The HOL `word` Library

W. Wong

University of Cambridge, Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England.

May 1993

Revised for HOL version 2.02 in February 1994
Revised for HOL98 in July/August 1999

Contents

Chapter 1

The `word` library

Bit vector (or word)[1] is one of the fundamental data objects in hardware specification and verification. The modelling of bit vectors is a key to the success of a hardware verification project. This library attempt to provide a general, flexible infrastructure for reasoning about words. The description begins with a discussion of approach used by the library to model words. This is followed by a summary of the facilities available in the library. Chapter 2 contains the reference entries of all ML functions, and the last chapter lists all theorems stored in the library.

1.1 Modelling bit vectors

An abstract model of words should encompass all their basic properties. It should be independent of any concrete representation. The basic abstract properties of words are:

- a word is a vector of n elements;
- the size of a given word n is constant;
- all elements are of the same type;
- an individual element is accessed via its index.

Suppose that w is a word of size n, it can be written as

$$w = [\![w_{n-1}w_{n-2}\ldots w_1w_0]\!]$$

where w_i represents the ith bit of the word w. We adopt the convention that the bits are indexed from the right hand side starting from 0. The index operation $w[i]$ accesses the ith bit of a word for all i less then n. A segment operation extracts a segment from a word. For example,

$$w[m,k] = [\![w_{k+m-1}\ldots w_k]\!] \tag{1.1}$$

where $(k+m) \leq n$ is a m-bit segment of the word w starting from the kth bit.

[1]The two terms, *bit vector* and *word* will be used interchangeably in this manual.

A word concatenation operation $\bullet$ can be defined as

$$\begin{aligned} A_\bullet B &= [\![a_{n-1}\dots a_0]\!]\bullet[\![b_{m-1}\dots b_0]\!] \\ &= [\![a_{n-1}\dots a_0 b_{m-1}\dots b_0]\!] \end{aligned} \tag{1.2}$$

which builds a word of size $n+m$ from two words of size n and m, respectively.

Since words of all sizes share these basic properties, a base type of some kind would be a starting point for modelling words. This base type should then be parameterized with the size and the type of the elements. This suggests a dependent type of the form

$:(\alpha, n)$`word`

where α is the type of the elements and n is the size. In the current version of the HOL system, it is possible to define a polymorphic type $:(\alpha)$`word` which takes the element type as a parameter, but it is not possible to parameterize a type with natural numbers. There is also difficulty in defining a real abstract type in the current version of HOL.

To overcome the difficulties mentioned above, the approach used in implementing the `word` library uses facilities available in the current version of HOL only. First of all, it defines a polymorphic type `:(*)word` to represent generic words. This allows one to use different types to represent the bits according to the requirements of one's applications. For example, `:(bool)word` is suitable for many hardware applications using two-value logic.

Dependent types are simulated using restricted universal quantifications. A restricted universal quantification is written in the form

$\forall x :: P.\, t[x]$

where if $x : \alpha$ then P can be any term of type $\alpha \to$ *bool*; this denotes the quantification of x over those values satisfying P. The semantics of this quantification is defined by the following equation:

$$\vdash_{\mathit{def}}\ \ \forall x :: P.\, t[x] = \forall x. P\, x \supset t[x] \tag{1.3}$$

Suppose that P is a predicate $\mathsf{PWORDLEN}\, n$ which returns T when applied to a word w if and only if w is an n-bit word, then the expression

$\forall w :: \mathsf{PWORDLEN}\, n.\ \dots$

can be read as '*for all n-bit words w, …*'. For a specific value of n, say 8, one can define a predicate $\mathsf{word8}$ by the definition

$\vdash_{\mathit{def}}\ \ \mathsf{word8} = \mathsf{PWORDLEN}\, 8.$

This predicate can then be used in expressions, such as $\forall w :: \mathsf{word8}.\ \dots$ Since the syntax of restricted quantification resembles the syntax of types closely and the semantics of

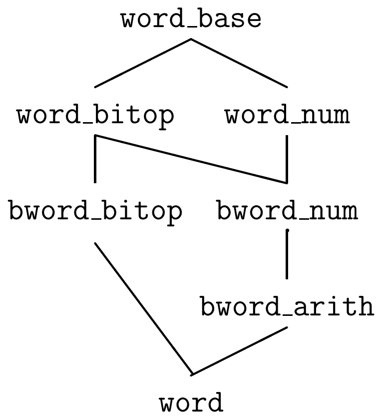

Figure 1.1: The ancestry of theories

the quantification is suitably defined, using this to simulate dependent types is very comprehensible.

As we cannot define a real abstract type in HOL, the list type is used as the underlying representation of the polymorphic type `:(*)word`. However, through disciplined use of system functions and properties derived for the new type, direct reference to the underlying representation is minimized. For example, when defining new constants, constant specification is used to specify the abstract properties of the new constant instead of using constant definition which needs access to the representation. In the development of the library, the proofs of some basic theorems about words have to refer to the underlying lists. After a small number of basic theorems are derived, one can proceed to reason about words on a more abstract level without resorting to the underlying representation.

1.2 The library

The `word` library consists of several theories and some ML functions implementing tactics and conversions which manipulate words. The ancestry of the theories is illustrated in Figure 1.1. The theories whose names begin with `word_` contains definitions of generic constants and theorems asserting general properties of words. These generic constants are polymorphic and can be applied to words of any types. There are three such theories in the library, namely `word_base`, `word_bitop` and `word_num`. As boolean words are used most often, the theories whose names begin with `bword_` are about this type of words. The subsections below describe individual theories in more detail.

$\mathsf{PWORDLEN}$`:num -> ((*)word -> bool)`
$\mathsf{PWORDLEN}\, n\, w = \mathsf{T}$ iff w is an n-bit word

$\mathsf{WORDLEN}$`:(*)word -> num`
$\mathsf{WORDLEN}\, w = n$

BIT `:num -> ((*)word -> *)`
$\mathsf{BIT}\, i\ [\![a_{n-1} \ldots a_i \ldots a_0]\!] = a_i$

WSEG`:num -> (num -> ((*)word -> (*)word))`
$\mathsf{WSEG}\, m\, k\ [\![a_{n-1} \ldots a_{k+m-1} \ldots a_k \ldots a_0]\!] = [\![a_{k+m-1} \ldots a_k]\!]$

WCAT`:(*)word # (*)word -> (*)word`
$\mathsf{WCAT}([\![a_{n-1} \ldots a_0]\!], [\![b_{m-1} \ldots b_0]\!]) = [\![a_{n-1} \ldots a_0 b_{m-1} \ldots b_0]\!]$

Table 1.1: Basic constants in the theory `word_base`

1.2.1 The basics: the theory `word_base`

First of all, the polymorphic type `:(*)word` is defined in this theory. It is defined using `define_type` with the following specification:

```
`word = WORD (*)list`
```

The basic constants denoting the functions of indexing, segmenting and concatenation of words described in Section 1.1 are BIT, WSEG and WCAT, respectively. The predicate PWORDLEN for discriminating the size of words and a function named WORDLEN returning the size of a word are also defined in this theory. The types and specifications of these constants are listed in Table 1.1. Several constants denoting some simple functions on words are also defined for convenient, such as MSB for most significant bit. These are listed in Table 1.2.

A number of theorems stating the properties of the basic constants are stored in this theory. Some of the more important ones are discussed below. The theorem `WSEG_PWORDLEN` states that the size of the word resulting from taking an m-bit segment from an n-bit word is m providing that $k + m \leq n$ where k is the starting bit.

HOL Theorem (`WSEG_PWORDLEN`)

$$\vdash \forall n. \forall w :: \mathsf{PWORDLEN}\, n.$$
$$\quad \forall m\, k.\, m + k \leq n \supset \mathsf{PWORDLEN}\, m\, (\mathsf{WSEG}\, m\, k\, w)$$

A nested WSEG expression can be simplified providing that the sizes and starting bits satisfy certain conditions. This is asserted by the theorem `WSEG_WSEG`.

LSB `:(*)word -> *`

$\vdash \forall n. \forall w :: \mathsf{PWORDLEN}\, n.\, 0 < n \supset (\mathsf{LSB}\, w = \mathsf{BIT}\, 0\, w)$

MSB `:(*)word -> *`

$$\begin{array}{l} \vdash \forall n. \forall w :: \mathsf{PWORDLEN}\, n. \\ \quad 0 < n \supset (\mathsf{MSB}\, w = \mathsf{BIT}\, (\mathsf{PRE}\, n)\, w) \end{array}$$

WSPLIT `:num -> ((*)word -> (*)word # (*)word)`

$$\begin{array}{l} \vdash (\forall n. \forall w :: \mathsf{PWORDLEN}\, n. \\ \quad \forall m.\, m \leq n \supset (\mathsf{WCAT}\, (\mathsf{WSPLIT}\, m\, w) = w)) \wedge \\ \quad (\forall n\, m. \forall w_1 :: \mathsf{PWORDLEN}\, n. \forall 2_2 :: \mathsf{PWORDLEN}\, m. \\ \quad\quad \mathsf{WSPLIT}\, m\, (\mathsf{WCAT}\, (w_1, w_2)) = w_1, w_2)) \end{array}$$

$$\begin{array}{l} \vdash \forall n. \forall w :: \mathsf{PWORDLEN}\, n. \\ \quad \forall k.\, k \leq n \supset (\mathsf{WSPLIT}\, k\, w = \mathsf{WSEG}\, (n-k)\, k\, w, \mathsf{WSEG}\, k\, 0\, w) \end{array}$$

Table 1.2: Other constants in the theory `word_base`

HOL Theorem (`WSEG_WSEG`)

$$\begin{array}{l} \vdash \forall n. \forall w :: \mathsf{PWORDLEN}\, n. \\ \quad \forall m_1\, k_1\, m_2\, k_2.\, m_1 + k_1 \leq n \wedge m_2 + k_2 \leq m_1 \supset \\ \quad\quad (\mathsf{WSEG}\, m_2\, k_2\, (\mathsf{WSEG}\, m_1\, k_1\, w) = \mathsf{WSEG}\, m_2\, (k_1 + k_2)\, w) \end{array}$$

The theorem `WCAT_PWORDLEN` states that the size of the result of the word concatenation operation is the sum of the sizes of its operands.

HOL Theorem (`WCAT_PWORDLEN`)

$$\begin{array}{l} \vdash \forall n_1. \forall w_1 :: \mathsf{PWORDLEN}\, n_1. \\ \quad \forall n_2. \forall w_2 :: \mathsf{PWORDLEN}\, n_2. \\ \quad\quad \mathsf{PWORDLEN}\, (n_1 + n_2)\, (\mathsf{WCAT}\, (w_1, w_2)) \end{array}$$

The associativity of the WCAT operation is asserted by the theorem `WCAT_ASSOC`.

HOL Theorem (`WCAT_ASSOC`)

$$\begin{array}{l} \vdash \forall w_1\, w_2\, w_3. \\ \quad \mathsf{WCAT}\, (w_1, \mathsf{WCAT}\, (w_2, w_3)) = \mathsf{WCAT}\, (\mathsf{WCAT}\, (w_1, w_2), w_3) \end{array}$$

The theorem WSEG_WCAT_WSEG asserts that taking a segment from a word which is built by concatenating two words w_1 and w_2 is equivalent to taking the appropriate segments from each word and then concatenating them provided that the segment spans across the boundary of the two words.

HOL Theorem (WSEG_WCAT_WSEG)

$$\begin{array}{l}
\vdash \forall n_1 \forall n_2.\, \forall w_1 :: \mathsf{PWORDLEN}\, n_1.\, \forall w_2 :: \mathsf{PWORDLEN}\, n_2. \\
\quad \forall m\, k. \\
\qquad m + k \leq n_1 + n_2 \wedge k < n_2 \wedge n_2 \leq m + k \supset \\
\qquad\quad (\mathsf{WSEG}\, m\, k\, (\mathsf{WCAT}\, (w_1, w_2)) = \\
\qquad\qquad \mathsf{WCAT}\, (\mathsf{WSEG}\, ((m + k) - n_2)\, 0\, w_1, \mathsf{WSEG}\, (n_2 - k)\, k\, w_2))
\end{array}$$

1.2.2 Generic bitwise operators: the theory word_bitop

Definitions in this theory include predicates for bitwise operators, predicates on properties of bits and shift operators.

Two predicates, PBITOP and PBITBOP are defined for quantifying bitwise operators. When applied to a suitably typed word function op, they will return T if and only if op is a bitwise unary or binary operator, respectively. The meaning of *bitwise* is that the operator preserves the size and the operation on each bit is independent of other bits. Note that as these predicates are polymorphic the type of the bits can be anything. The exact definitions of these predicates are as follows:

PBITOP : `((*)word -> (**)word) -> bool`
PBITOP op = T iff op is a bitwise unary operator

$$\begin{array}{l}
\vdash_{def} \quad \forall op.\, \mathsf{PBITOP}\, op = \\
\quad (\forall n.\, \forall w :: \mathsf{PWORDLEN}\, n. \\
\qquad \mathsf{PWORDLEN}\, n\, (op\, w) \wedge \\
\qquad (\forall m\, k.\, m + k \leq n \supset (op\, (\mathsf{WSEG}\, m\, k\, w) = \mathsf{WSEG}\, m\, k\, (op\, w)))
\end{array}$$

PBITBOP : `((*)word -> (**)word -> (***)word) -> bool`
PBITBOP op = T iff op is a bitwise binary operator

$$\begin{array}{l}
\vdash_{def} \quad \forall op.\, \mathsf{PBITBOP}\, op = \\
\quad (\forall n.\, \forall w_1 :: \mathsf{PWORDLEN}\, n.\, \forall w_2 \mathsf{PWORDLEN}\, n. \\
\qquad \mathsf{PWORDLEN}\, n\, (op\, w_1\, w_2) \wedge \\
\qquad (\forall m\, k.\, m + k \leq n \supset \\
\qquad\quad (op\, (\mathsf{WSEG}\, m\, k\, w_1)\, (\mathsf{WSEG}\, m\, k\, w_2) = \mathsf{WSEG}\, m\, k\, (op\, w_1\, w_2)))
\end{array}$$

SHR `:bool -> * -> (*)word -> ((*)word # *)`

$$\mathsf{SHR}\, f\, b\, [\![a_{n-1} \ldots a_1 a_0]\!] = \begin{cases} ([\![a_{n-1} a_{n-1} \ldots a_1]\!], a_0) & \text{if } f = \mathsf{T} \\ ([\![b\, a_{n-1} \ldots a_1]\!], a_0) & \text{if } f = \mathsf{F} \end{cases}$$

SHL `:bool -> (*)word -> * -> (* # (*)word)`

$$\mathsf{SHL}\, f\ [\![a_{n-1} a_{n-2} \ldots a_0]\!]\ b = \begin{cases} (a_{n-1}, [\![a_{n-2} \ldots a_0 a_0]\!]) & \text{if } f = \mathsf{T} \\ (a_{n-1}, [\![a_{n-2} \ldots a_0\, b]\!]) & \text{if } f = \mathsf{F} \end{cases}$$

Table 1.3: Shift operators

Two higher-order functions, FORALLBITS and EXISTSABIT, are defined for testing whether the bits of a word have certain properties. The term FORALLBITS $P\,w$ evaluates to T if and only if all the bits in the word w satisfy the predicate P. The term EXISTSABIT $P\,w$ evaluates to T if and only if there exists one or more bits in the word w satisfying the predicate P. The higher-order function WMAP defined in this theory is analogous to the function MAP on lists. The meaning of the expression WMAP $f\,w$ is to apply the function f to each bit of the word w.

Also in this theory are the definitions of two generic shift operators: SHL and SHR. Their types and specification is listed in Table 1.3. Both take three arguments and return a pair. The first argument is a boolean value indicating the kind of operation to be performed. The second and the third arguments to SHR are a single bit and a word, respectively. The order of these two arguments to SHL is reversed. Depending on the value of the boolean and single bit argument, these operators can perform either a logical shift, an arithmetic shift or a rotation operation. If the boolean argument is T, the single bit argument is not used. SHR shifts its operand one bit to the right and the left-most bit is duplicated to fill the vacant position, thus, implementing an arithmetic shift. If the boolean argument is F, SHR fills the vacant position with the single bit argument. If this bit is the right-most bit of the operand, a rotation is performed. If it has value 0, it results in a logical shift. The operation performed by SHL is similar. The pair returned by these operators consists of a word which is the operation result and a single bit which is the bit shifted out of the operand.

A number of theorems asserting the operational behaviour of these operators and their relationship with the basic constants WCAT and WSEG are stored in this theory. The theorems `SHR_WSEG` and `SHL_WSEG` state the equivalence between a shift expression and a combination of WCAT and WSEG. Thus, an expression involving shift operators can be simplified to one which only involves the basic word operations.

HOL Theorem (SHR_WSEG)

$$\begin{array}{l}
\vdash \forall n.\, \forall w :: \mathsf{PWORDLEN}\, n.\\
\quad \forall m\, k.\, m + k \leq n \supset 0 < m \supset\\
\quad\quad (\forall f\, b.\, \mathsf{SHR}\, f\, b\, (\mathsf{WSEG}\, m\, k\, w) =\\
\quad\quad\quad (f \Rightarrow \mathsf{WCAT}\, (\mathsf{WSEG}\, 1\, (k + (m-1))\, w, \mathsf{WSEG}\, (m-1)\, (k+1)\, w) \mid\\
\quad\quad\quad\quad \mathsf{WCAT}\, (\mathsf{WORD}\, [b], \mathsf{WSEG}\, (m-1)\, (k+1)\, w)),\\
\quad\quad\quad \mathsf{BIT}\, k\, w)
\end{array}$$

HOL Theorem (SHL_WSEG)

$$\begin{array}{l}
\vdash \forall n.\, \forall w :: \mathsf{PWORDLEN}\, n.\\
\quad \forall m\, k.\, m + k \leq n \supset 0 < m \supset\\
\quad\quad (\forall f\, b.\, \mathsf{SHL}\, f\, (\mathsf{WSEG}\, m\, k\, w)\, b =\\
\quad\quad\quad \mathsf{BIT}\, (k + (m-1))\, w,\\
\quad\quad\quad (f \Rightarrow \mathsf{WCAT}\, (\mathsf{WSEG}\, (m-1)\, k\, w, \mathsf{WSEG}\, 1\, k\, w) \mid\\
\quad\quad\quad\quad \mathsf{WCAT}\, (\mathsf{WSEG}\, (m-1)\, k\, w, \mathsf{WORD}\, [b])))
\end{array}$$

1.2.3 Boolean bitwise operators: the theory `bword_bitop`

In this theory, a small set of boolean bitwise operators are defined and theorems asserting that they are bitwise operators are proved. The boolean bitwise operators are:

WNOT	`:bool word -> bool word`	bitwise negation
WAND	`:bool word -> bool word -> bool word`	bitwise AND
WOR	`:bool word -> bool word -> bool word`	bitwise OR
WXOR	`:bool word -> bool word -> bool word`	bitwise exclusive-OR

The theorems stating that they are bitwise are:

`PBITOP_WNOT`	$\vdash$ PBITOP WNOT
`PBITBOP_WAND`	$\vdash$ PBITBOP WAND
`PBITBOP_WOR`	$\vdash$ PBITBOP WOR
`PBITBOP_WXOR`	$\vdash$ PBITBOP WXOR

1.2.4 Natural numbers and words: the theory `word_num`

Words are often interpreted as natural numbers. In this theory, two constants are defined to map generic words to natural numbers and vice versa:

NVAL`:(* -> num) -> num -> (*)word -> num`

NVAL f b w returns the numeric value of w. f is a function mapping a bit to its numeric value and b is the base or radix of the word.

NWORD:`num -> (num -> *) -> num -> num -> (*)word`

NWORD n f' b m returns an n-bit word representing the value of m. f' is a function mapping a number to a bit and b is the base.

The upper bound of the numeric value of a word is stated by the theorem `NVAL_MAX`.

HOL Theorem (`NVAL_MAX`)

$$\begin{array}{l} \vdash \forall f\, b.\, (\forall x.\, f\, x < b) \supset \\ \quad \forall n.\, \forall w :: \mathsf{PWORDLEN}\, n.\, \mathsf{NVAL}\, f\, b\, w < (b\, \mathsf{EXP}\, n) \end{array}$$

Provided that the bit value function f satisfies $\forall x.\, f\, x < b$, the numeric value of a word w is always less than b^n. The theorem `NVAL_WCAT` states that the value of a word can be calculated from the values of its segments.

HOL Theorem (`NVAL_WCAT`)

$$\begin{array}{l} \vdash \forall n\, m.\, \forall w_1 :: \mathsf{PWORDLEN}\, n. \\ \quad \forall w_2 :: \mathsf{PWORDLEN}\, m. \\ \quad\quad \forall f\, b. \\ \quad\quad\quad \mathsf{NVAL}\, f\, b\, (\mathsf{WCAT}\, (w_1, w_2)) = \\ \quad\quad\quad\quad (\mathsf{NVAL}\, f\, b\, w_1 \times (b\, \mathsf{EXP}\, m)) + (\mathsf{NVAL}\, f\, b\, w_2) \end{array}$$

The theorem stating the size of the result of mapping from natural number to word is `NWORD_PWORDLEN`.

HOL Theorem (`NWORD_PWORDLEN`)

$$\vdash \forall n\, f\, b\, m.\, \mathsf{PWORDLEN}\, n\, (\mathsf{NWORD}\, n\, f\, b\, m)$$

1.2.5 Boolean words and numbers: the theory `bword_num`

In this theory, two functions mapping between a single bit and number are defined first. Then, the constants denoting the mapping between boolean words and natural numbers are defined in terms of these bit mapping functions and the generic word–num mapping functions described in Section 1.2.4.

BV `:bool -> num`

$\vdash \forall b.\, \mathsf{BV}\, b = (b \Rightarrow \mathsf{SUC}\, 0 \mid 0)$

VB `:num -> bool`

$\vdash \forall n.\, \mathsf{VB}\, n = \neg((n\, \mathsf{MOD}\, 2) = 0)$

BNVAL:`bool word -> num`

BNVAL w returns the numeric value of w. $\vdash_{def}$ BNVAL w = NVAL BV $2\ w$

NBWORD:`num -> num -> bool word`

NBWORD n m returns a n-bit word representing the value of m. $\vdash_{def}$ NBWORD $n\ m$ = NWORD n VB $2\ m$

The functions BNVAL and NBWORD are inverse to each other in the set of numbers less than 2^n where n is the size of the word. The following theorems state the basic properties of these mapping functions.

HOL Theorem (`VB_BV`)

$$\vdash \forall x.\, \mathsf{VB}\,(\mathsf{BV}\,x) = x$$

HOL Theorem (`BV_VB`)

$$\vdash \forall x.\, x < 2 \supset (\mathsf{BV}\,(\mathsf{VB}\,x) = x)$$

HOL Theorem (`NBWORD_BNVAL`)

$$\vdash \forall n.\, \forall w :: \mathsf{PWORDLEN}\,n.\, \mathsf{NBWORD}\,n\,(\mathsf{BNVAL}\,w) = w$$

HOL Theorem (`BNVAL_NBWORD`)

$$\vdash \forall n\,m.\\ m < (2\,\mathsf{EXP}\,n) \supset (\mathsf{BNVAL}\,(\mathsf{NBWORD}\,n\,m) = m)$$

HOL Theorem (`PWORDLEN_NBWORD`)

$$\vdash \forall n\,m.\, \mathsf{PWORDLEN}\,n\,(\mathsf{NBWORD}\,n\,m)$$

HOL Theorem (`NBWORD_MOD`)

$$\vdash \forall n\,m.\, \mathsf{NBWORD}\,n\,(m\,\mathsf{MOD}\,(2\,\mathsf{EXP}\,n)) = \mathsf{NBWORD}\,n\,m$$

The theorem `NBWORD_SUC` asserts the fact that converting a number m to a word can be performed bit by bit recursively.

HOL Theorem (`NBWORD_SUC`)

$$\vdash \forall n\,m.\, \mathsf{NBWORD}\,(\mathsf{SUC}\,n)\,m = \\ \mathsf{WCAT}\,(\mathsf{NBWORD}\,n\,(m\,\mathsf{DIV}\,2), \mathsf{WORD}\,[\mathsf{VB}\,(m\,\mathsf{MOD}\,2)])$$

The theorem `WSEG_NBWORD` states that taking an m-bit segment of an n-bit word mapped to by NBWORD from a number l is equivalent to mapping the quotient of l divided by 2^k to an m-bit word.

HOL Theorem (`WSEG_NBWORD`)

$$\vdash \forall m\,k\,n.\, m + k \leq n \supset \\ (\forall l.\, \mathsf{WSEG}\,m\,k\,(\mathsf{NBWORD}\,n\,l) = \mathsf{NBWORD}\,m\,(l\,\mathsf{DIV}\,(2\,\mathsf{EXP}\,k)))$$

1.2.6 Boolean word arithmetic: the theory bword_arith

This theory is about addition of boolean words. Two methods of computing the carry value of each bit are defined: ACARRY uses addition and ICARRY uses logical operations $\wedge$ and $\vee$. The theorem asserting the equivalence of these methods is ACARRY_EQ_ICARRY.

The theorem ADD_WORD_SPLIT states that addition of two words can be carried out in segments.

HOL Theorem (ADD_WORD_SPLIT)

$$\begin{aligned}
&\vdash \forall n_1\, n_2.\, \forall w_1\, w_2 :: \mathsf{PWORDLEN}\,(n_1+n_2).\, \forall cin.\\
&\quad \mathsf{NBWORD}\,(n_1+n_2)\,(\mathsf{BNVAL}\,w_1 + \mathsf{BNVAL}\,w_2 + \mathsf{BV}\,cin) =\\
&\quad \mathsf{WCAT}\,(\mathsf{NBWORD}\,n_1\,(\mathsf{BNVAL}\,(\mathsf{WSEG}\,n_1\,n_2\,w_1) + \mathsf{BNVAL}\,(\mathsf{WSEG}\,n_1\,n_2\,w_2) +\\
&\qquad \mathsf{BV}\,(\mathsf{ACARRY}\,n_2\,w_1\,w_2\,cin)),\\
&\qquad \mathsf{NBWORD}\,n_2\,(\mathsf{BNVAL}\,(\mathsf{WSEG}\,n_2\,0\,w_1) + \mathsf{BNVAL}\,(\mathsf{WSEG}\,n_2\,0\,w_2) +\\
&\qquad \mathsf{BV}\,cin))
\end{aligned}$$

The theorem WSEG_NBWORD_ADD asserts that taking a segment of the sum of two words is equal to taking the corresponding segments of the words then summing them up.

HOL Theorem (WSEG_NBWORD_ADD)

$$\begin{aligned}
&\vdash \forall n.\, \forall w_1\, w_2 :: \mathsf{PWORDLEN}\,n.\, \forall m\, k\, cin.\, m+k \leq n \supset\\
&\quad (\mathsf{WSEG}\,m\,k\,(\mathsf{NBWORD}\,n\,(\mathsf{BNVAL}\,w_1 + \mathsf{BNVAL}\,w_2 + \mathsf{BV}\,cin)) =\\
&\quad \mathsf{NBWORD}\,m\,(\mathsf{BNVAL}\,(\mathsf{WSEG}\,m\,k\,w_1) + \mathsf{BNVAL}\,(\mathsf{WSEG}\,m\,k\,w_2) +\\
&\qquad \mathsf{BV}\,(\mathsf{ACARRY}\,k\,w_1\,w_2\,cin)))
\end{aligned}$$

1.2.7 Proof tools

The word library currently has a small set of tools in the form of conversions and tactics for manipulating words. These include the following:

BIT_CONV : conv When applied to a term as the left hand side of the following theorem, this conversion returns the theorem

$$\vdash \mathsf{BIT}\,k\,(\mathsf{WORD}[w_{n-1};\ldots;w_k;\ldots;w_0]) = w_k$$

WSEG_CONV : conv When applied to a term as the left hand side of the following theorem, this conversion returns the theorem

$$\vdash \mathsf{WSEG}\,m\,k\,(\mathsf{WORD}[w_{n-1};\ldots;w_k;\ldots;w_0]) = [w_{m+k-1};\ldots;w_k]$$

`WSEG_WSEG_CONV :(term -> conv)` When applied to a term as the left hand side of the following theorem, the conversion `WSEG_WSEG_CONV "n"` returns the theorem

$$\mathsf{PWORDLEN}\,n\,w\ \vdash \mathsf{WSEG}\,m_2\,k_2(\mathsf{WSEG}\,m_1\,k_1 w) = \mathsf{WSEG}\,m_2\,k\,w$$

where $k = k_1 + k_2$ and n, k_1, k_2, m_1 and m_2 are numeric constants and satisfy the following relations: $k_1 + m_1 \leq n$ and $k_2 + m_2 \leq m_1$.

`PWORDLEN_CONV :(term list -> conv)` When applied to the term $\mathsf{PWORDLEN}\,m\,tm$, the conversion `PWORDLEN_CONV tms` returns a theorem asserting the size of the word tm. This theorem is in the form

$$A \vdash \mathsf{PWORDLEN}\,m\,tm = \mathsf{T}$$

where the exact form of A, tm and the term list argument `tms` is given in the table below:

tm	`tms`	theorem
$\mathsf{WORD}[b_{n-1};\ldots;b_0]$	`[ ]`	$\vdash \mathsf{PWORDLEN}\,n\,(\mathsf{WORD}[b_{n-1};\ldots;b_0])$
$\mathsf{WSEG}\,m\,k\,tm'$	`[ "n" ]`	$\mathsf{PWORDLEN}\,n\,tm'$ $\vdash \mathsf{PWORDLEN}\,m\,(\mathsf{WSEG}\,m\,k\,tm')$
$\mathsf{WCAT}(tm', tm'')$	`["n1";"n2"]`	$\mathsf{PWORDLEN}\,n_1\,tm', \mathsf{PWORDLEN}\,n_2\,tm''$ $\vdash \mathsf{PWORDLEN}\,n\,(\mathsf{WCAT}(tm', tm''))$ where $n = n1 + n2$
$\mathsf{WNOT}\,tm'$	`[ ]`	$\mathsf{PWORDLEN}\,n\,tm'$ $\vdash \mathsf{PWORDLEN}\,n\,(\mathsf{WNOT}\,tm')$
$\mathsf{WAND}\,tm'\,tm''$	`[ ]`	$\mathsf{PWORDLEN}\,n\,tm', \mathsf{PWORDLEN}\,n\,tm''$ $\vdash \mathsf{PWORDLEN}\,n\,(\mathsf{WAND}\,tm'\,tm'')$
$\mathsf{WOR}\,tm'\,tm''$	`[ ]`	$\mathsf{PWORDLEN}\,n\,tm', \mathsf{PWORDLEN}\,n\,tm''$ $\vdash \mathsf{PWORDLEN}\,n\,(\mathsf{WOR}\,tm'\,tm'')$
$\mathsf{WXOR}\,tm'\,tm''$	`[ ]`	$\mathsf{PWORDLEN}\,n\,tm', \mathsf{PWORDLEN}\,n\,tm''$ $\vdash \mathsf{PWORDLEN}\,n\,(\mathsf{WXOR}\,tm'\,tm'')$

`PWORDLEN_bitop_CONV :conv` When applied to a term $\mathsf{PWORDLEN}\,n\,tm$ where tm involves only bitwise operators and variables, this conversion returns the theorem

$$\ldots, \mathsf{PWORDLEN}\,n\,w_i, \ldots \vdash \mathsf{PWORDLEN}\,n\,tm = \mathsf{T}$$

where there is one assumption $\mathsf{PWORDLEN}\,n\,w_i$ for each simple variable w_i in tm. This conversion automatically descends into the subterms until it reaches all variables.

`PWORDLEN_TAC :(term list -> tactic)` When applied to a goal of the form PWORDLEN `n tm`, the tactic `PWORDLEN_TAC tms` solves it if the conversion `PWORDLEN_CONV tms` returns a theorem without assumptions. Otherwise, the assumptions of the theorem returned by the conversion become the new subgoals.

1.3 Working with words

The basic technique for reasoning about words with the `word` library is by structural induction on the size of the word. Since the structure of words is linear and symmetric, structural induction can be carried out from either end using the WCAT operation as the basic constructor. In addition, structural analysis can be done at any position of a word. In general, there are three theorems associated with each basic word function: one for each kind of structural analysis. Considering the function NBWORD as an example, the theorem `NBWORD_SUC` described in Section 1.2.5 is for structural induction from the right hand end. The theorem `NBWORD_SUC_LEFT` shown below is for structural induction from the left hand end, and the theorem `NBWORD_SPLIT` is for structural analysis at any position.

HOL Theorem (`NBWORD_SUC_LEFT`)

$$\begin{aligned}&\vdash \forall n\, m.\, \mathsf{NBWORD}\,(\mathsf{SUC}\, n)\, m = \\ &\quad \mathsf{WCAT}\,(\mathsf{WORD}\,[\mathsf{VB}\,((m\ \mathsf{DIV}\ (2)\ \mathsf{EXP}\ (n))\ \mathsf{MOD}\ 2)], \mathsf{NBWORD}\, n\, m)\end{aligned}$$

HOL Theorem (`NBWORD_SPLIT`)

$$\begin{aligned}&\vdash \forall n_1\, n_2\, m.\, \mathsf{NBWORD}\,(n_1 + n_2)\, m = \\ &\quad \mathsf{WCAT}\,(\mathsf{NBWORD}\, n_1\, (m\ \mathsf{DIV}\ (2)\ \mathsf{EXP}\ (n_2)), \mathsf{NBWORD}\, n_2\, m)\end{aligned}$$

The following example uses structural induction from the right hand end to prove a theorem about taking an n-bit segment of an $(n+1)$-bit word which is the result of converting a natural number using the function NBWORD. We first set up the goal

$$?-\quad \forall n\, m.\, \mathsf{WSEG}\, n\, 0 (\mathsf{NBWORD}(\mathsf{SUC}\, n)\, m) = \mathsf{NBWORD}\, n\, m.$$

Then, the induction tactic `INDUCT_TAC` is applied to the size of the word. This generates two subgoals. The first subgoal, corresponding to the base case of the induction, is

$$?-\quad \mathsf{WSEG}\, 0\, 0 (\mathsf{NBWORD}(\mathsf{SUC}\, 0)\, m) = \mathsf{NBWORD}\, 0\, m.$$

This is trivial to solve since a zero-bit segment of a word is WORD[ ] and converting a number to a zero-bit word always gives the same result. The second subgoal corresponding to the step case of the induction is

$$?-\quad \forall m.\, \mathsf{WSEG}\,(\mathsf{SUC}\, n)\, 0\, (\mathsf{NBWORD}\,(\mathsf{SUC}\,(\mathsf{SUC}\, n))\, m) = \mathsf{NBWORD}\,(\mathsf{SUC}\, n)\, m$$

The right hand end induction theorem for NBWORD, `NBWORD_SUC`, can now be used to rewrite the goal. Rewriting the resulting goal further with the theorem `WSEG_WCAT_WSEG` and simplifying the result reduces it to

$$?-\quad \mathsf{WSEG}\, n\, 0\, (\mathsf{NBWORD}\,(\mathsf{SUC}\, n)\, (m\ \mathsf{DIV}\ 2)) = \mathsf{NBWORD}\, n\, (m\ \mathsf{DIV}\ 2).$$

```
let WSEG_NBWORD_SUC = PROVE(
    "!n m. (WSEG n 0(NBWORD (SUC n) m) = NBWORD n m)",
    INDUCT_TAC THENL[
     REWRITE_TAC[NBWORD0;WSEG0];
     GEN_TAC THEN PURE_ONCE_REWRITE_TAC[NBWORD_SUC]
     THEN RESQ_REWRITE1_TAC (SPECL["SUC n"; "1"] WSEG_WCAT_WSEG) THENL[
      MATCH_ACCEPT_TAC PWORDLEN_NBWORD;
      MATCH_ACCEPT_TAC PWORDLEN1;
      PURE_ONCE_REWRITE_TAC[GSYM ADD1] THEN PURE_ONCE_REWRITE_TAC[ADD_0]
      THEN MATCH_ACCEPT_TAC LESS_EQ_SUC_REFL;
      CONV_TAC (RAND_CONV num_CONV) THEN MATCH_ACCEPT_TAC LESS_0;
      CONV_TAC ((RATOR_CONV o RAND_CONV) num_CONV)
      THEN PURE_REWRITE_TAC[ADD_0;LESS_EQ_MONO]
      THEN MATCH_ACCEPT_TAC ZERO_LESS_EQ;
      PURE_REWRITE_TAC[SUB_0;ADD_0;SUC_SUB1]
      THEN PURE_ONCE_ASM_REWRITE_TAC[]
      THEN RESQ_REWRITE1_TAC (SPEC "1" WSEG_WORD_LENGTH)
      THEN REFL_TAC]]);;
```

Figure 1.2: A proof of the theorem WSEG_NBWORD_SUC

The induction hypothesis can then be used to solve the goal. However, as the theorem WSEG_WCAT_WSEG is restricted universally quantified, ordinary rewriting tactics, such as REWRITE_TAC, cannot use it to rewrite the goal. Special tactics are required. The res_quan library provides the basic facilities for manipulating restricted quantifications[1]. The complete proof is listed in Figure 1.2.

Chapter 2

ML Functions in the word Library

This chapter provides documentation on all the ML functions that are made available in HOL when the word library is loaded. This documentation is also available online via the help facility.

BIT_CONV

```
BIT_CONV : conv
```

Synopsis

Computes by inference the result of accessing a bit in a word.

Description

For any word of the form WORD[b(n-1);...;bk;...;b0], the result of evaluating

```
BIT_CONV "BIT k (WORD [b(n-1);...;bk;...;b0])"
```

is the theorem

```
|- BIT k (WORD [b(n-1);...;bk;...;b0]) = bk
```

The bits are indexed form the end of the list and starts from 0.

Failure

BIT_CONV tm fails if tm is not of the form "BIT k w" where w is as described above, or k is not less than the size of the word.

See also

WSEG_CONV

PWORDLEN_bitop_CONV

```
PWORDLEN_bitop_CONV : conv
```

Synopsis
Computes by inference the predicate asserting the size of a word.

Description
For a term tm of type :(bool)word involving only a combination of bitwise operators WNOT, WAND, WOR, WXOR and variables, the result of evaluating

```
PWORDLEN_bitop_CONV  "PWORDLEN n tm"
```

is the theorem

```
..., PWORDLEN n vi, ... |- PWORDLEN n tm = T
```

Each free variable occurred in tm will have a corresponding clause in the assumption. This conversion recursively descends into the subterms of tm until it reaches all simple variables.

Failure
PWORDLEN_bitop_CONV tm fails if constants other than those mentioned above occur in tm.

See also
PWORDLEN_CONV, PWORDLEN_TAC

PWORDLEN_CONV

```
PWORDLEN_CONV : term list -> conv
```

Synopsis
Computes by inference the predicate asserting the size of a word.

Description
For any term tm of type :(*)word, the result of evaluating

```
PWORDLEN_CONV tms "PWORDLEN n tm"
```

where n must be a numeric constant, is the theorem

```
A |- PWORDLEN n tm = T
```

where the new assumption(s) A depends on the actual form of the term tm.

If tm is an application of the unary bitwise operator WNOT, i.e., tm = WNOT tm', then A will be PWORDLEN n tm'. If tm is an application of one of the binary bitwise operators: WAND, WOR and WXOR, then A will be PWORDLEN n tm', PWORDLEN n tm''. If tm is WORD [b(n-1);...;b0], then A is empty. The length of the list must agree with n. In all above cases, the term list argument is irrelevant. An empty list could be supplied.

If tm is WSEG n k tm', then the term list tms should be [N] which indicates the size of tm', and the assumption A will be PWORDLEN N tm'.

If tm is WCAT(tm',tm''), then the term list tms should be [n1;n2] which tells the sizes of the words to be concatenated. The assumption will be PWORDLEN n1 tm', PWORDLEN n2 tm''. The value of n must be the sum of n1 and n2.

Failure

PWORDLEN_CONV tms tm fails if tm is not of the form described above.

See also

PWORDLEN_bitop_CONV, PWORDLEN_TAC

PWORDLEN_TAC

```
PWORDLEN_TAC : term list -> tactic
```

Synopsis

Tactic to solve a goal about the size of a word.

Description

When applied to a goal A ?- PWORDLEN n tm, the tactic PWORDLEN_TAC tms solves it if the conversion PWORDLEN_CONV tms returns a theorem

```
A' |- PWORDLEN n tm
```

where A' is either empty or every clause in it occurs in the assumption of the goal A. Otherwise, each clause in A' which does not appear in A becomes a new subgoal.

Failure

PWORDLEN_TAC tms fails if the corresponding conversion PWORDLEN_CONV fails.

See also

PWORDLEN_CONV

WSEG_CONV

```
WSEG_CONV : conv
```

Synopsis
Computes by inference the result of taking a segment from a word.

Description
For any word of the form `WORD[b(n-1);...;bk;...;b0]`, the result of evaluating

```
WSEG_CONV "WSEG m k (WORD [b(n-1);...;bk;...;b0])",
```

where `m` and `k` must be numeric constants, is the theorem

```
|- WSEG m k (WORD [b(n-1);...;bk;...;b0]) = [b(m+k-1);...;bk]
```

The bits are indexed form the end of the list and starts from 0.

Failure
`WSEG_CONV tm` fails if `tm` is not of the form described above, or `m + k` is not less than the size of the word.

See also
`BIT_CONV`, `WSEG_WSEG_CONV`

WSEG_WSEG_CONV

```
WSEG_WSEG_CONV : term -> conv
```

Synopsis
Computes by inference the result of taking a segment from a segment of a word.

Description
For any word `w` of size `n`, the result of evaluating

```
WSEG_WSEG_CONV "n" "WSEG m2 k2 (WSEG m1 k1 w)"
```

where `m2`, `k2`, `m1` and `k1` must be numeric constants, is the theorem

```
PWORDLEN n w |- WSEG m2 k2 (WSEG m1 k1 w) = WSEG m2 k w
```

where `k` is a numeric constant whose value is the sum of `k1` and `k2`.

Failure

`WSEG_WSEG_CONV tm` fails if `tm` is not of the form described above, or the relations `k1 + m1 <= n` and `k2 + m2 <= m1` are not satisfied.

See also

`BIT_CONV`, `WSEG_CONV`

Chapter 3

Pre-proved Theorems

The sections that follow list all theorems in the theory word. The theorems listed in this chapter will be available by name at the top-level when the theories in which they are declared are open-ed.

3.1 The theory word_base

```
BIT0 (word_base)
  |- !b. BIT 0 (WORD [b]) = b

BIT_DEF (word_base)
  |- !k l. BIT k (WORD l) = ELL k l

BIT_EQ_IMP_WORD_EQ (word_base)
  |- !n (w1::PWORDLEN n) (w2::PWORDLEN n).
       (!k. k < n ==> (BIT k w1 = BIT k w2)) ==> (w1 = w2)

BIT_WCAT1 (word_base)
  |- !n (w::PWORDLEN n) b. BIT n (WCAT (WORD [b],w)) = b

BIT_WCAT_FST (word_base)
  |- !n1 n2 (w1::PWORDLEN n1) (w2::PWORDLEN n2) k.
       n2 <= k /\ k < n1 + n2 ==> (BIT k (WCAT (w1,w2)) = BIT (k - n2) w1)

BIT_WCAT_SND (word_base)
  |- !n1 n2 (w1::PWORDLEN n1) (w2::PWORDLEN n2) k.
       k < n2 ==> (BIT k (WCAT (w1,w2)) = BIT k w2)

BIT_WSEG (word_base)
  |- !n (w::PWORDLEN n) m k j.
       m + k <= n ==> j < m ==> (BIT j (WSEG m k w) = BIT (j + k) w)

ii_internalword_base0_def (word_base)
  |- ii_internalword_base0 =
     (\a. ii_internal_mk_word ((\a. CONSTR 0 a (\n. BOTTOM)) a))
```

```
LSB  (word_base)
   |- !n (w::PWORDLEN n). 0 < n ==> (LSB w = BIT 0 w)

LSB_DEF  (word_base)
   |- !l. LSB (WORD l) = LAST l

MSB  (word_base)
   |- !n (w::PWORDLEN n). 0 < n ==> (MSB w = BIT (PRE n) w)

MSB_DEF  (word_base)
   |- !l. MSB (WORD l) = HD l

PWORDLEN  (word_base)
   |- !n w. PWORDLEN n w = (WORDLEN w = n)

PWORDLEN0  (word_base)
   |- !w. PWORDLEN 0 w ==> (w = WORD [])

PWORDLEN1  (word_base)
   |- !x. PWORDLEN 1 (WORD [x])

PWORDLEN_DEF  (word_base)
   |- !n l. PWORDLEN n (WORD l) = (n = LENGTH l)

WCAT0  (word_base)
   |- !w. (WCAT (WORD [],w) = w) /\ (WCAT (w,WORD []) = w)

WCAT_11  (word_base)
   |- !m n (wm1::PWORDLEN m) (wm2::PWORDLEN m) (wn1::PWORDLEN n)
        (wn2::PWORDLEN n).
       (WCAT (wm1,wn1) = WCAT (wm2,wn2)) = (wm1 = wm2) /\ (wn1 = wn2)

WCAT_ASSOC  (word_base)
   |- !w1 w2 w3. WCAT (w1,WCAT (w2,w3)) = WCAT (WCAT (w1,w2),w3)

WCAT_DEF  (word_base)
   |- !l1 l2. WCAT (WORD l1,WORD l2) = WORD (APPEND l1 l2)

WCAT_PWORDLEN  (word_base)
   |- !n1 (w1::PWORDLEN n1) n2 (w2::PWORDLEN n2).
       PWORDLEN (n1 + n2) (WCAT (w1,w2))

WCAT_WSEG_WSEG  (word_base)
   |- !n (w::PWORDLEN n) m1 m2 k.
       m1 + (m2 + k) <= n ==>
       (WCAT (WSEG m2 (m1 + k) w,WSEG m1 k w) = WSEG (m1 + m2) k w)
```

```
WORD  (word_base)
  |- WORD = ii_internalword_base0

WORDLEN_DEF  (word_base)
  |- !l. WORDLEN (WORD l) = LENGTH l

WORDLEN_SUC_WCAT  (word_base)
  |- !n w.
       PWORDLEN (SUC n) w ==>
       ?(b::PWORDLEN 1) (w'::PWORDLEN n). w = WCAT (b,w')

WORDLEN_SUC_WCAT_BIT_WSEG  (word_base)
  |- !n (w::PWORDLEN (SUC n)). w = WCAT (WORD [BIT n w],WSEG n 0 w)

WORDLEN_SUC_WCAT_BIT_WSEG_RIGHT  (word_base)
  |- !n (w::PWORDLEN (SUC n)). w = WCAT (WSEG n 1 w,WORD [BIT 0 w])

WORDLEN_SUC_WCAT_WSEG_WSEG  (word_base)
  |- !w::PWORDLEN (SUC n). w = WCAT (WSEG 1 n w,WSEG n 0 w)

WORDLEN_SUC_WCAT_WSEG_WSEG_RIGHT  (word_base)
  |- !w::PWORDLEN (SUC n). w = WCAT (WSEG n 1 w,WSEG 1 0 w)

WORD_11  (word_base)
  |- !l l'. (WORD l = WORD l') = (l = l')

word_11  (word_base)
  |- !a a'. (WORD a = WORD a') = (a = a')

word_Ax  (word_base)
  |- !f. ?fn. !a. fn (WORD a) = f a

word_Axiom  (word_base)
  |- !f. ?fn. !a. fn (WORD a) = f a

word_cases  (word_base)
  |- !w. ?l. w = WORD l

word_case_cong  (word_base)
  |- !f' f M' M.
       (M = M') /\ (!a. (M' = WORD a) ==> (f a = f' a)) ==>
       (word_case f M = word_case f' M')

word_case_def  (word_base)
  |- !f a. word_case f (WORD a) = f a
```

```
WORD_CONS_WCAT  (word_base)
   |- !x l. WORD (x::l) = WCAT (WORD [x],WORD l)

WORD_DEF  (word_base)
   |- !l. WORD l = ABS_word (Node l [])

word_induct  (word_base)
   |- !P. (!l. P (WORD l)) ==> !w. P w

word_induction  (word_base)
   |- !P. (!l. P (WORD l)) ==> !w. P w

word_ISO_DEF  (word_base)
   |- (!a. ABS_word (REP_word a) = a) /\
      !r.
        TRP (\v tl. (?l. v = l) /\ (LENGTH tl = 0)) r =
        (REP_word (ABS_word r) = r)

word_nchotomy  (word_base)
   |- !w. ?l. w = WORD l

WORD_PARTITION  (word_base)
   |- (!n (w::PWORDLEN n) m. m <= n ==> (WCAT (WSPLIT m w) = w)) /\
      !n m (w1::PWORDLEN n) (w2::PWORDLEN m).
        WSPLIT m (WCAT (w1,w2)) = (w1,w2)

word_repfns  (word_base)
   |- (!a. ii_internal_mk_word (ii_internal_dest_word a) = a) /\
      !r.
        (\a0.
           !'word'.
             (!a0.
                (?a. a0 = (\a. CONSTR 0 a (\n. BOTTOM)) a) ==>
                'word' a0) ==>
             'word' a0) r =
        (ii_internal_dest_word (ii_internal_mk_word r) = r)

word_size_def  (word_base)
   |- !f a. word_size f (WORD a) = 1 + list_size f a

word_size_full_def  (word_base)
   |- !f a. word_size f (WORD a) = 1 + list_size f a

WORD_SNOC_WCAT  (word_base)
   |- !l x. WORD (SNOC x l) = WCAT (WORD l,WORD [x])
```

```
WORD_SPLIT  (word_base)
  |- !n1 n2 (w::PWORDLEN (n1 + n2)). w = WCAT (WSEG n1 n2 w,WSEG n2 0 w)

word_TY_DEF  (word_base)
  |- ?rep.
       TYPE_DEFINITION
         (\a0.
            !'word'.
              (!a0.
                 (?a. a0 = (\a. CONSTR 0 a (\n. BOTTOM)) a) ==>
                 'word' a0) ==>
              'word' a0) rep

WSEG0  (word_base)
  |- !k w. WSEG 0 k w = WORD []

WSEG_BIT  (word_base)
  |- !n (w::PWORDLEN n) k. k < n ==> (WSEG 1 k w = WORD [BIT k w])

WSEG_DEF  (word_base)
  |- !m k l. WSEG m k (WORD l) = WORD (LASTN m (BUTLASTN k l))

WSEG_PWORDLEN  (word_base)
  |- !n (w::PWORDLEN n) m k. m + k <= n ==> PWORDLEN m (WSEG m k w)

WSEG_WCAT1  (word_base)
  |- !n1 n2 (w1::PWORDLEN n1) (w2::PWORDLEN n2).
       WSEG n1 n2 (WCAT (w1,w2)) = w1

WSEG_WCAT2  (word_base)
  |- !n1 n2 (w1::PWORDLEN n1) (w2::PWORDLEN n2).
       WSEG n2 0 (WCAT (w1,w2)) = w2

WSEG_WCAT_WSEG  (word_base)
  |- !n1 n2 (w1::PWORDLEN n1) (w2::PWORDLEN n2) m k.
       m + k <= n1 + n2 /\ k < n2 /\ n2 <= m + k ==>
       (WSEG m k (WCAT (w1,w2)) =
        WCAT (WSEG (m + k - n2) 0 w1,WSEG (n2 - k) k w2))

WSEG_WCAT_WSEG1  (word_base)
  |- !n1 n2 (w1::PWORDLEN n1) (w2::PWORDLEN n2) m k.
       m <= n1 /\ n2 <= k ==>
       (WSEG m k (WCAT (w1,w2)) = WSEG m (k - n2) w1)

WSEG_WCAT_WSEG2  (word_base)
  |- !n1 n2 (w1::PWORDLEN n1) (w2::PWORDLEN n2) m k.
       m + k <= n2 ==> (WSEG m k (WCAT (w1,w2)) = WSEG m k w2)
```

WSEG_WORDLEN (word_base)
```
|- !n (w::PWORDLEN n) m k. m + k <= n ==> (WORDLEN (WSEG m k w) = m)
```

WSEG_WORD_LENGTH (word_base)
```
|- !n (w::PWORDLEN n). WSEG n 0 w = w
```

WSEG_WSEG (word_base)
```
|- !n (w::PWORDLEN n) m1 k1 m2 k2.
     m1 + k1 <= n /\ m2 + k2 <= m1 ==>
     (WSEG m2 k2 (WSEG m1 k1 w) = WSEG m2 (k1 + k2) w)
```

WSPLIT_DEF (word_base)
```
|- !m l. WSPLIT m (WORD l) = (WORD (BUTLASTN m l),WORD (LASTN m l))
```

WSPLIT_PWORDLEN (word_base)
```
|- !n (w::PWORDLEN n) m.
     m <= n ==>
     PWORDLEN (n - m) (FST (WSPLIT m w)) /\
     PWORDLEN m (SND (WSPLIT m w))
```

WSPLIT_WSEG (word_base)
```
|- !n (w::PWORDLEN n) k.
     k <= n ==> (WSPLIT k w = (WSEG (n - k) k w,WSEG k 0 w))
```

WSPLIT_WSEG1 (word_base)
```
|- !n (w::PWORDLEN n) k.
     k <= n ==> (FST (WSPLIT k w) = WSEG (n - k) k w)
```

WSPLIT_WSEG2 (word_base)
```
|- !n (w::PWORDLEN n) k. k <= n ==> (SND (WSPLIT k w) = WSEG k 0 w)
```

3.2 The theory word_bitop

EXISTSABIT (word_bitop)
```
|- !n (w::PWORDLEN n) P. EXISTSABIT P w = ?k. k < n /\ P (BIT k w)
```

EXISTSABIT_DEF (word_bitop)
```
|- !P l. EXISTSABIT P (WORD l) = SOME_EL P l
```

EXISTSABIT_WCAT (word_bitop)
```
|- !w1 w2 P.
     EXISTSABIT P (WCAT (w1,w2)) = EXISTSABIT P w1 \/ EXISTSABIT P w2
```

```
EXISTSABIT_WSEG  (word_bitop)
  |- !n (w::PWORDLEN n) m k.
       m + k <= n ==> !P. EXISTSABIT P (WSEG m k w) ==> EXISTSABIT P w

FORALLBITS  (word_bitop)
  |- !n (w::PWORDLEN n) P. FORALLBITS P w = !k. k < n ==> P (BIT k w)

FORALLBITS_DEF  (word_bitop)
  |- !P l. FORALLBITS P (WORD l) = ALL_EL P l

FORALLBITS_WCAT  (word_bitop)
  |- !w1 w2 P.
       FORALLBITS P (WCAT (w1,w2)) = FORALLBITS P w1 /\ FORALLBITS P w2

FORALLBITS_WSEG  (word_bitop)
  |- !n (w::PWORDLEN n) P.
       FORALLBITS P w ==> !m k. m + k <= n ==> FORALLBITS P (WSEG m k w)

NOT_EXISTSABIT  (word_bitop)
  |- !P w. ~EXISTSABIT P w = FORALLBITS ($~ o P) w

NOT_FORALLBITS  (word_bitop)
  |- !P w. ~FORALLBITS P w = EXISTSABIT ($~ o P) w

PBITBOP_DEF  (word_bitop)
  |- !op.
       PBITBOP op =
       !n (w1::PWORDLEN n) (w2::PWORDLEN n).
         PWORDLEN n (op w1 w2) /\
         !m k.
           m + k <= n ==>
           (op (WSEG m k w1) (WSEG m k w2) = WSEG m k (op w1 w2))

PBITBOP_EXISTS  (word_bitop)
  |- !f. ?fn. !l1 l2. fn (WORD l1) (WORD l2) = WORD (MAP2 f l1 l2)

PBITBOP_PWORDLEN  (word_bitop)
  |- !(op::PBITBOP) n (w1::PWORDLEN n) (w2::PWORDLEN n).
       PWORDLEN n (op w1 w2)

PBITBOP_WSEG  (word_bitop)
  |- !(op::PBITBOP) n (w1::PWORDLEN n) (w2::PWORDLEN n) m k.
       m + k <= n ==>
       (op (WSEG m k w1) (WSEG m k w2) = WSEG m k (op w1 w2))
```

```
PBITOP_BIT  (word_bitop)
  |- !(op::PBITOP) n (w::PWORDLEN n) k.
       k < n ==> (op (WORD [BIT k w]) = WORD [BIT k (op w)])
```

```
PBITOP_DEF  (word_bitop)
  |- !op.
       PBITOP op =
       !n (w::PWORDLEN n).
         PWORDLEN n (op w) /\
         !m k. m + k <= n ==> (op (WSEG m k w) = WSEG m k (op w))
```

```
PBITOP_PWORDLEN  (word_bitop)
  |- !(op::PBITOP) n (w::PWORDLEN n). PWORDLEN n (op w)
```

```
PBITOP_WSEG  (word_bitop)
  |- !(op::PBITOP) n (w::PWORDLEN n) m k.
       m + k <= n ==> (op (WSEG m k w) = WSEG m k (op w))
```

```
SHL_DEF  (word_bitop)
  |- !f w b.
       SHL f w b =
       (BIT (PRE (WORDLEN w)) w,
        WCAT
          (WSEG (PRE (WORDLEN w)) 0 w,
           (if f then WSEG 1 0 w else WORD [b])))
```

```
SHL_WSEG  (word_bitop)
  |- !n (w::PWORDLEN n) m k.
       m + k <= n ==>
       0 < m ==>
       !f b.
         SHL f (WSEG m k w) b =
         (BIT (k + (m - 1)) w,
          (if f then
             WCAT (WSEG (m - 1) k w,WSEG 1 k w)
           else
             WCAT (WSEG (m - 1) k w,WORD [b])))
```

```
SHL_WSEG_1F  (word_bitop)
  |- !n (w::PWORDLEN n) m k.
       m + k <= n ==>
       0 < m ==>
       !b.
         SHL F (WSEG m k w) b =
         (BIT (k + (m - 1)) w,WCAT (WSEG (m - 1) k w,WORD [b]))
```

SHL_WSEG_NF (word_bitop)
```
  |- !n (w::PWORDLEN n) m k.
       m + k <= n ==>
       0 < m ==>
       0 < k ==>
       (SHL F (WSEG m k w) (BIT (k - 1) w) =
        (BIT (k + (m - 1)) w,WSEG m (k - 1) w))
```

SHR_DEF (word_bitop)
```
  |- !f b w.
       SHR f b w =
       (WCAT
          ((if f then WSEG 1 (PRE (WORDLEN w)) w else WORD [b]),
           WSEG (PRE (WORDLEN w)) 1 w),BIT 0 w)
```

SHR_WSEG (word_bitop)
```
  |- !n (w::PWORDLEN n) m k.
       m + k <= n ==>
       0 < m ==>
       !f b.
         SHR f b (WSEG m k w) =
         ((if f then
             WCAT (WSEG 1 (k + (m - 1)) w,WSEG (m - 1) (k + 1) w)
           else
             WCAT (WORD [b],WSEG (m - 1) (k + 1) w)),BIT k w)
```

SHR_WSEG_1F (word_bitop)
```
  |- !n (w::PWORDLEN n) m k.
       m + k <= n ==>
       0 < m ==>
       !b.
         SHR F b (WSEG m k w) =
         (WCAT (WORD [b],WSEG (m - 1) (k + 1) w),BIT k w)
```

SHR_WSEG_NF (word_bitop)
```
  |- !n (w::PWORDLEN n) m k.
       m + k < n ==>
       0 < m ==>
       (SHR F (BIT (m + k) w) (WSEG m k w) = (WSEG m (k + 1) w,BIT k w))
```

WMAP_0 (word_bitop)
```
  |- !f. WMAP f (WORD []) = WORD []
```

WMAP_BIT (word_bitop)
```
  |- !n (w::PWORDLEN n) k. k < n ==> !f. BIT k (WMAP f w) = f (BIT k w)
```

```
WMAP_DEF   (word_bitop)
   |- !f l. WMAP f (WORD l) = WORD (MAP f l)

WMAP_o   (word_bitop)
   |- !w f g. WMAP g (WMAP f w) = WMAP (g o f) w

WMAP_PBITOP   (word_bitop)
   |- !f. PBITOP (WMAP f)

WMAP_PWORDLEN   (word_bitop)
   |- !(w::PWORDLEN n) f. PWORDLEN n (WMAP f w)

WMAP_WCAT   (word_bitop)
   |- !w1 w2 f. WMAP f (WCAT (w1,w2)) = WCAT (WMAP f w1,WMAP f w2)

WMAP_WSEG   (word_bitop)
   |- !n (w::PWORDLEN n) m k.
        m + k <= n ==> !f. WMAP f (WSEG m k w) = WSEG m k (WMAP f w)

WSEG_SHL   (word_bitop)
   |- !n (w::PWORDLEN (SUC n)) m k.
        0 < k /\ m + k <= SUC n ==>
        !b. WSEG m k (SND (SHL f w b)) = WSEG m (k - 1) w

WSEG_SHL_0   (word_bitop)
   |- !n (w::PWORDLEN (SUC n)) m b.
        0 < m /\ m <= SUC n ==>
        (WSEG m 0 (SND (SHL f w b)) =
         WCAT (WSEG (m - 1) 0 w,(if f then WSEG 1 0 w else WORD [b])))
```

3.3 The theory word_num

```
LVAL   (word_num)
   |- (!f b. LVAL f b [] = 0) /\
      !l f b x. LVAL f b (x::l) = f x * b EXP LENGTH l + LVAL f b l

LVAL_DEF   (word_num)
   |- !f b l. LVAL f b l = FOLDL (\e x. b * e + f x) 0 l

LVAL_MAX   (word_num)
   |- !l f b. (!x. f x < b) ==> LVAL f b l < b EXP LENGTH l

LVAL_SNOC   (word_num)
   |- !l h f b. LVAL f b (SNOC h l) = LVAL f b l * b + f h
```

```
NLIST_DEF   (word_num)
   |- (!frep b m. NLIST 0 frep b m = []) /\
      !n frep b m.
        NLIST (SUC n) frep b m =
        SNOC (frep (m MOD b)) (NLIST n frep b (m DIV b))

NVAL0   (word_num)
   |- !f b. NVAL f b (WORD []) = 0

NVAL1   (word_num)
   |- !f b x. NVAL f b (WORD [x]) = f x

NVAL_DEF   (word_num)
   |- !f b l. NVAL f b (WORD l) = LVAL f b l

NVAL_MAX   (word_num)
   |- !f b. (!x. f x < b) ==> !n (w::PWORDLEN n). NVAL f b w < b EXP n

NVAL_WCAT   (word_num)
   |- !n m (w1::PWORDLEN n) (w2::PWORDLEN m) f b.
        NVAL f b (WCAT (w1,w2)) = NVAL f b w1 * b EXP m + NVAL f b w2

NVAL_WCAT1   (word_num)
   |- !w f b x. NVAL f b (WCAT (w,WORD [x])) = NVAL f b w * b + f x

NVAL_WCAT2   (word_num)
   |- !n (w::PWORDLEN n) f b x.
        NVAL f b (WCAT (WORD [x],w)) = f x * b EXP n + NVAL f b w

NVAL_WORDLEN_0   (word_num)
   |- !(w::PWORDLEN 0) fv r. NVAL fv r w = 0

NWORD_DEF   (word_num)
   |- !n frep b m. NWORD n frep b m = WORD (NLIST n frep b m)

NWORD_LENGTH   (word_num)
   |- !n f b m. WORDLEN (NWORD n f b m) = n

NWORD_PWORDLEN   (word_num)
   |- !n f b m. PWORDLEN n (NWORD n f b m)
```

3.4 The theory bword_bitop

```
PBITBOP_WAND   (bword_bitop)
   |- PBITBOP $WAND
```

```
PBITBOP_WOR  (bword_bitop)
   |- PBITBOP $WOR

PBITBOP_WXOR  (bword_bitop)
   |- PBITBOP $WXOR

PBITOP_WNOT  (bword_bitop)
   |- PBITOP WNOT

WAND_DEF  (bword_bitop)
   |- !l1 l2. WORD l1 WAND WORD l2 = WORD (MAP2 $/\ l1 l2)

WCAT_WNOT  (bword_bitop)
   |- !n1 n2 (w1::PWORDLEN n1) (w2::PWORDLEN n2).
        WCAT (WNOT w1,WNOT w2) = WNOT (WCAT (w1,w2))

WNOT_DEF  (bword_bitop)
   |- !l. WNOT (WORD l) = WORD (MAP $~ l)

WNOT_WNOT  (bword_bitop)
   |- !w. WNOT (WNOT w) = w

WOR_DEF  (bword_bitop)
   |- !l1 l2. WORD l1 WOR WORD l2 = WORD (MAP2 $\/ l1 l2)

WXOR_DEF  (bword_bitop)
   |- !l1 l2. WORD l1 WXOR WORD l2 = WORD (MAP2 (\x y. ~(x = y)) l1 l2)
```

3.5 The theory bword_num

```
ADD_BNVAL_LEFT  (bword_num)
   |- !n (w1::PWORDLEN (SUC n)) (w2::PWORDLEN (SUC n)).
        BNVAL w1 + BNVAL w2 =
        (BV (BIT n w1) + BV (BIT n w2)) * 2 EXP n +
        (BNVAL (WSEG n 0 w1) + BNVAL (WSEG n 0 w2))

ADD_BNVAL_RIGHT  (bword_num)
   |- !n (w1::PWORDLEN (SUC n)) (w2::PWORDLEN (SUC n)).
        BNVAL w1 + BNVAL w2 =
        (BNVAL (WSEG n 1 w1) + BNVAL (WSEG n 1 w2)) * 2 +
        (BV (BIT 0 w1) + BV (BIT 0 w2))
```

ADD_BNVAL_SPLIT (bword_num)
```
  |- !n1 n2 (w1::PWORDLEN (n1 + n2)) (w2::PWORDLEN (n1 + n2)).
       BNVAL w1 + BNVAL w2 =
       (BNVAL (WSEG n1 n2 w1) + BNVAL (WSEG n1 n2 w2)) * 2 EXP n2 +
       (BNVAL (WSEG n2 0 w1) + BNVAL (WSEG n2 0 w2))
```

BIT_NBWORD0 (bword_num)
```
  |- !k n. k < n ==> (BIT k (NBWORD n 0) = F)
```

BNVAL0 (bword_num)
```
  |- BNVAL (WORD []) = 0
```

BNVAL_11 (bword_num)
```
  |- !w1 w2.
       (WORDLEN w1 = WORDLEN w2) ==> (BNVAL w1 = BNVAL w2) ==> (w1 = w2)
```

BNVAL_DEF (bword_num)
```
  |- !l. BNVAL (WORD l) = LVAL BV 2 l
```

BNVAL_MAX (bword_num)
```
  |- !n (w::PWORDLEN n). BNVAL w < 2 EXP n
```

BNVAL_NBWORD (bword_num)
```
  |- !n m. m < 2 EXP n ==> (BNVAL (NBWORD n m) = m)
```

BNVAL_NVAL (bword_num)
```
  |- !w. BNVAL w = NVAL BV 2 w
```

BNVAL_ONTO (bword_num)
```
  |- !w. ?n. BNVAL w = n
```

BNVAL_WCAT (bword_num)
```
  |- !n m (w1::PWORDLEN n) (w2::PWORDLEN m).
       BNVAL (WCAT (w1,w2)) = BNVAL w1 * 2 EXP m + BNVAL w2
```

BNVAL_WCAT1 (bword_num)
```
  |- !n (w::PWORDLEN n) x. BNVAL (WCAT (w,WORD [x])) = BNVAL w * 2 + BV x
```

BNVAL_WCAT2 (bword_num)
```
  |- !n (w::PWORDLEN n) x.
       BNVAL (WCAT (WORD [x],w)) = BV x * 2 EXP n + BNVAL w
```

BV_DEF (bword_num)
```
  |- !b. BV b = (if b then SUC 0 else 0)
```

BV_LESS_2 (bword_num)
```
  |- !x. BV x < 2
```

```
BV_VB   (bword_num)
   |- !x. x < 2 ==> (BV (VB x) = x)

DOUBL_EQ_SHL   (bword_num)
   |- !n.
        0 < n ==>
        !(w::PWORDLEN n) b.
          NBWORD n (BNVAL w + BNVAL w + BV b) = SND (SHL F w b)

EQ_NBWORD0_SPLIT   (bword_num)
   |- !n (w::PWORDLEN n) m.
        m <= n ==>
        ((w = NBWORD n 0) =
         (WSEG (n - m) m w = NBWORD (n - m) 0) /\
         (WSEG m 0 w = NBWORD m 0))

MSB_NBWORD   (bword_num)
   |- !n m. BIT n (NBWORD (SUC n) m) = VB ((m DIV 2 EXP n) MOD 2)

NBWORD0   (bword_num)
   |- !m. NBWORD 0 m = WORD []

NBWORD_BNVAL   (bword_num)
   |- !n (w::PWORDLEN n). NBWORD n (BNVAL w) = w

NBWORD_DEF   (bword_num)
   |- !n m. NBWORD n m = WORD (NLIST n VB 2 m)

NBWORD_MOD   (bword_num)
   |- !n m. NBWORD n (m MOD 2 EXP n) = NBWORD n m

NBWORD_SPLIT   (bword_num)
   |- !n1 n2 m.
        NBWORD (n1 + n2) m = WCAT (NBWORD n1 (m DIV 2 EXP n2),NBWORD n2 m)

NBWORD_SUC   (bword_num)
   |- !n m.
        NBWORD (SUC n) m = WCAT (NBWORD n (m DIV 2),WORD [VB (m MOD 2)])

NBWORD_SUC_FST   (bword_num)
   |- !n m.
        NBWORD (SUC n) m =
        WCAT (WORD [VB ((m DIV 2 EXP n) MOD 2)],NBWORD n m)

NBWORD_SUC_WSEG   (bword_num)
   |- !n (w::PWORDLEN (SUC n)). NBWORD n (BNVAL w) = WSEG n 0 w
```

PWORDLEN_NBWORD (bword_num)
```
|- !n m. PWORDLEN n (NBWORD n m)
```

VB_BV (bword_num)
```
|- !x. VB (BV x) = x
```

VB_DEF (bword_num)
```
|- !n. VB n = ~(n MOD 2 = 0)
```

WCAT_NBWORD_0 (bword_num)
```
|- !n1 n2. WCAT (NBWORD n1 0,NBWORD n2 0) = NBWORD (n1 + n2) 0
```

WORDLEN_NBWORD (bword_num)
```
|- !n m. WORDLEN (NBWORD n m) = n
```

WSEG_NBWORD (bword_num)
```
|- !m k n.
     m + k <= n ==> !l. WSEG m k (NBWORD n l) = NBWORD m (l DIV 2 EXP k)
```

WSEG_NBWORD_SUC (bword_num)
```
|- !n m. WSEG n 0 (NBWORD (SUC n) m) = NBWORD n m
```

WSPLIT_NBWORD_0 (bword_num)
```
|- !n m.
     m <= n ==> (WSPLIT m (NBWORD n 0) = (NBWORD (n - m) 0,NBWORD m 0))
```

ZERO_WORD_VAL (bword_num)
```
|- !n (w::PWORDLEN n). (w = NBWORD n 0) = (BNVAL w = 0)
```

3.6 The theory bword_arith

ACARRY_ACARRY_WSEG (bword_arith)
```
|- !n (w1::PWORDLEN n) (w2::PWORDLEN n) cin m k1 k2.
     k1 < m /\ k2 < n /\ m + k2 <= n ==>
     (ACARRY k1 (WSEG m k2 w1) (WSEG m k2 w2) (ACARRY k2 w1 w2 cin) =
      ACARRY (k1 + k2) w1 w2 cin)
```

ACARRY_DEF (bword_arith)
```
|- (!w1 w2 cin. ACARRY 0 w1 w2 cin = cin) /\
   !n w1 w2 cin.
     ACARRY (SUC n) w1 w2 cin =
     VB
       ((BV (BIT n w1) + BV (BIT n w2) + BV (ACARRY n w1 w2 cin)) DIV 2)
```

```
ACARRY_EQ_ADD_DIV  (bword_arith)
  |- !n (w1::PWORDLEN n) (w2::PWORDLEN n) k.
       k < n ==>
       (BV (ACARRY k w1 w2 cin) =
        (BNVAL (WSEG k 0 w1) + BNVAL (WSEG k 0 w2) + BV cin) DIV 2 EXP k)

ACARRY_EQ_ICARRY  (bword_arith)
  |- !n (w1::PWORDLEN n) (w2::PWORDLEN n) cin k.
       k <= n ==> (ACARRY k w1 w2 cin = ICARRY k w1 w2 cin)

ACARRY_MSB  (bword_arith)
  |- !n (w1::PWORDLEN n) (w2::PWORDLEN n) cin.
       ACARRY n w1 w2 cin =
       BIT n (NBWORD (SUC n) (BNVAL w1 + BNVAL w2 + BV cin))

ACARRY_WSEG  (bword_arith)
  |- !n (w1::PWORDLEN n) (w2::PWORDLEN n) cin k m.
       k < m /\ m <= n ==>
       (ACARRY k (WSEG m 0 w1) (WSEG m 0 w2) cin = ACARRY k w1 w2 cin)

ADD_NBWORD_EQ0_SPLIT  (bword_arith)
  |- !n1 n2 (w1::PWORDLEN (n1 + n2)) (w2::PWORDLEN (n1 + n2)) cin.
       (NBWORD (n1 + n2) (BNVAL w1 + BNVAL w2 + BV cin) =
        NBWORD (n1 + n2) 0) =
       (NBWORD n1
          (BNVAL (WSEG n1 n2 w1) + BNVAL (WSEG n1 n2 w2) +
           BV (ACARRY n2 w1 w2 cin)) =
        NBWORD n1 0) /\
       (NBWORD n2 (BNVAL (WSEG n2 0 w1) + BNVAL (WSEG n2 0 w2) + BV cin) =
        NBWORD n2 0)

ADD_WORD_SPLIT  (bword_arith)
  |- !n1 n2 (w1::PWORDLEN (n1 + n2)) (w2::PWORDLEN (n1 + n2)) cin.
       NBWORD (n1 + n2) (BNVAL w1 + BNVAL w2 + BV cin) =
       WCAT
         (NBWORD n1
            (BNVAL (WSEG n1 n2 w1) + BNVAL (WSEG n1 n2 w2) +
             BV (ACARRY n2 w1 w2 cin)),
          NBWORD n2
            (BNVAL (WSEG n2 0 w1) + BNVAL (WSEG n2 0 w2) + BV cin))

ICARRY_DEF  (bword_arith)
  |- (!w1 w2 cin. ICARRY 0 w1 w2 cin = cin) /\
     !n w1 w2 cin.
       ICARRY (SUC n) w1 w2 cin =
       BIT n w1 /\ BIT n w2 \/
       (BIT n w1 \/ BIT n w2) /\ ICARRY n w1 w2 cin
```

ICARRY_WSEG (bword_arith)
```
  |- !n (w1::PWORDLEN n) (w2::PWORDLEN n) cin k m.
       k < m /\ m <= n ==>
       (ICARRY k (WSEG m 0 w1) (WSEG m 0 w2) cin = ICARRY k w1 w2 cin)
```

WSEG_NBWORD_ADD (bword_arith)
```
  |- !n (w1::PWORDLEN n) (w2::PWORDLEN n) m k cin.
       m + k <= n ==>
       (WSEG m k (NBWORD n (BNVAL w1 + BNVAL w2 + BV cin)) =
        NBWORD m
          (BNVAL (WSEG m k w1) + BNVAL (WSEG m k w2) +
           BV (ACARRY k w1 w2 cin)))
```

References

[1] W. Wong. *The HOL* res_quan *Library*. Computer Laboratory, University of Cambridge, 1993.

Index