NASA Technical Memorandum 110274

A Bitvectors Library For PVS

Ricky W. Butler
Paul S. Miner
Langley Research Center, Hampton, Virginia

Mandayam K. Srivas
SRI International, Menlo Park, California

Dave A. Greve
Steven P. Miller
Rockwell Collins, Cedar Rapids, Iowa

August 1996
## Contents

1 Introduction 3

2 Fundamental Definition of a Bitvector 3

3 Natural Number Interpretations of a Bitvector 4

4 Bitwise Logical Operations on Bitvectors 5

5 Bitvector Concatenation 6

6 Extraction Operator 7

7 Shift Operations on Bitvectors 8

8 Bitvector Rotation 8

9 Zero and Sign-Extend Operators 9

10 Theorems Involving Concatenation and Extraction 10

11 2's Complement Interpretations of a Bitvector 11

12 Bitvector Arithmetic 12
   12.1 Definition of Arithmetic Operators 12
   12.2 Arithmetic Properties of Shifting 13
   12.3 Theorems about 2's Complement Arithmetic 14

13 Overflow 15

14 Library Organization 15
1 Introduction

The method used for specifying the parallel data lines of a hardware device is fundamental to any hardware verification. These lines consist of an ordered set of 0's and 1's, usually called bits. The ordered set of bits is referred to as a bitvector. Although a human reader of a circuit design automatically “interprets” these bitvectors as natural numbers, 2’s complement integers, characters, or some other encoded object, a formal model must explicitly account for these interpretations. For example, if bv is a bitvector, a function, say bv2nat, must be applied to bv in order to convert it to a natural number, i.e. bv2nat(bv).

The bitvectors library has been developed for PVS [1, 2, 3, 4, 5, 6] with several goals in mind:

- All of the common functions that interpret and operate on bitvectors should be defined in a manner that is simple and reusable.
- The library should not introduce new axioms. In this way the library will be consistent if PVS is consistent.
- The library should provide a complete set of operators on bit-vectors that hide the particular bitvector implementation used. Thus, if the definition of the bitvector type were change from its current functional form to another form (e.g., a list form), the interface to the user would remain the same.
- The library should be organized in a manner that supports a variety of hardware, without imposing a heavy overhead. In other words, specific parts of the library should be accessible without being exposed to extraneous definitions.
- The library should facilitate the connection to different hardware design tools.

Similar libraries have been constructed for many other systems including the Boyer-Moore theorem prover [7] and the Cambridge Higher Order Logic (HOL) system [8].

The bitvectors library is available via the World Wide Web at


in the file bitvectors.dmp.

2 Fundamental Definition of a Bitvector

There are several methods one could use to define a bitvector in PVS. Three reasonable candidates are:

- a list of bits
- a finite sequence of bits
- a function from \{0,1,2, ...,N-1\} into \{0,1\}. 

3
The third method has been used in this library. A bit is defined as:

\[\text{bit : TYPE} = \{n : \text{nat} \mid n \leq 1\}\]

and a bit-vector is defined as

\[\text{bvec : TYPE} = [\text{below}(N) \rightarrow \text{bit}]\]

Thus the type bvec is a function from below(N) to bit. The domain of the function is specified using the type below which is predefined in the PVS prelude as:

\[\text{below}(i) : \text{TYPE} = \{s : \text{nat} \mid s < i\}\]

The symbol N is a constant natural number representing the length of the bitvector. It is imported into the basic theory using PVS's theory parameterization capability:

\[
\text{bv[N : nat]} : \text{THEORY}
\begin{align*}
\text{BEGIN} \\
\text{bit : TYPE} = \{n : \text{nat} \mid n \leq 1\} \\
\text{bvec : TYPE} = [\text{below}(N) \rightarrow \text{bit}] \\
\text{END bv}
\end{align*}
\]

This definition allows the use of empty bitvectors, which is primarily useful when using the concatenation operators defined in a subsequent section.

A bitvector of length N is defined as follows:

\[\text{bv : VAR bvec[N]}\]

and the ith bit can be retrieved in two ways: \(\text{bv}(i)\) or \(\text{bv}^i\). The latter method has the advantage that it is implementation independent. The ^ operator is defined as follows:

\[\wedge(\text{bv : bvec, (i : below(N))}) : \text{bit} = \text{bv}(i)\]

3  **Natural Number Interpretations of a Bitvector**

A bitvector is interpreted as a natural number through use of a function named bv2nat. This function is defined as follows:

\[
\text{bv_nat[N : nat]} : \text{THEORY}
\begin{align*}
\text{BEGIN} \\
\text{IMPORTING bv[N], exp2}
\end{align*}
\]
\text{bv2nat\_rec}(n: \text{upto}(N), \text{bv}: \text{bvec}): \text{RECURSIVE} \text{ nat } = \\
\text{IF} \ n = 0 \ \text{THEN} \ 0 \\
\text{ELSE} \ \text{exp2}(n-1) \ast \text{bv}\text{\textasciitilde}(n-1) + \text{bv2nat\_rec}(n - 1, \text{bv}) \\
\text{ENDIF} \\
\text{MEASURE} \ n \\

\text{bv2nat}(\text{bv}: \text{bvec}): \text{below}(\text{exp2}(N)) = \text{bv2nat\_rec}(N, \text{bv})

where \text{exp2} is the power of 2 function defined in the \text{exp2} theory:

\text{exp2}(n: \text{nat}): \text{RECURSIVE} \text{ posnat} = \text{IF} \ n = 0 \ \text{THEN} \ 1 \ \text{ELSE} \ 2 \ast \text{exp2}(n - 1) \ \text{ENDIF} \\
\text{MEASURE} \ n \\

The \text{bv2nat} function returns a natural number that is less than \(2^N\). Note that this fact is contained in the type of the function\(^1\). The \text{bv2nat} function is defined in terms of a recursive function \text{bv2nat\_rec}. The function \text{bv2nat\_rec} is equivalent to

\[
\text{bv2nat\_rec}(n, \text{bv}) = \sum_{i=0}^{n-1} 2^i \text{bv}\text{\textasciitilde}i
\]

Note that this definition designates that the 0th bit is the least significant bit and the \(N-1\) bit is the most significant bit.

The \text{bv2nat} function is bijective (i.e. is a one-to-one correspondence):

\text{bv2nat\_bij} : \text{THEOREM} \text{ bijective?}(\text{bv2nat})

and thus an inverse function \text{nat2bv} exists:

\text{nat2bv}(\text{val}: \text{below}(\text{exp2}(N))): \text{bvec} = \text{inverse}(\text{bv2nat})(\text{val})

Thus, the following relationship exists between these functions:

\text{bv2nat\_inv} : \text{THEOREM} \text{ bv2nat}(\text{nat2bv}(\text{val})) = \text{val}

### 4 Bitwise Logical Operations on Bitvectors

The bitwise logical operations on bitvectors are defined in the \text{bv\_bitwise} theory as follows:

\(^1\)The PVS system provides a powerful type theory that is heavily exploited in this library. We have deliberately packed as much information as possible into the types of the functions. This provides two major benefits: (1) The information is automatically available in proofs, and (2) many theorems can be stated concisely, without explicit constraints.
OR(bv1,bv2: bvec[N]) : bvec = (LAMBDA i: bv1(i) OR bv2(i));
AND(bv1,bv2: bvec[N]): bvec = (LAMBDA i: bv1(i) AND bv2(i));
IFF(bv1,bv2: bvec[N]): bvec = (LAMBDA i: bv1(i) IFF bv2(i));
NOT(bv: bvec[N]) : bvec = (LAMBDA i: NOT bv(i));
XOR(bv1,bv2: bvec[N]): bvec = (LAMBDA i: XOR(bv1(i),bv2(i)));

If the user wishes to avoid the use of the underlying bitvector implementation, the following lemmas can be used rather than expanding these functions:

bv, bv1, bv2: VAR bvec[N]

bv_OR : LEMMA (bv OR bv2)~i = (bv1~i OR bv2~i)
bv_AND : LEMMA (bv1 AND bv2)~i = (bv1~i AND bv2~i)
bv_IFF : LEMMA (bv1 IFF bv2)~i = (bv1~i IFF bv2~i)
bv_XOR : LEMMA XOR(bv1,bv2)~i = XOR(bv1~i,bv2~i)
bv_NOT : LEMMA (NOT bv)~i = NOT bv~i

5 Bitvector Concatenation

The concatenation operator o on bitvectors is defined in the bv_concat theory as follows:

bv_concat [n:nat, m:nat ]: THEORY
BEGIN
    o(bvn: bvec[n], bvm: bvec[m]): bvec[n+m] =
        (LAMBDA (nm: below(n+m)): IF nm < m THEN bvm(nm)
            ELSE bv'n(mn - m)
            ENDF

The result of concatenating a bitvector of length n with a bitvector of length m is a new bitvector of length n+m. The zero-length bitvector is the identity. The following theorems, which establish that the triple (bvec, o, null bv) is a monoid, are proved in the theory bv_concat_lems.
null bv: bvec[0] \% zero-length bit-vector

concat_identity_r : LEMMA (FORALL (n: nat), (bvn:bvec[n])):
  bvn o null bv = bvn

concat_identity_l : LEMMA (FORALL (n: nat), (bvn:bvec[n])):
  null bv o bvn = bvn

concat_associative : LEMMA (FORALL (m,n,p: nat), (bvm:bvec[m]),
  (bvn:bvec[n]), (bvp:bvec[p]):
  (bvm o bvn) o bvp = bvm o (bvn o bvp))

The bv_concat_lemms theory also provides a lemma not_over_concat

not_over_concat : LEMMA (FORALL (n: nat), (a,b: bvec[n])):
  (NOT (a o b)) = (NOT a) o (NOT b))

that shows that NOT distributes over the o operator and a lemma bvconcat2nat that provides
the result of applying bv2nat to a concatenated bitvector:

bvn: VAR bvec[n]
bvm: VAR bvec[m]
nm: VAR below(n+m)

bvconcat2nat: THEOREM bv2nat[n+m](bvn o bvm)
  = bv2nat[n](bvn) * exp2(m) + bv2nat[m](bvm)

6 Extraction Operator

The operator \(^{(i,j)}\) extracts a contiguous fragment of a bitvector between two given positions.

\(^{(bv: bvec[N], sp:[ii: below(N), upto(ii))]}(bv: bvec[N], sp:[ii: below(N), upto(ii))]}\):
  bvec[proj_1(sp) - proj_2(sp) + 1] =
  (LAMBDA (ii: below(proj_1(sp) - proj_2(sp) + 1)):
    bv(ii + proj_2(sp)))

Although the definition looks formidable, the behavior is quite simple. The first argument is
a bitvector of length N. The second argument designates the subfield that is to be extracted.
For example, suppose bv = (t,u,v,w,x,y,z) with z as the least significant bit. Then,
bv\(^{(4,2)}\) is the bitvector of length 3 that contains the bits 4, 3 and 2. In other words,
bv\(^{(4,2)}\) = (v,w,x).
7 Shift Operations on Bitvectors

The left and shift operations on a bitvector are defined as follows:

\[
\text{right_shift}(i: \text{nat}, \text{bv}: \text{bvec}[N]): \text{bvec}[N] = \\
\quad \text{IF } i = 0 \text{ THEN } \text{bv} \\
\qquad \text{ELSIF } i < N \text{ THEN } \text{bv}[i] \circ \text{bv}[N-1, i] \\
\qquad \text{ELSE } \text{bv}[0[N] \text{ ENDF}
\]

\[
\text{left_shift}(i: \text{nat}, \text{bv}: \text{bvec}[N]): \text{bvec}[N] = \\
\quad \text{IF } i = 0 \text{ THEN } \text{bv} \\
\qquad \text{ELSIF } i < N \text{ THEN } \text{bv}[-(N-i-1, 0) \circ \text{bv}[i] \\
\qquad \text{ELSE } \text{bv}[0[N] \text{ ENDF}
\]

The right_shift operation shifts a bit vector by a given number of positions to the right, filling 0's in the shifted bits. The left_shift operation shifts a bit vector by a given number of positions to the left, filling 0's in the shifted bits.

8 Bitvector Rotation

The rotation operations on a bitvector are defined in the bv_rotate theory as follows:

\[
\text{rotate_right}(k: \text{upto}(N), \text{bv}: \text{bvec}[N]): \text{bvec}[N] = \\
\quad \text{IF } (k = 0) \text{ OR } (k = N) \text{ THEN } \text{bv} \\
\qquad \text{ELSE } \text{bv}[-(k-1, 0) \circ \text{bv}[-(N-1, k) \text{ ENDF}
\]

\[
\text{rotate_left}(k: \text{upto}(N), \text{bv}: \text{bvec}[N]): \text{bvec}[N] = \\
\quad \text{IF } (k=0) \text{ OR } (k = N) \text{ THEN } \text{bv} \\
\qquad \text{ELSE } \text{bv}[-(N-k-1, 0) \circ \text{bv}[-(N-1,N-k) \text{ ENDF}
\]

The following lemmas relate the fields of the rotated bitvector with the original bitvector:

\[
\text{rotate_right}_\text{lem} : \text{LEMMA } \text{rotate_right}(k, \text{bv})^i = \\
\quad \text{IF } i+k < N \text{ THEN } \text{bv}[-(i+k) \text{ ELSE } \text{bv}[-(i+k-N) \text{ ENDF}
\]

\[
\text{rotate_left}_\text{lem} : \text{LEMMA } \text{rotate_left}(k, \text{bv})^i = \\
\quad \text{IF } i-k >= 0 \text{ THEN } \text{bv}[-(i-k) \text{ ELSE } \text{bv}[-(N+i-k) \text{ ENDF}
\]

The 1-bit rotation functions are defined in terms of these as follows:

\[
\text{rot_r1}(\text{bv}: \text{bvec}[N]): \text{bvec}[N] = \text{rotate_right}(1, \text{bv})
\]

\[
\text{rot_l1}(\text{bv}: \text{bvec}[N]): \text{bvec}[N] = \text{rotate_left}(1, \text{bv})
\]

The \text{rotate_right}(1, \text{bv}) and \text{rotate_left}(1, \text{bv}) functions can also be expressed in terms of \text{rot_r1} and \text{rot_l1} as follows:
iterate_rot_r1 : LEMMA iterate(rot_r1,k)(bv) = rotate_right(k,bv)

iterate_rot_l1 : LEMMA iterate(rot_l1,k)(bv) = rotate_left(k,bv)

where iterate is defined in the PVS prelude as follows:

function_iterate[T: TYPE]: THEORY
BEGIN
  f: VAR [T -> T]
  m, n: VAR nat
  x: VAR T

  iterate(f, n)(x): RECURSIVE T =
  IF n = 0 THEN x ELSE iterate(f, n-1)(f(x)) ENDIF
  MEASURE n

END function_iterate

9 Zero and Sign-Extend Operators

The zero_extend operator expands a bit-vector of length N into a bitvector of length k filling the upper bits with zeros:

zero_extend(k: above(N)): [bvec[N] -> bvec[k]] =
  (LAMBDA bv: bvecO[k-N] o bv)

Thus, the natural number interpretation remains the same:

zero_extend_lem : THEOREM bv2nat[k](zero_extend(k)(bv)) = bv2nat(bv)

The sign_extend operator returns a function that extends a bit vector to length k by repeating the most significant bit of the given bit vector:

sign_extend(k: above(N)): [bvec[N] -> bvec[k]] =
  (LAMBDA bv: IF bv(N-1) = 1 THEN bvec1[k-N] o bv ELSE bvec0[k-N] o bv ENDIF)

The 2's complement interpretation remains the same:

sign_extend_lem : THEOREM bv2int[k](sign_extend(k)(bv)) = bv2int(bv)

These higher-order functions are defined in the theory bv_extend.

The following useful theorem has been proved about the sign_extend function:
sign_to_zero : THEOREM sign_extend(k)(bv) =  
   IF bv(N-1) = 1 THEN NOT(zero_extend(k)(NOT(bv))) 
   ELSE zero_extend(k)(bv) 
   ENDIF

A function zero_extend_lsend is also defined to return a function that extends a bit vector to length k by padding 0's at the least significant end of bv. That is, the bv2nat interpretation of the argument increases by $2^{(k-N)}$:

zero_extend_lsend(k: above(N)): [bv[N] -> bv[k]] =  
   (LAMBDA bv: bv o bvvec0[k-N])

zero_extend_lsend: THEOREM bv2nat(zero_extend_lsend(k)(bv)) 
   = bv2nat(bv) * exp2(k-N)

A higher-order function, lsb_extend, returns a function that extends a bit vector to length k by repeating the least significant bit of the bit vector at its least significant end.

lsb_extend(k: above(N)): [bv[N] -> bv[k]] =  
   (LAMBDA bv: IF bv'0 = 0 THEN bv o bvvec0[k-N] 
   ELSE bv o bvvec1[k-N] ENDIF)

The lemmas about the extend functions are proved in the theory bv_extend_lems.

10 Theorems Involving Concatenation and Extraction

The following properties of ^ and o are proved in the theory bv_manipulations:

bvn: VAR bvvec[n]  
bvm: VAR bvvec[m]

caret_concat_bot : THEOREM i < m IMPLIES (bvn o bvm)^-(i,j) = bvm^-(i,j))

caret_concat_top : THEOREM i >= m AND j >= m IMPLIES 
   (bvn o bvm)^-(i,j) = bvn^-(i-m, j-m))

caret_concat_all : THEOREM i >= m AND j < m IMPLIES 
   (bvn o bvm)^-(i,j) = bvn^-(i-m,0) o bvm^-(m-i,j))

bv_decomposition : THEOREM bvn^-(n-1,k+1) o bvn^-(k,0) = bvn

concat_bottom : THEOREM (bvn o bvm)^-((m-1), 0) = bvm

concat_top : THEOREM (bvn o bvm)^-((n+m-1), m) = bvn
The first two theorems simplify formulas involving concatenation and extraction when the part to be extracted is completely within one of the parts being joined together. The formula caret_concat_all moves an extraction within the concatenation. The last two theorems are similar to the first two, except that the extraction involves the complete parts.

11 2’s Complement Interpretations of a Bitvector

The 2's complement interpretation of a bitvector of length \( N \) enables the representation of integers from \(-2^{N-1}\) to \(2^{N-1} - 1\). The basic definitions for 2's complement arithmetic are defined in the bv_int theory.

Two constants are defined to represent the minimum and maximum values:

\[
\begin{align*}
\text{minint} & : \text{int} = -\exp2(N-1) \\
\text{maxint} & : \text{int} = \exp2(N-1) - 1
\end{align*}
\]

The range of values is defined as follows:

\[
\begin{align*}
in\_\text{rng}_{\text{2s comp}}(i: \text{int}) & : \text{bool} = (\text{minint} \leq i \text{ AND } i \leq \text{maxint}) \\
rng\_\text{2s comp} & : \text{TYPE} = i: \text{int} | \text{minint} \leq i \text{ AND } i \leq \text{maxint}
\end{align*}
\]

The 2's complement interpretation function, \( \text{bv2int} \), is defined as follows:

\[
\begin{align*}
\text{bv2int}(bv: \text{bvec}) : \text{rng}_{\text{2s comp}} & = \text{IF } \text{bv2nat}(bv) < \exp2(N-1) \text{ THEN } \text{bv2nat}(bv) \\
& \text{ELSE } \text{bv2nat}(bv) - \exp2(N) \text{ ENDIF}
\end{align*}
\]

The \( \text{bv2int} \) function can also be expressed as follows:

\[
\text{bv2int lem} : \text{THEOREM } \text{bv2int}(bv) = \text{bv2nat}(bv) - \exp2(N) * \text{bv}(N - 1)
\]

The \( \text{bv2int} \) function is bijective (i.e. is a one-to-one correspondence):

\[
\text{bv2int bij} : \text{THEOREM } \text{bijective?}(\text{bv2int})
\]

and thus an inverse function \( \text{int2bv} \) exists:

\[
\text{int2bv}(val: \text{below}(\exp2(N))): \text{bvec} = \text{inverse(\text{bv2int})(val)}
\]

The following relationship exists between these functions:

\[
\text{bv2int inv} : \text{THEOREM } \text{bv2int}(\text{int2bv}(iv))=iv;
\]

The \( \text{int2bv} \) functions can also be translated into \( \text{nat2bv} \) as follows:

\[
\begin{align*}
\text{ii: VAR rng}_{\text{2s comp}} & \\
\text{int2bv}_{\text{2nat}} : \text{LEMMA } \text{int2bv}(ii) & = \text{IF } ii \geq 0 \text{ THEN } \text{nat2bv}[N](ii) \\
& \text{ELSE } \text{nat2bv}[N](ii+\exp2(N)) \text{ ENDIF}
\end{align*}
\]

11
12 Bitvector Arithmetic

An important advantage of 2’s complement arithmetic is that the + operation for the natural number interpretation and the 2’s complement interpretation is the same. Thus, the same hardware can be used for both cases. This property and others is developed in the following subsections.

12.1 Definition of Arithmetic Operators

Operations are defined to increment and decrement a bitvector by an integer in the theory bv_arith_nat. This operations are overloaded on the + and - symbols:

\[ +\text{by}(bv: \text{bvec}, i: \text{int}): \text{bvec} = \text{nat2bv(mod(bv2nat(bv) + i, exp2(N)))} \]

\[ -(bv: \text{bvec}, i: \text{int}): \text{bvec} = bv + (-i) \]

The addition of two bit vectors is defined as follows:

\[ +(bv1: \text{bvec}, bv2: \text{bvec}): \text{bvec} = \]
\[ \text{IF } bv2nat(bv1) + bv2nat(bv2) < \text{exp2(N)} \]
\[ \text{THEN } \text{nat2bv(bv2nat(bv1) + bv2nat(bv2))} \]
\[ \text{ELSE } \text{nat2bv(bv2nat(bv1) + bv2nat(bv2) - exp2(N))} \]
\[ \text{ENDIF} \]

This definition leads immediately to the following theorems:

\[ \text{bv_add} \text{LEMMA } bv2nat(bv1 + bv2) = \]
\[ \text{IF } bv2nat(bv1) + bv2nat(bv2) < \text{exp2(N)} \]
\[ \text{THEN } bv2nat(bv1) + bv2nat(bv2) \]
\[ \text{ELSE } bv2nat(bv1) + bv2nat(bv2) - \text{exp2(N)} \text{ ENDIF} \]

\[ \text{bv_addcomm} \text{THEOREM } bv1 + bv2 = bv2 + bv1 \]

The first lemma provides the natural number interpretation for the + operation. The next theorem shows that it is commutative. Other useful lemmas about bitvector addition are also provided:

\[ k, k1, k2: \text{VAR int} \]

\[ \text{bv_add_two_consts: THEOREM } (bv1 + k1) + (bv2 + k2) = (bv1 + bv2) + (k1 + k2) \]
\[ \text{bv_add_const_assoc: THEOREM } bv1 + (bv2 + k) = (bv1 + bv2) + k \]
\[ \text{bv_add_2_consts: LEMMA } (bv + k1) + k2 = bv + (k1 + k2) \]

\[ \text{bv_both_sides: THEOREM } (bv1 + bv3 = bv2 + bv3) \text{ IFF } bv1 = bv2 \]

\[ \text{bv_add_assoc: THEOREM } bv1 + (bv2 + bv3) = (bv1 + bv2) + bv3 \]
The * is overloaded to represent the unsigned multiplication of two n-bit bvecs:

\[ *(bv1: \text{bvec}[N], bv2: \text{bvec}[N]): \text{bvec}[2*N] = \text{nat2bv}[2*N](\text{bv2nat}(bv1) * \text{bv2nat}(bv2)) \]

This definition leads immediately to the following theorem, which provides the natural number interpretation for the * operation:

\[ \text{bv_mult}: \text{LEMMA bv2nat}(bv1 * bv2) = \text{bv2nat}(bv1) * \text{bv2nat}(bv2) \]

The carryout function is defined as follows:

\[ \text{carryout}(bv1: \text{bvec}, bv2: \text{bvec}, Cin: \text{bvec}[1]): \text{bvec}[1] = \]

\[ (\text{LAMBDA}(bb: \text{below}(1)):
   \text{bool2bit}(\text{bv2nat}(bv1) + \text{bv2nat}(bv2) + \text{bv2nat}(Cin) \geq \exp2(N)) \) ; \]

The carryout function indicates when the + operation will exceed the capacity of the bitvector. Note that the carryout returns a bvec[1].

The inequalities over bitvectors are defined as follows:

\[ < (bv1: \text{bvec}, bv2: \text{bvec}): \text{bool} = \text{bv2nat}(bv1) < \text{bv2nat}(bv2) ; \]
\[ <=(bv1: \text{bvec}, bv2: \text{bvec}): \text{bool} = \text{bv2nat}(bv1) \leq \text{bv2nat}(bv2) ; \]

\[ > (bv1: \text{bvec}, bv2: \text{bvec}): \text{bool} = \text{bv2nat}(bv1) > \text{bv2nat}(bv2) ; \]
\[ \geq(bv1: \text{bvec}, bv2: \text{bvec}): \text{bool} = \text{bv2nat}(bv1) \geq \text{bv2nat}(bv2) ; \]

The following lemmas about the bitvector order relations are provided:

\[ \text{bv\_smallest}: \text{LEMMA (FORALL bv: bv} \geq \text{bvec0)} \]
\[ \text{bv\_greatest}: \text{LEMMA (FORALL bv: bv} \leq \text{bvec1)} \]

12.2 Arithmetic Properties of Shifting

The following theorems (available in bv_arith_extract) give the numerical properties of left and right shifting:

\[ ss: \text{VAR below}(N) \]
\[ bv: \text{VAR} \text{bvec}[N] \]

\[ \text{bv\_shift}: \text{THEOREM bv2nat(bv}^{\sim}(N-1, ss)) = \text{div(bv2nat(bv), } \exp2(ss)) \]
\[ \text{bv\_bottom}: \text{THEOREM bv2nat(bv}^{\sim}(ss, 0)) = \text{mod(bv2nat(bv), } \exp2(ss+1)) \]
\[ \text{right\_shift\_lem}: \text{THEOREM bv2nat(right\_shift(ss, bv))} = \text{div(bv2nat(bv),} \exp2(ss)) \]
\[ \text{left\_shift\_lem}: \text{THEOREM bv2nat(left\_shift(ss, bv))} = \]
\[ \quad \text{bv2nat(bv}^{\sim}(N-ss-1, 0)) \times \exp2(ss) \]
The bv_shift theorem establishes that the extraction of the upper bits is equivalent to dividing by a power of 2 under the natural number interpretation\(^2\). This theorem is closely related to the right_shift_lem. The bv_bottom theorem establishes that the extraction of the lower bits is equivalent to a power of 2 mod operation under the natural number interpretation.

The arithmetic right shift operator is defined in bv_arith_shift as follows:

\[
\text{arith_shift_right}(k: \text{upto}(N), \text{by}: \text{bvec}[N]): \text{bvec}[N] = \text{right_shift_with}(k, \text{fill}[k](\text{bv}^\wedge(N-1)), \text{bv})
\]

Note that it fills the upper \(k\) bits with the \((N-1)st\) bit of the original bitvector. The following theorem shows the 2's complement result of an arithmetic right shift:

\[
k: \text{VAR upto}(N)
\]
\[
\text{arith_shift_right_int}: \text{LEMMA } \text{bv2int(arith_shift_right}(k,\text{bv})) = \text{floor(bv2int(bv)/exp2(k))}
\]

### 12.3 Theorems about 2's Complement Arithmetic

The 2's complement negation of a bit vector is defined in bv_arithmetic as follows:

\[
-(\text{bv}: \text{bvec}): \text{bvec} = \text{int2bv( IF bv2int(bv) = minint THEN bv2int(bv) ELSE -(bv2int(bv)) ENDIF )}
\]

The following property relates this operator to bv2int:

\[
\text{unaryminus}: \text{LEMMA } \text{bv2int(-bv)} = \text{IF bv2int(bv) = minint THEN bv2int(bv) ELSE -(bv2int(bv)) ENDIF}
\]

The subtraction of two bit vectors is defined (in bv_arithmetic) using bitvector addition as follows:

\[
-(\text{bv1}, \text{bv2}): \text{bvec} = (\text{bv1 + (-bv2)})
\]

If the result is in the range of 2s complement integers, addition of two bit vectors is the same as for a natural number interpretation:

\[
\text{intaddlem}: \text{THEOREM } \text{in_rng_2s_comp(bv2int(bv1) + bv2int(bv2))}
\]
\[
\text{IMPLIES bv2int(bv1 + bv2) = bv2int(bv1) + bv2int(bv2)}
\]

This is the relationship that enables one to use the same hardware for natural number addition as 2's complement addition.

The 2s complement of a bitvector is its 1's complement + 1:

\[
\text{twos_compl}: \text{THEOREM } -\text{bv2int(bv)} = \text{bv2int(NOT bv)} + 1;
\]

The 1's complement of a bitvector \(bv\) is the bitwise NOT, i.e. \(\text{NOT } bv\).

\(^2\)The div function over natural numbers is defined by \(\text{div(n,m)}: \text{nat} = \text{floor(n/m)}\)
13 Overflow

Arithmetic overflow occurs when the result of an operation cannot be represented within the bitvector. The conditions for 2's complement overflow are define in the bv_overflow theory:

\[
\text{overflow}(bv1, bv2, b) : \text{bool} = (bv2int(bv1) + bv2int(bv2) + b) > \text{maxint}[N] \\
\text{OR} (bv2int(bv1) + bv2int(bv2) + b) < \text{minint}[N]
\]

The following theorem provides the relationships between the top bits of the operands and the result when overflow occurs.

\[
\text{overflow_def} : \text{THEOREM} \\ \ \text{overflow}(bv1, bv2, b) = \\
((bv1 \wedge (N - 1) = bv2 \wedge (N - 1)) \\
\text{AND} (bv1 \wedge (N - 1) /= (bv1 + bv2 + b) \wedge (N - 1)))
\]

The following theorems define the result of bitvector arithmetic when overflow occurs:

\[
\text{not_in_rng} : \text{THEOREM} \ \ \text{NOT in_rng}_2s\text{_comp}(bv2int(bv1) + bv2int(bv2)) \\
\text{IMPLIES} \ bv2int(bv1 + bv2) = \\
bv2nat(bv1) + bv2nat(bv2) - \exp2(N)
\]

\[
\text{not_in_rng_int} : \text{THEOREM} \ \ \text{NOT in_rng}_2s\text{_comp}(bv2int(bv1) + bv2int(bv2)) \\
\text{IMPLIES} \ bv2int(bv1 + bv2) = \\
bv2int(bv1) + bv2int(bv2) + \exp2(N) * bv1(N - 1) \\
+ \exp2(N) * bv2(N - 1) \\
- \exp2(N)
\]

14 Library Organization

The top of the bitvectors library is located in the theory bv_top. It imports the following theories:
bv provides basic definition of bitvector type bvec
bv_nat interpretes bvec as a natural number
bv_int interpretes bvec as an integer
bv_arithmetic defines basic operators (i.e. + - >) over bitvectors
bv_arith_nat defines bitvector plus, etc
bv_arith_extract defines arithmetic over extractors
bv_extractors defines extractor operator that
bv_extractors_lems provides lemmas about operator
bv_concat defines concatenation operator o creates smaller bitvectors from larger
bv_concat_lems establishes that concat is a monoid
bv_constants defines some useful bitvector constants
bv_manipulations provides lemmas concerning and o
bv_bitwise defines bit-wise logical operations on bitvectors
bv_bitwise_lems provides lemmas about bit-wise logical operations
bv_shift defines shift operations
bv_extend provides zero and sign extend operations
bv_extend_lems provide lemmas about extend operations
bv_fract defines fractional interpretation of a bitvector
bv_overflow relates overflow to top bits

A graphical display of the import chain is shown in figure 1.

References


Figure 1: Importing Structure of Bitvectors Library


**Title:** A Bitvectors Library For PVS

**Abstract:**

This paper describes a bitvectors library that has been developed for PVS. The library defines a bitvector as a function from a subrange of the integers into \( \{0, 1\} \). The library provides functions that interpret a bitvector as a natural number, as a 2's complement number, as a vector of logical values and as a 2's complement fraction. The library provides a concatenation operator and an extractor. Shift, extend and rotate operations are also defined. Fundamental properties of each of these operations have been proved in PVS.

**Subject Terms:**

Hardware Verification, Bitvectors, Formal Methods, and Formal Verification.