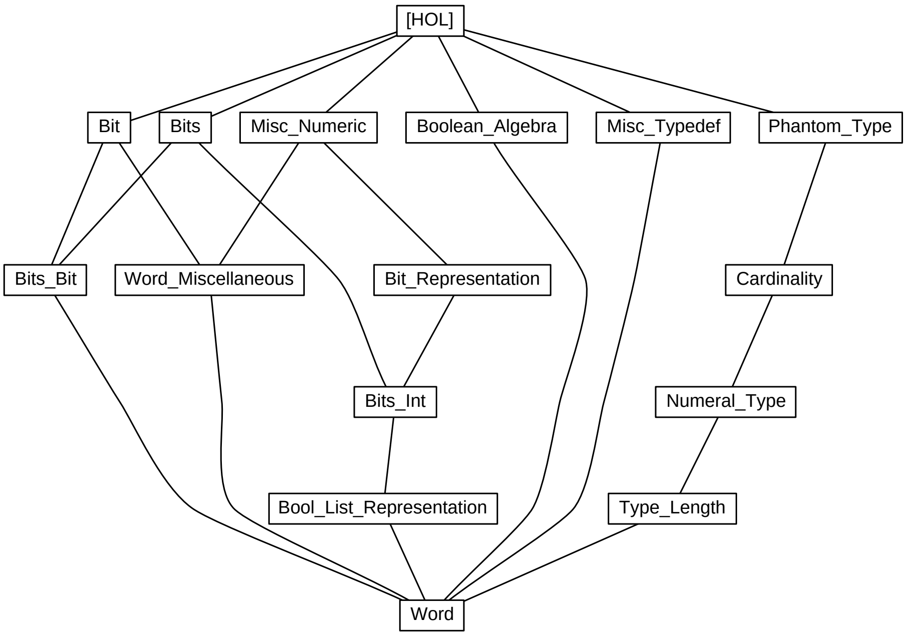
Jeremy Dawson, Paul Graunke, Brian Huffman, Gerwin Klein, and John Matthews

April 17, 2016

Abstract

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

Contents


[HOL]
Bit
Bits
Misc_Numeric
Boolean_Algebra
Misc_Typedef
Phantom_Type
Bits_Bit
Word_Miscellaneous
Bit_Representation
Cardinality
Bits_Int
Numeral_Type
Bool_List_Representation
Type_Length
Word

1 A generic phantom type

theory *Phantom-Type*
imports *Main*
begin

datatype ('a, 'b) *phantom* = *phantom* (*of-phantom*: 'b)

lemma *type-definition-phantom'*: *type-definition of-phantom phantom UNIV*
⟨*proof*⟩

lemma *phantom-comp-of-phantom* [*simp*]: *phantom* ∘ *of-phantom* = *id*
  and *of-phantom-comp-phantom* [*simp*]: *of-phantom* ∘ *phantom* = *id*
⟨*proof*⟩

syntax *-Phantom* :: *type* ⇒ *logic* ((*1Phantom*/(*1*'(-'))))
translations
  Phantom('t) => *CONST phantom* :: - ⇒ ('t, -) *phantom*

⟨*ML*⟩

lemma *of-phantom-inject* [*simp*]:
  of-phantom x = *of-phantom y* ⟷ *x* = *y*
⟨*proof*⟩

end

2 Cardinality of types

theory *Cardinality*
imports *Phantom-Type*
begin

2.1 Preliminary lemmas

lemma (**in** *type-definition*) *univ*:
  UNIV = *Abs* ‘ *A*
⟨*proof*⟩

lemma (**in** *type-definition*) *card*: *card* (*UNIV* :: 'b *set*) = *card A*
  ⟨*proof*⟩

lemma *finite-range-Some*: *finite* (*range* (*Some* :: 'a ⇒ 'a *option*)) = *finite* (*UNIV* :: 'a *set*)
⟨*proof*⟩

lemma *infinite-literal*: ¬ *finite* (*UNIV* :: *String.literal set*)
⟨*proof*⟩

2.2 Cardinalities of types

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

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

⟨*ML*⟩

lemma *card-prod* [*simp*]: $CARD('a \times 'b) = CARD('a) * CARD('b)$
⟨*proof*⟩

lemma *card-UNIV-sum*: $CARD('a + 'b) = (if\ CARD('a) \neq 0 \wedge CARD('b) \neq 0\ then\ CARD('a) + CARD('b)\ else\ 0)$
⟨*proof*⟩

lemma *card-sum* [*simp*]: $CARD('a + 'b) = CARD('a{::}finite) + CARD('b{::}finite)$
⟨*proof*⟩

lemma *card-UNIV-option*: $CARD('a\ option) = (if\ CARD('a) = 0\ then\ 0\ else\ CARD('a) + 1)$
⟨*proof*⟩

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

lemma *card-UNIV-set*: $CARD('a\ set) = (if\ CARD('a) = 0\ then\ 0\ else\ 2\ \hat{}\ CARD('a))$
⟨*proof*⟩

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

lemma *card-nat* [*simp*]: $CARD(nat) = 0$
⟨*proof*⟩

lemma *card-fun*: $CARD('a \Rightarrow 'b) = (if\ CARD('a) \neq 0 \wedge CARD('b) \neq 0 \vee CARD('b) = 1\ then\ CARD('b)\ \hat{}\ CARD('a)\ else\ 0)$
⟨*proof*⟩

corollary *finite-UNIV-fun*:
$finite\ (UNIV :: ('a \Rightarrow 'b)\ set) \longleftrightarrow$
$finite\ (UNIV :: 'a\ set) \wedge finite\ (UNIV :: 'b\ set) \vee CARD('b) = 1$
(**is** *?lhs* ⟷ *?rhs*)
⟨*proof*⟩

lemma *card-literal*: $CARD(String.literal) = 0$
⟨*proof*⟩

2.3 Classes with at least 1 and 2

Class finite already captures "at least 1"

lemma *zero-less-card-finite* [*simp*]: *0 < CARD('a::finite)*
⟨*proof*⟩

lemma *one-le-card-finite* [*simp*]: *Suc 0 ≤ CARD('a::finite)*
⟨*proof*⟩

Class for cardinality "at least 2"

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

lemma *one-less-card*: *Suc 0 < CARD('a::card2)*
⟨*proof*⟩

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

2.4 A type class for deciding finiteness of types

type-synonym *'a finite-UNIV = ('a, bool) phantom*

class *finite-UNIV =*
fixes *finite-UNIV :: ('a, bool) phantom*
assumes *finite-UNIV*: *finite-UNIV = Phantom('a) (finite (UNIV :: 'a set))*

lemma *finite-UNIV-code* [*code-unfold*]:
finite (UNIV :: 'a :: finite-UNIV set)
⟷ of-phantom (finite-UNIV :: 'a finite-UNIV)
⟨*proof*⟩

2.5 A type class for computing the cardinality of types

definition *is-list-UNIV :: 'a list ⇒ bool*
where *is-list-UNIV xs = (let c = CARD('a) in if c = 0 then False else size (remdups xs) = c)*

lemma *is-list-UNIV-iff*: *is-list-UNIV xs ⟷ set xs = UNIV*
⟨*proof*⟩

type-synonym *'a card-UNIV = ('a, nat) phantom*

class *card-UNIV = finite-UNIV +*
fixes *card-UNIV :: 'a card-UNIV*
assumes *card-UNIV*: *card-UNIV = Phantom('a) CARD('a)*

2.6 Instantiations for *card-UNIV*

instantiation *nat :: card-UNIV* **begin**
definition *finite-UNIV = Phantom(nat) False*
definition *card-UNIV = Phantom(nat) 0*
instance ⟨*proof*⟩

end

instantiation *int* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*int*) *False*
definition *card-UNIV* = *Phantom*(*int*) *0*
instance ⟨*proof*⟩
end

instantiation *natural* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*natural*) *False*
definition *card-UNIV* = *Phantom*(*natural*) *0*
instance
⟨*proof*⟩
end

instantiation *integer* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*integer*) *False*
definition *card-UNIV* = *Phantom*(*integer*) *0*
instance
⟨*proof*⟩
end

instantiation *list* :: (*type*) *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a list*) *False*
definition *card-UNIV* = *Phantom*('*a list*) *0*
instance ⟨*proof*⟩
end

instantiation *unit* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*unit*) *True*
definition *card-UNIV* = *Phantom*(*unit*) *1*
instance ⟨*proof*⟩
end

instantiation *bool* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*bool*) *True*
definition *card-UNIV* = *Phantom*(*bool*) *2*
instance ⟨*proof*⟩
end

instantiation *char* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*char*) *True*
definition *card-UNIV* = *Phantom*(*char*) *256*
instance ⟨*proof*⟩
end

instantiation *prod* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* × '*b*)
(*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *of-phantom* (*finite-UNIV* :: '*b*

finite-UNIV))
instance ⟨*proof*⟩
end

instantiation *prod* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* × '*b*)
(*of-phantom* (*card-UNIV* :: '*a card-UNIV*) ∗ *of-phantom* (*card-UNIV* :: '*b card-UNIV*))
instance ⟨*proof*⟩
end

instantiation *sum* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* + '*b*)
(*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *of-phantom* (*finite-UNIV* :: '*b finite-UNIV*))
instance
⟨*proof*⟩
end

instantiation *sum* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* + '*b*)
(*let ca* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0 then ca* + *cb else 0*)
instance ⟨*proof*⟩
end

instantiation *fun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* ⇒ '*b*)
(*let cb* = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in cb = *1* ∨ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *cb* ≠ *0*)
instance
⟨*proof*⟩
end

instantiation *fun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* ⇒ '*b*)
(*let ca* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0* ∨ *cb* = *1 then cb* ^ *ca else 0*)
instance ⟨*proof*⟩
end

instantiation *option* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a option*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance ⟨*proof*⟩
end

instantiation *option* :: (*card-UNIV*) *card-UNIV* **begin**

definition *card-UNIV* = *Phantom*('a *option*)
 (*let c* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*) *in if c* ≠ *0 then Suc c else 0*)
instance ⟨*proof*⟩
end

instantiation *String.literal* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*String.literal*) *False*
definition *card-UNIV* = *Phantom*(*String.literal*) *0*
instance
 ⟨*proof*⟩
end

instantiation *set* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('a *set*) (*of-phantom* (*finite-UNIV* :: 'a *finite-UNIV*))
instance ⟨*proof*⟩
end

instantiation *set* :: (*card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('a *set*)
 (*let c* = *of-phantom* (*card-UNIV* :: 'a *card-UNIV*) *in if c* = *0 then 0 else 2* ^ *c*)
instance ⟨*proof*⟩
end

lemma *UNIV-finite-1*: *UNIV* = *set* [*finite-1*.a_1]
⟨*proof*⟩

lemma *UNIV-finite-2*: *UNIV* = *set* [*finite-2*.a_1, *finite-2*.a_2]
⟨*proof*⟩

lemma *UNIV-finite-3*: *UNIV* = *set* [*finite-3*.a_1, *finite-3*.a_2, *finite-3*.a_3]
⟨*proof*⟩

lemma *UNIV-finite-4*: *UNIV* = *set* [*finite-4*.a_1, *finite-4*.a_2, *finite-4*.a_3, *finite-4*.a_4]
⟨*proof*⟩

lemma *UNIV-finite-5*:
 UNIV = *set* [*finite-5*.a_1, *finite-5*.a_2, *finite-5*.a_3, *finite-5*.a_4, *finite-5*.a_5]
⟨*proof*⟩

instantiation *Enum.finite-1* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-1*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-1*) *1*
instance
 ⟨*proof*⟩
end

instantiation *Enum.finite-2* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-2*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-2*) *2*

instance
⟨*proof*⟩
end

instantiation *Enum.finite-3* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-3*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-3*) *3*
instance
⟨*proof*⟩
end

instantiation *Enum.finite-4* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-4*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-4*) *4*
instance
⟨*proof*⟩
end

instantiation *Enum.finite-5* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*Enum.finite-5*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-5*) *5*
instance
⟨*proof*⟩
end

2.7 Code setup for sets

Implement *CARD*(*'a*) via *card-UNIV-class.card-UNIV* and provide implementations for *finite*, *card*, *op* ⊆, and *op* =if the calling context already provides *finite-UNIV* and *card-UNIV* instances. If we implemented the latter always via *card-UNIV-class.card-UNIV*, we would require instances of essentially all element types, i.e., a lot of instantiation proofs and – at run time – possibly slow dictionary constructions.

context
begin

qualified definition *card-UNIV'* :: *'a card-UNIV*
where [*code del*]: *card-UNIV'* = *Phantom*(*'a*) *CARD*(*'a*)

lemma *CARD-code* [*code-unfold*]:
CARD(*'a*) = *of-phantom* (*card-UNIV'* :: *'a card-UNIV*)
⟨*proof*⟩

lemma *card-UNIV'-code* [*code*]:
card-UNIV' = *card-UNIV*
⟨*proof*⟩

end

lemma *card-Compl*:
finite A $\Longrightarrow$ *card* $(- A)$ = *card* (*UNIV* :: *'a set*) $-$ *card* (*A* :: *'a set*)
⟨*proof*⟩

context fixes *xs* :: *'a* :: *finite-UNIV list*
begin

qualified definition *finite'* :: *'a set* $\Rightarrow$ *bool*
where [*simp*, *code del*, *code-abbrev*]: *finite'* = *finite*

lemma *finite'-code* [*code*]:
finite' (*set xs*) $\longleftrightarrow$ *True*
finite' (*List.coset xs*) $\longleftrightarrow$ *of-phantom* (*finite-UNIV* :: *'a finite-UNIV*)
⟨*proof*⟩

end

context fixes *xs* :: *'a* :: *card-UNIV list*
begin

qualified definition *card'* :: *'a set* $\Rightarrow$ *nat*
where [*simp*, *code del*, *code-abbrev*]: *card'* = *card*

lemma *card'-code* [*code*]:
card' (*set xs*) = *length* (*remdups xs*)
card' (*List.coset xs*) = *of-phantom* (*card-UNIV* :: *'a card-UNIV*) $-$ *length* (*remdups xs*)
⟨*proof*⟩ **definition** *subset'* :: *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool*
where [*simp*, *code del*, *code-abbrev*]: *subset'* = *op* $\subseteq$

lemma *subset'-code* [*code*]:
subset' A (*List.coset ys*) $\longleftrightarrow$ ($\forall y \in$ *set ys*. $y \notin A$)
subset' (*set ys*) *B* $\longleftrightarrow$ ($\forall y \in$ *set ys*. $y \in B$)
subset' (*List.coset xs*) (*set ys*) $\longleftrightarrow$ (*let n* = *CARD*(*'a*) *in* $n > 0 \wedge$ *card*(*set* (*xs* @ *ys*)) = *n*)
⟨*proof*⟩ **definition** *eq-set* :: *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool*
where [*simp*, *code del*, *code-abbrev*]: *eq-set* = *op* =

lemma *eq-set-code* [*code*]:
fixes *ys*
defines *rhs* $\equiv$
let n = *CARD*(*'a*)
in if n = *0 then False else*
let xs' = *remdups xs*; *ys'* = *remdups ys*
in length xs' + *length ys'* = *n* $\wedge$ ($\forall x \in$ *set xs'*. $x \notin$ *set ys'*) $\wedge$ ($\forall y \in$ *set ys'*. $y \notin$ *set xs'*)
shows *eq-set* (*List.coset xs*) (*set ys*) $\longleftrightarrow$ *rhs*
and *eq-set* (*set ys*) (*List.coset xs*) $\longleftrightarrow$ *rhs*

and *eq-set* (*set xs*) (*set ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
and *eq-set* (*List.coset xs*) (*List.coset ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
⟨*proof*⟩

end

Provide more informative exceptions than Match for non-rewritten cases. If generated code raises one these exceptions, then a code equation calls the mentioned operator for an element type that is not an instance of *card-UNIV* and is therefore not implemented via *card-UNIV-class.card-UNIV*. Constrain the element type with sort *card-UNIV* to change this.

lemma *card-coset-error* [*code*]:
card (*List.coset xs*) =
Code.abort (*STR ''card* (*List.coset* -) *requires type class instance card-UNIV''*)
(λ-. *card* (*List.coset xs*))
⟨*proof*⟩

lemma *coset-subseteq-set-code* [*code*]:
List.coset xs ⊆ *set ys* ⟷
(*if xs* = [] ∧ *ys* = [] *then False*
else Code.abort
(*STR ''subset-eq* (*List.coset* -) (*List.set* -) *requires type class instance card-UNIV''*)
(λ-. *List.coset xs* ⊆ *set ys*))
⟨*proof*⟩

notepad begin — test code setup
⟨*proof*⟩
end

end

3 Numeral Syntax for Types

theory *Numeral-Type*
imports *Cardinality*
begin

3.1 Numeral Types

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

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

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

⟨*proof*⟩

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

lemma *infinite-num0*: ¬ *finite* (*UNIV* :: *num0 set*)
⟨*proof*⟩

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

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

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

instance *num1* :: *finite*
⟨*proof*⟩

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

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

3.2 Locales for for modular arithmetic subtypes

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

lemma *size0*: *0* < *n*
⟨*proof*⟩

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

lemma *Rep-less-n*: *Rep x* < *n*

⟨*proof*⟩

lemma *Rep-le-n*: *Rep x* $\leq$ *n*
⟨*proof*⟩

lemma *Rep-inject-sym*: *x* = *y* $\longleftrightarrow$ *Rep x* = *Rep y*
⟨*proof*⟩

lemma *Rep-inverse*: *Abs* (*Rep x*) = *x*
⟨*proof*⟩

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

lemma *Rep-Abs-mod*: *Rep* (*Abs* (*m mod n*)) = *m mod n*
⟨*proof*⟩

lemma *Rep-Abs-0*: *Rep* (*Abs 0*) = *0*
⟨*proof*⟩

lemma *Rep-0*: *Rep 0* = *0*
⟨*proof*⟩

lemma *Rep-Abs-1*: *Rep* (*Abs 1*) = *1*
⟨*proof*⟩

lemma *Rep-1*: *Rep 1* = *1*
⟨*proof*⟩

lemma *Rep-mod*: *Rep x mod n* = *Rep x*
⟨*proof*⟩

lemmas *Rep-simps* =
  Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

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

end

locale *mod-ring* = *mod-type n Rep Abs*
  for *n* :: *int*
  and *Rep* :: *'a*::{*comm-ring-1*} $\Rightarrow$ *int*
  and *Abs* :: *int* $\Rightarrow$ *'a*::{*comm-ring-1*}
begin

lemma *of-nat-eq*: *of-nat k* = *Abs* (*int k mod n*)
⟨*proof*⟩

lemma *of-int-eq*: *of-int z* = *Abs* (*z mod n*)
⟨*proof*⟩

lemma *Rep-numeral*:
Rep (*numeral w*) = *numeral w mod n*
⟨*proof*⟩

lemma *iszero-numeral*:
iszero (*numeral w*::'*a*) ⟷ *numeral w mod n* = *0*
⟨*proof*⟩

lemma *cases*:
assumes *1*: ⋀*z*. ⟦(*x*::'*a*) = *of-int z*; *0* ≤ *z*; *z* < *n*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *induct*:
(⋀*z*. ⟦*0* ≤ *z*; *z* < *n*⟧ ⟹ *P* (*of-int z*)) ⟹ *P* (*x*::'*a*)
⟨*proof*⟩

end

3.3 Ring class instances

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

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

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

instance
⟨*proof*⟩

end

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

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

definition *Abs-bit1'* :: *int* ⇒ '*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 $x + y = \mathit{Abs\text{-}bit0}'\ (\mathit{Rep\text{-}bit0}\ x + \mathit{Rep\text{-}bit0}\ y)$
definition $x * y = \mathit{Abs\text{-}bit0}'\ (\mathit{Rep\text{-}bit0}\ x * \mathit{Rep\text{-}bit0}\ y)$
definition $x - y = \mathit{Abs\text{-}bit0}'\ (\mathit{Rep\text{-}bit0}\ x - \mathit{Rep\text{-}bit0}\ y)$
definition $- x = \mathit{Abs\text{-}bit0}'\ (- \mathit{Rep\text{-}bit0}\ x)$

definition $\mathit{0} = \mathit{Abs\text{-}bit1}\ \mathit{0}$
definition $\mathit{1} = \mathit{Abs\text{-}bit1}\ \mathit{1}$
definition $x + y = \mathit{Abs\text{-}bit1}'\ (\mathit{Rep\text{-}bit1}\ x + \mathit{Rep\text{-}bit1}\ y)$
definition $x * y = \mathit{Abs\text{-}bit1}'\ (\mathit{Rep\text{-}bit1}\ x * \mathit{Rep\text{-}bit1}\ y)$
definition $x - y = \mathit{Abs\text{-}bit1}'\ (\mathit{Rep\text{-}bit1}\ x - \mathit{Rep\text{-}bit1}\ y)$
definition $- x = \mathit{Abs\text{-}bit1}'\ (- \mathit{Rep\text{-}bit1}\ x)$

instance ⟨*proof*⟩

end

interpretation *bit0*:
mod-type int CARD('a::finite bit0)
Rep-bit0 :: *'a::finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ *'a::finite bit0*
⟨*proof*⟩

interpretation *bit1*:
mod-type int CARD('a::finite bit1)
Rep-bit1 :: *'a::finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ *'a::finite bit1*
⟨*proof*⟩

instance *bit0* :: (*finite*) *comm-ring-1*
⟨*proof*⟩

instance *bit1* :: (*finite*) *comm-ring-1*
⟨*proof*⟩

interpretation *bit0*:
mod-ring int CARD('a::finite bit0)
Rep-bit0 :: *'a::finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ *'a::finite bit0*
⟨*proof*⟩

interpretation *bit1*:
mod-ring int CARD('a::finite bit1)
Rep-bit1 :: *'a::finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ *'a::finite bit1*
⟨*proof*⟩

Set up cases, induction, and arithmetic

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

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

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

lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit0*] **for** *dummy* :: *'a::finite*
lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit1*] **for** *dummy* :: *'a::finite*

3.4 Order instances

instantiation *bit0* **and** *bit1* :: (*finite*) *linorder* **begin**
definition $a < b \longleftrightarrow$ *Rep-bit0* $a <$ *Rep-bit0* b
definition $a \leq b \longleftrightarrow$ *Rep-bit0* $a \leq$ *Rep-bit0* b
definition $a < b \longleftrightarrow$ *Rep-bit1* $a <$ *Rep-bit1* b
definition $a \leq b \longleftrightarrow$ *Rep-bit1* $a \leq$ *Rep-bit1* b

instance
⟨*proof*⟩
end

lemma (**in** *preorder*) *tranclp-less*: $op <^{++} = op <$
⟨*proof*⟩

instance *bit0* **and** *bit1* :: (*finite*) *wellorder*
⟨*proof*⟩

3.5 Code setup and type classes for code generation

Code setup for *num0* and *num1*

definition *Num0* :: *num0* **where** *Num0* = *Abs-num0 0*
code-datatype *Num0*

instantiation *num0* :: *equal* **begin**
definition *equal-num0* :: *num0* ⇒ *num0* ⇒ *bool*
where *equal-num0* = *op* =
instance ⟨*proof*⟩
end

lemma *equal-num0-code* [*code*]:
equal-class.equal Num0 Num0 = *True*
⟨*proof*⟩

code-datatype *1* :: *num1*

instantiation *num1* :: *equal* **begin**
definition *equal-num1* :: *num1* ⇒ *num1* ⇒ *bool*
where *equal-num1* = *op* =

instance ⟨*proof*⟩
end

lemma *equal-num1-code* [*code*]:
equal-class.equal (*1* :: *num1*) *1* = *True*
⟨*proof*⟩

instantiation *num1* :: *enum* **begin**
definition *enum-class.enum* = [*1* :: *num1*]
definition *enum-class.enum-all P* = *P* (*1* :: *num1*)
definition *enum-class.enum-ex P* = *P* (*1* :: *num1*)
instance
⟨*proof*⟩
end

instantiation *num0* **and** *num1* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*num0*) *False*
definition *card-UNIV* = *Phantom*(*num0*) *0*
definition *finite-UNIV* = *Phantom*(*num1*) *True*
definition *card-UNIV* = *Phantom*(*num1*) *1*
instance
⟨*proof*⟩
end

Code setup for *'a bit0* and *'a bit1*

declare
bit0.Rep-inverse[*code abstype*]
bit0.Rep-0[*code abstract*]
bit0.Rep-1[*code abstract*]

lemma *Abs-bit0'-code* [*code abstract*]:
Rep-bit0 (*Abs-bit0' x* :: *'a* :: *finite bit0*) = *x mod int* (*CARD*(*'a bit0*))
⟨*proof*⟩

lemma *inj-on-Abs-bit0*:
inj-on (*Abs-bit0* :: *int* ⇒ *'a bit0*) {*0*..<*2* ∗ *int CARD*(*'a* :: *finite*)}
⟨*proof*⟩

declare
bit1.Rep-inverse[*code abstype*]
bit1.Rep-0[*code abstract*]
bit1.Rep-1[*code abstract*]

lemma *Abs-bit1'-code* [*code abstract*]:
Rep-bit1 (*Abs-bit1' x* :: *'a* :: *finite bit1*) = *x mod int* (*CARD*(*'a bit1*))
⟨*proof*⟩

lemma *inj-on-Abs-bit1*:
inj-on (*Abs-bit1* :: *int* ⇒ *'a bit1*) {*0*..<*1* + *2* ∗ *int CARD*(*'a* :: *finite*)}

⟨*proof*⟩

instantiation *bit0* **and** *bit1* :: (*finite*) *equal* **begin**

definition *equal-class.equal x y* ⟷ *Rep-bit0 x* = *Rep-bit0 y*
definition *equal-class.equal x y* ⟷ *Rep-bit1 x* = *Rep-bit1 y*

instance
⟨*proof*⟩

end

instantiation *bit0* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit0 list*) = *map* (*Abs-bit0'* ∘ *int*) (*upt 0* (*CARD*(*'a bit0*)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)

instance
⟨*proof*⟩

end

instantiation *bit1* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit1 list*) = *map* (*Abs-bit1'* ∘ *int*) (*upt 0* (*CARD*(*'a bit1*)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)

instance
⟨*proof*⟩

end

instantiation *bit0* **and** *bit1* :: (*finite*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*'a bit0*) *True*
definition *finite-UNIV* = *Phantom*(*'a bit1*) *True*
instance ⟨*proof*⟩
end

instantiation *bit0* **and** *bit1* :: ({*finite*,*card-UNIV*}) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*(*'a bit0*) (*2* ∗ *of-phantom* (*card-UNIV* :: *'a card-UNIV*))
definition *card-UNIV* = *Phantom*(*'a bit1*) (*1* + *2* ∗ *of-phantom* (*card-UNIV* :: *'a card-UNIV*))
instance ⟨*proof*⟩
end

3.6 Syntax

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

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

⟨*ML*⟩

3.7 Examples

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

end

4 Assigning lengths to types by typeclasses

theory *Type-Length*
imports ~~/*src*/*HOL*/*Library*/*Numeral-Type*
begin

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

class *len0* =
fixes *len-of* :: *'a itself* ⇒ *nat*

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

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

instantiation *num0* **and** *num1* :: *len0*
begin

definition
len-num0: *len-of* (*x*::*num0 itself*) = *0*

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

instance ⟨*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*⟩

end

5 Boolean Algebras

theory *Boolean-Algebra*
imports *Main*
begin

locale *boolean* =
  fixes *conj* :: '*a* ⇒ '*a* ⇒ '*a* (**infixr** ⊓ *70*)
  fixes *disj* :: '*a* ⇒ '*a* ⇒ '*a* (**infixr** ⊔ *65*)
  fixes *compl* :: '*a* ⇒ '*a* (∼ - [*81*] *80*)
  fixes *zero* :: '*a* (**0**)
  fixes *one* :: '*a* (**1**)
  assumes *conj-assoc*: (*x* ⊓ *y*) ⊓ *z* = *x* ⊓ (*y* ⊓ *z*)
  assumes *disj-assoc*: (*x* ⊔ *y*) ⊔ *z* = *x* ⊔ (*y* ⊔ *z*)
  assumes *conj-commute*: *x* ⊓ *y* = *y* ⊓ *x*
  assumes *disj-commute*: *x* ⊔ *y* = *y* ⊔ *x*
  assumes *conj-disj-distrib*: *x* ⊓ (*y* ⊔ *z*) = (*x* ⊓ *y*) ⊔ (*x* ⊓ *z*)
  assumes *disj-conj-distrib*: *x* ⊔ (*y* ⊓ *z*) = (*x* ⊔ *y*) ⊓ (*x* ⊔ *z*)
  assumes *conj-one-right* [*simp*]: *x* ⊓ **1** = *x*
  assumes *disj-zero-right* [*simp*]: *x* ⊔ **0** = *x*
  assumes *conj-cancel-right* [*simp*]: *x* ⊓ ∼ *x* = **0**
  assumes *disj-cancel-right* [*simp*]: *x* ⊔ ∼ *x* = **1**
begin

sublocale *conj*: *abel-semigroup conj*
  ⟨*proof*⟩

sublocale *disj*: *abel-semigroup disj*
⟨*proof*⟩

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

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

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

lemma *dual*: *boolean disj conj compl one zero*
⟨*proof*⟩

5.1 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*: $[\![x \sqcap y = \mathbf{0}; x \sqcup y = \mathbf{1}]\!] \Longrightarrow \sim x = y$
⟨*proof*⟩

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

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

5.2 Conjunction

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

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

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

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

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

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

⟨*proof*⟩

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

lemma *conj-disj-distrib2*:
$(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
⟨*proof*⟩

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

5.3 Disjunction

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

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

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

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

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

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

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

lemma *disj-conj-distrib2*:
$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
⟨*proof*⟩

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

5.4 De Morgan's Laws

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

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

end

5.5 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

sublocale *xor*: *abel-semigroup xor*
⟨*proof*⟩

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

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

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

lemma *xor-zero-right* [*simp*]: $x \oplus \mathbf{0} = x$
⟨*proof*⟩

lemma *xor-zero-left* [*simp*]: $\mathbf{0} \oplus x = x$
⟨*proof*⟩

lemma *xor-one-right* [*simp*]: $x \oplus \mathbf{1} = \sim x$
⟨*proof*⟩

lemma *xor-one-left* [*simp*]: $\mathbf{1} \oplus x = \sim x$
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

end

end

6 Syntactic classes for bitwise operations

theory *Bits*
imports *Main*
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 ~~ 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* :: $'a \Rightarrow bool$

end

7 The Field of Integers mod 2

theory *Bit*
imports *Main*

begin

7.1 Bits as a datatype

typedef *bit* = *UNIV* :: *bool set*
morphisms *set Bit*
⟨*proof*⟩

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

definition *zero-bit-def*:
0 = *Bit False*

definition *one-bit-def*:
1 = *Bit True*

instance ⟨*proof*⟩

end

old-rep-datatype *0*::*bit* *1*::*bit*
⟨*proof*⟩

lemma *Bit-set-eq* [*simp*]:
Bit (*set b*) = *b*
⟨*proof*⟩

lemma *set-Bit-eq* [*simp*]:
set (*Bit P*) = *P*
⟨*proof*⟩

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

lemma *Bit-inject* [*simp*]:
$Bit\ P = Bit\ Q \longleftrightarrow (P \longleftrightarrow Q)$
⟨*proof*⟩

lemma *set* [*iff*]:
¬ *set 0*
set 1
⟨*proof*⟩

lemma [*code*]:
$set\ 0 \longleftrightarrow False$
$set\ 1 \longleftrightarrow True$
⟨*proof*⟩

lemma *set-iff*:
set b $\longleftrightarrow$ *b* = *1*
⟨*proof*⟩

lemma *bit-eq-iff-set*:
b = *0* $\longleftrightarrow$ ¬ *set b*
b = *1* $\longleftrightarrow$ *set b*
⟨*proof*⟩

lemma *Bit* [*simp*, *code*]:
Bit False = *0*
Bit True = *1*
⟨*proof*⟩

lemma *bit-not-0-iff* [*iff*]:
(*x*::*bit*) ≠ *0* $\longleftrightarrow$ *x* = *1*
⟨*proof*⟩

lemma *bit-not-1-iff* [*iff*]:
(*x*::*bit*) ≠ *1* $\longleftrightarrow$ *x* = *0*
⟨*proof*⟩

lemma [*code*]:
HOL.equal 0 b $\longleftrightarrow$ ¬ *set b*
HOL.equal 1 b $\longleftrightarrow$ *set b*
⟨*proof*⟩

7.2 Type *bit* forms a field

instantiation *bit* :: *field*
begin

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

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

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

definition *minus-bit-def* [*simp*]:
x − *y* = (*x* + *y* :: *bit*)

definition *inverse-bit-def* [*simp*]:
inverse x = (*x* :: *bit*)

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

$x\ div\ y = (x * y :: bit)$

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

instance
⟨*proof*⟩

end

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

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

Not sure whether the next two should be simp rules.

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

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

7.3 Numerals at type *bit*

All numerals reduce to either 0 or 1.

lemma *bit-minus1* [*simp*]: $-\ 1 = (1 :: bit)$
⟨*proof*⟩

lemma *bit-neg-numeral* [*simp*]: $(-\ numeral\ w :: bit) = numeral\ w$
⟨*proof*⟩

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

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

7.4 Conversion from *bit*

context *zero-neq-one*
begin

definition *of-bit* :: $bit \Rightarrow 'a$
where
of-bit b = case-bit 0 1 b

lemma *of-bit-eq* [*simp*, *code*]:

of-bit 0 = 0
of-bit 1 = 1
⟨*proof*⟩

lemma *of-bit-eq-iff*:
of-bit x = of-bit y ⟷ *x = y*
⟨*proof*⟩

end

context *semiring-1*
begin

lemma *of-nat-of-bit-eq*:
of-nat (*of-bit b*) = *of-bit b*
⟨*proof*⟩

end

context *ring-1*
begin

lemma *of-int-of-bit-eq*:
of-int (*of-bit b*) = *of-bit b*
⟨*proof*⟩

end

hide-const (**open**) *set*

end

8 Bit operations in $\mathcal{Z}_\in$

theory *Bits-Bit*
imports *Bits ~~/src/HOL/Library/Bit*
begin

instantiation *bit* :: *bit*
begin

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

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

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

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

instance ⟨*proof*⟩

end

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

lemma *bit-extra-simps* [*simp*]:
x AND 0 = (0::bit)
x AND 1 = (x::bit)
x OR 1 = (1::bit)
x OR 0 = (x::bit)
x XOR 1 = NOT (x::bit)
x XOR 0 = (x::bit)
⟨*proof*⟩

lemma *bit-ops-comm*:
(x::bit) AND y = y AND x
(x::bit) OR y = y OR x
(x::bit) XOR y = y XOR x
⟨*proof*⟩

lemma *bit-ops-same* [*simp*]:
(x::bit) AND x = x
(x::bit) OR x = x
(x::bit) XOR x = 0
⟨*proof*⟩

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

lemma *bit-or-def*: *(b::bit) OR c = NOT (NOT b AND NOT c)*
⟨*proof*⟩

lemma *bit-xor-def*: *(b::bit) XOR c = (b AND NOT c) OR (NOT b AND c)*
⟨*proof*⟩

lemma *bit-NOT-eq-1-iff* [*simp*]: *NOT (b::bit) = 1 ⟷ b = 0*
⟨*proof*⟩

lemma *bit-AND-eq-1-iff* [*simp*]: *(a::bit) AND b = 1 ⟷ a = 1 ∧ b = 1*

⟨*proof*⟩

end

9 Useful Numerical Lemmas

theory *Misc-Numeric*
imports *Main*
begin

lemma *mod-2-neq-1-eq-eq-0*:
fixes $k :: int$
shows $k \ mod \ 2 \neq 1 \longleftrightarrow k \ mod \ 2 = 0$
⟨*proof*⟩

lemma *z1pmod2*:
fixes $b :: int$
shows $(2 * b + 1) \ mod \ 2 = (1::int)$
⟨*proof*⟩

lemma *diff-le-eq'*:
$a - b \leq c \longleftrightarrow a \leq b + (c::int)$
⟨*proof*⟩

lemma *emep1*:
fixes $n \ d :: int$
shows $even \ n \Longrightarrow even \ d \Longrightarrow 0 \leq d \Longrightarrow (n + 1) \ mod \ d = (n \ mod \ d) + 1$
⟨*proof*⟩

lemma *int-mod-ge*:
$a < n \Longrightarrow 0 < (n :: int) \Longrightarrow a \leq a \ mod \ n$
⟨*proof*⟩

lemma *int-mod-ge'*:
$b < 0 \Longrightarrow 0 < (n :: int) \Longrightarrow b + n \leq b \ mod \ n$
⟨*proof*⟩

lemma *int-mod-le'*:
$(0::int) \leq b - n \Longrightarrow b \ mod \ n \leq b - n$
⟨*proof*⟩

lemma *zless2*:
$0 < (2 :: int)$
⟨*proof*⟩

lemma *zless2p*:
$0 < (2 \hat{} \ n :: int)$
⟨*proof*⟩

lemma *zle2p*:
$0 \leq (2 \mathbin{\hat{}} n :: int)$
⟨*proof*⟩

lemma *m1mod2k*:
$- 1 \ mod \ 2 \mathbin{\hat{}} n = (2 \mathbin{\hat{}} n - 1 :: int)$
⟨*proof*⟩

lemma *p1mod22k'*:
fixes $b :: int$
shows $(1 + 2 * b) \ mod \ (2 * 2 \mathbin{\hat{}} n) = 1 + 2 * (b \ mod \ 2 \mathbin{\hat{}} n)$
⟨*proof*⟩

lemma *p1mod22k*:
fixes $b :: int$
shows $(2 * b + 1) \ mod \ (2 * 2 \mathbin{\hat{}} n) = 2 * (b \ mod \ 2 \mathbin{\hat{}} n) + 1$
⟨*proof*⟩

lemma *int-mod-lem*:
$(0 :: int) < n ==> (0 <= b \ \& \ b < n) = (b \ mod \ n = b)$
⟨*proof*⟩

end

10 Integers as implict bit strings

theory *Bit-Representation*
imports *Misc-Numeric*
begin

10.1 Constructors and destructors for binary integers

definition $Bit :: int \Rightarrow bool \Rightarrow int$ (**infixl** *BIT 90*)
where
$k \ BIT \ b = (if \ b \ then \ 1 \ else \ 0) + k + k$

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

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

lemma *Bit-B0-2t*: $k \ BIT \ False = 2 * k$
⟨*proof*⟩

lemma *Bit-B1-2t*: $k \ BIT \ True = 2 * k + 1$
⟨*proof*⟩

definition *bin-last* :: *int* ⇒ *bool*
where
bin-last w ⟷ *w mod 2* = *1*

lemma *bin-last-odd*:
bin-last = *odd*
⟨*proof*⟩

definition *bin-rest* :: *int* ⇒ *int*
where
bin-rest w = *w div 2*

lemma *bin-rl-simp* [*simp*]:
bin-rest w BIT bin-last w = *w*
⟨*proof*⟩

lemma *bin-rest-BIT* [*simp*]: *bin-rest* (*x BIT b*) = *x*
⟨*proof*⟩

lemma *bin-last-BIT* [*simp*]: *bin-last* (*x BIT b*) = *b*
⟨*proof*⟩

lemma *BIT-eq-iff* [*iff*]: *u BIT b* = *v BIT c* ⟷ *u* = *v* ∧ *b* = *c*
⟨*proof*⟩

lemma *BIT-bin-simps* [*simp*]:
numeral k BIT False = *numeral* (*Num.Bit0 k*)
numeral k BIT True = *numeral* (*Num.Bit1 k*)
(− *numeral k*) *BIT False* = − *numeral* (*Num.Bit0 k*)
(− *numeral k*) *BIT True* = − *numeral* (*Num.BitM k*)
⟨*proof*⟩

lemma *BIT-special-simps* [*simp*]:
shows *0 BIT False* = *0* **and** *0 BIT True* = *1*
and *1 BIT False* = *2* **and** *1 BIT True* = *3*
and (− *1*) *BIT False* = − *2* **and** (− *1*) *BIT True* = − *1*
⟨*proof*⟩

lemma *Bit-eq-0-iff*: *w BIT b* = *0* ⟷ *w* = *0* ∧ ¬ *b*
⟨*proof*⟩

lemma *Bit-eq-m1-iff*: *w BIT b* = −*1* ⟷ *w* = −*1* ∧ *b*
⟨*proof*⟩

lemma *BitM-inc*: *Num.BitM* (*Num.inc w*) = *Num.Bit1 w*
⟨*proof*⟩

lemma *expand-BIT*:

numeral (Num.Bit0 w) = numeral w BIT False
numeral (Num.Bit1 w) = numeral w BIT True
− numeral (Num.Bit0 w) = (− numeral w) BIT False
− numeral (Num.Bit1 w) = (− numeral (w + Num.One)) BIT True
⟨*proof*⟩

lemma *bin-last-numeral-simps* [*simp*]:
¬ bin-last 0
bin-last 1
bin-last (− 1)
bin-last Numeral1
¬ bin-last (numeral (Num.Bit0 w))
bin-last (numeral (Num.Bit1 w))
¬ bin-last (− numeral (Num.Bit0 w))
bin-last (− numeral (Num.Bit1 w))
⟨*proof*⟩

lemma *bin-rest-numeral-simps* [*simp*]:
bin-rest 0 = 0
bin-rest 1 = 0
bin-rest (− 1) = − 1
bin-rest Numeral1 = 0
bin-rest (numeral (Num.Bit0 w)) = numeral w
bin-rest (numeral (Num.Bit1 w)) = numeral w
bin-rest (− numeral (Num.Bit0 w)) = − numeral w
bin-rest (− numeral (Num.Bit1 w)) = − numeral (w + Num.One)
⟨*proof*⟩

lemma *less-Bits*:
v BIT b < w BIT c ⟷ v < w ∨ v ≤ w ∧ ¬ b ∧ c
⟨*proof*⟩

lemma *le-Bits*:
v BIT b ≤ w BIT c ⟷ v < w ∨ v ≤ w ∧ (¬ b ∨ c)
⟨*proof*⟩

lemma *pred-BIT-simps* [*simp*]:
x BIT False − 1 = (x − 1) BIT True
x BIT True − 1 = x BIT False
⟨*proof*⟩

lemma *succ-BIT-simps* [*simp*]:
x BIT False + 1 = x BIT True
x BIT True + 1 = (x + 1) BIT False
⟨*proof*⟩

lemma *add-BIT-simps* [*simp*]:
x BIT False + y BIT False = (x + y) BIT False
x BIT False + y BIT True = (x + y) BIT True

x BIT True + y BIT False = (x + y) BIT True
x BIT True + y BIT True = (x + y + 1) BIT False
⟨*proof*⟩

lemma *mult-BIT-simps* [*simp*]:
x BIT False ∗ y = (x ∗ y) BIT False
x ∗ y BIT False = (x ∗ y) BIT False
x BIT True ∗ y = (x ∗ y) BIT False + y
⟨*proof*⟩

lemma *B-mod-2'*:
X = 2 ==> (w BIT True) mod X = 1 & (w BIT False) mod X = 0
⟨*proof*⟩

lemma *bin-ex-rl*: *EX w b. w BIT b = bin*
⟨*proof*⟩

lemma *bin-exhaust*:
assumes *Q*: ⋀*x b. bin = x BIT b* ⟹ *Q*
shows *Q*
⟨*proof*⟩

primrec *bin-nth* **where**
Z: *bin-nth w 0* ⟷ *bin-last w*
| *Suc*: *bin-nth w (Suc n)* ⟷ *bin-nth (bin-rest w) n*

lemma *bin-abs-lem*:
bin = (w BIT b) ==> bin ~= −1 --> bin ~= 0 -->
nat |w| < nat |bin|
⟨*proof*⟩

lemma *bin-induct*:
assumes *PPls*: *P 0*
and *PMin*: *P (− 1)*
and *PBit*: !!*bin bit. P bin ==> P (bin BIT bit)*
shows *P bin*
⟨*proof*⟩

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

lemma *bin-eq-iff*:
x = y ⟷ (∀ *n. bin-nth x n = bin-nth y n*)

⟨*proof*⟩

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

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

lemma *bin-nth-minus1* [*simp*]: *bin-nth* (− *1*) *n*
⟨*proof*⟩

lemma *bin-nth-0-BIT*: *bin-nth* (*w BIT b*) *0* $\longleftrightarrow$ *b*
⟨*proof*⟩

lemma *bin-nth-Suc-BIT*: *bin-nth* (*w BIT b*) (*Suc n*) = *bin-nth w n*
⟨*proof*⟩

lemma *bin-nth-minus* [*simp*]: *0* < *n* ==> *bin-nth* (*w BIT b*) *n* = *bin-nth w* (*n* − *1*)
⟨*proof*⟩

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

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

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

10.2 Truncating binary integers

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

lemma *bin-sign-simps* [*simp*]:
bin-sign 0 = *0*
bin-sign 1 = *0*
bin-sign (− *1*) = − *1*
bin-sign (*numeral k*) = *0*
bin-sign (− *numeral k*) = −*1*
bin-sign (*w BIT b*) = *bin-sign w*

⟨proof⟩

lemma *bin-sign-rest* [*simp*]:
bin-sign (*bin-rest w*) = *bin-sign w*
⟨proof⟩

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

primrec *sbintrunc* :: *nat* => *int* => *int* **where**
Z : *sbintrunc 0 bin* = (*if bin-last bin then* −*1 else 0*)
| *Suc* : *sbintrunc* (*Suc n*) *bin* = *sbintrunc n* (*bin-rest bin*) *BIT* (*bin-last bin*)

lemma *sign-bintr*: *bin-sign* (*bintrunc n w*) = *0*
⟨proof⟩

lemma *bintrunc-mod2p*: *bintrunc n w* = (*w mod 2* ^ *n*)
⟨proof⟩

lemma *sbintrunc-mod2p*: *sbintrunc n w* = (*w* + *2* ^ *n*) *mod 2* ^ (*Suc n*) − *2* ^ *n*
⟨proof⟩

10.3 Simplifications for (s)bintrunc

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

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

lemma *sbintrunc-n-minus1* [*simp*]: *sbintrunc n* (− *1*) = −*1*
⟨proof⟩

lemma *bintrunc-Suc-numeral*:
bintrunc (*Suc n*) *1* = *1*
bintrunc (*Suc n*) (− *1*) = *bintrunc n* (− *1*) *BIT True*
bintrunc (*Suc n*) (*numeral* (*Num.Bit0 w*)) = *bintrunc n* (*numeral w*) *BIT False*
bintrunc (*Suc n*) (*numeral* (*Num.Bit1 w*)) = *bintrunc n* (*numeral w*) *BIT True*
bintrunc (*Suc n*) (− *numeral* (*Num.Bit0 w*)) =
bintrunc n (− *numeral w*) *BIT False*
bintrunc (*Suc n*) (− *numeral* (*Num.Bit1 w*)) =
bintrunc n (− *numeral* (*w* + *Num.One*)) *BIT True*
⟨proof⟩

lemma *sbintrunc-0-numeral* [*simp*]:
sbintrunc 0 1 = −*1*
sbintrunc 0 (*numeral* (*Num.Bit0 w*)) = *0*
sbintrunc 0 (*numeral* (*Num.Bit1 w*)) = −*1*

sbintrunc 0 (− numeral (Num.Bit0 w)) = 0
sbintrunc 0 (− numeral (Num.Bit1 w)) = −1
⟨proof⟩

lemma *sbintrunc-Suc-numeral*:
sbintrunc (Suc n) 1 = 1
sbintrunc (Suc n) (numeral (Num.Bit0 w)) =
  sbintrunc n (numeral w) BIT False
sbintrunc (Suc n) (numeral (Num.Bit1 w)) =
  sbintrunc n (numeral w) BIT True
sbintrunc (Suc n) (− numeral (Num.Bit0 w)) =
  sbintrunc n (− numeral w) BIT False
sbintrunc (Suc n) (− numeral (Num.Bit1 w)) =
  sbintrunc n (− numeral (w + Num.One)) BIT True
⟨proof⟩

lemma *bin-sign-lem*: (bin-sign (sbintrunc n bin) = −1) = bin-nth bin n
⟨proof⟩

lemma *nth-bintr*: bin-nth (bintrunc m w) n = (n < m & bin-nth w n)
⟨proof⟩

lemma *nth-sbintr*:
bin-nth (sbintrunc m w) n =
    (if n < m then bin-nth w n else bin-nth w m)
⟨proof⟩

lemma *bin-nth-Bit*:
bin-nth (w BIT b) n = (n = 0 & b | (EX m. n = Suc m & bin-nth w m))
⟨proof⟩

lemma *bin-nth-Bit0*:
bin-nth (numeral (Num.Bit0 w)) n ⟷
  (∃ m. n = Suc m ∧ bin-nth (numeral w) m)
⟨proof⟩

lemma *bin-nth-Bit1*:
bin-nth (numeral (Num.Bit1 w)) n ⟷
  n = 0 ∨ (∃ m. n = Suc m ∧ bin-nth (numeral w) m)
⟨proof⟩

lemma *bintrunc-bintrunc-l*:
n <= m ==> (bintrunc m (bintrunc n w) = bintrunc n w)
⟨proof⟩

lemma *sbintrunc-sbintrunc-l*:
n <= m ==> (sbintrunc m (sbintrunc n w) = sbintrunc n w)
⟨proof⟩

lemma *bintrunc-bintrunc-ge*:
n <= *m* ==> (*bintrunc n* (*bintrunc m w*) = *bintrunc n w*)
⟨*proof*⟩

lemma *bintrunc-bintrunc-min* [*simp*]:
bintrunc m (*bintrunc n w*) = *bintrunc* (*min m n*) *w*
⟨*proof*⟩

lemma *sbintrunc-sbintrunc-min* [*simp*]:
sbintrunc m (*sbintrunc n w*) = *sbintrunc* (*min m n*) *w*
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

lemmas *sbintrunc-0-BIT-B0* [*simp*] =
sbintrunc.Z [**where** *bin=w BIT False*,
simplified bin-last-numeral-simps bin-rest-numeral-simps] **for** *w*

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

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

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

lemma *bintrunc-minus*:
0 < *n* ==> *bintrunc* (*Suc* (*n* − *1*)) *w* = *bintrunc n w*
⟨*proof*⟩

lemma *sbintrunc-minus*:
0 < *n* ==> *sbintrunc* (*Suc* (*n* − *1*)) *w* = *sbintrunc n w*
⟨*proof*⟩

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

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

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

lemmas *bmsts* = *bintrunc-minus-simps*(*1*−*3*) [*THEN thobini1* [*THEN* [*2*] *trans*]]
lemmas *bintrunc-Pls-minus-I* = *bmsts*(*1*)
lemmas *bintrunc-Min-minus-I* = *bmsts*(*2*)
lemmas *bintrunc-BIT-minus-I* = *bmsts*(*3*)

lemma *bintrunc-Suc-lem*:
bintrunc (*Suc n*) *x* = *y* ==> *m* = *Suc n* ==> *bintrunc m x* = *y*
⟨*proof*⟩

lemmas *bintrunc-Suc-Ialts* =
bintrunc-Min-I [*THEN bintrunc-Suc-lem*]
bintrunc-BIT-I [*THEN bintrunc-Suc-lem*]

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

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

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

lemma *sbintrunc-Suc-lem*:
sbintrunc (*Suc n*) *x* = *y* ==> *m* = *Suc n* ==> *sbintrunc m x* = *y*
⟨*proof*⟩

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

lemma *sbintrunc-bintrunc-lt*:
m > *n* ==> *sbintrunc n* (*bintrunc m w*) = *sbintrunc n w*
⟨*proof*⟩

lemma *bintrunc-sbintrunc-le*:
m <= *Suc n* ==> *bintrunc m* (*sbintrunc n w*) = *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* ⟹ *bintrunc n* (*sbintrunc* (*n* − *1*) *w*) = *bintrunc n w*
⟨*proof*⟩

lemma *sbintrunc-bintrunc'* [*simp*]:
0 < *n* ⟹ *sbintrunc* (*n* − *1*) (*bintrunc n w*) = *sbintrunc* (*n* − *1*) *w*
⟨*proof*⟩

lemma *bin-sbin-eq-iff*:
bintrunc (*Suc n*) *x* = *bintrunc* (*Suc n*) *y* ⟷
sbintrunc n x = *sbintrunc n y*
⟨*proof*⟩

lemma *bin-sbin-eq-iff'*:
0 < *n* ⟹ *bintrunc n x* = *bintrunc n y* ⟷
sbintrunc (*n* − *1*) *x* = *sbintrunc* (*n* − *1*) *y*
⟨*proof*⟩

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

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

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

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

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

lemma *bintrunc-numeral-simps* [*simp*]:
bintrunc (*numeral* k) (*numeral* (*Num.Bit0* w)) =
bintrunc (*pred-numeral* k) (*numeral* w) *BIT False*
bintrunc (*numeral* k) (*numeral* (*Num.Bit1* w)) =
bintrunc (*pred-numeral* k) (*numeral* w) *BIT True*
bintrunc (*numeral* k) (− *numeral* (*Num.Bit0* w)) =
bintrunc (*pred-numeral* k) (− *numeral* w) *BIT False*
bintrunc (*numeral* k) (− *numeral* (*Num.Bit1* w)) =
bintrunc (*pred-numeral* k) (− *numeral* (w + *Num.One*)) *BIT True*
bintrunc (*numeral* k) *1* = *1*
⟨*proof*⟩

lemma *sbintrunc-numeral-simps* [*simp*]:
sbintrunc (*numeral* k) (*numeral* (*Num.Bit0* w)) =
sbintrunc (*pred-numeral* k) (*numeral* w) *BIT False*
sbintrunc (*numeral* k) (*numeral* (*Num.Bit1* w)) =
sbintrunc (*pred-numeral* k) (*numeral* w) *BIT True*
sbintrunc (*numeral* k) (− *numeral* (*Num.Bit0* w)) =
sbintrunc (*pred-numeral* k) (− *numeral* w) *BIT False*
sbintrunc (*numeral* k) (− *numeral* (*Num.Bit1* w)) =
sbintrunc (*pred-numeral* k) (− *numeral* (w + *Num.One*)) *BIT True*
sbintrunc (*numeral* k) *1* = *1*
⟨*proof*⟩

lemma *no-bintr-alt1*: *bintrunc* n = (λw. w *mod* *2* ˆ n :: *int*)
⟨*proof*⟩

lemma *range-bintrunc*: *range* (*bintrunc* n) = {i. *0* <= i & i < *2* ˆ n}
⟨*proof*⟩

lemma *no-sbintr-alt2*:
sbintrunc n = (%w. (w + *2* ˆ n) *mod* *2* ˆ *Suc* n − *2* ˆ n :: *int*)
⟨*proof*⟩

lemma *range-sbintrunc*:
range (*sbintrunc* n) = {i. − (*2* ˆ n) <= i & i < *2* ˆ n}
⟨*proof*⟩

lemma *sb-inc-lem*:

$(a::int) + 2\hat{}k < 0 \Longrightarrow a + 2\hat{}k + 2\hat{}(Suc\ k) <= (a + 2\hat{}k)\ mod\ 2\hat{}(Suc\ k)$
⟨proof⟩

lemma *sb-inc-lem'*:
$(a::int) < -(2\hat{}k) \Longrightarrow a + 2\hat{}k + 2\hat{}(Suc\ k) <= (a + 2\hat{}k)\ mod\ 2\hat{}(Suc\ k)$
⟨proof⟩

lemma *sbintrunc-inc*:
$x < -(2\hat{}n) ==> x + 2\hat{}(Suc\ n) <= sbintrunc\ n\ x$
⟨proof⟩

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

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

lemma *sbintrunc-dec*:
$x >= (2\ \hat{}\ n) ==> x - 2\ \hat{}\ (Suc\ n) >= sbintrunc\ n\ x$
⟨proof⟩

lemmas *zmod-uminus'* = *zminus-zmod* [**where** *m*=*c*] **for** *c*
lemmas *zpower-zmod'* = *power-mod* [**where** *b*=*c* **and** *n*=*k*] **for** *c k*

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

lemmas *brdmods'* [*symmetric*] =
zpower-zmod' [*symmetric*]
trans [*OF mod-add-left-eq mod-add-right-eq*]
trans [*OF mod-diff-left-eq mod-diff-right-eq*]
trans [*OF mod-mult-right-eq mod-mult-left-eq*]
zmod-uminus' [*symmetric*]
mod-add-left-eq [**where** *b* = *1::int*]
mod-diff-left-eq [**where** *b* = *1::int*]

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

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

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

lemma *bintr-lt2p*: *bintrunc n w* < *2* ^ *n*
⟨*proof*⟩

lemma *bintr-Min*: *bintrunc n* (− *1*) = *2* ^ *n* − *1*
⟨*proof*⟩

lemma *sbintr-ge*: − (*2* ^ *n*) ≤ *sbintrunc n w*
⟨*proof*⟩

lemma *sbintr-lt*: *sbintrunc n w* < *2* ^ *n*
⟨*proof*⟩

lemma *sign-Pls-ge-0*:
(*bin-sign bin* = *0*) = (*bin* >= (*0* :: *int*))
⟨*proof*⟩

lemma *sign-Min-lt-0*:
(*bin-sign bin* = −*1*) = (*bin* < (*0* :: *int*))
⟨*proof*⟩

lemma *bin-rest-trunc*:
(*bin-rest* (*bintrunc n bin*)) = *bintrunc* (*n* − *1*) (*bin-rest bin*)
⟨*proof*⟩

lemma *bin-rest-power-trunc*:
(*bin-rest* ^^ *k*) (*bintrunc n bin*) =
bintrunc (*n* − *k*) ((*bin-rest* ^^ *k*) *bin*)
⟨*proof*⟩

lemma *bin-rest-trunc-i*:
bintrunc n (*bin-rest bin*) = *bin-rest* (*bintrunc* (*Suc n*) *bin*)
⟨*proof*⟩

lemma *bin-rest-strunc*:
bin-rest (*sbintrunc* (*Suc n*) *bin*) = *sbintrunc n* (*bin-rest bin*)
⟨*proof*⟩

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

lemma *sbintrunc-rest* [*simp*]:
sbintrunc n (*bin-rest* (*sbintrunc n bin*)) = *bin-rest* (*sbintrunc n bin*)
⟨*proof*⟩

lemma *bintrunc-rest'*:
bintrunc n o bin-rest o bintrunc n = *bin-rest o bintrunc n*
⟨*proof*⟩

lemma *sbintrunc-rest'* :
sbintrunc n o bin-rest o sbintrunc n = bin-rest o sbintrunc n
⟨*proof*⟩

lemma *rco-lem*:
f o g o f = g o f ==> f o (g o f) ^^ n = g ^^ n o f
⟨*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*]

10.4 Splitting and concatenation

primrec *bin-split* :: *nat* ⇒ *int* ⇒ *int* × *int* **where**
Z: *bin-split 0 w = (w, 0)*
| *Suc*: *bin-split (Suc n) w = (let (w1, w2) = bin-split n (bin-rest w)*
in (w1, w2 BIT bin-last w))

lemma [*code*]:
bin-split (Suc n) w = (let (w1, w2) = bin-split n (bin-rest w) in (w1, w2 BIT bin-last w))
bin-split 0 w = (w, 0)
⟨*proof*⟩

primrec *bin-cat* :: *int* ⇒ *nat* ⇒ *int* ⇒ *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*

end

11 Bitwise Operations on Binary Integers

theory *Bits-Int*
imports *Bits Bit-Representation*
begin

11.1 Logical operations

bit-wise logical operations on the int type

instantiation *int* :: *bit*
begin

definition *int-not-def*:
bitNOT = (λ*x*::*int*. − *x* − *1*)

function *bitAND-int* **where**
  bitAND-int x y =
    (if x = 0 then 0 else if x = −1 then y else
      (bin-rest x AND bin-rest y) BIT (bin-last x ∧ bin-last y))
  ⟨proof⟩

termination
  ⟨proof⟩

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

definition *int-or-def*:
  bitOR = (λx y::int. NOT (NOT x AND NOT y))

definition *int-xor-def*:
  bitXOR = (λx y::int. (x AND NOT y) OR (NOT x AND y))

instance *⟨proof⟩*

end

11.1.1 Basic simplification rules

lemma *int-not-BIT* [*simp*]:
  NOT (w BIT b) = (NOT w) BIT (¬ b)
  ⟨proof⟩

lemma *int-not-simps* [*simp*]:
  NOT (0::int) = −1
  NOT (1::int) = −2
  NOT (− 1::int) = 0
  NOT (numeral w::int) = − numeral (w + Num.One)
  NOT (− numeral (Num.Bit0 w)::int) = numeral (Num.BitM w)
  NOT (− numeral (Num.Bit1 w)::int) = numeral (Num.Bit0 w)
  ⟨proof⟩

lemma *int-not-not* [*simp*]: *NOT (NOT (x::int)) = x*
  ⟨proof⟩

lemma *int-and-0* [*simp*]: *(0::int) AND x = 0*
  ⟨proof⟩

lemma *int-and-m1* [*simp*]: *(−1::int) AND x = x*
  ⟨proof⟩

lemma *int-and-Bits* [*simp*]:
  (x BIT b) AND (y BIT c) = (x AND y) BIT (b ∧ c)
  ⟨proof⟩

lemma *int-or-zero* [*simp*]: $(0::int)\ OR\ x = x$
⟨*proof*⟩

lemma *int-or-minus1* [*simp*]: $(-1::int)\ OR\ x = -1$
⟨*proof*⟩

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

lemma *int-xor-zero* [*simp*]: $(0::int)\ XOR\ x = x$
⟨*proof*⟩

lemma *int-xor-Bits* [*simp*]:
$(x\ BIT\ b)\ XOR\ (y\ BIT\ c) = (x\ XOR\ y)\ BIT\ ((b \lor c) \land \neg\ (b \land c))$
⟨*proof*⟩

11.1.2 Binary destructors

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

lemma *bin-last-NOT* [*simp*]: $bin\text{-}last\ (NOT\ x) \longleftrightarrow \neg\ bin\text{-}last\ x$
⟨*proof*⟩

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

lemma *bin-last-AND* [*simp*]: $bin\text{-}last\ (x\ AND\ y) \longleftrightarrow bin\text{-}last\ x \land bin\text{-}last\ y$
⟨*proof*⟩

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

lemma *bin-last-OR* [*simp*]: $bin\text{-}last\ (x\ OR\ y) \longleftrightarrow bin\text{-}last\ x \lor bin\text{-}last\ y$
⟨*proof*⟩

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

lemma *bin-last-XOR* [*simp*]: $bin\text{-}last\ (x\ XOR\ y) \longleftrightarrow (bin\text{-}last\ x \lor bin\text{-}last\ y) \land \neg\ (bin\text{-}last\ x \land bin\text{-}last\ y)$
⟨*proof*⟩

lemma *bin-nth-ops*:
$!!x\ y.\ bin\text{-}nth\ (x\ AND\ y)\ n = (bin\text{-}nth\ x\ n\ \&\ bin\text{-}nth\ y\ n)$
$!!x\ y.\ bin\text{-}nth\ (x\ OR\ y)\ n = (bin\text{-}nth\ x\ n\ |\ bin\text{-}nth\ y\ n)$
$!!x\ y.\ bin\text{-}nth\ (x\ XOR\ y)\ n = (bin\text{-}nth\ x\ n\ {}^\sim{=}\ bin\text{-}nth\ y\ n)$
$!!x.\ bin\text{-}nth\ (NOT\ x)\ n = ({}^\sim\ bin\text{-}nth\ x\ n)$

⟨*proof*⟩

11.1.3 Derived properties

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

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

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

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

lemma *bin-ops-comm*:
shows
int-and-comm: $!!y::int.\ x\ AND\ y = y\ AND\ x$ **and**
int-or-comm: $!!y::int.\ x\ OR\ y = y\ OR\ x$ **and**
int-xor-comm: $!!y::int.\ x\ XOR\ y = y\ XOR\ x$
⟨*proof*⟩

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

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

lemma *bbw-ao-absorb*:
$!!y::int.\ x\ AND\ (y\ OR\ x) = x\ \&\ x\ OR\ (y\ AND\ x) = x$
⟨*proof*⟩

lemma *bbw-ao-absorbs-other*:
$x\ AND\ (x\ OR\ y) = x \wedge (y\ AND\ x)\ OR\ x = (x::int)$
$(y\ OR\ x)\ AND\ x = x \wedge x\ OR\ (x\ AND\ y) = (x::int)$

$(x\ OR\ y)\ AND\ x = x \wedge (x\ AND\ y)\ OR\ x = (x::int)$
⟨proof⟩

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

lemma *int-xor-not*:
$!!y::int.\ (NOT\ x)\ XOR\ y = NOT\ (x\ XOR\ y)$ &
$x\ XOR\ (NOT\ y) = NOT\ (x\ XOR\ y)$
⟨proof⟩

lemma *int-and-assoc*:
$(x\ AND\ y)\ AND\ (z::int) = x\ AND\ (y\ AND\ z)$
⟨proof⟩

lemma *int-or-assoc*:
$(x\ OR\ y)\ OR\ (z::int) = x\ OR\ (y\ OR\ z)$
⟨proof⟩

lemma *int-xor-assoc*:
$(x\ XOR\ y)\ XOR\ (z::int) = x\ XOR\ (y\ XOR\ z)$
⟨proof⟩

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

lemma *bbw-lcs* [*simp*]:
$(y::int)\ AND\ (x\ AND\ z) = x\ AND\ (y\ AND\ z)$
$(y::int)\ OR\ (x\ OR\ z) = x\ OR\ (y\ OR\ z)$
$(y::int)\ XOR\ (x\ XOR\ z) = x\ XOR\ (y\ XOR\ z)$
⟨proof⟩

lemma *bbw-not-dist*:
$!!y::int.\ NOT\ (x\ OR\ y) = (NOT\ x)\ AND\ (NOT\ y)$
$!!y::int.\ NOT\ (x\ AND\ y) = (NOT\ x)\ OR\ (NOT\ y)$
⟨proof⟩

lemma *bbw-oa-dist*:
$!!y\ z::int.\ (x\ AND\ y)\ OR\ z =$
$(x\ OR\ z)\ AND\ (y\ OR\ z)$
⟨proof⟩

lemma *bbw-ao-dist*:
$!!y\ z::int.\ (x\ OR\ y)\ AND\ z =$
$(x\ AND\ z)\ OR\ (y\ AND\ z)$
⟨proof⟩

11.1.4 Simplification with numerals

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

lemma *bin-rl-eqI*: ⟦*bin-rest x* = *bin-rest y*; *bin-last x* = *bin-last y*⟧ ⟹ *x* = *y*
⟨*proof*⟩

lemma *bin-rest-neg-numeral-BitM* [*simp*]:
bin-rest (− *numeral* (*Num.BitM w*)) = − *numeral w*
⟨*proof*⟩

lemma *bin-last-neg-numeral-BitM* [*simp*]:
bin-last (− *numeral* (*Num.BitM w*))
⟨*proof*⟩

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

lemma *int-and-numerals* [*simp*]:
numeral (*Num.Bit0 x*) *AND numeral* (*Num.Bit0 y*) = (*numeral x AND numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *AND numeral* (*Num.Bit1 y*) = (*numeral x AND numeral y*) *BIT False*
numeral (*Num.Bit1 x*) *AND numeral* (*Num.Bit0 y*) = (*numeral x AND numeral y*) *BIT False*
numeral (*Num.Bit1 x*) *AND numeral* (*Num.Bit1 y*) = (*numeral x AND numeral y*) *BIT True*
numeral (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit0 y*) = (*numeral x AND* − *numeral y*) *BIT False*
numeral (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit1 y*) = (*numeral x AND* − *numeral* (*y* + *Num.One*)) *BIT False*
numeral (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit0 y*) = (*numeral x AND* − *numeral y*) *BIT False*
numeral (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit1 y*) = (*numeral x AND* − *numeral* (*y* + *Num.One*)) *BIT True*
− *numeral* (*Num.Bit0 x*) *AND numeral* (*Num.Bit0 y*) = (− *numeral x AND numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *AND numeral* (*Num.Bit1 y*) = (− *numeral x AND numeral y*) *BIT False*
− *numeral* (*Num.Bit1 x*) *AND numeral* (*Num.Bit0 y*) = (− *numeral* (*x* + *Num.One*) *AND numeral y*) *BIT False*
− *numeral* (*Num.Bit1 x*) *AND numeral* (*Num.Bit1 y*) = (− *numeral* (*x* + *Num.One*) *AND numeral y*) *BIT True*
− *numeral* (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit0 y*) = (− *numeral x AND* − *numeral y*) *BIT False*
− *numeral* (*Num.Bit0 x*) *AND* − *numeral* (*Num.Bit1 y*) = (− *numeral x AND* − *numeral* (*y* + *Num.One*)) *BIT False*
− *numeral* (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit0 y*) = (− *numeral* (*x* + *Num.One*) *AND* − *numeral y*) *BIT False*
− *numeral* (*Num.Bit1 x*) *AND* − *numeral* (*Num.Bit1 y*) = (− *numeral* (*x* + *Num.One*) *AND* − *numeral* (*y* + *Num.One*)) *BIT True*
(*1*::*int*) *AND numeral* (*Num.Bit0 y*) = *0*
(*1*::*int*) *AND numeral* (*Num.Bit1 y*) = *1*
(*1*::*int*) *AND* − *numeral* (*Num.Bit0 y*) = *0*

(1::int) AND − numeral (Num.Bit1 y) = 1
numeral (Num.Bit0 x) AND (1::int) = 0
numeral (Num.Bit1 x) AND (1::int) = 1
− numeral (Num.Bit0 x) AND (1::int) = 0
− numeral (Num.Bit1 x) AND (1::int) = 1
⟨*proof*⟩

lemma *int-or-numerals* [*simp*]:
numeral (Num.Bit0 x) OR numeral (Num.Bit0 y) = (numeral x OR numeral y) BIT False
numeral (Num.Bit0 x) OR numeral (Num.Bit1 y) = (numeral x OR numeral y) BIT True
numeral (Num.Bit1 x) OR numeral (Num.Bit0 y) = (numeral x OR numeral y) BIT True
numeral (Num.Bit1 x) OR numeral (Num.Bit1 y) = (numeral x OR numeral y) BIT True
numeral (Num.Bit0 x) OR − numeral (Num.Bit0 y) = (numeral x OR − numeral y) BIT False
numeral (Num.Bit0 x) OR − numeral (Num.Bit1 y) = (numeral x OR − numeral (y + Num.One)) BIT True
numeral (Num.Bit1 x) OR − numeral (Num.Bit0 y) = (numeral x OR − numeral y) BIT True
numeral (Num.Bit1 x) OR − numeral (Num.Bit1 y) = (numeral x OR − numeral (y + Num.One)) BIT True
− numeral (Num.Bit0 x) OR numeral (Num.Bit0 y) = (− numeral x OR numeral y) BIT False
− numeral (Num.Bit0 x) OR numeral (Num.Bit1 y) = (− numeral x OR numeral y) BIT True
− numeral (Num.Bit1 x) OR numeral (Num.Bit0 y) = (− numeral (x + Num.One) OR numeral y) BIT True
− numeral (Num.Bit1 x) OR numeral (Num.Bit1 y) = (− numeral (x + Num.One) OR numeral y) BIT True
− numeral (Num.Bit0 x) OR − numeral (Num.Bit0 y) = (− numeral x OR − numeral y) BIT False
− numeral (Num.Bit0 x) OR − numeral (Num.Bit1 y) = (− numeral x OR − numeral (y + Num.One)) BIT True
− numeral (Num.Bit1 x) OR − numeral (Num.Bit0 y) = (− numeral (x + Num.One) OR − numeral y) BIT True
− numeral (Num.Bit1 x) OR − numeral (Num.Bit1 y) = (− numeral (x + Num.One) OR − numeral (y + Num.One)) BIT True
(1::int) OR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)
(1::int) OR numeral (Num.Bit1 y) = numeral (Num.Bit1 y)
(1::int) OR − numeral (Num.Bit0 y) = − numeral (Num.BitM y)
(1::int) OR − numeral (Num.Bit1 y) = − numeral (Num.Bit1 y)
numeral (Num.Bit0 x) OR (1::int) = numeral (Num.Bit1 x)
numeral (Num.Bit1 x) OR (1::int) = numeral (Num.Bit1 x)
− numeral (Num.Bit0 x) OR (1::int) = − numeral (Num.BitM x)
− numeral (Num.Bit1 x) OR (1::int) = − numeral (Num.Bit1 x)
⟨*proof*⟩

lemma *int-xor-numerals* [*simp*]:
numeral (Num.Bit0 x) XOR numeral (Num.Bit0 y) = (numeral x XOR numeral y) BIT False
numeral (Num.Bit0 x) XOR numeral (Num.Bit1 y) = (numeral x XOR numeral y) BIT True
numeral (Num.Bit1 x) XOR numeral (Num.Bit0 y) = (numeral x XOR numeral y) BIT True
numeral (Num.Bit1 x) XOR numeral (Num.Bit1 y) = (numeral x XOR numeral y) BIT False
numeral (Num.Bit0 x) XOR − numeral (Num.Bit0 y) = (numeral x XOR − numeral y) BIT False
numeral (Num.Bit0 x) XOR − numeral (Num.Bit1 y) = (numeral x XOR − numeral (y + Num.One)) BIT True
numeral (Num.Bit1 x) XOR − numeral (Num.Bit0 y) = (numeral x XOR − numeral y) BIT True
numeral (Num.Bit1 x) XOR − numeral (Num.Bit1 y) = (numeral x XOR − numeral (y + Num.One)) BIT False
− numeral (Num.Bit0 x) XOR numeral (Num.Bit0 y) = (− numeral x XOR numeral y) BIT False
− numeral (Num.Bit0 x) XOR numeral (Num.Bit1 y) = (− numeral x XOR numeral y) BIT True
− numeral (Num.Bit1 x) XOR numeral (Num.Bit0 y) = (− numeral (x + Num.One) XOR numeral y) BIT True
− numeral (Num.Bit1 x) XOR numeral (Num.Bit1 y) = (− numeral (x + Num.One) XOR numeral y) BIT False
− numeral (Num.Bit0 x) XOR − numeral (Num.Bit0 y) = (− numeral x XOR − numeral y) BIT False
− numeral (Num.Bit0 x) XOR − numeral (Num.Bit1 y) = (− numeral x XOR − numeral (y + Num.One)) BIT True
− numeral (Num.Bit1 x) XOR − numeral (Num.Bit0 y) = (− numeral (x + Num.One) XOR − numeral y) BIT True
− numeral (Num.Bit1 x) XOR − numeral (Num.Bit1 y) = (− numeral (x + Num.One) XOR − numeral (y + Num.One)) BIT False
(1::int) XOR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)
(1::int) XOR numeral (Num.Bit1 y) = numeral (Num.Bit0 y)
(1::int) XOR − numeral (Num.Bit0 y) = − numeral (Num.BitM y)
(1::int) XOR − numeral (Num.Bit1 y) = − numeral (Num.Bit0 (y + Num.One))
numeral (Num.Bit0 x) XOR (1::int) = numeral (Num.Bit1 x)
numeral (Num.Bit1 x) XOR (1::int) = numeral (Num.Bit0 x)
− numeral (Num.Bit0 x) XOR (1::int) = − numeral (Num.BitM x)
− numeral (Num.Bit1 x) XOR (1::int) = − numeral (Num.Bit0 (x + Num.One))
⟨*proof*⟩

11.1.5 Interactions with arithmetic

lemma *plus-and-or* [*rule-format*]:
ALL y::int. (x AND y) + (x OR y) = x + y
⟨*proof*⟩

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

lemmas *int-and-le* =
xtrans(*3*) [*OF* *bbw-ao-absorbs* (*2*) [*THEN* *conjunct2*, *symmetric*] *le-int-or*]

lemma *bin-add-not*: *x* + *NOT* *x* = (−*1*::*int*)
⟨*proof*⟩

11.1.6 Truncating results of bit-wise operations

lemma *bin-trunc-ao*:
!!*x* *y*. (*bintrunc* *n* *x*) *AND* (*bintrunc* *n* *y*) = *bintrunc* *n* (*x* *AND* *y*)
!!*x* *y*. (*bintrunc* *n* *x*) *OR* (*bintrunc* *n* *y*) = *bintrunc* *n* (*x* *OR* *y*)
⟨*proof*⟩

lemma *bin-trunc-xor*:
!!*x* *y*. *bintrunc* *n* (*bintrunc* *n* *x* *XOR* *bintrunc* *n* *y*) =
bintrunc *n* (*x* *XOR* *y*)
⟨*proof*⟩

lemma *bin-trunc-not*:
!!*x*. *bintrunc* *n* (*NOT* (*bintrunc* *n* *x*)) = *bintrunc* *n* (*NOT* *x*)
⟨*proof*⟩

lemma *bintr-bintr-i*:
x = *bintrunc* *n* *y* ==> *bintrunc* *n* *x* = *bintrunc* *n* *y*
⟨*proof*⟩

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

11.2 Setting and clearing bits

primrec
bin-sc :: *nat* => *bool* => *int* => *int*
where
Z: *bin-sc* *0* *b* *w* = *bin-rest* *w* *BIT* *b*
| *Suc*: *bin-sc* (*Suc* *n*) *b* *w* = *bin-sc* *n* *b* (*bin-rest* *w*) *BIT* *bin-last* *w*

lemma *bin-nth-sc* [*simp*]:
bin-nth (*bin-sc* *n* *b* *w*) *n* ⟷ *b*
⟨*proof*⟩

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

bin-sc n c (bin-sc n b w) = bin-sc n c w
⟨*proof*⟩

lemma *bin-sc-sc-diff*:
m ~= n ==>
bin-sc m c (bin-sc n b w) = bin-sc n b (bin-sc m c w)
⟨*proof*⟩

lemma *bin-nth-sc-gen*:
bin-nth (bin-sc n b w) m = (if m = n then b else bin-nth w m)
⟨*proof*⟩

lemma *bin-sc-nth* [*simp*]:
(bin-sc n (bin-nth w n) w) = w
⟨*proof*⟩

lemma *bin-sign-sc* [*simp*]:
bin-sign (bin-sc n b w) = bin-sign w
⟨*proof*⟩

lemma *bin-sc-bintr* [*simp*]:
bintrunc m (bin-sc n x (bintrunc m (w))) = bintrunc m (bin-sc n x w)
⟨*proof*⟩

lemma *bin-clr-le*:
bin-sc n False w <= w
⟨*proof*⟩

lemma *bin-set-ge*:
bin-sc n True w >= w
⟨*proof*⟩

lemma *bintr-bin-clr-le*:
bintrunc n (bin-sc m False w) <= bintrunc n w
⟨*proof*⟩

lemma *bintr-bin-set-ge*:
bintrunc n (bin-sc m True w) >= bintrunc n w
⟨*proof*⟩

lemma *bin-sc-FP* [*simp*]: *bin-sc n False 0 = 0*
⟨*proof*⟩

lemma *bin-sc-TM* [*simp*]: *bin-sc n True (− 1) = − 1*
⟨*proof*⟩

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

lemma *bin-sc-minus*:

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

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

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

11.3 Splitting and concatenation

definition *bin-rcat :: nat ⇒ int list ⇒ int*
where
bin-rcat n = foldl (λu v. bin-cat u n v) 0

fun *bin-rsplit-aux :: nat ⇒ nat ⇒ int ⇒ int list ⇒ int list*
where
bin-rsplit-aux n m c bs =
(if m = 0 | n = 0 then bs else
let (a, b) = bin-split n c
in bin-rsplit-aux n (m − n) a (b # bs))

definition *bin-rsplit :: nat ⇒ nat × int ⇒ int list*
where
bin-rsplit n w = bin-rsplit-aux n (fst w) (snd w) []

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

definition *bin-rsplitl :: nat ⇒ nat × int ⇒ int list*
where
bin-rsplitl n w = bin-rsplitl-aux n (fst w) (snd w) []

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

lemma *bin-sign-cat:*
bin-sign (bin-cat x n y) = bin-sign x
⟨*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*:
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*:
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*:
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*:
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-zero* [*simp*]: *bin-cat 0 n w* = *bintrunc n w*
⟨*proof*⟩

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

lemma *bintr-cat*: *bintrunc m* (*bin-cat a n b*) =
bin-cat (*bintrunc* (*m* − *n*) *a*) *n* (*bintrunc* (*min m n*) *b*)
⟨*proof*⟩

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

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

lemma *split-bintrunc*:
bin-split n c = (*a*, *b*) ==> *b* = *bintrunc n c*
⟨*proof*⟩

lemma *bin-cat-split*:
bin-split n w = (*u*, *v*) ==> *w* = *bin-cat u n v*
⟨*proof*⟩

lemma *bin-split-cat*:
bin-split n (*bin-cat v n w*) = (*v*, *bintrunc n w*)

⟨*proof*⟩

lemma *bin-split-zero* [*simp*]: *bin-split n 0 = (0, 0)*
⟨*proof*⟩

lemma *bin-split-minus1* [*simp*]:
bin-split n (− 1) = (− 1, bintrunc n (− 1))
⟨*proof*⟩

lemma *bin-split-trunc*:
bin-split (min m n) c = (a, b) ==>
bin-split n (bintrunc m c) = (bintrunc (m − n) a, b)
⟨*proof*⟩

lemma *bin-split-trunc1*:
bin-split n c = (a, b) ==>
bin-split n (bintrunc m c) = (bintrunc (m − n) a, bintrunc m b)
⟨*proof*⟩

lemma *bin-cat-num*:
bin-cat a n b = a ∗ 2 ˆ n + bintrunc n b
⟨*proof*⟩

lemma *bin-split-num*:
bin-split n b = (b div 2 ˆ n, b mod 2 ˆ n)
⟨*proof*⟩

11.4 Miscellaneous lemmas

lemma *nth-2p-bin*:
bin-nth (2 ˆ n) m = (m = n)
⟨*proof*⟩

lemma *ex-eq-or*:
(EX m. n = Suc m & (m = k | P m)) = (n = Suc k | (EX m. n = Suc m & P m))
⟨*proof*⟩

lemma *power-BIT*: *2 ˆ (Suc n) − 1 = (2 ˆ n − 1) BIT True*
⟨*proof*⟩

lemma *mod-BIT*:
bin BIT bit mod 2 ˆ Suc n = (bin mod 2 ˆ n) BIT bit
⟨*proof*⟩

lemma *AND-mod*:
fixes *x* :: *int*

shows *x AND 2 ^ n − 1 = x mod 2 ^ n*
⟨*proof*⟩

end

12 Bool lists and integers

theory *Bool-List-Representation*
imports *Main Bits-Int*
begin

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

lemma *map2-Nil* [*simp*, *code*]:
map2 f [] *ys* = []
⟨*proof*⟩

lemma *map2-Nil2* [*simp*, *code*]:
map2 f xs [] = []
⟨*proof*⟩

lemma *map2-Cons* [*simp*, *code*]:
map2 f (*x* # *xs*) (*y* # *ys*) = *f x y* # *map2 f xs ys*
⟨*proof*⟩

12.1 Operations on lists of booleans

primrec *bl-to-bin-aux* :: *bool list* ⇒ *int* ⇒ *int*
where
Nil: *bl-to-bin-aux* [] *w* = *w*
| *Cons*: *bl-to-bin-aux* (*b* # *bs*) *w* =
bl-to-bin-aux bs (*w BIT b*)

definition *bl-to-bin* :: *bool list* ⇒ *int*
where
bl-to-bin-def: *bl-to-bin bs* = *bl-to-bin-aux bs 0*

primrec *bin-to-bl-aux* :: *nat* ⇒ *int* ⇒ *bool list* ⇒ *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*) # *bl*)

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

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

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

12.2 Arithmetic in terms of bool lists

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

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

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

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

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

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

lemma *bin-to-bl-aux-zero-minus-simp* [*simp*]:
0 < *n* ⟹ *bin-to-bl-aux n 0 bl* =
bin-to-bl-aux (*n* − *1*) *0* (*False* # *bl*)
⟨*proof*⟩

lemma *bin-to-bl-aux-minus1-minus-simp* [*simp*]:
0 < n ==> bin-to-bl-aux n (− 1) bl =
bin-to-bl-aux (n − 1) (− 1) (True # bl)
⟨*proof*⟩

lemma *bin-to-bl-aux-one-minus-simp* [*simp*]:
0 < n ⟹ bin-to-bl-aux n 1 bl =
bin-to-bl-aux (n − 1) 0 (True # bl)
⟨*proof*⟩

lemma *bin-to-bl-aux-Bit-minus-simp* [*simp*]:
0 < n ==> bin-to-bl-aux n (w BIT b) bl =
bin-to-bl-aux (n − 1) w (b # bl)
⟨*proof*⟩

lemma *bin-to-bl-aux-Bit0-minus-simp* [*simp*]:
0 < n ==> bin-to-bl-aux n (numeral (Num.Bit0 w)) bl =
bin-to-bl-aux (n − 1) (numeral w) (False # bl)
⟨*proof*⟩

lemma *bin-to-bl-aux-Bit1-minus-simp* [*simp*]:
0 < n ==> bin-to-bl-aux n (numeral (Num.Bit1 w)) bl =
bin-to-bl-aux (n − 1) (numeral w) (True # bl)
⟨*proof*⟩

Link between bin and bool list.

lemma *bl-to-bin-aux-append*:
bl-to-bin-aux (bs @ cs) w = bl-to-bin-aux cs (bl-to-bin-aux bs w)
⟨*proof*⟩

lemma *bin-to-bl-aux-append*:
bin-to-bl-aux n w bs @ cs = bin-to-bl-aux n w (bs @ cs)
⟨*proof*⟩

lemma *bl-to-bin-append*:
bl-to-bin (bs @ cs) = bl-to-bin-aux cs (bl-to-bin bs)
⟨*proof*⟩

lemma *bin-to-bl-aux-alt*:
bin-to-bl-aux n w bs = bin-to-bl n w @ bs
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

⟨*proof*⟩

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

lemma *bin-to-bl-bintr*:
bin-to-bl n (*bintrunc m bin*) =
replicate (*n* − *m*) *False* @ *bin-to-bl* (*min n m*) *bin*
⟨*proof*⟩

lemma *bl-to-bin-rep-False*: *bl-to-bin* (*replicate n False*) = *0*
⟨*proof*⟩

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

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

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

lemma *sign-bl-bin*: *bin-sign* (*bl-to-bin bs*) = *0*
⟨*proof*⟩

lemma *bl-sbin-sign-aux*:
hd (*bin-to-bl-aux* (*Suc n*) *w bs*) =
(*bin-sign* (*sbintrunc n w*) = −*1*)
⟨*proof*⟩

lemma *bl-sbin-sign*:
hd (*bin-to-bl* (*Suc n*) *w*) = (*bin-sign* (*sbintrunc n w*) = −*1*)
⟨*proof*⟩

lemma *bin-nth-of-bl-aux*:
bin-nth (*bl-to-bin-aux bl w*) *n* =
(*n* < *size bl* & *rev bl* ! *n* | *n* >= *length bl* & *bin-nth w* (*n* − *size bl*))
⟨*proof*⟩

lemma *bin-nth-of-bl*: *bin-nth* (*bl-to-bin bl*) *n* = (*n* < *length bl* & *rev bl* ! *n*)
⟨*proof*⟩

lemma *bin-nth-bl*: *n* < *m* ⟹ *bin-nth w n* = *nth* (*rev* (*bin-to-bl m w*)) *n*
⟨*proof*⟩

lemma *nth-rev*:
$n < length\ xs \Longrightarrow rev\ xs\ !\ n = xs\ !\ (length\ xs - 1 - n)$
⟨*proof*⟩

lemma *nth-rev-alt*: $n < length\ ys \Longrightarrow ys\ !\ n = rev\ ys\ !\ (length\ ys - Suc\ n)$
⟨*proof*⟩

lemma *nth-bin-to-bl-aux*:
$n < m + length\ bl \Longrightarrow (bin\text{-}to\text{-}bl\text{-}aux\ m\ w\ bl)\ !\ n =$
$(if\ n < m\ then\ bin\text{-}nth\ w\ (m - 1 - n)\ else\ bl\ !\ (n - m))$
⟨*proof*⟩

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

lemma *bl-to-bin-lt2p-aux*:
$bl\text{-}to\text{-}bin\text{-}aux\ bs\ w < (w + 1) * (2\ \hat{}\ length\ bs)$
⟨*proof*⟩

lemma *bl-to-bin-lt2p-drop*:
$bl\text{-}to\text{-}bin\ bs < 2\ \hat{}\ length\ (dropWhile\ Not\ bs)$
⟨*proof*⟩

lemma *bl-to-bin-lt2p*: $bl\text{-}to\text{-}bin\ bs < 2\ \hat{}\ length\ bs$
⟨*proof*⟩

lemma *bl-to-bin-ge2p-aux*:
$bl\text{-}to\text{-}bin\text{-}aux\ bs\ w >= w * (2\ \hat{}\ length\ bs)$
⟨*proof*⟩

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

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

lemma *butlast-bin-rest*:
$butlast\ bl = bin\text{-}to\text{-}bl\ (length\ bl - Suc\ 0)\ (bin\text{-}rest\ (bl\text{-}to\text{-}bin\ bl))$
⟨*proof*⟩

lemma *butlast-rest-bl2bin-aux*:
$bl\ \tilde{}= [] \Longrightarrow$
$bl\text{-}to\text{-}bin\text{-}aux\ (butlast\ bl)\ w = bin\text{-}rest\ (bl\text{-}to\text{-}bin\text{-}aux\ bl\ w)$
⟨*proof*⟩

lemma *butlast-rest-bl2bin*:
$bl\text{-}to\text{-}bin\ (butlast\ bl) = bin\text{-}rest\ (bl\text{-}to\text{-}bin\ bl)$
⟨*proof*⟩

lemma *trunc-bl2bin-aux*:
bintrunc m (bl-to-bin-aux bl w) =
bl-to-bin-aux (drop (length bl − m) bl) (bintrunc (m − length bl) w)
⟨*proof*⟩

lemma *trunc-bl2bin*:
bintrunc m (bl-to-bin bl) = bl-to-bin (drop (length bl − m) bl)
⟨*proof*⟩

lemma *trunc-bl2bin-len* [*simp*]:
bintrunc (length bl) (bl-to-bin bl) = bl-to-bin bl
⟨*proof*⟩

lemma *bl2bin-drop*:
bl-to-bin (drop k bl) = bintrunc (length bl − k) (bl-to-bin bl)
⟨*proof*⟩

lemma *nth-rest-power-bin*:
bin-nth ((bin-rest ^^ k) w) n = bin-nth w (n + k)
⟨*proof*⟩

lemma *take-rest-power-bin*:
m <= n ==> take m (bin-to-bl n w) = bin-to-bl m ((bin-rest ^^ (n − m)) w)
⟨*proof*⟩

lemma *hd-butlast*: *size xs > 1 ==> hd (butlast xs) = hd xs*
⟨*proof*⟩

lemma *last-bin-last'*:
size xs > 0 ⟹ last xs ⟷ bin-last (bl-to-bin-aux xs w)
⟨*proof*⟩

lemma *last-bin-last*:
size xs > 0 ==> last xs ⟷ bin-last (bl-to-bin xs)
⟨*proof*⟩

lemma *bin-last-last*:
bin-last w ⟷ last (bin-to-bl (Suc n) w)
⟨*proof*⟩

lemma *bl-xor-aux-bin*:
map2 (%x y. x ~= y) (bin-to-bl-aux n v bs) (bin-to-bl-aux n w cs) =
bin-to-bl-aux n (v XOR w) (map2 (%x y. x ~= y) bs cs)
⟨*proof*⟩

lemma *bl-or-aux-bin*:

map2 (op |) (bin-to-bl-aux n v bs) (bin-to-bl-aux n w cs) =
bin-to-bl-aux n (v OR w) (map2 (op |) bs cs)
⟨*proof*⟩

lemma *bl-and-aux-bin*:
map2 (op &) (bin-to-bl-aux n v bs) (bin-to-bl-aux n w cs) =
bin-to-bl-aux n (v AND w) (map2 (op &) bs cs)
⟨*proof*⟩

lemma *bl-not-aux-bin*:
map Not (bin-to-bl-aux n w cs) =
bin-to-bl-aux n (NOT w) (map Not cs)
⟨*proof*⟩

lemma *bl-not-bin*: *map Not (bin-to-bl n w) = bin-to-bl n (NOT w)*
⟨*proof*⟩

lemma *bl-and-bin*:
map2 (op $\wedge$*) (bin-to-bl n v) (bin-to-bl n w) = bin-to-bl n (v AND w)*
⟨*proof*⟩

lemma *bl-or-bin*:
map2 (op $\vee$*) (bin-to-bl n v) (bin-to-bl n w) = bin-to-bl n (v OR w)*
⟨*proof*⟩

lemma *bl-xor-bin*:
map2 ($\lambda x\ y.\ x \neq y$*) (bin-to-bl n v) (bin-to-bl n w) = bin-to-bl n (v XOR w)*
⟨*proof*⟩

lemma *drop-bin2bl-aux*:
drop m (bin-to-bl-aux n bin bs) =
bin-to-bl-aux (n − m) bin (drop (m − n) bs)
⟨*proof*⟩

lemma *drop-bin2bl*: *drop m (bin-to-bl n bin) = bin-to-bl (n − m) bin*
⟨*proof*⟩

lemma *take-bin2bl-lem1*:
take m (bin-to-bl-aux m w bs) = bin-to-bl m w
⟨*proof*⟩

lemma *take-bin2bl-lem*:
take m (bin-to-bl-aux (m + n) w bs) =
take m (bin-to-bl (m + n) w)
⟨*proof*⟩

lemma *bin-split-take*:
bin-split n c = (a, b) $\Longrightarrow$
bin-to-bl m a = take m (bin-to-bl (m + n) c)

⟨*proof*⟩

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

lemma *nth-takefill*: $m < n \Longrightarrow$
$takefill\ fill\ n\ l\ !\ m = (if\ m < length\ l\ then\ l\ !\ m\ else\ fill)$
⟨*proof*⟩

lemma *takefill-alt*:
$takefill\ fill\ n\ l = take\ n\ l\ @\ replicate\ (n - length\ l)\ fill$
⟨*proof*⟩

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

lemma *takefill-le'*:
$n = m + k \Longrightarrow takefill\ x\ m\ (takefill\ x\ n\ l) = takefill\ x\ m\ l$
⟨*proof*⟩

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

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

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

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

lemma *take-takefill* [*simp*]:
$m \leq n \Longrightarrow take\ m\ (takefill\ fill\ n\ w) = takefill\ fill\ m\ w$
⟨*proof*⟩

lemma *takefill-append*:
$takefill\ fill\ (m + length\ xs)\ (xs\ @\ w) = xs\ @\ (takefill\ fill\ m\ w)$
⟨*proof*⟩

lemma *takefill-same'*:
$l = length\ xs ==> takefill\ fill\ l\ xs = xs$
⟨*proof*⟩

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

lemma *takefill-bintrunc*:
takefill False n bl = *rev* (*bin-to-bl n* (*bl-to-bin* (*rev bl*)))
⟨*proof*⟩

lemma *bl-bin-bl-rtf*:
bin-to-bl n (*bl-to-bin bl*) = *rev* (*takefill False n* (*rev bl*))
⟨*proof*⟩

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

lemma *tf-rev*:
n + *k* = *m* + *length bl* ==> *takefill x m* (*rev* (*takefill y n bl*)) =
rev (*takefill y m* (*rev* (*takefill x k* (*rev bl*))))
⟨*proof*⟩

lemma *takefill-minus*:
0 < *n* ==> *takefill fill* (*Suc* (*n* − *1*)) *w* = *takefill fill n w*
⟨*proof*⟩

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

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

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

lemma *takefill-numeral-Nil* [*simp*]:
takefill fill (*numeral k*) [] = *fill* # *takefill fill* (*pred-numeral k*) []
⟨*proof*⟩

lemma *takefill-numeral-Cons* [*simp*]:
takefill fill (*numeral k*) (*x* # *xs*) = *x* # *takefill fill* (*pred-numeral k*) *xs*
⟨*proof*⟩

lemma *bl-to-bin-aux-cat*:
!!*nv v*. *bl-to-bin-aux bs* (*bin-cat w nv v*) =
bin-cat w (*nv* + *length bs*) (*bl-to-bin-aux bs v*)
⟨*proof*⟩

lemma *bin-to-bl-aux-cat*:
!!*w bs*. *bin-to-bl-aux* (*nv* + *nw*) (*bin-cat v nw w*) *bs* =
bin-to-bl-aux nv v (*bin-to-bl-aux nw w bs*)
⟨*proof*⟩

lemma *bl-to-bin-aux-alt*:
bl-to-bin-aux bs w = *bin-cat w* (*length bs*) (*bl-to-bin bs*)
⟨*proof*⟩

lemma *bin-to-bl-cat*:
bin-to-bl (*nv* + *nw*) (*bin-cat v nw w*) =
bin-to-bl-aux nv v (*bin-to-bl nw w*)
⟨*proof*⟩

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

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

lemma *bl-to-bin-app-cat*:
bl-to-bin (*bsa* @ *bs*) = *bin-cat* (*bl-to-bin bsa*) (*length bs*) (*bl-to-bin bs*)
⟨*proof*⟩

lemma *bin-to-bl-cat-app*:
bin-to-bl (*n* + *nw*) (*bin-cat w nw wa*) = *bin-to-bl n w* @ *bin-to-bl nw wa*
⟨*proof*⟩

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*)) =
(*bl-to-bin* (*replicate n True*)) + *1*
⟨*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*
⟨*proof*⟩

lemma *bl-of-nth-nth-le*:
  n ≤ length xs ⟹ bl-of-nth n (nth (rev xs)) = drop (length xs − n) xs
  ⟨*proof*⟩

lemma *bl-of-nth-nth* [*simp*]: *bl-of-nth (length xs) (op ! (rev xs)) = xs*
  ⟨*proof*⟩

lemma *size-rbl-pred*: *length (rbl-pred bl) = length bl*
  ⟨*proof*⟩

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

lemma *size-rbl-add*:
  !!*cl. length (rbl-add bl cl) = length bl*
  ⟨*proof*⟩

lemma *size-rbl-mult*:
  !!*cl. length (rbl-mult bl cl) = length bl*
  ⟨*proof*⟩

lemmas *rbl-sizes* [*simp*] =
  size-rbl-pred size-rbl-succ size-rbl-add size-rbl-mult

lemmas *rbl-Nils* =
  rbl-pred.Nil rbl-succ.Nil rbl-add.Nil rbl-mult.Nil

lemma *rbl-pred*:
  rbl-pred (rev (bin-to-bl n bin)) = rev (bin-to-bl n (bin − 1))
  ⟨*proof*⟩

lemma *rbl-succ*:
  rbl-succ (rev (bin-to-bl n bin)) = rev (bin-to-bl n (bin + 1))
  ⟨*proof*⟩

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))
  ⟨*proof*⟩

lemma *rbl-add-app2*:
  !!*blb. length blb >= length bla ==>*
    rbl-add bla (blb @ blc) = rbl-add bla blb
  ⟨*proof*⟩

lemma *rbl-add-take2*:
  !!*blb. length blb >= length bla ==>*
    rbl-add bla (take (length bla) blb) = rbl-add bla blb
  ⟨*proof*⟩

lemma *rbl-add-long*:
m >= n ==> rbl-add (rev (bin-to-bl n bina)) (rev (bin-to-bl m binb)) =
rev (bin-to-bl n (bina + binb))
⟨proof⟩

lemma *rbl-mult-app2*:
!!blb. length blb >= length bla ==>
rbl-mult bla (blb @ blc) = rbl-mult bla blb
⟨proof⟩

lemma *rbl-mult-take2*:
length blb >= length bla ==>
rbl-mult bla (take (length bla) blb) = rbl-mult bla blb
⟨proof⟩

lemma *rbl-mult-gt1*:
m >= length bl ==> rbl-mult bl (rev (bin-to-bl m binb)) =
rbl-mult bl (rev (bin-to-bl (length bl) binb))
⟨proof⟩

lemma *rbl-mult-gt*:
m > n ==> rbl-mult (rev (bin-to-bl n bina)) (rev (bin-to-bl m binb)) =
rbl-mult (rev (bin-to-bl n bina)) (rev (bin-to-bl n binb))
⟨proof⟩

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

lemma *rbbl-Cons*:
b # rev (bin-to-bl n x) = rev (bin-to-bl (Suc n) (x BIT b))
⟨proof⟩

lemma *rbl-mult*: !!bina binb.
rbl-mult (rev (bin-to-bl n bina)) (rev (bin-to-bl n binb)) =
rev (bin-to-bl n (bina * binb))
⟨proof⟩

lemma *rbl-add-split*:
P (rbl-add (y # ys) (x # xs)) =
(ALL ws. length ws = length ys --> ws = rbl-add ys xs -->
(y --> ((x --> P (False # rbl-succ ws)) & (~ x --> P (True # ws)))) &
(~ y --> P (x # ws)))
⟨proof⟩

lemma *rbl-mult-split*:
P (rbl-mult (y # ys) xs) =
(ALL ws. length ws = Suc (length ys) --> ws = False # rbl-mult ys xs -->

(y --> P (rbl-add ws xs)) & (~ y --> P ws))

⟨*proof*⟩

12.3 Repeated splitting or concatenation

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

lemma *bin-cat-foldl-lem*:
foldl (%*u*. *bin-cat u n*) *x xs* =
bin-cat x (*size xs* ∗ *n*) (*foldl* (%*u*. *bin-cat u n*) *y xs*)
⟨*proof*⟩

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

lemmas *bin-rsplit-aux-simps* = *bin-rsplit-aux.simps bin-rsplitl-aux.simps*
lemmas *rsplit-aux-simps* = *bin-rsplit-aux-simps*

lemmas *th-if-simp1* = *if-split* [**where** *P* = *op* = *l*, *THEN iffD1*, *THEN conjunct1*, *THEN mp*] **for** *l*
lemmas *th-if-simp2* = *if-split* [**where** *P* = *op* = *l*, *THEN iffD1*, *THEN conjunct2*, *THEN mp*] **for** *l*

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

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

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

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

lemma *bin-rsplit-aux-append*:
bin-rsplit-aux n m c (*bs* @ *cs*) = *bin-rsplit-aux n m c bs* @ *cs*
⟨*proof*⟩

lemma *bin-rsplitl-aux-append*:
bin-rsplitl-aux n m c (*bs* @ *cs*) = *bin-rsplitl-aux n m c bs* @ *cs*
⟨*proof*⟩

lemmas *rsplit-aux-apps* [**where** *bs* = []] =
bin-rsplit-aux-append bin-rsplitl-aux-append

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

lemmas *rsplit-aux-alts* = *rsplit-aux-apps* [unfolded append-Nil rsplit-def-auxs [symmetric]]

lemma bin-split-minus: 0 < n ==> bin-split (Suc (n − 1)) w = bin-split n w
⟨proof⟩

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

lemma bin-split-pred-simp [simp]:
(0::nat) < numeral bin ⟹
bin-split (numeral bin) w =
(let (w1, w2) = bin-split (numeral bin − 1) (bin-rest w)
in (w1, w2 BIT bin-last w))
⟨proof⟩

lemma bin-rsplit-aux-simp-alt:
bin-rsplit-aux n m c bs =
(if m = 0 ∨ n = 0
then bs
else let (a, b) = bin-split n c in bin-rsplit n (m − n, a) @ b # bs)
⟨proof⟩

lemmas bin-rsplit-simp-alt =
trans [OF bin-rsplit-def bin-rsplit-aux-simp-alt]

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

lemma bin-rsplit-size-sign' [rule-format] :
⟦n > 0; rev sw = bin-rsplit n (nw, w)⟧ ⟹
(ALL v: set sw. bintrunc n v = v)
⟨proof⟩

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

lemma bin-nth-rsplit [rule-format] :
n > 0 ==> m < n ==> (ALL w k nw. rev sw = bin-rsplit n (nw, w) −−>
k < size sw −−> bin-nth (sw ! k) m = bin-nth w (k ∗ n + m))
⟨proof⟩

lemma bin-rsplit-all:
0 < nw ==> nw <= n ==> bin-rsplit n (nw, w) = [bintrunc n w]
⟨proof⟩

lemma bin-rsplit-l [rule-format] :
ALL bin. bin-rsplitl n (m, bin) = bin-rsplit n (m, bintrunc m bin)
⟨proof⟩

lemma bin-rsplit-rcat [rule-format] :

*n > 0 --> bin-rsplit n (n * size ws, bin-rcat n ws) = map (bintrunc n) ws*
⟨*proof*⟩

lemma *bin-rsplit-aux-len-le* [*rule-format*] :
∀ *ws m. n ≠ 0 ⟶ ws = bin-rsplit-aux n nw w bs ⟶*
*length ws ≤ m ⟷ nw + length bs * n ≤ m * n*
⟨*proof*⟩

lemma *bin-rsplit-len-le*:
*n ≠ 0 --> ws = bin-rsplit n (nw, w) --> (length ws <= m) = (nw <= m * n)*
⟨*proof*⟩

lemma *bin-rsplit-aux-len*:
n ≠ 0 ⟹ length (bin-rsplit-aux n nw w cs) =
(nw + n − 1) div n + length cs
⟨*proof*⟩

lemma *bin-rsplit-len*:
n≠0 ==> length (bin-rsplit n (nw, w)) = (nw + n − 1) div n
⟨*proof*⟩

lemma *bin-rsplit-aux-len-indep*:
n ≠ 0 ⟹ length bs = length cs ⟹
length (bin-rsplit-aux n nw v bs) =
length (bin-rsplit-aux n nw w cs)
⟨*proof*⟩

lemma *bin-rsplit-len-indep*:
n≠0 ==> length (bin-rsplit n (nw, v)) = length (bin-rsplit n (nw, w))
⟨*proof*⟩

Even more bit operations

instantiation *int* :: *bitss*
begin

definition [*iff*]:
i !! *n ⟷ bin-nth i n*

definition
lsb i = (*i* :: *int*) !! *0*

definition
set-bit i n b = bin-sc n b i

definition
set-bits f =
(if ∃ n. ∀ n′≥n. ¬ f n′ then
let n = LEAST n. ∀ n′≥n. ¬ f n′

in bl-to-bin (*rev* (*map f* [*0*..<*n*]))
else if ∃ *n*. ∀ *n′*≥*n*. *f n′ then*
let n = *LEAST n*. ∀ *n′*≥*n*. *f n′*
in sbintrunc n (*bl-to-bin* (*True* # *rev* (*map f* [*0*..<*n*])))
else 0 :: *int*)

definition
shiftl x n = (*x* :: *int*) ∗ *2* ˆ *n*

definition
shiftr x n = (*x* :: *int*) *div 2* ˆ *n*

definition
msb x ⟷ (*x* :: *int*) < *0*

instance ⟨*proof*⟩

end

end

13 Type Definition Theorems

theory *Misc-Typedef*
imports *Main*
begin

14 More lemmas about normal type definitions

lemma
tdD1: *type-definition Rep Abs A* ⟹ ∀ *x*. *Rep x* ∈ *A* **and**
tdD2: *type-definition Rep Abs A* ⟹ ∀ *x*. *Abs* (*Rep x*) = *x* **and**
tdD3: *type-definition Rep Abs A* ⟹ ∀ *y*. *y* ∈ *A* ⟶ *Rep* (*Abs y*) = *y*
⟨*proof*⟩

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

context *type-definition*
begin

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

lemma *Abs-eqD*: *Abs x* = *Abs y* ==> *x* ∈ *A* ==> *y* ∈ *A* ==> *x* = *y*
⟨*proof*⟩

lemma *Abs-inverse′*:

r : A ==> Abs r = a ==> Rep a = r
⟨*proof*⟩

lemma *Rep-comp-inverse*:
Rep o f = g ==> Abs o g = f
⟨*proof*⟩

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

lemma *Rep-inverse'*: *Rep a = r ==> Abs r = a*
⟨*proof*⟩

lemma *comp-Abs-inverse*:
f o Abs = g ==> g o Rep = f
⟨*proof*⟩

lemma *set-Rep*:
A = range Rep
⟨*proof*⟩

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

lemma *Abs-inj-on*: *inj-on Abs A*
⟨*proof*⟩

lemma *image*: *Abs ' A = UNIV*
⟨*proof*⟩

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

lemma *fns1*:
Rep o fa = fr o Rep | fa o Abs = Abs o fr ==> Abs o fr o Rep = fa
⟨*proof*⟩

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

lemma *fns4*:
Rep o fa o Abs = fr ==>
Rep o fa = fr o Rep & fa o Abs = Abs o fr
⟨*proof*⟩

end

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

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

14.1 Extended form of type definition predicate

lemma *td-conds*:
norm o norm = norm ==> (fr o norm = norm o fr) =
(norm o fr o norm = fr o norm & norm o fr o norm = norm o fr)
⟨*proof*⟩

lemma *fn-comm-power*:
fa o tr = tr o fr ==> fa ^^ n o tr = tr o fr ^^ n
⟨*proof*⟩

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

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

lemma *Abs-norm* [*simp*]:
Abs (norm x) = Abs x
⟨*proof*⟩

lemma *td-th*:
g o Abs = f ==> f (Rep x) = g x
⟨*proof*⟩

lemma *eq-norm′*: *Rep o Abs = norm*
⟨*proof*⟩

lemma *norm-Rep* [*simp*]: *norm (Rep x) = Rep x*
⟨*proof*⟩

lemmas *td = td-thm*

lemma *set-iff-norm*: *w : A* ⟷ *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 o norm = Abs
norm o Rep = Rep
norm o norm = norm
⟨*proof*⟩

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

lemma *fns5*:
Rep o fa o Abs = fr ==>
fr o norm = fr & norm o fr = fr
⟨*proof*⟩

lemma *fns2*:
Abs o fr o Rep = fa ==>
(norm o fr o norm = fr o norm) = (Rep o fa = fr o Rep)
⟨*proof*⟩

lemma *fns3*:
Abs o fr o Rep = fa ==>
(norm o fr o norm = norm o fr) = (fa o Abs = Abs o fr)
⟨*proof*⟩

lemma *fns*:
fr o norm = norm o fr ==>
(fa o Abs = Abs o fr) = (Rep o fa = fr o Rep)
⟨*proof*⟩

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

end

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

end

15 Miscellaneous lemmas, of at least doubtful value

theory *Word-Miscellaneous*
imports *Main ~~/src/HOL/Library/Bit Misc-Numeric*
begin

lemma *power-minus-simp*:
$0 < n \Longrightarrow a \,\hat{}\, n = a * a \,\hat{}\, (n - 1)$
⟨*proof*⟩

lemma *funpow-minus-simp*:
$0 < n \Longrightarrow f \,\hat{}\hat{}\, n = f \circ f \,\hat{}\hat{}\, (n - 1)$
⟨*proof*⟩

lemma *power-numeral*:
$a \,\hat{}\, numeral\ k = a * a \,\hat{}\, (pred\text{-}numeral\ k)$
⟨*proof*⟩

lemma *funpow-numeral* [*simp*]:
$f \,\hat{}\hat{}\, numeral\ k = f \circ f \,\hat{}\hat{}\, (pred\text{-}numeral\ k)$
⟨*proof*⟩

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

lemma *rco-alt*: $(f\ o\ g)\ \hat{}\hat{}\ n\ o\ f = f\ o\ (g\ o\ f)\ \hat{}\hat{}\ n$
⟨*proof*⟩

lemma *list-exhaust-size-gt0*:
assumes y: $\bigwedge a\ list.\ y = a \# list \Longrightarrow P$
shows $0 < length\ y \Longrightarrow P$
⟨*proof*⟩

lemma *list-exhaust-size-eq0*:
assumes y: $y = [] \Longrightarrow P$
shows $length\ y = 0 \Longrightarrow P$
⟨*proof*⟩

lemma *size-Cons-lem-eq*:
$y = xa \# list ==> size\ y = Suc\ k ==> size\ list = k$
⟨*proof*⟩

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

lemma *not-B1-is-B0*: $y \neq (1{::}bit) \Longrightarrow y = (0{::}bit)$
⟨*proof*⟩

lemma *B1-ass-B0*:
assumes y: $y = (0{::}bit) \Longrightarrow y = (1{::}bit)$
shows $y = (1{::}bit)$
⟨*proof*⟩
lemmas *n2s-ths* [*THEN eq-reflection*] = *add-2-eq-Suc add-2-eq-Suc'*

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

lemma *and-len*: *xs = ys ==> xs = ys & length xs = length ys*
⟨*proof*⟩

lemma *size-if*: *size (if p then xs else ys) = (if p then size xs else size ys)*
⟨*proof*⟩

lemma *tl-if*: *tl (if p then xs else ys) = (if p then tl xs else tl ys)*
⟨*proof*⟩

lemma *hd-if*: *hd (if p then xs else ys) = (if p then hd xs else hd ys)*
⟨*proof*⟩

lemma *if-Not-x*: *(if p then ~ x else x) = (p = (~ x))*
⟨*proof*⟩

lemma *if-x-Not*: *(if p then x else ~ x) = (p = x)*
⟨*proof*⟩

lemma *if-same-and*: *(If p x y & If p u v) = (if p then x & u else y & v)*
⟨*proof*⟩

lemma *if-same-eq*: *(If p x y = (If p u v)) = (if p then x = (u) else y = (v))*
⟨*proof*⟩

lemma *if-same-eq-not*:
(If p x y = (~ If p u v)) = (if p then x = (~u) else y = (~v))
⟨*proof*⟩

lemma *if-Cons*: *(if p then x # xs else y # ys) = If p x y # If p xs ys*
⟨*proof*⟩

lemma *if-single*:
(if xc then [xab] else [an]) = [if xc then xab else an]
⟨*proof*⟩

lemma *if-bool-simps*:
If p True y = (p | y) & If p False y = (~p & y) &
If p y True = (p --> y) & If p y False = (p & y)
⟨*proof*⟩

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

lemmas *seqr = eq-reflection* [**where** *x = size w*] **for** *w*

lemma *the-elemI*: *y = {x} ==> the-elem y = x*
⟨*proof*⟩

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

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

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* ==> *j* − *i* = (*k* :: *nat*) ⟨*proof*⟩

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

lemma *nmod2*: *n mod* (*2*::*int*) = *0* | *n mod 2* = *1*
⟨*proof*⟩

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

lemma *le-diff-eq'*: (*a* ≤ *c* − *b*) = (*b* + *a* ≤ (*c*::*int*)) ⟨*proof*⟩

lemma *less-diff-eq'*: (*a* < *c* − *b*) = (*b* + *a* < (*c*::*int*)) ⟨*proof*⟩

lemma *diff-less-eq'*: (*a* − *b* < *c*) = (*a* < *b* + (*c*::*int*)) ⟨*proof*⟩

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

lemma *z1pdiv2*:
(*2* ∗ *b* + *1*) *div 2* = (*b*::*int*) ⟨*proof*⟩

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

lemma *axxbyy*:
a + *m* + *m* = *b* + *n* + *n* ==> (*a* = *0* | *a* = *1*) ==> (*b* = *0* | *b* = *1*) ==>
a = *b* & *m* = (*n* :: *int*) ⟨*proof*⟩

lemma *axxmod2*:
(*1* + *x* + *x*) *mod 2* = (*1* :: *int*) & (*0* + *x* + *x*) *mod 2* = (*0* :: *int*) ⟨*proof*⟩

lemma *axxdiv2*:

(1 + x + x) div 2 = (x :: int) & (0 + x + x) div 2 = (x :: int) ⟨*proof*⟩

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

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

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

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

lemma *mod-plus-right*:
((a + x) mod m = (b + x) mod m) = (a mod m = b mod (m :: nat))
⟨*proof*⟩

lemma *nat-minus-mod*: *(n − n mod m) mod m = (0 :: nat)*
⟨*proof*⟩

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

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

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

lemma *nat-mod-eq*:
!!b. b < n ==> a mod n = b mod n ==> a mod n = (b :: nat)
⟨*proof*⟩

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

lemma *nat-mod-lem*:
(0 :: nat) < n ==> b < n = (b mod n = b)
⟨*proof*⟩

lemma *mod-nat-add*:
(x :: nat) < z ==> y < z ==>
(x + y) mod z = (if x + y < z then x + y else x + y − z)
⟨*proof*⟩

lemma *mod-nat-sub*:
(x :: nat) < z ==> (x − y) mod z = x − y
⟨*proof*⟩

lemma *int-mod-eq*:
(0 :: int) <= b ==> b < n ==> a mod n = b mod n ==> a mod n = b
⟨*proof*⟩

lemmas *int-mod-eq′* = *mod-pos-pos-trivial*

lemma *int-mod-le*: (0::int) <= a ==> a mod n <= a
⟨*proof*⟩

lemma *mod-add-if-z*:
(x :: int) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>
(x + y) mod z = (if x + y < z then x + y else x + y − z)
⟨*proof*⟩

lemma *mod-sub-if-z*:
(x :: int) < z ==> y < z ==> 0 <= y ==> 0 <= x ==> 0 <= z ==>
(x − y) mod z = (if y <= x then x − y else x − y + z)
⟨*proof*⟩

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 ~= (0 :: int) ==> (a + m * n) div m = a div m + n
⟨*proof*⟩

lemma *mod-power-lem*:
a > 1 ==> a ^ n mod a ^ m = (if m <= n then 0 else (a :: int) ^ n)
⟨*proof*⟩

lemma *pl-pl-rels*:
a + b = c + d ==>
a >= c & b <= d | a <= c & b >= (d :: nat) ⟨*proof*⟩

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

lemma *minus-eq*: (m − k = m) = (k = 0 | m = (0 :: nat)) ⟨*proof*⟩

lemma *pl-pl-mm*: (a :: nat) + b = c + d ==> a − c = d − b ⟨*proof*⟩

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

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 ==> (a >= b * c) = (a div c >= (b :: nat))
⟨proof⟩

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

lemma *div-mult-le*: (a :: nat) div b * b <= a
⟨proof⟩

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

lemma *given-quot*: f > (0 :: nat) ==> (f * l + (f − 1)) div f = l
⟨proof⟩

lemma *given-quot-alt*: f > (0 :: nat) ==> (l * f + f − Suc 0) div f = l
⟨proof⟩

lemma *diff-mod-le*: (a::nat) < d ==> b dvd d ==> a − a mod b <= d − b
⟨proof⟩

lemma *less-le-mult'*:
w * c < b * c ==> 0 ≤ c ==> (w + 1) * c ≤ b * (c::int)
⟨proof⟩

lemma *less-le-mult*:
w * c < b * c ⟹ 0 ≤ c ⟹ w * c + c ≤ b * (c::int)
⟨proof⟩

lemmas *less-le-mult-minus* = *iffD2* [*OF le-diff-eq less-le-mult*,
simplified left-diff-distrib]

lemma *gen-minus*: 0 < n ==> f n = f (Suc (n − 1))
⟨proof⟩

lemma *mpl-lem*: j <= (i :: nat) ==> k < j ==> i − j + k < i ⟨proof⟩

lemma *nonneg-mod-div*:
0 <= a ==> 0 <= b ==> 0 <= (a mod b :: int) & 0 <= a div b
⟨proof⟩

declare *iszero-0* [*intro*]

lemma *min-pm* [*simp*]:
min a b + (a − b) = (a :: nat)
⟨proof⟩

lemma *min-pm1* [*simp*]:
a − b + min a b = (a :: nat)

⟨*proof*⟩

lemma *rev-min-pm* [*simp*]:
min b a + (*a* − *b*) = (*a* :: *nat*)
⟨*proof*⟩

lemma *rev-min-pm1* [*simp*]:
a − *b* + *min b a* = (*a* :: *nat*)
⟨*proof*⟩

lemma *min-minus* [*simp*]:
min m (*m* − *k*) = (*m* − *k* :: *nat*)
⟨*proof*⟩

lemma *min-minus′* [*simp*]:
min (*m* − *k*) *m* = (*m* − *k* :: *nat*)
⟨*proof*⟩

end

16 A type of finite bit strings

theory *Word*
imports
Type-Length
~~/src/HOL/Library/Boolean-Algebra
Bits-Bit
Bool-List-Representation
Misc-Typedef
Word-Miscellaneous
begin

See `Examples/WordExamples.thy` for examples.

16.1 Type definition

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

lemma *uint-nonnegative*:
0 ≤ *uint w*
⟨*proof*⟩

lemma *uint-bounded*:
fixes *w* :: *′a*::*len0 word*
shows *uint w* < *2* ^ *len-of TYPE*(*′a*)
⟨*proof*⟩

lemma *uint-idem*:

fixes w :: $'a$::*len0 word*
shows *uint w mod 2 ^ len-of TYPE('a) = uint w*
⟨*proof*⟩

lemma *word-uint-eq-iff*:
$a = b \longleftrightarrow$ *uint a = uint b*
⟨*proof*⟩

lemma *word-uint-eqI*:
uint a = uint b $\Longrightarrow a = b$
⟨*proof*⟩

definition *word-of-int* :: *int* ⇒ $'a$::*len0 word*
where
— representation of words using unsigned or signed bins, only difference in these is the type class
word-of-int k = Abs-word (k mod 2 ^ len-of TYPE('a))

lemma *uint-word-of-int*:
uint (word-of-int k :: $'a$::*len0 word) = k mod 2 ^ len-of TYPE('a)*
⟨*proof*⟩

lemma *word-of-int-uint*:
word-of-int (uint w) = w
⟨*proof*⟩

lemma *split-word-all*:
$(\bigwedge x$::$'a$::*len0 word. PROP P x*$) \equiv (\bigwedge x$*. PROP P (word-of-int x*$))$
⟨*proof*⟩

16.2 Type conversions and casting

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

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

definition *uints* :: *nat* ⇒ *int set*
where
— the sets of integers representing the words
uints n = range (bintrunc n)

definition *sints* :: *nat* ⇒ *int set*
where
sints n = range (sbintrunc (n − 1))

lemma *uints-num*:
uints n = {*i*. *0* ≤ *i* ∧ *i* < *2* ^ *n*}
⟨*proof*⟩

lemma *sints-num*:
sints n = {*i*. − (*2* ^ (*n* − *1*)) ≤ *i* ∧ *i* < *2* ^ (*n* − *1*)}
⟨*proof*⟩

definition *unats* :: *nat* ⇒ *nat set*
where
unats n = {*i*. *i* < *2* ^ *n*}

definition *norm-sint* :: *nat* ⇒ *int* ⇒ *int*
where
norm-sint n w = (*w* + *2* ^ (*n* − *1*)) *mod 2* ^ *n* − *2* ^ (*n* − *1*)

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

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

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

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

instance ⟨*proof*⟩

end

lemma *word-size-gt-0* [*iff*]:
0 < *size* (*w*::*'a*::*len word*)
⟨*proof*⟩

lemmas *lens-gt-0* = *word-size-gt-0 len-gt-0*

lemma *lens-not-0* [*iff*]:
shows *size* (*w*::*'a*::*len word*) ≠ *0*
and *len-of TYPE*(*'a*::*len*) ≠ *0*
⟨*proof*⟩

definition *source-size* :: (*'a*::*len0 word* ⇒ *'b*) ⇒ *nat*
where

— whether a cast (or other) function is to a longer or shorter length
[*code del*]: *source-size c* = (*let arb* = *undefined*; *x* = *c arb in size arb*)

definition *target-size* :: (*'a* ⇒ *'b*::*len0 word*) ⇒ *nat*
where
[*code del*]: *target-size c* = *size* (*c undefined*)

definition *is-up* :: (*'a*::*len0 word* ⇒ *'b*::*len0 word*) ⇒ *bool*
where
is-up c ⟷ *source-size c* ≤ *target-size c*

definition *is-down* :: (*'a* :: *len0 word* ⇒ *'b* :: *len0 word*) ⇒ *bool*
where
is-down c ⟷ *target-size c* ≤ *source-size c*

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

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

definition *word-reverse* :: *'a*::*len0 word* ⇒ *'a word*
where
word-reverse w = *of-bl* (*rev* (*to-bl w*))

definition *word-int-case* :: (*int* ⇒ *'b*) ⇒ *'a*::*len0 word* ⇒ *'b*
where
word-int-case f w = *f* (*uint w*)

translations
case x of XCONST of-int y => *b* == *CONST word-int-case* (%*y*. *b*) *x*
case x of (*XCONST of-int* :: *'a*) *y* => *b* => *CONST word-int-case* (%*y*. *b*) *x*

16.3 Correspondence relation for theorem transfer

definition *cr-word* :: *int* ⇒ *'a*::*len0 word* ⇒ *bool*
where
cr-word = (λ*x y*. *word-of-int x* = *y*)

lemma *Quotient-word*:
Quotient (λ*x y*. *bintrunc* (*len-of TYPE*(*'a*)) *x* = *bintrunc* (*len-of TYPE*(*'a*)) *y*)
word-of-int uint (*cr-word* :: - ⇒ *'a*::*len0 word* ⇒ *bool*)
⟨*proof*⟩

lemma *reflp-word*:
reflp (λ*x y*. *bintrunc* (*len-of TYPE*(*'a*::*len0*)) *x* = *bintrunc* (*len-of TYPE*(*'a*)) *y*)

⟨*proof*⟩

setup-lifting *Quotient-word reflp-word*

TODO: The next lemma could be generated automatically.

lemma *uint-transfer* [*transfer-rule*]:
(*rel-fun pcr-word op* =) (*bintrunc* (*len-of TYPE*('*a*)))
(*uint* :: '*a*::*len0 word* ⇒ *int*)
⟨*proof*⟩

16.4 Basic code generation setup

definition *Word* :: *int* ⇒ '*a*::*len0 word*
where
[*code-post*]: *Word* = *word-of-int*

lemma [*code abstype*]:
Word (*uint w*) = *w*
⟨*proof*⟩

declare *uint-word-of-int* [*code abstract*]

instantiation *word* :: (*len0*) *equal*
begin

definition *equal-word* :: '*a word* ⇒ '*a word* ⇒ *bool*
where
equal-word k l ⟷ *HOL.equal* (*uint k*) (*uint l*)

instance ⟨*proof*⟩

end

notation *fcomp* (**infixl** ∘> *60*)
notation *scomp* (**infixl** ∘→ *60*)

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

definition
random-word i = *Random.range i* ∘→ (λ*k*. *Pair* (
let j = *word-of-int* (*int-of-integer* (*integer-of-natural k*)) :: '*a word*
in (*j*, λ-::*unit*. *Code-Evaluation.term-of j*)))

instance ⟨*proof*⟩

end

no-notation *fcomp* (**infixl** ∘> *60*)

no-notation *scomp* (**infixl** ∘→ *60*)

16.5 Type-definition locale instantiations

lemmas *uint-0* = *uint-nonnegative*
lemmas *uint-lt* = *uint-bounded*
lemmas *uint-mod-same* = *uint-idem*

lemma *td-ext-uint*:
td-ext (*uint* :: *'a word* ⇒ *int*) *word-of-int* (*uints* (*len-of TYPE*('*a*::*len0*)))
(λ*w*::*int*. *w mod 2 ^ len-of TYPE*('*a*))
⟨*proof*⟩

interpretation *word-uint*:
td-ext uint::'*a*::*len0 word* ⇒ *int*
word-of-int
uints (*len-of TYPE*('*a*::*len0*))
λ*w*. *w mod 2 ^ len-of TYPE*('*a*::*len0*)
⟨*proof*⟩

lemmas *td-uint* = *word-uint.td-thm*
lemmas *int-word-uint* = *word-uint.eq-norm*

lemma *td-ext-ubin*:
td-ext (*uint* :: *'a word* ⇒ *int*) *word-of-int* (*uints* (*len-of TYPE*('*a*::*len0*)))
(*bintrunc* (*len-of TYPE*('*a*)))
⟨*proof*⟩

interpretation *word-ubin*:
td-ext uint::'*a*::*len0 word* ⇒ *int*
word-of-int
uints (*len-of TYPE*('*a*::*len0*))
bintrunc (*len-of TYPE*('*a*::*len0*))
⟨*proof*⟩

16.6 Arithmetic operations

lift-definition *word-succ* :: '*a*::*len0 word* ⇒ '*a word* **is** λ*x*. *x* + *1*
⟨*proof*⟩

lift-definition *word-pred* :: '*a*::*len0 word* ⇒ '*a word* **is** λ*x*. *x* − *1*
⟨*proof*⟩

instantiation *word* :: (*len0*) {*neg-numeral*, *Divides.div*, *comm-monoid-mult*, *comm-ring*}
begin

lift-definition *zero-word* :: '*a word* **is** *0* ⟨*proof*⟩

lift-definition *one-word* :: '*a word* **is** *1* ⟨*proof*⟩

lift-definition *plus-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op* +
⟨*proof*⟩

lift-definition *minus-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op* −
⟨*proof*⟩

lift-definition *uminus-word* :: *'a word* ⇒ *'a word* **is** *uminus*
⟨*proof*⟩

lift-definition *times-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *op* ∗
⟨*proof*⟩

definition
word-div-def: *a div b* = *word-of-int* (*uint a div uint b*)

definition
word-mod-def: *a mod b* = *word-of-int* (*uint a mod uint b*)

instance
⟨*proof*⟩

end

Legacy theorems:

lemma *word-arith-wis* [*code*]: **shows**
word-add-def: *a* + *b* = *word-of-int* (*uint a* + *uint b*) **and**
word-sub-wi: *a* − *b* = *word-of-int* (*uint a* − *uint b*) **and**
word-mult-def: *a* ∗ *b* = *word-of-int* (*uint a* ∗ *uint b*) **and**
word-minus-def: − *a* = *word-of-int* (− *uint a*) **and**
word-succ-alt: *word-succ a* = *word-of-int* (*uint a* + *1*) **and**
word-pred-alt: *word-pred a* = *word-of-int* (*uint a* − *1*) **and**
word-0-wi: *0* = *word-of-int 0* **and**
word-1-wi: *1* = *word-of-int 1*
⟨*proof*⟩

lemmas *arths* =
bintr-ariths [*THEN word-ubin.norm-eq-iff* [*THEN iffD1*], *folded word-ubin.eq-norm*]

lemma *wi-homs*:
shows
wi-hom-add: *word-of-int a* + *word-of-int b* = *word-of-int* (*a* + *b*) **and**
wi-hom-sub: *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* (*a* + *1*) **and**
wi-hom-pred: *word-pred* (*word-of-int a*) = *word-of-int* (*a* − *1*)
⟨*proof*⟩

lemmas *wi-hom-syms* = *wi-homs* [*symmetric*]

lemmas *word-of-int-homs* = *wi-homs word-0-wi word-1-wi*

lemmas *word-of-int-hom-syms* = *word-of-int-homs* [*symmetric*]

instance *word* :: (*len*) *comm-ring-1*
⟨*proof*⟩

lemma *word-of-nat*: *of-nat n* = *word-of-int* (*int n*)
⟨*proof*⟩

lemma *word-of-int*: *of-int* = *word-of-int*
⟨*proof*⟩

definition *udvd* :: *'a::len word => 'a::len word => bool* (**infixl** *udvd 50*)
where
a udvd b = (*EX n>=0. uint b* = *n* ∗ *uint a*)

16.7 Ordering

instantiation *word* :: (*len0*) *linorder*
begin

definition
word-le-def: $a \leq b \longleftrightarrow uint\ a \leq uint\ b$

definition
word-less-def: $a < b \longleftrightarrow uint\ a < uint\ b$

instance
⟨*proof*⟩

end

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

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

16.8 Bit-wise operations

instantiation *word* :: (*len0*) *bits*
begin

lift-definition *bitNOT-word* :: *'a word ⇒ 'a word* **is** *bitNOT*
⟨*proof*⟩

lift-definition *bitAND-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitAND*
⟨*proof*⟩

lift-definition *bitOR-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitOR*
⟨*proof*⟩

lift-definition *bitXOR-word* :: *'a word* ⇒ *'a word* ⇒ *'a word* **is** *bitXOR*
⟨*proof*⟩

definition
word-test-bit-def: *test-bit a* = *bin-nth* (*uint a*)

definition
word-set-bit-def: *set-bit a n x* =
word-of-int (*bin-sc n x* (*uint a*))

definition
word-set-bits-def: (*BITS n*. *f n*) = *of-bl* (*bl-of-nth* (*len-of TYPE* (*'a*)) *f*)

definition
word-lsb-def: *lsb a* ⟷ *bin-last* (*uint a*)

definition *shiftl1* :: *'a word* ⇒ *'a word*
where
shiftl1 w = *word-of-int* (*uint w BIT False*)

definition *shiftr1* :: *'a word* ⇒ *'a word*
where
— shift right as unsigned or as signed, ie logical or arithmetic
shiftr1 w = *word-of-int* (*bin-rest* (*uint w*))

definition
shiftl-def: *w* << *n* = (*shiftl1* ^^ *n*) *w*

definition
shiftr-def: *w* >> *n* = (*shiftr1* ^^ *n*) *w*

instance ⟨*proof*⟩

end

lemma [*code*]: **shows**
word-not-def: *NOT* (*a*::*'a*::*len0 word*) = *word-of-int* (*NOT* (*uint a*)) **and**
word-and-def: (*a*::*'a word*) *AND b* = *word-of-int* (*uint a AND uint b*) **and**
word-or-def: (*a*::*'a word*) *OR b* = *word-of-int* (*uint a OR uint b*) **and**
word-xor-def: (*a*::*'a word*) *XOR b* = *word-of-int* (*uint a XOR uint b*)
⟨*proof*⟩

instantiation *word* :: (*len*) *bitss*

begin

definition
word-msb-def:
$msb\ a \longleftrightarrow bin\text{-}sign\ (sint\ a) = -1$

instance ⟨*proof*⟩

end

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

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

16.9 Shift operations

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

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

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

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

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

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

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

16.10 Rotation

definition *rotater1* :: *'a list => 'a list*
where
rotater1 ys =
(case ys of [] => [] | x # xs => last ys # butlast ys)

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

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

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

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

16.11 Split and cat operations

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

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

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

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

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

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

16.12 Theorems about typedefs

lemma *sint-sbintrunc'*:
sint (*word-of-int bin* :: *'a word*) =
(*sbintrunc* (*len-of TYPE* (*'a* :: *len*) − *1*) *bin*)
⟨*proof*⟩

lemma *uint-sint*:
uint w = *bintrunc* (*len-of TYPE*(*'a*)) (*sint* (*w* :: *'a* :: *len word*))
⟨*proof*⟩

lemma *bintr-uint*:
fixes *w* :: *'a*::*len0 word*
shows *len-of TYPE*(*'a*) ≤ *n* ⟹ *bintrunc n* (*uint w*) = *uint w*
⟨*proof*⟩

lemma *wi-bintr*:
len-of TYPE(*'a*::*len0*) ≤ *n* ⟹
word-of-int (*bintrunc n w*) = (*word-of-int w* :: *'a word*)
⟨*proof*⟩

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*⟩

lemma *td-ext-sint*:
td-ext (*sint* :: *'a word* ⇒ *int*) *word-of-int* (*sints* (*len-of TYPE*(*'a*::*len*)))
(λ*w*. (*w* + *2* ˆ (*len-of TYPE*(*'a*) − *1*)) *mod 2* ˆ *len-of TYPE*(*'a*) −
2 ˆ (*len-of TYPE*(*'a*) − *1*))
⟨*proof*⟩

interpretation *word-sint*:
td-ext sint ::*'a*::*len word* => *int*
word-of-int
sints (*len-of TYPE*(*'a*::*len*))
%*w*. (*w* + *2*ˆ(*len-of TYPE*(*'a*::*len*) − *1*)) *mod 2*ˆ*len-of TYPE*(*'a*::*len*) −
2 ˆ (*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*]

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

lemma *to-bl-def'*:
(*to-bl* :: *'a* :: *len0 word* => *bool list*) =
bin-to-bl (*len-of TYPE*(*'a*)) *o uint*
⟨*proof*⟩

lemmas *word-reverse-no-def* [*simp*] = *word-reverse-def* [*of numeral w*] **for** *w*

lemma *uints-mod*: *uints n* = *range* (λ*w*. *w mod 2* ^ *n*)
⟨*proof*⟩

lemma *word-numeral-alt*:
numeral b = *word-of-int* (*numeral b*)
⟨*proof*⟩

declare *word-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *word-neg-numeral-alt*:
− *numeral b* = *word-of-int* (− *numeral b*)
⟨*proof*⟩

declare *word-neg-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *word-numeral-transfer* [*transfer-rule*]:
(*rel-fun op* = *pcr-word*) *numeral numeral*
(*rel-fun op* = *pcr-word*) (− *numeral*) (− *numeral*)
⟨*proof*⟩

lemma *uint-bintrunc* [*simp*]:
uint (*numeral bin* :: *'a word*) =
bintrunc (*len-of TYPE* (*'a* :: *len0*)) (*numeral bin*)
⟨*proof*⟩

lemma *uint-bintrunc-neg* [*simp*]: *uint* (− *numeral bin* :: *'a word*) =
bintrunc (*len-of TYPE* (*'a* :: *len0*)) (− *numeral bin*)
⟨*proof*⟩

lemma *sint-sbintrunc* [*simp*]:
sint (*numeral bin* :: *'a word*) =
sbintrunc (*len-of TYPE* (*'a* :: *len*) − *1*) (*numeral bin*)
⟨*proof*⟩

lemma *sint-sbintrunc-neg* [*simp*]: *sint* (− *numeral bin* :: *'a word*) =
sbintrunc (*len-of TYPE* (*'a* :: *len*) − *1*) (− *numeral bin*)
⟨*proof*⟩

lemma *unat-bintrunc* [*simp*]:
unat (*numeral bin* :: *'a* :: *len0 word*) =
nat (*bintrunc* (*len-of TYPE*(*'a*)) (*numeral bin*))
⟨*proof*⟩

lemma *unat-bintrunc-neg* [*simp*]:
unat (− *numeral bin* :: *'a* :: *len0 word*) =
nat (*bintrunc* (*len-of TYPE*(*'a*)) (− *numeral bin*))
⟨*proof*⟩

lemma *size-0-eq*: *size* (*w* :: *'a* :: *len0 word*) = *0* ⟹ *v* = *w*
⟨*proof*⟩

lemma *uint-ge-0* [*iff*]: *0* ≤ *uint* (*x*::*'a*::*len0 word*)
⟨*proof*⟩

lemma *uint-lt2p* [*iff*]: *uint* (*x*::*'a*::*len0 word*) < *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *sint-ge*: − (*2* ^ (*len-of TYPE*(*'a*) − *1*)) ≤ *sint* (*x*::*'a*::*len word*)
⟨*proof*⟩

lemma *sint-lt*: *sint* (*x*::*'a*::*len word*) < *2* ^ (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *sign-uint-Pls* [*simp*]:
bin-sign (*uint x*) = *0*
⟨*proof*⟩

lemma *uint-m2p-neg*: *uint* (*x*::*'a*::*len0 word*) − *2* ^ *len-of TYPE*(*'a*) < *0*
⟨*proof*⟩

lemma *uint-m2p-not-non-neg*:
¬ *0* ≤ *uint* (*x*::*'a*::*len0 word*) − *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *lt2p-lem*:
len-of TYPE(*'a*) ≤ *n* ⟹ *uint* (*w* :: *'a*::*len0 word*) < *2* ^ *n*
⟨*proof*⟩

lemma *uint-le-0-iff* [*simp*]: *uint x* ≤ *0* ⟷ *uint x* = *0*
⟨*proof*⟩

lemma *uint-nat*: *uint w* = *int* (*unat w*)
⟨*proof*⟩

lemma *uint-numeral*:
uint (*numeral b* :: *'a* :: *len0 word*) = *numeral b mod 2* ^ *len-of TYPE*(*'a*)

⟨*proof*⟩

lemma *uint-neg-numeral*:
uint (− *numeral b* :: ′*a* :: *len0 word*) = − *numeral b mod 2* ^ *len-of TYPE*(′*a*)
⟨*proof*⟩

lemma *unat-numeral*:
unat (*numeral b*::′*a*::*len0 word*) = *numeral b mod 2* ^ *len-of TYPE* (′*a*)
⟨*proof*⟩

lemma *sint-numeral*: *sint* (*numeral b* :: ′*a* :: *len word*) = (*numeral b* +
2 ^ (*len-of TYPE*(′*a*) − *1*)) *mod 2* ^ *len-of TYPE*(′*a*) −
2 ^ (*len-of TYPE*(′*a*) − *1*)
⟨*proof*⟩

lemma *word-of-int-0* [*simp*, *code-post*]:
word-of-int 0 = *0*
⟨*proof*⟩

lemma *word-of-int-1* [*simp*, *code-post*]:
word-of-int 1 = *1*
⟨*proof*⟩

lemma *word-of-int-neg-1* [*simp*]: *word-of-int* (− *1*) = − *1*
⟨*proof*⟩

lemma *word-of-int-numeral* [*simp*] :
(*word-of-int* (*numeral bin*) :: ′*a* :: *len0 word*) = (*numeral bin*)
⟨*proof*⟩

lemma *word-of-int-neg-numeral* [*simp*]:
(*word-of-int* (− *numeral bin*) :: ′*a* :: *len0 word*) = (− *numeral bin*)
⟨*proof*⟩

lemma *word-int-case-wi*:
word-int-case f (*word-of-int i* :: ′*b word*) =
f (*i mod 2* ^ *len-of TYPE*(′*b*::*len0*))
⟨*proof*⟩

lemma *word-int-split*:
P (*word-int-case f x*) =
(*ALL i*. *x* = (*word-of-int i* :: ′*b* :: *len0 word*) &
0 <= *i* & *i* < *2* ^ *len-of TYPE*(′*b*) --> *P* (*f i*))
⟨*proof*⟩

lemma *word-int-split-asm*:
P (*word-int-case f x*) =
(~ (*EX n*. *x* = (*word-of-int n* :: ′*b*::*len0 word*) &
0 <= *n* & *n* < *2* ^ *len-of TYPE*(′*b*::*len0*) & ~ *P* (*f n*)))

⟨*proof*⟩

lemmas *uint-range′* = *word-uint.Rep* [*unfolded uints-num mem-Collect-eq*]
lemmas *sint-range′* = *word-sint.Rep* [*unfolded One-nat-def sints-num mem-Collect-eq*]

lemma *uint-range-size*: *0* <= *uint w* & *uint w* < *2* ^ *size w*
⟨*proof*⟩

lemma *sint-range-size*:
− (*2* ^ (*size w* − *Suc 0*)) <= *sint w* & *sint w* < *2* ^ (*size w* − *Suc 0*)
⟨*proof*⟩

lemma *sint-above-size*: *2* ^ (*size* (*w*::′*a*::*len word*) − *1*) ≤ *x* ⟹ *sint w* < *x*
⟨*proof*⟩

lemma *sint-below-size*:
x ≤ − (*2* ^ (*size* (*w*::′*a*::*len word*) − *1*)) ⟹ *x* ≤ *sint w*
⟨*proof*⟩

16.13 Testing bits

lemma *test-bit-eq-iff*: (*test-bit* (*u*::′*a*::*len0 word*) = *test-bit v*) = (*u* = *v*)
⟨*proof*⟩

lemma *test-bit-size* [*rule-format*] : (*w*::′*a*::*len0 word*) !! *n* −−> *n* < *size w*
⟨*proof*⟩

lemma *word-eq-iff*:
fixes *x y* :: ′*a*::*len0 word*
shows *x* = *y* ⟷ (∀ *n*<*len-of TYPE*(′*a*). *x* !! *n* = *y* !! *n*)
⟨*proof*⟩

lemma *word-eqI* [*rule-format*]:
fixes *u* :: ′*a*::*len0 word*
shows (*ALL n*. *n* < *size u* −−> *u* !! *n* = *v* !! *n*) ⟹ *u* = *v*
⟨*proof*⟩

lemma *word-eqD*: (*u*::′*a*::*len0 word*) = *v* ⟹ *u* !! *x* = *v* !! *x*
⟨*proof*⟩

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*::′*a*::*len0 word*)) *n* ⟹ *n* < *len-of TYPE*(′*a*)
⟨*proof*⟩

lemma *bin-nth-sint*:
  fixes $w :: 'a::len\ word$
  shows $len\text{-}of\ TYPE('a) \leq n \Longrightarrow$
    $bin\text{-}nth\ (sint\ w)\ n = bin\text{-}nth\ (sint\ w)\ (len\text{-}of\ TYPE('a) - 1)$
  ⟨*proof*⟩

lemma *td-bl*:
  type-definition $(to\text{-}bl :: 'a::len0\ word => bool\ list)$
    of-bl
    $\{bl.\ length\ bl = len\text{-}of\ TYPE('a)\}$
  ⟨*proof*⟩

interpretation *word-bl*:
  type-definition $to\text{-}bl :: 'a::len0\ word => bool\ list$
    of-bl
    $\{bl.\ length\ bl = len\text{-}of\ TYPE('a::len0)\}$
  ⟨*proof*⟩

lemmas *word-bl-Rep'* = *word-bl.Rep* [*unfolded mem-Collect-eq*, *iff*]

lemma *word-size-bl*: $size\ w = size\ (to\text{-}bl\ w)$
  ⟨*proof*⟩

lemma *to-bl-use-of-bl*:
  $(to\text{-}bl\ w = bl) = (w = of\text{-}bl\ bl \wedge length\ bl = length\ (to\text{-}bl\ w))$
  ⟨*proof*⟩

lemma *to-bl-word-rev*: $to\text{-}bl\ (word\text{-}reverse\ w) = rev\ (to\text{-}bl\ w)$
  ⟨*proof*⟩

lemma *word-rev-rev* [*simp*] : $word\text{-}reverse\ (word\text{-}reverse\ w) = w$
  ⟨*proof*⟩

lemma *word-rev-gal*: $word\text{-}reverse\ w = u \Longrightarrow word\text{-}reverse\ u = w$
  ⟨*proof*⟩

lemma *word-rev-gal'*: $u = word\text{-}reverse\ w \Longrightarrow w = word\text{-}reverse\ u$
  ⟨*proof*⟩

lemma *length-bl-gt-0* [*iff*]: $0 < length\ (to\text{-}bl\ (x::'a::len\ word))$
  ⟨*proof*⟩

lemma *bl-not-Nil* [*iff*]: $to\text{-}bl\ (x::'a::len\ word) \neq []$
  ⟨*proof*⟩

lemma *length-bl-neq-0* [*iff*]: $length\ (to\text{-}bl\ (x::'a::len\ word)) \neq 0$
  ⟨*proof*⟩

lemma *hd-bl-sign-sint*: *hd* (*to-bl w*) = (*bin-sign* (*sint w*) = −*1*)
⟨*proof*⟩

lemma *of-bl-drop'*:
lend = *length bl* − *len-of TYPE* (*'a* :: *len0*) ⟹
of-bl (*drop lend bl*) = (*of-bl bl* :: *'a word*)
⟨*proof*⟩

lemma *test-bit-of-bl*:
(*of-bl bl*::*'a*::*len0 word*) !! *n* = (*rev bl* ! *n* ∧ *n* < *len-of TYPE*(*'a*) ∧ *n* < *length bl*)
⟨*proof*⟩

lemma *no-of-bl*:
(*numeral bin* ::*'a*::*len0 word*) = *of-bl* (*bin-to-bl* (*len-of TYPE* (*'a*)) (*numeral bin*))
⟨*proof*⟩

lemma *uint-bl*: *to-bl w* = *bin-to-bl* (*size w*) (*uint w*)
⟨*proof*⟩

lemma *to-bl-bin*: *bl-to-bin* (*to-bl w*) = *uint w*
⟨*proof*⟩

lemma *to-bl-of-bin*:
to-bl (*word-of-int bin*::*'a*::*len0 word*) = *bin-to-bl* (*len-of TYPE*(*'a*)) *bin*
⟨*proof*⟩

lemma *to-bl-numeral* [*simp*]:
to-bl (*numeral bin*::*'a*::*len0 word*) =
bin-to-bl (*len-of TYPE*(*'a*)) (*numeral bin*)
⟨*proof*⟩

lemma *to-bl-neg-numeral* [*simp*]:
to-bl (− *numeral bin*::*'a*::*len0 word*) =
bin-to-bl (*len-of TYPE*(*'a*)) (− *numeral bin*)
⟨*proof*⟩

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

lemma *uint-bl-bin*:
fixes *x* :: *'a*::*len0 word*
shows *bl-to-bin* (*bin-to-bl* (*len-of TYPE*(*'a*)) (*uint x*)) = *uint x*
⟨*proof*⟩

lemma *uints-unats*: *uints n* = *int* ` *unats n*
⟨*proof*⟩

lemma *unats-uints*: *unats n* = *nat* ‘ *uints n*
⟨*proof*⟩

lemmas *bintr-num* = *word-ubin.norm-eq-iff*
[*of numeral a numeral b*, *symmetric*, *folded word-numeral-alt*] **for** *a b*
lemmas *sbintr-num* = *word-sbin.norm-eq-iff*
[*of numeral a numeral b*, *symmetric*, *folded word-numeral-alt*] **for** *a b*

lemma *num-of-bintr'*:
bintrunc (*len-of TYPE*('*a* :: *len0*)) (*numeral a*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: '*a word*)
⟨*proof*⟩

lemma *num-of-sbintr'*:
sbintrunc (*len-of TYPE*('*a* :: *len*) − *1*) (*numeral a*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: '*a word*)
⟨*proof*⟩

lemma *num-abs-bintr*:
(*numeral x* :: '*a word*) =
word-of-int (*bintrunc* (*len-of TYPE*('*a*::*len0*)) (*numeral x*))
⟨*proof*⟩

lemma *num-abs-sbintr*:
(*numeral x* :: '*a word*) =
word-of-int (*sbintrunc* (*len-of TYPE*('*a*::*len*) − *1*) (*numeral x*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *ucast-bintr* [*simp*]:
ucast (*numeral w* ::'*a*::*len0 word*) =
word-of-int (*bintrunc* (*len-of TYPE*('*a*)) (*numeral w*))
⟨*proof*⟩

lemma *scast-sbintr* [*simp*]:
scast (*numeral w* ::′*a*::*len word*) =
word-of-int (*sbintrunc* (*len-of TYPE*(′*a*) − *Suc 0*) (*numeral w*))
⟨*proof*⟩

lemma *source-size*: *source-size* (*c*::′*a*::*len0 word* ⇒ -) = *len-of TYPE*(′*a*)
⟨*proof*⟩

lemma *target-size*: *target-size* (*c*::- ⇒ ′*b*::*len0 word*) = *len-of TYPE*(′*b*)
⟨*proof*⟩

lemma *is-down*:
fixes *c* :: ′*a*::*len0 word* ⇒ ′*b*::*len0 word*
shows *is-down c* ⟷ *len-of TYPE*(′*b*) ≤ *len-of TYPE*(′*a*)
⟨*proof*⟩

lemma *is-up*:
fixes *c* :: ′*a*::*len0 word* ⇒ ′*b*::*len0 word*
shows *is-up c* ⟷ *len-of TYPE*(′*a*) ≤ *len-of TYPE*(′*b*)
⟨*proof*⟩

lemmas *is-up-down* = *trans* [*OF is-up is-down* [*symmetric*]]

lemma *down-cast-same* [*OF refl*]: *uc* = *ucast* ⟹ *is-down uc* ⟹ *uc* = *scast*
⟨*proof*⟩

lemma *word-rev-tf*:
to-bl (*of-bl bl*::′*a*::*len0 word*) =
rev (*takefill False* (*len-of TYPE*(′*a*)) (*rev bl*))
⟨*proof*⟩

lemma *word-rep-drop*:
to-bl (*of-bl bl*::′*a*::*len0 word*) =
replicate (*len-of TYPE*(′*a*) − *length bl*) *False* @
drop (*length bl* − *len-of TYPE*(′*a*)) *bl*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *ucast-up-app* [*OF refl*]:
uc = *ucast* ⟹ *source-size uc* + *n* = *target-size uc* ⟹
to-bl (*uc w*) = *replicate n False* @ (*to-bl w*)
⟨*proof*⟩

lemma *ucast-down-drop* [*OF refl*]:
uc = *ucast* ⟹ *source-size uc* = *target-size uc* + *n* ⟹
to-bl (*uc w*) = *drop n* (*to-bl w*)
⟨*proof*⟩

lemma *scast-down-drop* [*OF refl*]:
sc = *scast* ⟹ *source-size sc* = *target-size sc* + *n* ⟹
to-bl (*sc w*) = *drop n* (*to-bl w*)
⟨*proof*⟩

lemma *sint-up-scast* [*OF refl*]:
sc = *scast* ⟹ *is-up sc* ⟹ *sint* (*sc w*) = *sint w*
⟨*proof*⟩

lemma *uint-up-ucast* [*OF refl*]:
uc = *ucast* ⟹ *is-up uc* ⟹ *uint* (*uc w*) = *uint w*
⟨*proof*⟩

lemma *ucast-up-ucast* [*OF refl*]:
uc = *ucast* ⟹ *is-up uc* ⟹ *ucast* (*uc w*) = *ucast w*
⟨*proof*⟩

lemma *scast-up-scast* [*OF refl*]:
sc = *scast* ⟹ *is-up sc* ⟹ *scast* (*sc w*) = *scast w*
⟨*proof*⟩

lemma *ucast-of-bl-up* [*OF refl*]:
w = *of-bl bl* ⟹ *size bl* <= *size w* ⟹ *ucast w* = *of-bl bl*
⟨*proof*⟩

lemmas *ucast-up-ucast-id* = *trans* [*OF ucast-up-ucast ucast-id*]
lemmas *scast-up-scast-id* = *trans* [*OF scast-up-scast scast-id*]

lemmas *isduu* = *is-up-down* [**where** *c* = *ucast*, *THEN iffD2*]
lemmas *isdus* = *is-up-down* [**where** *c* = *scast*, *THEN iffD2*]
lemmas *ucast-down-ucast-id* = *isduu* [*THEN ucast-up-ucast-id*]
lemmas *scast-down-scast-id* = *isdus* [*THEN ucast-up-ucast-id*]

lemma *up-ucast-surj*:
is-up (*ucast* :: *'b::len0 word* => *'a::len0 word*) ⟹
surj (*ucast* :: *'a word* => *'b word*)
⟨*proof*⟩

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*) $\Longrightarrow$
inj-on (*ucast* :: *'a word* => *'b word*) *A*
⟨*proof*⟩

lemma *down-ucast-inj*:
is-down (*ucast* :: *'b*::*len0 word* => *'a*::*len0 word*) $\Longrightarrow$
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-wi* [*OF refl*]:
uc = *ucast* $\Longrightarrow$ *is-down uc* $\Longrightarrow$ *uc* (*word-of-int x*) = *word-of-int x*
⟨*proof*⟩

lemma *ucast-down-no* [*OF refl*]:
uc = *ucast* $\Longrightarrow$ *is-down uc* $\Longrightarrow$ *uc* (*numeral bin*) = *numeral bin*
⟨*proof*⟩

lemma *ucast-down-bl* [*OF refl*]:
uc = *ucast* $\Longrightarrow$ *is-down uc* $\Longrightarrow$ *uc* (*of-bl bl*) = *of-bl bl*
⟨*proof*⟩

lemmas *slice-def'* = *slice-def* [*unfolded word-size*]
lemmas *test-bit-def'* = *word-test-bit-def* [*THEN fun-cong*]

lemmas *word-log-defs* = *word-and-def word-or-def word-xor-def word-not-def*

16.14 Word Arithmetic

lemma *word-less-alt*: (*a* < *b*) = (*uint a* < *uint b*)
⟨*proof*⟩

lemma *signed-linorder*: *class.linorder word-sle word-sless*
⟨*proof*⟩

interpretation *signed*: *linorder word-sle word-sless*
⟨*proof*⟩

lemma *udvdI*:
$0 \leq n \Longrightarrow$ *uint b* = *n* ∗ *uint a* $\Longrightarrow$ *a udvd b*
⟨*proof*⟩

lemmas *word-div-no* [*simp*] = *word-div-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-mod-no* [*simp*] = *word-mod-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-less-no* [*simp*] = *word-less-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-le-no* [*simp*] = *word-le-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-sless-no* [*simp*] = *word-sless-def* [*of numeral a numeral b*] **for** *a b*

lemmas *word-sle-no* [*simp*] = *word-sle-def* [*of numeral a numeral b*] **for** *a b*

lemma *word-m1-wi*: − *1* = *word-of-int* (− *1*)
⟨*proof*⟩

lemma *word-0-bl* [*simp*]: *of-bl* [] = *0*
⟨*proof*⟩

lemma *word-1-bl*: *of-bl* [*True*] = *1*
⟨*proof*⟩

lemma *uint-eq-0* [*simp*]: *uint 0* = *0*
⟨*proof*⟩

lemma *of-bl-0* [*simp*]: *of-bl* (*replicate n False*) = *0*
⟨*proof*⟩

lemma *to-bl-0* [*simp*]:
to-bl (*0*::'*a*::*len0 word*) = *replicate* (*len-of TYPE*('*a*)) *False*
⟨*proof*⟩

lemma *uint-0-iff*:
uint x = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *unat-0-iff*:
unat x = *0* ⟷ *x* = *0*
⟨*proof*⟩

lemma *unat-0* [*simp*]:
unat 0 = *0*
⟨*proof*⟩

lemma *size-0-same'*:
size w = *0* ⟹ *w* = (*v* :: '*a* :: *len0 word*)
⟨*proof*⟩

lemmas *size-0-same* = *size-0-same'* [*unfolded word-size*]

lemmas *unat-eq-0* = *unat-0-iff*
lemmas *unat-eq-zero* = *unat-0-iff*

lemma *unat-gt-0*: (*0* < *unat x*) = (*x* ~= *0*)

⟨*proof*⟩

lemma *ucast-0* [*simp*]: *ucast 0 = 0*
⟨*proof*⟩

lemma *sint-0* [*simp*]: *sint 0 = 0*
⟨*proof*⟩

lemma *scast-0* [*simp*]: *scast 0 = 0*
⟨*proof*⟩

lemma *sint-n1* [*simp*] : *sint* (− *1*) = − *1*
⟨*proof*⟩

lemma *scast-n1* [*simp*]: *scast* (− *1*) = − *1*
⟨*proof*⟩

lemma *uint-1* [*simp*]: *uint* (*1*::′*a*::*len word*) = *1*
⟨*proof*⟩

lemma *unat-1* [*simp*]: *unat* (*1*::′*a*::*len word*) = *1*
⟨*proof*⟩

lemma *ucast-1* [*simp*]: *ucast* (*1*::′*a*::*len word*) = *1*
⟨*proof*⟩

16.15 Transferring goals from words to ints

lemma *word-ths*:
shows
word-succ-p1: *word-succ a* = *a* + *1* **and**
word-pred-m1: *word-pred a* = *a* − *1* **and**
word-pred-succ: *word-pred* (*word-succ a*) = *a* **and**
word-succ-pred: *word-succ* (*word-pred a*) = *a* **and**
word-mult-succ: *word-succ a* ∗ *b* = *b* + *a* ∗ *b*
⟨*proof*⟩

lemma *uint-cong*: *x* = *y* ⟹ *uint x* = *uint y*
⟨*proof*⟩

lemma *uint-word-ariths*:
fixes *a b* :: ′*a*::*len0 word*
shows *uint* (*a* + *b*) = (*uint a* + *uint b*) *mod 2* ˆ *len-of TYPE*(′*a*::*len0*)
and *uint* (*a* − *b*) = (*uint a* − *uint b*) *mod 2* ˆ *len-of TYPE*(′*a*)
and *uint* (*a* ∗ *b*) = *uint a* ∗ *uint b mod 2* ˆ *len-of TYPE*(′*a*)
and *uint* (− *a*) = − *uint a mod 2* ˆ *len-of TYPE*(′*a*)
and *uint* (*word-succ a*) = (*uint a* + *1*) *mod 2* ˆ *len-of TYPE*(′*a*)
and *uint* (*word-pred a*) = (*uint a* − *1*) *mod 2* ˆ *len-of TYPE*(′*a*)
and *uint* (*0* :: ′*a word*) = *0 mod 2* ˆ *len-of TYPE*(′*a*)

and *uint* (*1* :: *'a word*) = *1 mod 2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemma *uint-word-arith-bintrs*:
fixes *a b* :: *'a*::*len0 word*
shows *uint* (*a* + *b*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* + *uint b*)
and *uint* (*a* − *b*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* − *uint b*)
and *uint* (*a* ∗ *b*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* ∗ *uint b*)
and *uint* (− *a*) = *bintrunc* (*len-of TYPE*(*'a*)) (− *uint a*)
and *uint* (*word-succ a*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* + *1*)
and *uint* (*word-pred a*) = *bintrunc* (*len-of TYPE*(*'a*)) (*uint a* − *1*)
and *uint* (*0* :: *'a word*) = *bintrunc* (*len-of TYPE*(*'a*)) *0*
and *uint* (*1* :: *'a word*) = *bintrunc* (*len-of TYPE*(*'a*)) *1*
⟨*proof*⟩

lemma *sint-word-ariths*:
fixes *a b* :: *'a*::*len word*
shows *sint* (*a* + *b*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* + *sint b*)
and *sint* (*a* − *b*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* − *sint b*)
and *sint* (*a* ∗ *b*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* ∗ *sint b*)
and *sint* (− *a*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (− *sint a*)
and *sint* (*word-succ a*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* + *1*)
and *sint* (*word-pred a*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*sint a* − *1*)
and *sint* (*0* :: *'a word*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) *0*
and *sint* (*1* :: *'a word*) = *sbintrunc* (*len-of TYPE*(*'a*) − *1*) *1*
⟨*proof*⟩

lemmas *uint-div-alt* = *word-div-def* [*THEN trans* [*OF uint-cong int-word-uint*]]
lemmas *uint-mod-alt* = *word-mod-def* [*THEN trans* [*OF uint-cong int-word-uint*]]

lemma *word-pred-0-n1*: *word-pred 0* = *word-of-int* (− *1*)
⟨*proof*⟩

lemma *succ-pred-no* [*simp*]:
word-succ (*numeral w*) = *numeral w* + *1*
word-pred (*numeral w*) = *numeral w* − *1*
word-succ (− *numeral w*) = − *numeral w* + *1*
word-pred (− *numeral w*) = − *numeral w* − *1*
⟨*proof*⟩

lemma *word-sp-01* [*simp*] :
word-succ (− *1*) = *0* & *word-succ 0* = *1* & *word-pred 0* = − *1* & *word-pred 1* = *0*
⟨*proof*⟩

lemma *word-of-int-Ex*:
∃ *y*. *x* = *word-of-int y*
⟨*proof*⟩

16.16 Order on fixed-length words

lemma *word-zero-le* [*simp*] :
0 <= (y :: 'a :: len0 word)
⟨*proof*⟩

lemma *word-m1-ge* [*simp*] : *word-pred 0 >= y*
⟨*proof*⟩

lemma *word-n1-ge* [*simp*]: $y \le (-1{::}'a{::}len0\ word)$
⟨*proof*⟩

lemmas *word-not-simps* [*simp*] =
word-zero-le [*THEN leD*] *word-m1-ge* [*THEN leD*] *word-n1-ge* [*THEN leD*]

lemma *word-gt-0*: $0 < y \longleftrightarrow 0 \neq (y :: 'a :: len0\ word)$
⟨*proof*⟩

lemmas *word-gt-0-no* [*simp*] = *word-gt-0* [*of numeral y*] **for** *y*

lemma *word-sless-alt*: (*a <s b*) = (*sint a* < *sint b*)
⟨*proof*⟩

lemma *word-le-nat-alt*: (*a <= b*) = (*unat a <= unat b*)
⟨*proof*⟩

lemma *word-less-nat-alt*: (*a < b*) = (*unat a < unat b*)
⟨*proof*⟩

lemma *wi-less*:
(*word-of-int n* < (*word-of-int m* :: *'a* :: *len0 word*)) =
(*n mod 2 ^ len-of TYPE('a)* < *m mod 2 ^ len-of TYPE('a)*)
⟨*proof*⟩

lemma *wi-le*:
(*word-of-int n* <= (*word-of-int m* :: *'a* :: *len0 word*)) =
(*n mod 2 ^ len-of TYPE('a)* <= *m mod 2 ^ len-of TYPE('a)*)
⟨*proof*⟩

lemma *udvd-nat-alt*: *a udvd b* = (*EX n>=0. unat b = n ∗ unat a*)
⟨*proof*⟩

lemma *udvd-iff-dvd*: *x udvd y* ⟷ *unat x dvd unat y*
⟨*proof*⟩

lemmas *unat-mono* = *word-less-nat-alt* [*THEN iffD1*]

lemma *unat-minus-one*:
assumes $w \neq 0$
shows *unat* $(w - 1) =$ *unat* $w - 1$

⟨*proof*⟩

lemma *measure-unat*: *p* ~= *0* ⟹ *unat* (*p* − *1*) < *unat p*
⟨*proof*⟩

lemmas *uint-add-ge0* [*simp*] = *add-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]
lemmas *uint-mult-ge0* [*simp*] = *mult-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]

lemma *uint-sub-lt2p* [*simp*]:
uint (*x* :: *'a* :: *len0 word*) − *uint* (*y* :: *'b* :: *len0 word*) <
2 ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

16.17 Conditions for the addition (etc) of two words to overflow

lemma *uint-add-lem*:
(*uint x* + *uint y* < *2* ^ *len-of TYPE*(*'a*)) =
(*uint* (*x* + *y* :: *'a* :: *len0 word*) = *uint x* + *uint y*)
⟨*proof*⟩

lemma *uint-mult-lem*:
(*uint x* * *uint y* < *2* ^ *len-of TYPE*(*'a*)) =
(*uint* (*x* * *y* :: *'a* :: *len0 word*) = *uint x* * *uint y*)
⟨*proof*⟩

lemma *uint-sub-lem*:
(*uint x* >= *uint y*) = (*uint* (*x* − *y*) = *uint x* − *uint y*)
⟨*proof*⟩

lemma *uint-add-le*: *uint* (*x* + *y*) <= *uint x* + *uint y*
⟨*proof*⟩

lemma *uint-sub-ge*: *uint* (*x* − *y*) >= *uint x* − *uint y*
⟨*proof*⟩

lemma *mod-add-if-z*:
(*x* :: *int*) < *z* ==> *y* < *z* ==> *0* <= *y* ==> *0* <= *x* ==> *0* <= *z* ==>
(*x* + *y*) *mod z* = (*if x* + *y* < *z then x* + *y else x* + *y* − *z*)
⟨*proof*⟩

lemma *uint-plus-if'*:
uint ((*a*::*'a word*) + *b*) =
(*if uint a* + *uint b* < *2* ^ *len-of TYPE*(*'a*::*len0*) *then uint a* + *uint b*
else uint a + *uint b* − *2* ^ *len-of TYPE*(*'a*))
⟨*proof*⟩

lemma *mod-sub-if-z*:
(*x* :: *int*) < *z* ==> *y* < *z* ==> *0* <= *y* ==> *0* <= *x* ==> *0* <= *z* ==>

(x − y) mod z = (if y <= x then x − y else x − y + z)
⟨*proof*⟩

lemma *uint-sub-if′*:
uint ((a::′a word) − b) =
(if uint b ≤ uint a then uint a − uint b
else uint a − uint b + 2 ^ len-of TYPE(′a::len0))
⟨*proof*⟩

16.18 Definition of *uint-arith*

lemma *word-of-int-inverse*:
word-of-int r = a ⟹ 0 <= r ⟹ r < 2 ^ len-of TYPE(′a) ⟹
uint (a::′a::len0 word) = r
⟨*proof*⟩

lemma *uint-split*:
fixes *x::′a::len0 word*
shows *P (uint x) =*
(ALL i. word-of-int i = x & 0 <= i & i < 2^len-of TYPE(′a) −−> P i)
⟨*proof*⟩

lemma *uint-split-asm*:
fixes *x::′a::len0 word*
shows *P (uint x) =*
(~(EX i. word-of-int i = x & 0 <= i & i < 2^len-of TYPE(′a) & ~ P i))
⟨*proof*⟩

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

lemmas *uint-arith-simps =*
word-le-def word-less-alt
word-uint.Rep-inject [symmetric]
uint-sub-if′ uint-plus-if′

lemma *power-False-cong*: *False ⟹ a ^ b = c ^ d*
⟨*proof*⟩

⟨*ML*⟩

16.19 More on overflows and monotonicity

lemma *no-plus-overflow-uint-size*:
((x :: ′a :: len0 word) <= x + y) = (uint x + uint y < 2 ^ size x)
⟨*proof*⟩

lemmas *no-olen-add = no-plus-overflow-uint-size [unfolded word-size]*

lemma *no-ulen-sub*: $((x :: 'a :: len0\ word) >= x - y) = (uint\ y <= uint\ x)$
⟨*proof*⟩

lemma *no-olen-add'*:
fixes $x :: 'a::len0\ word$
shows $(x \leq y + x) = (uint\ y + uint\ x < 2\ \hat{}\ len\text{-}of\ TYPE('a))$
⟨*proof*⟩

lemmas *olen-add-eqv* = *trans* [*OF no-olen-add no-olen-add'* [*symmetric*]]

lemmas *uint-plus-simple-iff* = *trans* [*OF no-olen-add uint-add-lem*]
lemmas *uint-plus-simple* = *uint-plus-simple-iff* [*THEN iffD1*]
lemmas *uint-minus-simple-iff* = *trans* [*OF no-ulen-sub uint-sub-lem*]
lemmas *uint-minus-simple-alt* = *uint-sub-lem* [*folded word-le-def*]
lemmas *word-sub-le-iff* = *no-ulen-sub* [*folded word-le-def*]
lemmas *word-sub-le* = *word-sub-le-iff* [*THEN iffD2*]

lemma *word-less-sub1*:
$(x :: 'a :: len\ word)\ \tilde{}= 0 \Longrightarrow (1 < x) = (0 < x - 1)$
⟨*proof*⟩

lemma *word-le-sub1*:
$(x :: 'a :: len\ word)\ \tilde{}= 0 \Longrightarrow (1 <= x) = (0 <= x - 1)$
⟨*proof*⟩

lemma *sub-wrap-lt*:
$((x :: 'a :: len0\ word) < x - z) = (x < z)$
⟨*proof*⟩

lemma *sub-wrap*:
$((x :: 'a :: len0\ word) <= x - z) = (z = 0 \mid x < z)$
⟨*proof*⟩

lemma *plus-minus-not-NULL-ab*:
$(x :: 'a :: len0\ word) <= ab - c \Longrightarrow c <= ab \Longrightarrow c\ \tilde{}= 0 \Longrightarrow x + c\ \tilde{}= 0$
⟨*proof*⟩

lemma *plus-minus-no-overflow-ab*:
$(x :: 'a :: len0\ word) <= ab - c \Longrightarrow c <= ab \Longrightarrow x <= x + c$
⟨*proof*⟩

lemma *le-minus'*:
$(a :: 'a :: len0\ word) + c <= b \Longrightarrow a <= a + c \Longrightarrow c <= b - a$
⟨*proof*⟩

lemma *le-plus'*:
$(a :: 'a :: len0\ word) <= b \Longrightarrow c <= b - a \Longrightarrow a + c <= b$
⟨*proof*⟩

lemmas *le-plus* = *le-plus'* [*rotated*]

lemmas *le-minus* = *leD* [*THEN thin-rl*, *THEN le-minus'*]

lemma *word-plus-mono-right*:
$(y :: 'a :: len0\ word) <= z \Longrightarrow x <= x + z \Longrightarrow x + y <= x + z$
⟨*proof*⟩

lemma *word-less-minus-cancel*:
$y - x < z - x \Longrightarrow x <= z \Longrightarrow (y :: 'a :: len0\ word) < z$
⟨*proof*⟩

lemma *word-less-minus-mono-left*:
$(y :: 'a :: len0\ word) < z \Longrightarrow x <= y \Longrightarrow y - x < z - x$
⟨*proof*⟩

lemma *word-less-minus-mono*:
$a < c \Longrightarrow d < b \Longrightarrow a - b < a \Longrightarrow c - d < c$
$\Longrightarrow a - b < c - (d::'a::len\ word)$
⟨*proof*⟩

lemma *word-le-minus-cancel*:
$y - x <= z - x \Longrightarrow x <= z \Longrightarrow (y :: 'a :: len0\ word) <= z$
⟨*proof*⟩

lemma *word-le-minus-mono-left*:
$(y :: 'a :: len0\ word) <= z \Longrightarrow x <= y \Longrightarrow y - x <= z - x$
⟨*proof*⟩

lemma *word-le-minus-mono*:
$a <= c \Longrightarrow d <= b \Longrightarrow a - b <= a \Longrightarrow c - d <= c$
$\Longrightarrow a - b <= c - (d::'a::len\ word)$
⟨*proof*⟩

lemma *plus-le-left-cancel-wrap*:
$(x :: 'a :: len0\ word) + y' < x \Longrightarrow x + y < x \Longrightarrow (x + y' < x + y) = (y' < y)$
⟨*proof*⟩

lemma *plus-le-left-cancel-nowrap*:
$(x :: 'a :: len0\ word) <= x + y' \Longrightarrow x <= x + y \Longrightarrow$
$(x + y' < x + y) = (y' < y)$
⟨*proof*⟩

lemma *word-plus-mono-right2*:
$(a :: 'a :: len0\ word) <= a + b \Longrightarrow c <= b \Longrightarrow a <= a + c$
⟨*proof*⟩

lemma *word-less-add-right*:
$(x :: 'a :: len0\ word) < y - z \Longrightarrow z <= y \Longrightarrow x + z < y$

⟨*proof*⟩

lemma *word-less-sub-right*:
$(x :: 'a :: len0\ word) < y + z \Longrightarrow y <= x \Longrightarrow x - y < z$
⟨*proof*⟩

lemma *word-le-plus-either*:
$(x :: 'a :: len0\ word) <= y \mid x <= z \Longrightarrow y <= y + z \Longrightarrow x <= y + z$
⟨*proof*⟩

lemma *word-less-nowrapI*:
$(x :: 'a :: len0\ word) < z - k \Longrightarrow k <= z \Longrightarrow 0 < k \Longrightarrow x < x + k$
⟨*proof*⟩

lemma *inc-le*: $(i :: 'a :: len\ word) < m \Longrightarrow i + 1 <= m$
⟨*proof*⟩

lemma *inc-i*:
$(1 :: 'a :: len\ word) <= i \Longrightarrow i < m \Longrightarrow 1 <= (i + 1)\ \&\ i + 1 <= m$
⟨*proof*⟩

lemma *udvd-incr-lem*:
$up < uq \Longrightarrow up = ua + n * uint\ K \Longrightarrow$
$uq = ua + n' * uint\ K \Longrightarrow up + uint\ K <= uq$
⟨*proof*⟩

lemma *udvd-incr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
$uint\ q = ua + n' * uint\ K \Longrightarrow p + K <= q$
⟨*proof*⟩

lemma *udvd-decr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
$uint\ q = ua + n' * uint\ K \Longrightarrow p <= q - K$
⟨*proof*⟩

lemmas *udvd-incr-lem0* = *udvd-incr-lem* [**where** *ua=0*, *unfolded add-0-left*]
lemmas *udvd-incr0* = *udvd-incr'* [**where** *ua=0*, *unfolded add-0-left*]
lemmas *udvd-decr0* = *udvd-decr'* [**where** *ua=0*, *unfolded add-0-left*]

lemma *udvd-minus-le'*:
$xy < k \Longrightarrow z\ udvd\ xy \Longrightarrow z\ udvd\ k \Longrightarrow xy <= k - z$
⟨*proof*⟩

lemma *udvd-incr2-K*:
$p < a + s \Longrightarrow a <= a + s \Longrightarrow K\ udvd\ s \Longrightarrow K\ udvd\ p - a \Longrightarrow a <= p \Longrightarrow$
$0 < K \Longrightarrow p <= p + K\ \&\ p + K <= a + s$
⟨*proof*⟩

lemma *word-succ-rbl*:
to-bl w = bl ⟹ to-bl (word-succ w) = (rev (rbl-succ (rev bl)))
⟨*proof*⟩

lemma *word-pred-rbl*:
to-bl w = bl ⟹ to-bl (word-pred w) = (rev (rbl-pred (rev bl)))
⟨*proof*⟩

lemma *word-add-rbl*:
to-bl v = vbl ⟹ to-bl w = wbl ⟹
to-bl (v + w) = (rev (rbl-add (rev vbl) (rev wbl)))
⟨*proof*⟩

lemma *word-mult-rbl*:
to-bl v = vbl ⟹ to-bl w = wbl ⟹
*to-bl (v * w) = (rev (rbl-mult (rev vbl) (rev wbl)))*
⟨*proof*⟩

lemma *rtb-rbl-ariths*:
rev (to-bl w) = ys ⟹ rev (to-bl (word-succ w)) = rbl-succ ys
rev (to-bl w) = ys ⟹ rev (to-bl (word-pred w)) = rbl-pred ys
*rev (to-bl v) = ys ⟹ rev (to-bl w) = xs ⟹ rev (to-bl (v * w)) = rbl-mult ys xs*
rev (to-bl v) = ys ⟹ rev (to-bl w) = xs ⟹ rev (to-bl (v + w)) = rbl-add ys xs
⟨*proof*⟩

16.20 Arithmetic type class instantiations

lemmas *word-le-0-iff* [*simp*] =
word-zero-le [*THEN leD*, *THEN linorder-antisym-conv1*]

lemma *word-of-int-nat*:
0 <= x ⟹ word-of-int x = of-nat (nat x)
⟨*proof*⟩

lemma *iszero-word-no* [*simp*]:
iszero (numeral bin :: 'a :: len word) =
iszero (bintrunc (len-of TYPE('a)) (numeral bin))
⟨*proof*⟩

Use *iszero* to simplify equalities between word numerals.

lemmas *word-eq-numeral-iff-iszero* [*simp*] =
eq-numeral-iff-iszero [**where** *'a='a::len word*]

16.21 Word and nat

lemma *td-ext-unat* [*OF refl*]:

n = len-of TYPE ('a :: len) ⟹
td-ext (unat :: 'a word => nat) of-nat
(unats n) (%i. i mod 2 ^ n)
⟨*proof*⟩

lemmas *unat-of-nat = td-ext-unat* [*THEN td-ext.eq-norm*]

interpretation *word-unat*:
td-ext unat::'a::len word => nat
of-nat
unats (len-of TYPE('a::len))
%i. i mod 2 ^ len-of TYPE('a::len)
⟨*proof*⟩

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

lemmas *unat-lt2p* [*iff*] = *word-unat.Rep* [*unfolded unats-def mem-Collect-eq*]

lemma *unat-le*: *y <= unat (z :: 'a :: len word) ⟹ y : unats (len-of TYPE ('a))*
⟨*proof*⟩

lemma *word-nchotomy*:
ALL w. EX n. (w :: 'a :: len word) = of-nat n & n < 2 ^ len-of TYPE ('a)
⟨*proof*⟩

lemma *of-nat-eq*:
fixes *w :: 'a::len word*
shows *(of-nat n = w) = (∃ q. n = unat w + q ∗ 2 ^ len-of TYPE('a))*
⟨*proof*⟩

lemma *of-nat-eq-size*:
(of-nat n = w) = (EX q. n = unat w + q ∗ 2 ^ size w)
⟨*proof*⟩

lemma *of-nat-0*:
(of-nat m = (0::'a::len word)) = (∃ q. m = q ∗ 2 ^ len-of TYPE('a))
⟨*proof*⟩

lemma *of-nat-2p* [*simp*]:
of-nat (2 ^ len-of TYPE('a)) = (0::'a::len word)
⟨*proof*⟩

lemma *of-nat-gt-0*: *of-nat k ~= 0 ⟹ 0 < k*
⟨*proof*⟩

lemma *of-nat-neq-0*:
0 < k ⟹ k < 2 ^ len-of TYPE ('a :: len) ⟹ of-nat k ~= (0 :: 'a word)
⟨*proof*⟩

lemma *Abs-fnat-hom-add*:
of-nat a + of-nat b = of-nat (a + b)
⟨*proof*⟩

lemma *Abs-fnat-hom-mult*:
*of-nat a * of-nat b = (of-nat (a * b) :: 'a :: len word)*
⟨*proof*⟩

lemma *Abs-fnat-hom-Suc*:
word-succ (of-nat a) = of-nat (Suc a)
⟨*proof*⟩

lemma *Abs-fnat-hom-0*: *(0::'a::len word) = of-nat 0*
⟨*proof*⟩

lemma *Abs-fnat-hom-1*: *(1::'a::len word) = of-nat (Suc 0)*
⟨*proof*⟩

lemmas *Abs-fnat-homs =*
Abs-fnat-hom-add Abs-fnat-hom-mult Abs-fnat-hom-Suc
Abs-fnat-hom-0 Abs-fnat-hom-1

lemma *word-arith-nat-add*:
a + b = of-nat (unat a + unat b)
⟨*proof*⟩

lemma *word-arith-nat-mult*:
*a * b = of-nat (unat a * unat b)*
⟨*proof*⟩

lemma *word-arith-nat-Suc*:
word-succ a = of-nat (Suc (unat a))
⟨*proof*⟩

lemma *word-arith-nat-div*:
a div b = of-nat (unat a div unat b)
⟨*proof*⟩

lemma *word-arith-nat-mod*:
a mod b = of-nat (unat a mod unat b)
⟨*proof*⟩

lemmas *word-arith-nat-defs =*
word-arith-nat-add word-arith-nat-mult
word-arith-nat-Suc Abs-fnat-hom-0
Abs-fnat-hom-1 word-arith-nat-div
word-arith-nat-mod

lemma *unat-cong*: *x = y ⟹ unat x = unat y*

⟨*proof*⟩

lemmas *unat-word-ariths* = *word-arith-nat-defs*
[*THEN trans* [*OF unat-cong unat-of-nat*]]

lemmas *word-sub-less-iff* = *word-sub-le-iff*
[*unfolded linorder-not-less* [*symmetric*] *Not-eq-iff*]

lemma *unat-add-lem*:
(*unat x* + *unat y* < *2* ^ *len-of TYPE*(*'a*)) =
(*unat* (*x* + *y* :: *'a* :: *len word*) = *unat x* + *unat y*)
⟨*proof*⟩

lemma *unat-mult-lem*:
(*unat x* ∗ *unat y* < *2* ^ *len-of TYPE*(*'a*)) =
(*unat* (*x* ∗ *y* :: *'a* :: *len word*) = *unat x* ∗ *unat y*)
⟨*proof*⟩

lemmas *unat-plus-if'* = *trans* [*OF unat-word-ariths*(*1*) *mod-nat-add*, *simplified*]

lemma *le-no-overflow*:
x <= *b* ⟹ *a* <= *a* + *b* ⟹ *x* <= *a* + (*b* :: *'a* :: *len0 word*)
⟨*proof*⟩

lemmas *un-ui-le* = *trans* [*OF word-le-nat-alt* [*symmetric*] *word-le-def*]

lemma *unat-sub-if-size*:
unat (*x* − *y*) = (*if unat y* <= *unat x*
then unat x − *unat y*
else unat x + *2* ^ *size x* − *unat y*)
⟨*proof*⟩

lemmas *unat-sub-if'* = *unat-sub-if-size* [*unfolded word-size*]

lemma *unat-div*: *unat* ((*x* :: *'a* :: *len word*) *div y*) = *unat x div unat y*
⟨*proof*⟩

lemma *unat-mod*: *unat* ((*x* :: *'a* :: *len word*) *mod y*) = *unat x mod unat y*
⟨*proof*⟩

lemma *uint-div*: *uint* ((*x* :: *'a* :: *len word*) *div y*) = *uint x div uint y*
⟨*proof*⟩

lemma *uint-mod*: *uint* ((*x* :: *'a* :: *len word*) *mod y*) = *uint x mod uint y*
⟨*proof*⟩

16.22 Definition of *unat-arith* tactic

lemma *unat-split*:

fixes *x*::*'a*::*len word*
shows *P* (*unat x*) =
(*ALL n. of-nat n* = *x* & *n* < *2^len-of TYPE*(*'a*) --> *P n*)
⟨*proof*⟩

lemma *unat-split-asm*:
fixes *x*::*'a*::*len word*
shows *P* (*unat x*) =
(~(*EX n. of-nat n* = *x* & *n* < *2^len-of TYPE*(*'a*) & ~ *P n*))
⟨*proof*⟩

lemmas *of-nat-inverse* =
word-unat.Abs-inverse' [*rotated*, *unfolded unats-def*, *simplified*]

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

lemmas *unat-arith-simps* =
word-le-nat-alt word-less-nat-alt
word-unat.Rep-inject [*symmetric*]
unat-sub-if' unat-plus-if' unat-div unat-mod

⟨*ML*⟩

lemma *no-plus-overflow-unat-size*:
((*x* :: *'a* :: *len word*) <= *x* + *y*) = (*unat x* + *unat y* < *2* ^ *size x*)
⟨*proof*⟩

lemmas *no-olen-add-nat* = *no-plus-overflow-unat-size* [*unfolded word-size*]

lemmas *unat-plus-simple* = *trans* [*OF no-olen-add-nat unat-add-lem*]

lemma *word-div-mult*:
(*0* :: *'a* :: *len word*) < *y* ⟹ *unat x* ∗ *unat y* < *2* ^ *len-of TYPE*(*'a*) ⟹
x ∗ *y div y* = *x*
⟨*proof*⟩

lemma *div-lt'*: (*i* :: *'a* :: *len word*) <= *k div x* ⟹
unat i ∗ *unat x* < *2* ^ *len-of TYPE*(*'a*)
⟨*proof*⟩

lemmas *div-lt''* = *order-less-imp-le* [*THEN div-lt'*]

lemma *div-lt-mult*: (*i* :: *'a* :: *len word*) < *k div x* ⟹ *0* < *x* ⟹ *i* ∗ *x* < *k*
⟨*proof*⟩

lemma *div-le-mult*:
(*i* :: *'a* :: *len word*) <= *k div x* ⟹ *0* < *x* ⟹ *i* ∗ *x* <= *k*
⟨*proof*⟩

lemma *div-lt-uint'*:
  *(i :: 'a :: len word) <= k div x ⟹ uint i * uint x < 2 ^ len-of TYPE('a)*
  ⟨*proof*⟩

lemmas *div-lt-uint''* = *order-less-imp-le* [*THEN div-lt-uint'*]

lemma *word-le-exists'*:
  (x :: 'a :: len0 word) <= y ⟹
    (EX z. y = x + z & uint x + uint z < 2 ^ len-of TYPE('a))
  ⟨*proof*⟩

lemmas *plus-minus-not-NULL* = *order-less-imp-le* [*THEN plus-minus-not-NULL-ab*]

lemmas *plus-minus-no-overflow* =
  order-less-imp-le [*THEN plus-minus-no-overflow-ab*]

lemmas *mcs* = *word-less-minus-cancel word-less-minus-mono-left*
  word-le-minus-cancel word-le-minus-mono-left

lemmas *word-l-diffs* = *mcs* [**where** *y* = *w* + *x*, *unfolded add-diff-cancel*] **for** *w x*
lemmas *word-diff-ls* = *mcs* [**where** *z* = *w* + *x*, *unfolded add-diff-cancel*] **for** *w x*
lemmas *word-plus-mcs* = *word-diff-ls* [**where** *y* = *v* + *x*, *unfolded add-diff-cancel*]
for *v x*

lemmas *le-unat-uoi* = *unat-le* [*THEN word-unat.Abs-inverse*]

lemmas *thd* = *refl* [*THEN* [*2*] *split-div-lemma* [*THEN iffD2*], *THEN conjunct1*]

lemmas *uno-simps* [*THEN le-unat-uoi*] = *mod-le-divisor div-le-dividend dtle*

lemma *word-mod-div-equality*:
  *(n div b) * b + (n mod b) = (n :: 'a :: len word)*
  ⟨*proof*⟩

lemma *word-div-mult-le*: *a div b * b <= (a::'a::len word)*
  ⟨*proof*⟩

lemma *word-mod-less-divisor*: *0 < n ⟹ m mod n < (n :: 'a :: len word)*
  ⟨*proof*⟩

lemma *word-of-int-power-hom*:
  word-of-int a ^ n = (word-of-int (a ^ n) :: 'a :: len word)
  ⟨*proof*⟩

lemma *word-arith-power-alt*:
  a ^ n = (word-of-int (uint a ^ n) :: 'a :: len word)
  ⟨*proof*⟩

lemma *of-bl-length-less*:
length x = k ⟹ k < len-of TYPE('a) ⟹ (of-bl x :: 'a :: len word) < 2 ^ k
⟨*proof*⟩

16.23 Cardinality, finiteness of set of words

instance *word* :: (*len0*) *finite*
⟨*proof*⟩

lemma *card-word*: *CARD('a::len0 word) = 2 ^ len-of TYPE('a)*
⟨*proof*⟩

lemma *card-word-size*:
card (UNIV :: 'a :: len0 word set) = (2 ^ size (x :: 'a word))
⟨*proof*⟩

16.24 Bitwise Operations on Words

lemmas *bin-log-bintrs = bin-trunc-not bin-trunc-xor bin-trunc-and bin-trunc-or*

lemmas *wils1 = bin-log-bintrs [THEN word-ubin.norm-eq-iff [THEN iffD1], folded word-ubin.eq-norm, THEN eq-reflection]*

lemmas *word-log-binary-defs =*
word-and-def word-or-def word-xor-def

lemma *word-wi-log-defs*:
NOT word-of-int a = word-of-int (NOT a)
word-of-int a AND word-of-int b = word-of-int (a AND b)
word-of-int a OR word-of-int b = word-of-int (a OR b)
word-of-int a XOR word-of-int b = word-of-int (a XOR b)
⟨*proof*⟩

lemma *word-no-log-defs* [*simp*]:
NOT (numeral a) = word-of-int (NOT (numeral a))
NOT (− numeral a) = word-of-int (NOT (− numeral a))
numeral a AND numeral b = word-of-int (numeral a AND numeral b)
numeral a AND − numeral b = word-of-int (numeral a AND − numeral b)
− numeral a AND numeral b = word-of-int (− numeral a AND numeral b)
− numeral a AND − numeral b = word-of-int (− numeral a AND − numeral b)
numeral a OR numeral b = word-of-int (numeral a OR numeral b)
numeral a OR − numeral b = word-of-int (numeral a OR − numeral b)
− numeral a OR numeral b = word-of-int (− numeral a OR numeral b)
− numeral a OR − numeral b = word-of-int (− numeral a OR − numeral b)
numeral a XOR numeral b = word-of-int (numeral a XOR numeral b)

numeral a XOR $-$ *numeral b* $=$ *word-of-int* (*numeral a XOR* $-$ *numeral b*)
$-$ *numeral a XOR numeral b* $=$ *word-of-int* ($-$ *numeral a XOR numeral b*)
$-$ *numeral a XOR* $-$ *numeral b* $=$ *word-of-int* ($-$ *numeral a XOR* $-$ *numeral b*)
⟨*proof*⟩

Special cases for when one of the arguments equals 1.

lemma *word-bitwise-1-simps* [*simp*]:
NOT (*1*::′*a*::*len0 word*) $= -2$
1 AND numeral b $=$ *word-of-int* (*1 AND numeral b*)
1 AND $-$ *numeral b* $=$ *word-of-int* (*1 AND* $-$ *numeral b*)
numeral a AND 1 $=$ *word-of-int* (*numeral a AND 1*)
$-$ *numeral a AND 1* $=$ *word-of-int* ($-$ *numeral a AND 1*)
1 OR numeral b $=$ *word-of-int* (*1 OR numeral b*)
1 OR $-$ *numeral b* $=$ *word-of-int* (*1 OR* $-$ *numeral b*)
numeral a OR 1 $=$ *word-of-int* (*numeral a OR 1*)
$-$ *numeral a OR 1* $=$ *word-of-int* ($-$ *numeral a OR 1*)
1 XOR numeral b $=$ *word-of-int* (*1 XOR numeral b*)
1 XOR $-$ *numeral b* $=$ *word-of-int* (*1 XOR* $-$ *numeral b*)
numeral a XOR 1 $=$ *word-of-int* (*numeral a XOR 1*)
$-$ *numeral a XOR 1* $=$ *word-of-int* ($-$ *numeral a XOR 1*)
⟨*proof*⟩

Special cases for when one of the arguments equals -1.

lemma *word-bitwise-m1-simps* [*simp*]:
NOT ($-$*1*::′*a*::*len0 word*) $=$ *0*
($-$*1*::′*a*::*len0 word*) *AND x* $=$ *x*
x AND ($-$*1*::′*a*::*len0 word*) $=$ *x*
($-$*1*::′*a*::*len0 word*) *OR x* $= -1$
x OR ($-$*1*::′*a*::*len0 word*) $= -1$
($-$*1*::′*a*::*len0 word*) *XOR x* $=$ *NOT x*
x XOR ($-$*1*::′*a*::*len0 word*) $=$ *NOT x*
⟨*proof*⟩

lemma *uint-or*: *uint* (*x OR y*) $=$ (*uint x*) *OR* (*uint y*)
⟨*proof*⟩

lemma *uint-and*: *uint* (*x AND y*) $=$ (*uint x*) *AND* (*uint y*)
⟨*proof*⟩

lemma *test-bit-wi* [*simp*]:
(*word-of-int x*::′*a*::*len0 word*) !! *n* $\longleftrightarrow$ *n* $<$ *len-of TYPE*(′*a*) $\wedge$ *bin-nth x n*
⟨*proof*⟩

lemma *word-test-bit-transfer* [*transfer-rule*]:
(*rel-fun pcr-word* (*rel-fun op* $=$ *op* $=$))
(λ*x n*. *n* $<$ *len-of TYPE*(′*a*) $\wedge$ *bin-nth x n*) (*test-bit* :: ′*a*::*len0 word* $\Rightarrow$ -)
⟨*proof*⟩

lemma *word-ops-nth-size*:

$n < size\ (x::'a::len0\ word) \Longrightarrow$
$(x\ OR\ y)\ !!\ n = (x\ !!\ n\ |\ y\ !!\ n)\ \&$
$(x\ AND\ y)\ !!\ n = (x\ !!\ n\ \&\ y\ !!\ n)\ \&$
$(x\ XOR\ y)\ !!\ n = (x\ !!\ n\ \sim= y\ !!\ n)\ \&$
$(NOT\ x)\ !!\ n = (\sim\ x\ !!\ n)$
⟨*proof*⟩

lemma *word-ao-nth*:
fixes $x :: 'a::len0\ word$
shows $(x\ OR\ y)\ !!\ n = (x\ !!\ n\ |\ y\ !!\ n)\ \&$
$(x\ AND\ y)\ !!\ n = (x\ !!\ n\ \&\ y\ !!\ n)$
⟨*proof*⟩

lemma *test-bit-numeral* [*simp*]:
$(numeral\ w :: 'a::len0\ word)\ !!\ n \longleftrightarrow$
$n < len\text{-}of\ TYPE('a) \wedge bin\text{-}nth\ (numeral\ w)\ n$
⟨*proof*⟩

lemma *test-bit-neg-numeral* [*simp*]:
$(-\ numeral\ w :: 'a::len0\ word)\ !!\ n \longleftrightarrow$
$n < len\text{-}of\ TYPE('a) \wedge bin\text{-}nth\ (-\ numeral\ w)\ n$
⟨*proof*⟩

lemma *test-bit-1* [*simp*]: $(1::'a::len\ word)\ !!\ n \longleftrightarrow n = 0$
⟨*proof*⟩

lemma *nth-0* [*simp*]: $\sim\ (0::'a::len0\ word)\ !!\ n$
⟨*proof*⟩

lemma *nth-minus1* [*simp*]: $(-1::'a::len0\ word)\ !!\ n \longleftrightarrow n < len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemmas *bwsimps* =
wi-hom-add
word-wi-log-defs

lemma *word-bw-assocs*:
fixes $x :: 'a::len0\ word$
shows
$(x\ AND\ y)\ AND\ z = x\ AND\ y\ AND\ z$
$(x\ OR\ y)\ OR\ z = x\ OR\ y\ OR\ z$
$(x\ XOR\ y)\ XOR\ z = x\ XOR\ y\ XOR\ z$
⟨*proof*⟩

lemma *word-bw-comms*:
fixes $x :: 'a::len0\ word$
shows

x *AND* $y = y$ *AND* x
x *OR* $y = y$ *OR* x
x *XOR* $y = y$ *XOR* x
⟨*proof*⟩

lemma *word-bw-lcs*:
fixes x :: *'a::len0 word*
shows
y *AND* x *AND* $z = x$ *AND* y *AND* z
y *OR* x *OR* $z = x$ *OR* y *OR* z
y *XOR* x *XOR* $z = x$ *XOR* y *XOR* z
⟨*proof*⟩

lemma *word-log-esimps* [*simp*]:
fixes x :: *'a::len0 word*
shows
x *AND* $0 = 0$
x *AND* $-1 = x$
x *OR* $0 = x$
x *OR* $-1 = -1$
x *XOR* $0 = x$
x *XOR* $-1 =$ *NOT* x
0 *AND* $x = 0$
-1 *AND* $x = x$
0 *OR* $x = x$
-1 *OR* $x = -1$
0 *XOR* $x = x$
-1 *XOR* $x =$ *NOT* x
⟨*proof*⟩

lemma *word-not-dist*:
fixes x :: *'a::len0 word*
shows
NOT (x *OR* y) = *NOT* x *AND* *NOT* y
NOT (x *AND* y) = *NOT* x *OR* *NOT* y
⟨*proof*⟩

lemma *word-bw-same*:
fixes x :: *'a::len0 word*
shows
x *AND* $x = x$
x *OR* $x = x$
x *XOR* $x = 0$
⟨*proof*⟩

lemma *word-ao-absorbs* [*simp*]:
fixes x :: *'a::len0 word*
shows
x *AND* (y *OR* x) = x

x OR y AND x = x
x AND (x OR y) = x
y AND x OR x = x
(y OR x) AND x = x
x OR x AND y = x
(x OR y) AND x = x
x AND y OR x = x
⟨*proof*⟩

lemma *word-not-not* [*simp*]:
NOT NOT (x::'a::len0 word) = x
⟨*proof*⟩

lemma *word-ao-dist*:
fixes *x :: 'a::len0 word*
shows *(x OR y) AND z = x AND z OR y AND z*
⟨*proof*⟩

lemma *word-oa-dist*:
fixes *x :: 'a::len0 word*
shows *x AND y OR z = (x OR z) AND (y OR z)*
⟨*proof*⟩

lemma *word-add-not* [*simp*]:
fixes *x :: 'a::len0 word*
shows *x + NOT x = −1*
⟨*proof*⟩

lemma *word-plus-and-or* [*simp*]:
fixes *x :: 'a::len0 word*
shows *(x AND y) + (x OR y) = x + y*
⟨*proof*⟩

lemma *leoa*:
fixes *x :: 'a::len0 word*
shows *(w = (x OR y)) ⟹ (y = (w AND y))* ⟨*proof*⟩
lemma *leao*:
fixes *x' :: 'a::len0 word*
shows *(w' = (x' AND y')) ⟹ (x' = (x' OR w'))* ⟨*proof*⟩

lemma *word-ao-equiv*:
fixes *w w' :: 'a::len0 word*
shows *(w = w OR w') = (w' = w AND w')*
⟨*proof*⟩

lemma *le-word-or2*: *x <= x OR (y::'a::len0 word)*
⟨*proof*⟩

lemmas *le-word-or1 = xtr3 [OF word-bw-comms (2) le-word-or2]*

lemmas *word-and-le1* = *xtr3* [*OF word-ao-absorbs* (*4*) [*symmetric*] *le-word-or2*]
lemmas *word-and-le2* = *xtr3* [*OF word-ao-absorbs* (*8*) [*symmetric*] *le-word-or2*]

lemma *bl-word-not*: *to-bl* (*NOT w*) = *map Not* (*to-bl w*)
⟨*proof*⟩

lemma *bl-word-xor*: *to-bl* (*v XOR w*) = *map2 op* ~= (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *bl-word-or*: *to-bl* (*v OR w*) = *map2 op* | (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *bl-word-and*: *to-bl* (*v AND w*) = *map2 op* & (*to-bl v*) (*to-bl w*)
⟨*proof*⟩

lemma *word-lsb-alt*: *lsb* (*w*::′*a*::*len0 word*) = *test-bit w 0*
⟨*proof*⟩

lemma *word-lsb-1-0* [*simp*]: *lsb* (*1*::′*a*::*len word*) & ~ *lsb* (*0*::′*b*::*len0 word*)
⟨*proof*⟩

lemma *word-lsb-last*: *lsb* (*w*::′*a*::*len word*) = *last* (*to-bl w*)
⟨*proof*⟩

lemma *word-lsb-int*: *lsb w* = (*uint w mod 2* = *1*)
⟨*proof*⟩

lemma *word-msb-sint*: *msb w* = (*sint w* < *0*)
⟨*proof*⟩

lemma *msb-word-of-int*:
msb (*word-of-int x*::′*a*::*len word*) = *bin-nth x* (*len-of TYPE*(′*a*) − *1*)
⟨*proof*⟩

lemma *word-msb-numeral* [*simp*]:
msb (*numeral w*::′*a*::*len word*) = *bin-nth* (*numeral w*) (*len-of TYPE*(′*a*) − *1*)
⟨*proof*⟩

lemma *word-msb-neg-numeral* [*simp*]:
msb (− *numeral w*::′*a*::*len word*) = *bin-nth* (− *numeral w*) (*len-of TYPE*(′*a*) − *1*)
⟨*proof*⟩

lemma *word-msb-0* [*simp*]: ¬ *msb* (*0*::′*a*::*len word*)
⟨*proof*⟩

lemma *word-msb-1* [*simp*]: *msb* (*1*::′*a*::*len word*) ⟷ *len-of TYPE*(′*a*) = *1*
⟨*proof*⟩

lemma *word-msb-nth*:
msb (*w*::*'a*::*len word*) = *bin-nth* (*uint w*) (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemma *word-msb-alt*: *msb* (*w*::*'a*::*len word*) = *hd* (*to-bl w*)
⟨*proof*⟩

lemma *word-set-nth* [*simp*]:
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* !! (*len-of TYPE*(*'a*) − *1*)
⟨*proof*⟩

lemmas *msb0* = *len-gt-0* [*THEN diff-Suc-less*, *THEN word-ops-nth-size* [*unfolded word-size*]]
lemmas *msb1* = *msb0* [**where** *i* = *0*]
lemmas *word-ops-msb* = *msb1* [*unfolded msb-nth* [*symmetric*, *unfolded One-nat-def*]]

lemmas *lsb0* = *len-gt-0* [*THEN word-ops-nth-size* [*unfolded word-size*]]
lemmas *word-ops-lsb* = *lsb0* [*unfolded word-lsb-alt*]

lemma *td-ext-nth* [*OF refl refl refl*, *unfolded word-size*]:
n = *size* (*w*::′*a*::*len0 word*) ⟹ *ofn* = *set-bits* ⟹ [*w*, *ofn g*] = *l* ⟹
td-ext test-bit ofn {*f*. *ALL i*. *f i* −−> *i* < *n*} (%*h i*. *h i* & *i* < *n*)
⟨*proof*⟩

interpretation *test-bit*:
td-ext op !! :: ′*a*::*len0 word* => *nat* => *bool*
set-bits
{*f*. ∀ *i*. *f i* ⟶ *i* < *len-of TYPE*(′*a*::*len0*)}
(λ*h i*. *h i* ∧ *i* < *len-of TYPE*(′*a*::*len0*))
⟨*proof*⟩

lemmas *td-nth* = *test-bit.td-thm*

lemma *word-set-set-same* [*simp*]:
fixes *w* :: ′*a*::*len0 word*
shows *set-bit* (*set-bit w n x*) *n y* = *set-bit w n y*
⟨*proof*⟩

lemma *word-set-set-diff*:
fixes *w* :: ′*a*::*len0 word*
assumes *m* ~= *n*
shows *set-bit* (*set-bit w m x*) *n y* = *set-bit* (*set-bit w n y*) *m x*
⟨*proof*⟩

lemma *nth-sint*:
fixes *w* :: ′*a*::*len word*
defines *l* ≡ *len-of TYPE* (′*a*)
shows *bin-nth* (*sint w*) *n* = (*if n* < *l* − *1 then w* !! *n else w* !! (*l* − *1*))
⟨*proof*⟩

lemma *word-lsb-numeral* [*simp*]:
lsb (*numeral bin* :: ′*a* :: *len word*) ⟷ *bin-last* (*numeral bin*)
⟨*proof*⟩

lemma *word-lsb-neg-numeral* [*simp*]:
lsb (− *numeral bin* :: ′*a* :: *len word*) ⟷ *bin-last* (− *numeral bin*)
⟨*proof*⟩

lemma *set-bit-word-of-int*:
set-bit (*word-of-int x*) *n b* = *word-of-int* (*bin-sc n b x*)
⟨*proof*⟩

lemma *word-set-numeral* [*simp*]:
set-bit (*numeral bin*::′*a*::*len0 word*) *n b* =
word-of-int (*bin-sc n b* (*numeral bin*))
⟨*proof*⟩

lemma *word-set-neg-numeral* [*simp*]:
set-bit (− *numeral bin*::′*a*::*len0 word*) *n b* =
word-of-int (*bin-sc n b* (− *numeral bin*))
⟨*proof*⟩

lemma *word-set-bit-0* [*simp*]:
set-bit 0 n b = *word-of-int* (*bin-sc n b 0*)
⟨*proof*⟩

lemma *word-set-bit-1* [*simp*]:
set-bit 1 n b = *word-of-int* (*bin-sc n b 1*)
⟨*proof*⟩

lemma *setBit-no* [*simp*]:
setBit (*numeral bin*) *n* = *word-of-int* (*bin-sc n True* (*numeral bin*))
⟨*proof*⟩

lemma *clearBit-no* [*simp*]:
clearBit (*numeral bin*) *n* = *word-of-int* (*bin-sc n False* (*numeral bin*))
⟨*proof*⟩

lemma *to-bl-n1*:
to-bl (−*1*::′*a*::*len0 word*) = *replicate* (*len-of TYPE* (′*a*)) *True*
⟨*proof*⟩

lemma *word-msb-n1* [*simp*]: *msb* (−*1*::′*a*::*len word*)
⟨*proof*⟩

lemma *word-set-nth-iff*:
(*set-bit w n b* = *w*) = (*w* !! *n* = *b* | *n* >= *size* (*w*::′*a*::*len0 word*))
⟨*proof*⟩

lemma *test-bit-2p*:
(*word-of-int* (*2* ˆ *n*)::′*a*::*len word*) !! *m* ⟷ *m* = *n* ∧ *m* < *len-of TYPE*(′*a*)
⟨*proof*⟩

lemma *nth-w2p*:
((*2*::′*a*::*len word*) ˆ *n*) !! *m* ⟷ *m* = *n* ∧ *m* < *len-of TYPE*(′*a*::*len*)
⟨*proof*⟩

lemma *uint-2p*:
(*0*::′*a*::*len word*) < *2* ˆ *n* ⟹ *uint* (*2* ˆ *n*::′*a*::*len word*) = *2* ˆ *n*
⟨*proof*⟩

lemma *word-of-int-2p*: (*word-of-int* (*2* ˆ *n*) :: ′*a* :: *len word*) = *2* ˆ *n*
⟨*proof*⟩

lemma *bang-is-le*: *x* !! *m* ⟹ *2* ˆ *m* <= (*x* :: ′*a* :: *len word*)
⟨*proof*⟩

lemma *word-clr-le*:
fixes *w* :: *'a*::*len0 word*
shows *w* >= *set-bit w n False*
⟨*proof*⟩

lemma *word-set-ge*:
fixes *w* :: *'a*::*len word*
shows *w* <= *set-bit w n True*
⟨*proof*⟩

16.25 Shifting, Rotating, and Splitting Words

lemma *shiftl1-wi* [*simp*]: *shiftl1* (*word-of-int w*) = *word-of-int* (*w BIT False*)
⟨*proof*⟩

lemma *shiftl1-numeral* [*simp*]:
shiftl1 (*numeral w*) = *numeral* (*Num.Bit0 w*)
⟨*proof*⟩

lemma *shiftl1-neg-numeral* [*simp*]:
shiftl1 (− *numeral w*) = − *numeral* (*Num.Bit0 w*)
⟨*proof*⟩

lemma *shiftl1-0* [*simp*] : *shiftl1 0* = *0*
⟨*proof*⟩

lemma *shiftl1-def-u*: *shiftl1 w* = *word-of-int* (*uint w BIT False*)
⟨*proof*⟩

lemma *shiftl1-def-s*: *shiftl1 w* = *word-of-int* (*sint w BIT False*)
⟨*proof*⟩

lemma *shiftr1-0* [*simp*]: *shiftr1 0* = *0*
⟨*proof*⟩

lemma *sshiftr1-0* [*simp*]: *sshiftr1 0* = *0*
⟨*proof*⟩

lemma *sshiftr1-n1* [*simp*] : *sshiftr1* (− *1*) = − *1*
⟨*proof*⟩

lemma *shiftl-0* [*simp*] : (*0*::*'a*::*len0 word*) << *n* = *0*
⟨*proof*⟩

lemma *shiftr-0* [*simp*] : (*0*::*'a*::*len0 word*) >> *n* = *0*
⟨*proof*⟩

lemma *sshiftr-0* [*simp*] : *0* >>> *n* = *0*

⟨*proof*⟩

lemma *sshiftr-n1* [*simp*] : *−1 >>> n = −1*
⟨*proof*⟩

lemma *nth-shiftl1*: *shiftl1 w !! n = (n < size w & n > 0 & w !! (n − 1))*
⟨*proof*⟩

lemma *nth-shiftl′* [*rule-format*]:
ALL n. ((w::′a::len0 word) << m) !! n = (n < size w & n >= m & w !! (n − m))
⟨*proof*⟩

lemmas *nth-shiftl = nth-shiftl′* [*unfolded word-size*]

lemma *nth-shiftr1*: *shiftr1 w !! n = w !! Suc n*
⟨*proof*⟩

lemma *nth-shiftr*:
⋀n. ((w::′a::len0 word) >> m) !! n = w !! (n + m)
⟨*proof*⟩

lemma *uint-shiftr1*: *uint (shiftr1 w) = bin-rest (uint w)*
⟨*proof*⟩

lemma *nth-sshiftr1*:
sshiftr1 w !! n = (if n = size w − 1 then w !! n else w !! Suc n)
⟨*proof*⟩

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*⟩

16.25.1 shift functions in terms of lists of bools

lemmas *bshiftr1-numeral* [*simp*] =
bshiftr1-def [**where** *w*=*numeral w*, *unfolded to-bl-numeral*] **for** *w*

lemma *bshiftr1-bl*: *to-bl* (*bshiftr1 b w*) = *b* # *butlast* (*to-bl w*)
⟨*proof*⟩

lemma *shiftl1-of-bl*: *shiftl1* (*of-bl bl*) = *of-bl* (*bl* @ [*False*])
⟨*proof*⟩

lemma *shiftl1-bl*: *shiftl1* (*w*::′*a*::*len0 word*) = *of-bl* (*to-bl w* @ [*False*])
⟨*proof*⟩

lemma *bl-shiftl1*:
to-bl (*shiftl1* (*w* :: ′*a* :: *len word*)) = *tl* (*to-bl w*) @ [*False*]
⟨*proof*⟩

lemma *bl-shiftl1′*:
to-bl (*shiftl1 w*) = *tl* (*to-bl w* @ [*False*])
⟨*proof*⟩

lemma *shiftr1-bl*: *shiftr1 w* = *of-bl* (*butlast* (*to-bl w*))
⟨*proof*⟩

lemma *bl-shiftr1*:
to-bl (*shiftr1* (*w* :: ′*a* :: *len word*)) = *False* # *butlast* (*to-bl w*)
⟨*proof*⟩

lemma *bl-shiftr1′*:
to-bl (*shiftr1 w*) = *butlast* (*False* # *to-bl w*)
⟨*proof*⟩

lemma *shiftl1-rev*:
shiftl1 w = *word-reverse* (*shiftr1* (*word-reverse w*))
⟨*proof*⟩

lemma *shiftl-rev*:
shiftl w n = *word-reverse* (*shiftr* (*word-reverse w*) *n*)
⟨*proof*⟩

lemma *rev-shiftl*: *word-reverse w* << *n* = *word-reverse* (*w* >> *n*)
⟨*proof*⟩

lemma *shiftr-rev*: *w* >> *n* = *word-reverse* (*word-reverse w* << *n*)
⟨*proof*⟩

lemma *rev-shiftr*: *word-reverse w* >> *n* = *word-reverse* (*w* << *n*)

⟨*proof*⟩

lemma *bl-sshiftr1*:
to-bl (sshiftr1 (w :: 'a :: len word)) = hd (to-bl w) # butlast (to-bl w)
⟨*proof*⟩

lemma *drop-shiftr*:
drop n (to-bl ((w :: 'a :: len word) >> 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*:
n ≤ size w ⟹ 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*]
lemmas *take-sshiftr = take-sshiftr'* [*THEN conjunct2*]

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 *shiftl-of-bl*: *of-bl bl << n = of-bl (bl @ replicate n False)*
⟨*proof*⟩

lemma *shiftl-bl*:
(w::'a::len0 word) << (n::nat) = of-bl (to-bl w @ replicate n False)
⟨*proof*⟩

lemmas *shiftl-numeral* [*simp*] = *shiftl-def* [**where** *w=numeral w*] **for** *w*

lemma *bl-shiftl*:
to-bl (w << n) = drop n (to-bl w) @ replicate (min (size w) n) False
⟨*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* ˆ *n* ∗ *w*
⟨*proof*⟩

lemma *shiftr1-bintr* [*simp*]:
(*shiftr1* (*numeral w*) :: *'a* :: *len0 word*) =
word-of-int (*bin-rest* (*bintrunc* (*len-of TYPE* (*'a*)) (*numeral w*)))
⟨*proof*⟩

lemma *sshiftr1-sbintr* [*simp*]:
(*sshiftr1* (*numeral w*) :: *'a* :: *len word*) =
word-of-int (*bin-rest* (*sbintrunc* (*len-of TYPE* (*'a*) − *1*) (*numeral w*)))
⟨*proof*⟩

lemma *shiftr-no* [*simp*]:

(*numeral w*::*'a*::*len0 word*) >> *n* = *word-of-int*
((*bin-rest* ˆˆ *n*) (*bintrunc* (*len-of TYPE*(*'a*)) (*numeral w*)))
⟨*proof*⟩

lemma *sshiftr-no* [*simp*]:

(*numeral w*::*'a*::*len word*) >>> *n* = *word-of-int*
((*bin-rest* ˆˆ *n*) (*sbintrunc* (*len-of TYPE*(*'a*) − *1*) (*numeral w*)))
⟨*proof*⟩

lemma *shiftr1-bl-of*:
length bl ≤ *len-of TYPE*(*'a*) =⇒
shiftr1 (*of-bl bl*::*'a*::*len0 word*) = *of-bl* (*butlast bl*)
⟨*proof*⟩

lemma *shiftr-bl-of*:
length bl ≤ *len-of TYPE*(*'a*) =⇒
(*of-bl bl*::*'a*::*len0 word*) >> *n* = *of-bl* (*take* (*length bl* − *n*) *bl*)
⟨*proof*⟩

lemma *shiftr-bl*:
(*x*::*'a*::*len0 word*) >> *n* ≡ *of-bl* (*take* (*len-of TYPE*(*'a*) − *n*) (*to-bl x*))
⟨*proof*⟩

lemma *msb-shift*:

msb (w::'a::len word) ⟷ (w >> (len-of TYPE('a) − 1)) ≠ 0
⟨*proof*⟩

lemma *zip-replicate*:
n ≥ length ys ⟹ zip (replicate n x) ys = map (λy. (x, y)) ys
⟨*proof*⟩

lemma *align-lem-or* [*rule-format*] :
ALL x m. length x = n + m --> length y = n + m -->
drop m x = replicate n False --> take m y = replicate m False -->
map2 op | x y = take m x @ drop m y
⟨*proof*⟩

lemma *align-lem-and* [*rule-format*] :
ALL x m. length x = n + m --> length y = n + m -->
drop m x = replicate n False --> take m y = replicate m False -->
map2 op & x y = replicate (n + m) False
⟨*proof*⟩

lemma *aligned-bl-add-size* [*OF refl*]:
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*⟩

16.25.2 Mask

lemma *nth-mask* [*OF refl*, *simp*]:
m = mask n ⟹ test-bit m i = (i < n & i < size m)
⟨*proof*⟩

lemma *mask-bl*: *mask n = of-bl (replicate n True)*
⟨*proof*⟩

lemma *mask-bin*: *mask n = word-of-int (bintrunc n (− 1))*
⟨*proof*⟩

lemma *and-mask-bintr*: *w AND mask n = word-of-int (bintrunc n (uint w))*
⟨*proof*⟩

lemma *and-mask-wi*: *word-of-int i AND mask n = word-of-int (bintrunc n i)*
⟨*proof*⟩

lemma *and-mask-no*: *numeral i AND mask n = word-of-int (bintrunc n (numeral i))*
⟨*proof*⟩

lemma *bl-and-mask'*:
to-bl (w AND mask n :: 'a :: len word) =

replicate (*len-of TYPE*('*a*) − *n*) *False* @
drop (*len-of TYPE*('*a*) − *n*) (*to-bl w*)
⟨*proof*⟩

lemma *and-mask-mod-2p*: *w AND mask n* = *word-of-int* (*uint w mod 2* ˆ *n*)
⟨*proof*⟩

lemma *and-mask-lt-2p*: *uint* (*w AND mask n*) < *2* ˆ *n*
⟨*proof*⟩

lemma *eq-mod-iff*: *0* < (*n*::*int*) ⟹ *b* = *b mod n* ⟷ *0* ≤ *b* ∧ *b* < *n*
⟨*proof*⟩

lemma *mask-eq-iff*: (*w AND mask n*) = *w* ⟷ *uint w* < *2* ˆ *n*
⟨*proof*⟩

lemma *and-mask-dvd*: *2* ˆ *n dvd uint w* = (*w AND mask n* = *0*)
⟨*proof*⟩

lemma *and-mask-dvd-nat*: *2* ˆ *n dvd unat w* = (*w AND mask n* = *0*)
⟨*proof*⟩

lemma *word-2p-lem*:
n < *size w* ⟹ *w* < *2* ˆ *n* = (*uint* (*w* :: '*a* :: *len word*) < *2* ˆ *n*)
⟨*proof*⟩

lemma *less-mask-eq*: *x* < *2* ˆ *n* ⟹ *x AND mask n* = (*x* :: '*a* :: *len word*)
⟨*proof*⟩

lemmas *mask-eq-iff-w2p* = *trans* [*OF mask-eq-iff word-2p-lem* [*symmetric*]]

lemmas *and-mask-less'* = *iffD2* [*OF word-2p-lem and-mask-lt-2p*, *simplified word-size*]

lemma *and-mask-less-size*: *n* < *size x* ⟹ *x AND mask n* < *2*ˆ*n*
⟨*proof*⟩

lemma *word-mod-2p-is-mask* [*OF refl*]:
c = *2* ˆ *n* ⟹ *c* > *0* ⟹ *x mod c* = (*x* :: '*a* :: *len word*) *AND mask n*
⟨*proof*⟩

lemma *mask-eqs*:
(*a AND mask n*) + *b AND mask n* = *a* + *b AND mask n*
a + (*b AND mask n*) *AND mask n* = *a* + *b AND mask n*
(*a AND mask n*) − *b AND mask n* = *a* − *b AND mask n*
a − (*b AND mask n*) *AND mask n* = *a* − *b AND mask n*
a ∗ (*b AND mask n*) *AND mask n* = *a* ∗ *b AND mask n*
(*b AND mask n*) ∗ *a AND mask n* = *b* ∗ *a AND mask n*
(*a AND mask n*) + (*b AND mask n*) *AND mask n* = *a* + *b AND mask n*
(*a AND mask n*) − (*b AND mask n*) *AND mask n* = *a* − *b AND mask n*

(a AND mask n) ∗ (b AND mask n) AND mask n = a ∗ b AND mask n
− (a AND mask n) AND mask n = − a AND mask n
word-succ (a AND mask n) AND mask n = word-succ a AND mask n
word-pred (a AND mask n) AND mask n = word-pred a AND mask n
⟨*proof*⟩

lemma *mask-power-eq*:
(x AND mask n) ^ k AND mask n = x ^ k AND mask n
⟨*proof*⟩

16.25.3 Revcast

lemmas *revcast-def′* = *revcast-def* [*simplified*]
lemmas *revcast-def″* = *revcast-def′* [*simplified word-size*]
lemmas *revcast-no-def* [*simp*] = *revcast-def′* [**where** *w=numeral w*, *unfolded word-size*]
for *w*

lemma *to-bl-revcast*:
to-bl (revcast w :: 'a :: len0 word) =
takefill False (len-of TYPE ('a)) (to-bl w)
⟨*proof*⟩

lemma *revcast-rev-ucast* [*OF refl refl refl*]:
cs = [rc, uc] ⟹ rc = revcast (word-reverse w) ⟹ uc = ucast w ⟹
rc = word-reverse uc
⟨*proof*⟩

lemma *revcast-ucast*: *revcast w = word-reverse (ucast (word-reverse w))*
⟨*proof*⟩

lemma *ucast-revcast*: *ucast w = word-reverse (revcast (word-reverse w))*
⟨*proof*⟩

lemma *ucast-rev-revcast*: *ucast (word-reverse w) = word-reverse (revcast w)*
⟨*proof*⟩

lemmas *wsst-TYs = source-size target-size word-size*

lemma *revcast-down-uu* [*OF refl*]:
rc = revcast ⟹ source-size rc = target-size rc + n ⟹
rc (w :: 'a :: len word) = ucast (w >> n)
⟨*proof*⟩

lemma *revcast-down-us* [*OF refl*]:
rc = revcast ⟹ source-size rc = target-size rc + n ⟹
rc (w :: 'a :: len word) = ucast (w >>> n)
⟨*proof*⟩

lemma *revcast-down-su* [*OF refl*]:

rc = revcast ⟹ source-size rc = target-size rc + n ⟹
  rc (w :: 'a :: len word) = scast (w >> n)
⟨proof⟩

lemma *revcast-down-ss* [OF refl]:
rc = revcast ⟹ source-size rc = target-size rc + n ⟹
  rc (w :: 'a :: len word) = scast (w >>> n)
⟨proof⟩

lemma *cast-down-rev*:
uc = ucast ⟹ source-size uc = target-size uc + n ⟹
  uc w = revcast ((w :: 'a :: len word) << n)
⟨proof⟩

lemma *revcast-up* [OF refl]:
rc = revcast ⟹ source-size rc + n = target-size rc ⟹
  rc w = (ucast w :: 'a :: len word) << n
⟨proof⟩

lemmas *rc1* = revcast-up [THEN
  revcast-rev-ucast [symmetric, THEN trans, THEN word-rev-gal, symmetric]]
lemmas *rc2* = revcast-down-uu [THEN
  revcast-rev-ucast [symmetric, THEN trans, THEN word-rev-gal, symmetric]]

lemmas *ucast-up* =
 rc1 [simplified rev-shiftr [symmetric] revcast-ucast [symmetric]]
lemmas *ucast-down* =
 rc2 [simplified rev-shiftr revcast-ucast [symmetric]]

16.25.4 Slices

lemma *slice1-no-bin* [simp]:
 slice1 n (numeral w :: 'b word) = of-bl (takefill False n (bin-to-bl (len-of TYPE('b :: len0)) (numeral w)))
⟨proof⟩

lemma *slice-no-bin* [simp]:
 slice n (numeral w :: 'b word) = of-bl (takefill False (len-of TYPE('b :: len0) − n)
   (bin-to-bl (len-of TYPE('b :: len0)) (numeral w)))
⟨proof⟩

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]]*

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-alts =*
le-add-diff-inverse [symmetric, THEN su1]
le-add-diff-inverse2 [symmetric, THEN su1]

lemma *ucast-slice1*: *ucast w = slice1 (size w) w*
⟨*proof*⟩

lemma *ucast-slice*: *ucast w = slice 0 w*
⟨*proof*⟩

lemma *slice-id*: *slice 0 t = t*
⟨*proof*⟩

lemma *revcast-slice1 [OF refl]*:
rc = revcast w ⟹ slice1 (size rc) w = rc
⟨*proof*⟩

lemma *slice1-tf-tf'*:
to-bl (slice1 n w :: 'a :: len0 word) =
rev (takefill False (len-of TYPE('a)) (rev (takefill False n (to-bl w))))

⟨*proof*⟩

lemmas *slice1-tf-tf* = *slice1-tf-tf′* [*THEN word-bl.Rep-inverse′*, *symmetric*]

lemma *rev-slice1*:
n + *k* = *len-of TYPE*(′*a*) + *len-of TYPE*(′*b*) ⟹
slice1 n (*word-reverse w* :: ′*b* :: *len0 word*) =
word-reverse (*slice1 k w* :: ′*a* :: *len0 word*)
⟨*proof*⟩

lemma *rev-slice*:
n + *k* + *len-of TYPE*(′*a*::*len0*) = *len-of TYPE*(′*b*::*len0*) ⟹
slice n (*word-reverse* (*w*::′*b word*)) = *word-reverse* (*slice k w*::′*a word*)
⟨*proof*⟩

lemmas *sym-notr* =
not-iff [*THEN iffD2*, *THEN not-sym*, *THEN not-iff* [*THEN iffD1*]]

— problem posed by TPHOLs referee: criterion for overflow of addition of signed integers

lemma *sofl-test*:
(*sint* (*x* :: ′*a* :: *len word*) + *sint y* = *sint* (*x* + *y*)) =
((((*x*+*y*) *XOR x*) *AND* ((*x*+*y*) *XOR y*)) >> (*size x* − *1*) = *0*)
⟨*proof*⟩

16.26 Split and cat

lemmas *word-split-bin′* = *word-split-def*
lemmas *word-cat-bin′* = *word-cat-def*

lemma *word-rsplit-no*:
(*word-rsplit* (*numeral bin* :: ′*b* :: *len0 word*) :: ′*a word list*) =
map word-of-int (*bin-rsplit* (*len-of TYPE*(′*a* :: *len*))
(*len-of TYPE*(′*b*), *bintrunc* (*len-of TYPE*(′*b*)) (*numeral bin*)))
⟨*proof*⟩

lemmas *word-rsplit-no-cl* [*simp*] = *word-rsplit-no*
[*unfolded bin-rsplitl-def bin-rsplit-l* [*symmetric*]]

lemma *test-bit-cat*:
wc = *word-cat a b* ⟹ *wc* !! *n* = (*n* < *size wc* &
(*if n* < *size b then b* !! *n else a* !! (*n* − *size b*)))
⟨*proof*⟩

lemma *word-cat-bl*: *word-cat a b* = *of-bl* (*to-bl a* @ *to-bl b*)
⟨*proof*⟩

lemma *of-bl-append*:

(*of-bl* (*xs* @ *ys*) :: *'a* :: *len word*) = *of-bl xs* ∗ *2*ˆ(*length ys*) + *of-bl ys*
⟨*proof*⟩

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

lemma *of-bl-True* [*simp*]:
(*of-bl* (*True#xs*)::*'a*::*len word*) = *2*ˆ*length xs* + *of-bl xs*
⟨*proof*⟩

lemma *of-bl-Cons*:
of-bl (*x#xs*) = *of-bool x* ∗ *2*ˆ*length xs* + *of-bl xs*
⟨*proof*⟩

lemma *split-uint-lem*: *bin-split n* (*uint* (*w* :: *'a* :: *len0 word*)) = (*a*, *b*) ⟹
a = *bintrunc* (*len-of TYPE*(*'a*) − *n*) *a* & *b* = *bintrunc* (*len-of TYPE*(*'a*)) *b*
⟨*proof*⟩

lemma *word-split-bl'*:
std = *size c* − *size b* ⟹ (*word-split c* = (*a*, *b*)) ⟹
(*a* = *of-bl* (*take std* (*to-bl c*)) & *b* = *of-bl* (*drop std* (*to-bl c*)))
⟨*proof*⟩

lemma *word-split-bl*: *std* = *size c* − *size b* ⟹
(*a* = *of-bl* (*take std* (*to-bl c*)) & *b* = *of-bl* (*drop std* (*to-bl c*))) ⟷
word-split c = (*a*, *b*)
⟨*proof*⟩

lemma *word-split-bl-eq*:
(*word-split* (*c*::*'a*::*len word*) :: (*'c* :: *len0 word* ∗ *'d* :: *len0 word*)) =
(*of-bl* (*take* (*len-of TYPE*(*'a*::*len*) − *len-of TYPE*(*'d*::*len0*)) (*to-bl c*)),
of-bl (*drop* (*len-of TYPE*(*'a*) − *len-of TYPE*(*'d*)) (*to-bl c*)))
⟨*proof*⟩

lemma *test-bit-split'*:
word-split c = (*a*, *b*) −−> (*ALL n m*. *b* !! *n* = (*n* < *size b* & *c* !! *n*) &
a !! *m* = (*m* < *size a* & *c* !! (*m* + *size b*)))
⟨*proof*⟩

lemma *test-bit-split*:
word-split c = (*a*, *b*) ⟹
(∀ *n*::*nat*. *b* !! *n* ⟷ *n* < *size b* ∧ *c* !! *n*) ∧ (∀ *m*::*nat*. *a* !! *m* ⟷ *m* < *size a*
∧ *c* !! (*m* + *size b*))
⟨*proof*⟩

lemma *test-bit-split-eq*: *word-split c* = (*a*, *b*) ⟷
((*ALL n*::*nat*. *b* !! *n* = (*n* < *size b* & *c* !! *n*)) &
(*ALL m*::*nat*. *a* !! *m* = (*m* < *size a* & *c* !! (*m* + *size b*))))
⟨*proof*⟩

lemma *word-cat-id*: *word-cat a b = b*
⟨*proof*⟩
lemma *word-cat-hom*:
len-of TYPE('a::len0) <= len-of TYPE('b::len0) + len-of TYPE ('c::len0)
⟹
(word-cat (word-of-int w :: 'b word) (b :: 'c word) :: 'a word) =
word-of-int (bin-cat w (size b) (uint b))
⟨*proof*⟩

lemma *word-cat-split-alt*:
size w <= size u + size v ⟹ word-split w = (u, v) ⟹ word-cat u v = w
⟨*proof*⟩

lemmas *word-cat-split-size = sym [THEN [2] word-cat-split-alt [symmetric]]*

16.26.1 Split and slice

lemma *split-slices*:
word-split w = (u, v) ⟹ u = slice (size v) w & v = slice 0 w
⟨*proof*⟩

lemma *slice-cat1 [OF refl]*:
wc = word-cat a b ⟹ size wc >= size a + size b ⟹ slice (size b) wc = a
⟨*proof*⟩

lemmas *slice-cat2 = trans [OF slice-id word-cat-id]*

lemma *cat-slices*:
a = slice n c ⟹ b = slice 0 c ⟹ n = size b ⟹
size a + size b >= size c ⟹ word-cat a b = c
⟨*proof*⟩

lemma *word-split-cat-alt*:
w = word-cat u v ⟹ size u + size v <= size w ⟹ word-split w = (u, v)
⟨*proof*⟩

lemmas *word-cat-bl-no-bin [simp] =*
word-cat-bl [**where** *a=numeral a* **and** *b=numeral b*,
unfolded to-bl-numeral]
for *a b*

lemmas *word-split-bl-no-bin [simp] =*
word-split-bl-eq [**where** *c=numeral c*, *unfolded to-bl-numeral*] **for** *c*

this odd result arises from the fact that the statement of the result implies that the decoded words are of the same type, and therefore of the same length, as the original word

lemma *word-rsplit-same*: *word-rsplit w = [w]*

⟨*proof*⟩

lemma *word-rsplit-empty-iff-size*:
(*word-rsplit w* = []) = (*size w* = *0*)
⟨*proof*⟩

lemma *test-bit-rsplit*:
sw = *word-rsplit w* ⟹ *m* < *size* (*hd sw* :: *'a* :: *len word*) ⟹
k < *length sw* ⟹ (*rev sw* ! *k*) !! *m* = (*w* !! (*k* ∗ *size* (*hd sw*) + *m*))
⟨*proof*⟩

lemma *word-rcat-bl*: *word-rcat wl* = *of-bl* (*concat* (*map to-bl wl*))
⟨*proof*⟩

lemma *size-rcat-lem'*:
size (*concat* (*map to-bl wl*)) = *length wl* ∗ *size* (*hd wl*)
⟨*proof*⟩

lemmas *size-rcat-lem* = *size-rcat-lem'* [*unfolded word-size*]

lemmas *td-gal-lt-len* = *len-gt-0* [*THEN td-gal-lt*]

lemma *nth-rcat-lem*:
n < *length* (*wl*::*'a word list*) ∗ *len-of TYPE*(*'a*::*len*) ⟹
rev (*concat* (*map to-bl wl*)) ! *n* =
rev (*to-bl* (*rev wl* ! (*n div len-of TYPE*(*'a*)))) ! (*n mod len-of TYPE*(*'a*))
⟨*proof*⟩

lemma *test-bit-rcat*:
sw = *size* (*hd wl* :: *'a* :: *len word*) ⟹ *rc* = *word-rcat wl* ⟹ *rc* !! *n* =
(*n* < *size rc* & *n div sw* < *size wl* & (*rev wl*) ! (*n div sw*) !! (*n mod sw*))
⟨*proof*⟩

lemma *foldl-eq-foldr*:
foldl op + *x xs* = *foldr op* + (*x* # *xs*) (*0* :: *'a* :: *comm-monoid-add*)
⟨*proof*⟩

lemmas *test-bit-cong* = *arg-cong* [**where** *f* = *test-bit*, *THEN fun-cong*]

lemmas *test-bit-rsplit-alt* =
trans [*OF nth-rev-alt* [*THEN test-bit-cong*]
test-bit-rsplit [*OF refl asm-rl diff-Suc-less*]]

— lazy way of expressing that u and v, and su and sv, have same types
lemma *word-rsplit-len-indep* [*OF refl refl refl refl*]:
[*u*,*v*] = *p* ⟹ [*su*,*sv*] = *q* ⟹ *word-rsplit u* = *su* ⟹
word-rsplit v = *sv* ⟹ *length su* = *length sv*
⟨*proof*⟩

lemma *length-word-rsplit-size*:
  $n = len\text{-}of\ TYPE\ ('a :: len) \Longrightarrow$
    $(length\ (word\text{-}rsplit\ w :: 'a\ word\ list) <= m) = (size\ w <= m * n)$
  ⟨*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*:
  $n = len\text{-}of\ TYPE\ ('a :: len) \Longrightarrow$
    $length\ (word\text{-}rsplit\ w :: 'a\ word\ list) = (size\ w + n - 1)\ div\ n$
  ⟨*proof*⟩

lemma *length-word-rsplit-even-size*:
  $n = len\text{-}of\ TYPE\ ('a :: len) \Longrightarrow size\ w = m * n \Longrightarrow$
    $length\ (word\text{-}rsplit\ w :: 'a\ word\ list) = m$
  ⟨*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*:
  $[\![word\text{-}rcat\ (ws::'a::len\ word\ list) = (frcw::'b::len0\ word);$
     $size\ frcw = length\ ws * len\text{-}of\ TYPE('a)]\!]$
    $\Longrightarrow length\ (word\text{-}rsplit\ frcw::'a\ word\ list) = length\ ws$
  ⟨*proof*⟩

lemma *msrevs*:
  fixes $n::nat$
  shows $0 < n \Longrightarrow (k * n + m)\ div\ n = m\ div\ n + k$
  and $(k * n + m)\ mod\ n = m\ mod\ n$
  ⟨*proof*⟩

lemma *word-rsplit-rcat-size* [*OF refl*]:
  $word\text{-}rcat\ (ws :: 'a :: len\ word\ list) = frcw \Longrightarrow$
    $size\ frcw = length\ ws * len\text{-}of\ TYPE\ ('a) \Longrightarrow word\text{-}rsplit\ frcw = ws$
  ⟨*proof*⟩

16.27 Rotation

lemmas *rotater-0′* [*simp*] = *rotater-def* [**where** $n = 0$, *simplified*]

lemmas *word-rot-defs* = *word-roti-def word-rotr-def word-rotl-def*

lemma *rotate-eq-mod*:
m mod length xs = n mod length xs ⟹ rotate m xs = rotate n xs
⟨*proof*⟩

lemmas *rotate-eqs* =
trans [*OF rotate0* [*THEN fun-cong*] *id-apply*]
rotate-rotate [*symmetric*]
rotate-id
rotate-conv-mod
rotate-eq-mod

16.27.1 Rotation of list to right

lemma *rotate1-rl′*: *rotater1* (*l* @ [*a*]) = *a* # *l*
⟨*proof*⟩

lemma *rotate1-rl* [*simp*] : *rotater1* (*rotate1 l*) = *l*
⟨*proof*⟩

lemma *rotate1-lr* [*simp*] : *rotate1* (*rotater1 l*) = *l*
⟨*proof*⟩

lemma *rotater1-rev′*: *rotater1* (*rev xs*) = *rev* (*rotate1 xs*)
⟨*proof*⟩

lemma *rotater-rev′*: *rotater n* (*rev xs*) = *rev* (*rotate n xs*)
⟨*proof*⟩

lemma *rotater-rev*: *rotater n ys* = *rev* (*rotate n* (*rev ys*))
⟨*proof*⟩

lemma *rotater-drop-take*:
rotater n xs =
drop (*length xs* − *n mod length xs*) *xs* @
take (*length xs* − *n mod length xs*) *xs*
⟨*proof*⟩

lemma *rotater-Suc* [*simp*] :
rotater (*Suc n*) *xs* = *rotater1* (*rotater n xs*)
⟨*proof*⟩

lemma *rotate-inv-plus* [*rule-format*] :
ALL k. k = m + n −−> rotater k (*rotate n xs*) = *rotater m xs* &
rotate k (*rotater n xs*) = *rotate m xs* &
rotater n (*rotate k xs*) = *rotate m xs* &
rotate n (*rotater k xs*) = *rotater m xs*
⟨*proof*⟩

lemmas *rotate-inv-rel* = *le-add-diff-inverse2* [*symmetric*, *THEN rotate-inv-plus*]

lemmas *rotate-inv-eq* = *order-refl* [*THEN rotate-inv-rel*, *simplified*]

lemmas *rotate-lr* [*simp*] = *rotate-inv-eq* [*THEN conjunct1*]
lemmas *rotate-rl* [*simp*] = *rotate-inv-eq* [*THEN conjunct2*, *THEN conjunct1*]

lemma *rotate-gal*: (*rotater n xs* = *ys*) = (*rotate n ys* = *xs*)
⟨*proof*⟩

lemma *rotate-gal'*: (*ys* = *rotater n xs*) = (*xs* = *rotate n ys*)
⟨*proof*⟩

lemma *length-rotater* [*simp*]:
length (*rotater n xs*) = *length xs*
⟨*proof*⟩

lemma *restrict-to-left*:
assumes $x = y$
shows $(x = z) = (y = z)$
⟨*proof*⟩

lemmas *rrs0* = *rotate-eqs* [*THEN restrict-to-left*,
simplified rotate-gal [*symmetric*] *rotate-gal'* [*symmetric*]]
lemmas *rrs1* = *rrs0* [*THEN refl* [*THEN rev-iffD1*]]
lemmas *rotater-eqs* = *rrs1* [*simplified length-rotater*]
lemmas *rotater-0* = *rotater-eqs* (*1*)
lemmas *rotater-add* = *rotater-eqs* (*2*)

16.27.2 map, map2, commuting with rotate(r)

lemma *butlast-map*:
xs ~= [] ⟹ *butlast* (*map f xs*) = *map f* (*butlast xs*)
⟨*proof*⟩

lemma *rotater1-map*: *rotater1* (*map f xs*) = *map f* (*rotater1 xs*)
⟨*proof*⟩

lemma *rotater-map*:
rotater n (*map f xs*) = *map f* (*rotater n xs*)
⟨*proof*⟩

lemma *but-last-zip* [*rule-format*] :
ALL ys. *length xs* = *length ys* −−> *xs* ~= [] −−>
last (*zip xs ys*) = (*last xs*, *last ys*) &
butlast (*zip xs ys*) = *zip* (*butlast xs*) (*butlast ys*)
⟨*proof*⟩

lemma *but-last-map2* [*rule-format*] :

ALL ys. length xs = length ys --> xs ~= [] -->
last (map2 f xs ys) = f (last xs) (last ys) &
butlast (map2 f xs ys) = map2 f (butlast xs) (butlast ys)
⟨*proof*⟩

lemma *rotater1-zip*:
length xs = length ys ⟹
rotater1 (zip xs ys) = zip (rotater1 xs) (rotater1 ys)
⟨*proof*⟩

lemma *rotater1-map2*:
length xs = length ys ⟹
rotater1 (map2 f xs ys) = map2 f (rotater1 xs) (rotater1 ys)
⟨*proof*⟩

lemmas *lrth* =
box-equals [OF asm-rl length-rotater [symmetric]
length-rotater [symmetric],
THEN rotater1-map2]

lemma *rotater-map2*:
length xs = length ys ⟹
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-eqs [THEN to-bl-rotl [THEN trans]]*

lemmas *word-rotl-eqs* =
blrs0 [simplified word-bl-Rep′ word-bl.Rep-inject to-bl-rotl [symmetric]]

lemma *to-bl-rotr*:
to-bl (word-rotr n w) = rotater n (to-bl w)

⟨*proof*⟩

lemmas *brrs0* = *rotater-eqs* [*THEN to-bl-rotr* [*THEN trans*]]

lemmas *word-rotr-eqs* =
brrs0 [*simplified word-bl-Rep′ word-bl.Rep-inject to-bl-rotr* [*symmetric*]]

declare *word-rotr-eqs* (*1*) [*simp*]
declare *word-rotl-eqs* (*1*) [*simp*]

lemma
word-rot-rl [*simp*]:
word-rotl k (*word-rotr k v*) = *v* **and**
word-rot-lr [*simp*]:
word-rotr k (*word-rotl k v*) = *v*
⟨*proof*⟩

lemma
word-rot-gal:
(*word-rotr n v* = *w*) = (*word-rotl n w* = *v*) **and**
word-rot-gal′:
(*w* = *word-rotr n v*) = (*v* = *word-rotl n w*)
⟨*proof*⟩

lemma *word-rotr-rev*:
word-rotr n w = *word-reverse* (*word-rotl n* (*word-reverse w*))
⟨*proof*⟩

lemma *word-roti-0* [*simp*]: *word-roti 0 w* = *w*
⟨*proof*⟩

lemmas *abl-cong* = *arg-cong* [**where** *f* = *of-bl*]

lemma *word-roti-add*:
word-roti (*m* + *n*) *w* = *word-roti m* (*word-roti n w*)
⟨*proof*⟩

lemma *word-roti-conv-mod′*: *word-roti n w* = *word-roti* (*n mod int* (*size w*)) *w*
⟨*proof*⟩

lemmas *word-roti-conv-mod* = *word-roti-conv-mod′* [*unfolded word-size*]

16.27.3 "Word rotation commutes with bit-wise operations

locale *word-rotate*
begin

lemmas *word-rot-defs′* = *to-bl-rotl to-bl-rotr*

lemmas *blwl-syms* [*symmetric*] = *bl-word-not bl-word-and bl-word-or bl-word-xor*

lemmas *lbl-lbl* = *trans* [*OF word-bl-Rep′ word-bl-Rep′* [*symmetric*]]

lemmas *ths-map2* [*OF lbl-lbl*] = *rotate-map2 rotater-map2*

lemmas *ths-map* [**where** *xs* = *to-bl v*] = *rotate-map rotater-map* **for** *v*

lemmas *th1s* [*simplified word-rot-defs′* [*symmetric*]] = *ths-map2 ths-map*

lemma *word-rot-logs*:
word-rotl n (*NOT v*) = *NOT word-rotl n v*
word-rotr n (*NOT v*) = *NOT word-rotr n v*
word-rotl n (*x AND y*) = *word-rotl n x AND word-rotl n y*
word-rotr n (*x AND y*) = *word-rotr n x AND word-rotr n y*
word-rotl n (*x OR y*) = *word-rotl n x OR word-rotl n y*
word-rotr n (*x OR y*) = *word-rotr n x OR word-rotr n y*
word-rotl n (*x XOR y*) = *word-rotl n x XOR word-rotl n y*
word-rotr n (*x XOR y*) = *word-rotr n x XOR word-rotr n y*
⟨*proof*⟩
end

lemmas *word-rot-logs* = *word-rotate.word-rot-logs*

lemmas *bl-word-rotl-dt* = *trans* [*OF to-bl-rotl rotate-drop-take*,
simplified word-bl-Rep′]

lemmas *bl-word-rotr-dt* = *trans* [*OF to-bl-rotr rotater-drop-take*,
simplified word-bl-Rep′]

lemma *bl-word-roti-dt′*:
n = *nat* ((− *i*) *mod int* (*size* (*w* :: ′*a* :: *len word*))) ⟹
to-bl (*word-roti i w*) = *drop n* (*to-bl w*) @ *take n* (*to-bl w*)
⟨*proof*⟩

lemmas *bl-word-roti-dt* = *bl-word-roti-dt′* [*unfolded word-size*]

lemmas *word-rotl-dt* = *bl-word-rotl-dt* [*THEN word-bl.Rep-inverse′* [*symmetric*]]
lemmas *word-rotr-dt* = *bl-word-rotr-dt* [*THEN word-bl.Rep-inverse′* [*symmetric*]]
lemmas *word-roti-dt* = *bl-word-roti-dt* [*THEN word-bl.Rep-inverse′* [*symmetric*]]

lemma *word-rotx-0* [*simp*] : *word-rotr i 0* = *0* & *word-rotl i 0* = *0*
⟨*proof*⟩

lemma *word-roti-0′* [*simp*] : *word-roti n 0* = *0*
⟨*proof*⟩

lemmas *word-rotr-dt-no-bin′* [*simp*] =
word-rotr-dt [**where** *w*=*numeral w*, *unfolded to-bl-numeral*] **for** *w*

lemmas *word-rotl-dt-no-bin'* [*simp*] =
word-rotl-dt [**where** *w=numeral w*, *unfolded to-bl-numeral*] **for** *w*

declare *word-roti-def* [*simp*]

16.28 Maximum machine word

lemma *word-int-cases*:
obtains *n* **where** $(x :: 'a::len0\ word) = word\text{-}of\text{-}int\ n$ **and** $0 \leq n$ **and** $n < 2^{\wedge}len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemma *word-nat-cases* [*cases type*: *word*]:
obtains *n* **where** $(x :: 'a::len\ word) = of\text{-}nat\ n$ **and** $n < 2^{\wedge}len\text{-}of\ TYPE('a)$
⟨*proof*⟩

lemma *max-word-eq*: $(max\text{-}word::'a::len\ word) = 2^{\wedge}len\text{-}of\ TYPE('a) - 1$
⟨*proof*⟩

lemma *max-word-max* [*simp*,*intro*!]: $n \leq max\text{-}word$
⟨*proof*⟩

lemma *word-of-int-2p-len*: $word\text{-}of\text{-}int\ (2\ \hat{}\ len\text{-}of\ TYPE('a)) = (0::'a::len0\ word)$
⟨*proof*⟩

lemma *word-pow-0*:
$(2::'a::len\ word)\ \hat{}\ len\text{-}of\ TYPE('a) = 0$
⟨*proof*⟩

lemma *max-word-wrap*: $x + 1 = 0 \Longrightarrow x = max\text{-}word$
⟨*proof*⟩

lemma *max-word-minus*:
$max\text{-}word = (-1::'a::len\ word)$
⟨*proof*⟩

lemma *max-word-bl* [*simp*]:
$to\text{-}bl\ (max\text{-}word::'a::len\ word) = replicate\ (len\text{-}of\ TYPE('a))\ True$
⟨*proof*⟩

lemma *max-test-bit* [*simp*]:
$(max\text{-}word::'a::len\ word)\ !!\ n = (n < len\text{-}of\ TYPE('a))$
⟨*proof*⟩

lemma *word-and-max* [*simp*]:
$x\ AND\ max\text{-}word = x$

⟨*proof*⟩

lemma *word-or-max* [*simp*]:
x OR max-word = max-word
⟨*proof*⟩

lemma *word-ao-dist2*:
x AND (y OR z) = x AND y OR x AND (z::'a::len0 word)
⟨*proof*⟩

lemma *word-oa-dist2*:
x OR y AND z = (x OR y) AND (x OR (z::'a::len0 word))
⟨*proof*⟩

lemma *word-and-not* [*simp*]:
x AND NOT x = (0::'a::len0 word)
⟨*proof*⟩

lemma *word-or-not* [*simp*]:
x OR NOT x = max-word
⟨*proof*⟩

lemma *word-boolean*:
boolean (op AND) (op OR) bitNOT 0 max-word
⟨*proof*⟩

interpretation *word-bool-alg*:
boolean op AND op OR bitNOT 0 max-word
⟨*proof*⟩

lemma *word-xor-and-or*:
x XOR y = x AND NOT y OR NOT x AND (y::'a::len0 word)
⟨*proof*⟩

interpretation *word-bool-alg*:
boolean-xor op AND op OR bitNOT 0 max-word op XOR
⟨*proof*⟩

lemma *shiftr-x-0* [*iff*]:
(x::'a::len0 word) >> 0 = x
⟨*proof*⟩

lemma *shiftl-x-0* [*simp*]:
(x :: 'a :: len word) << 0 = x
⟨*proof*⟩

lemma *shiftl-1* [*simp*]:
(1::'a::len word) << n = 2^n
⟨*proof*⟩

lemma *uint-lt-0* [*simp*]:
  uint x < *0* = *False*
  ⟨*proof*⟩

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 (λ(*x*,*y*). *x* & *y*) (*zip xs* (*replicate n True*)) = *xs*
  ⟨*proof*⟩

lemma *map-replicate-False*:
  n = *length xs* ⟹ *map* (λ(*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*:
  $uint\ (x + y) =$
  $(if\ uint\ x + uint\ y < 2\hat{}size\ x\ then$
      $uint\ x + uint\ y$
   $else$
      $uint\ x + uint\ y - 2\hat{}size\ x)$
  ⟨*proof*⟩

lemma *unat-plus-if-size*:
  $unat\ (x + (y{::}'a{::}len\ word)) =$
  $(if\ unat\ x + unat\ y < 2\hat{}size\ x\ then$
      $unat\ x + unat\ y$
   $else$
      $unat\ x + unat\ y - 2\hat{}size\ x)$
  ⟨*proof*⟩

lemma *word-neq-0-conv*:
  fixes $w :: 'a :: len\ word$
  shows $(w \neq 0) = (0 < w)$
  ⟨*proof*⟩

lemma *max-lt*:
  $unat\ (max\ a\ b\ div\ c) = unat\ (max\ a\ b)\ div\ unat\ (c{::}\ 'a :: len\ word)$
  ⟨*proof*⟩

lemma *uint-sub-if-size*:
  $uint\ (x - y) =$
  $(if\ uint\ y \leq uint\ x\ then$
      $uint\ x - uint\ y$
   $else$
      $uint\ x - uint\ y + 2\hat{}size\ x)$
  ⟨*proof*⟩

lemma *unat-sub*:
  $b <= a \Longrightarrow unat\ (a - b) = unat\ a - unat\ b$
  ⟨*proof*⟩

lemmas *word-less-sub1-numberof* [*simp*] = *word-less-sub1* [*of numeral w*] **for** *w*
lemmas *word-le-sub1-numberof* [*simp*] = *word-le-sub1* [*of numeral w*] **for** *w*

lemma *word-of-int-minus*:
  $word\text{-}of\text{-}int\ (2\hat{}len\text{-}of\ TYPE('a) - i) = (word\text{-}of\text{-}int\ (-i){::}'a{::}len\ word)$
⟨*proof*⟩

lemmas *word-of-int-inj* =
  word-uint.Abs-inject [*unfolded uints-num*, *simplified*]

lemma *word-le-less-eq*:
  $(x\ {::}'z{::}len\ word) \leq y = (x = y \vee x < y)$

⟨*proof*⟩

lemma *mod-plus-cong*:
assumes *1*: $(b::int) = b'$
and *2*: $x \ mod \ b' = x' \ mod \ b'$
and *3*: $y \ mod \ b' = y' \ mod \ b'$
and *4*: $x' + y' = z'$
shows $(x + y) \ mod \ b = z' \ mod \ b'$
⟨*proof*⟩

lemma *mod-minus-cong*:
assumes *1*: $(b::int) = b'$
and *2*: $x \ mod \ b' = x' \ mod \ b'$
and *3*: $y \ mod \ b' = y' \ mod \ b'$
and *4*: $x' - y' = z'$
shows $(x - y) \ mod \ b = z' \ mod \ b'$
⟨*proof*⟩

lemma *word-induct-less*:
$[\![P \ (0::'a::len \ word); \bigwedge n. \ [\![n < m; P \ n]\!] \Longrightarrow P \ (1 + n)]\!] \Longrightarrow P \ m$
⟨*proof*⟩

lemma *word-induct*:
$[\![P \ (0::'a::len \ word); \bigwedge n. \ P \ n \Longrightarrow P \ (1 + n)]\!] \Longrightarrow P \ m$
⟨*proof*⟩

lemma *word-induct2* [*induct type*]:
$[\![P \ 0; \bigwedge n. \ [\![1 + n \neq 0; P \ n]\!] \Longrightarrow P \ (1 + n)]\!] \Longrightarrow P \ (n::'b::len \ word)$
⟨*proof*⟩

16.29 Recursion combinator for words

definition *word-rec* :: $'a \Rightarrow ('b::len \ word \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'b \ word \Rightarrow 'a$
where
word-rec forZero forSuc n = *rec-nat forZero* (*forSuc* ∘ *of-nat*) (*unat n*)

lemma *word-rec-0*: *word-rec z s 0* = *z*
⟨*proof*⟩

lemma *word-rec-Suc*:
$1 + n \neq (0::'a::len \ word) \Longrightarrow$ *word-rec z s* $(1 + n) = s \ n$ (*word-rec z s n*)
⟨*proof*⟩

lemma *word-rec-Pred*:
$n \neq 0 \Longrightarrow$ *word-rec z s n* $= s \ (n - 1)$ (*word-rec z s* $(n - 1)$)
⟨*proof*⟩

lemma *word-rec-in*:
f (*word-rec z* (λ-. *f*) *n*) = *word-rec* (*f z*) (λ-. *f*) *n*

⟨*proof*⟩

lemma *word-rec-in2*:
$f\ n\ (word\text{-}rec\ z\ f\ n) = word\text{-}rec\ (f\ 0\ z)\ (f \circ op + 1)\ n$
⟨*proof*⟩

lemma *word-rec-twice*:
$m \leq n \Longrightarrow word\text{-}rec\ z\ f\ n = word\text{-}rec\ (word\text{-}rec\ z\ f\ (n - m))\ (f \circ op + (n - m))\ m$
⟨*proof*⟩

lemma *word-rec-id*: $word\text{-}rec\ z\ (\lambda\text{-}.\ id)\ n = z$
⟨*proof*⟩

lemma *word-rec-id-eq*: $\forall\, m < n.\ f\ m = id \Longrightarrow word\text{-}rec\ z\ f\ n = z$
⟨*proof*⟩

lemma *word-rec-max*:
$\forall\, m{\geq}n.\ m \neq -\ 1 \longrightarrow f\ m = id \Longrightarrow word\text{-}rec\ z\ f\ (-\ 1) = word\text{-}rec\ z\ f\ n$
⟨*proof*⟩

lemma *unatSuc*:
$1 + n \neq (0{::}'a{::}len\ word) \Longrightarrow unat\ (1 + n) = Suc\ (unat\ n)$
⟨*proof*⟩

declare *bin-to-bl-def* [*simp*]

⟨*ML*⟩

hide-const (**open**) *Word*

end

References

[1] Jeremy Dawson. Isabelle theories for machine words. In Michael Goldsmith and Bill Roscoe, editors, *Seventh International Workshop on Automated Verification of Critical Systems (AVOCS'07)*, Electronic Notes in Theoretical Computer Science, page 15, Oxford, September 2007. Elsevier. to appear.