Internal Note #57
Computational Logic, Inc.
August 19, 1988

A Library For Hardware Verification

Bill Bevier

1. Introduction

This note describes the library of Boyer-Moore events which I have been working on for the last couple of months. The library is oriented toward hardware verification, but the early events contain facts about arithmetic and lists which are useful for proofs in many domains.

The library consists of a stack of "theories". At the bottom are facts about sets, bags, numbers and lists. Then bit vectors and integers are introduced. Finally, facts about the integer and natural number interpretation of bit vectors occur. In this note we describe each theory by giving the definitions which appear in a theory. The last section describes how to get your hands on the library. You may browse the event files which contribute to this library to get an idea of the lemmas which are proved.

Reference is made below to various `DEFTHEORY`s. I've made use of the `DEFTHEORY` mechanism to group events so that they can be enabled and disabled in clusters.

The library is still experimental. It may be discovered that some new definitions are desirable, or some important lemmas are not included. The library is known to be incomplete in the latter theories. I will play the role of maintainer of this library, and issue releases if enough users come aboard. I want feedback from users to determine how useful the libraries are for various applications.

2. Sets and Bags

This theory has been developed only in a rudimentary way. It includes definitions necessary to prove some arithmetic `META` lemmas, and some facts I used in the proof of Kit. Besides the definitions presented here, the theory includes facts about the function `MEMBER`, which is defined during `BOOT-STRAP`. The lemmas which have been proved about this theory are named in the deftheory `SETS-AND-BAGS`.

`DELETE` deletes the first occurence of an element in a list. `BAGDIFF` computes the bag difference of two lists. `BAGINT` returns the bag intersection of two lists. `OCCURRENCES` counts the number of occurrences of an item in a list. `PERMUTATION` is an equivalence relation on lists, recognizing two lists which contain the same elements in the same number. `SETP` recognizes a list in which no element occurs more than once. `SUBBAGP` determines if one list is a subbag of another.

```
(DEFN DELETE (X L)
  (IF (LISTP L)
      (IF (EQUAL X (CAR L))
          (CDR L)
          (CONS (CAR L) (DELETE X (CDR L))))
      L))

(DEFN BAGDIFF (X Y)
  (IF (LISTP Y)
      (IF (MEMBER (CAR Y) X)
          (BAGDIFF (DELETE (CAR Y) X) (CDR Y))
          (BAGDIFF X (CDR Y)))
      X))

(DEFN BAGINT (X Y)
  (IF (LISTP X)
      (IF (MEMBER (CAR X) Y)
          (CONS (CAR X) (BAGINT (CDR X) (DELETE (CAR X) Y)))
          (BAGINT (CDR X) Y))
      NIL))

(DEFN OCCURRENCES (X L)
  (IF (LISTP L)
      (IF (EQUAL X (CAR L))
          (ADD1 (OCCURRENCES X (CDR L)))
          (OCCURRENCES X (CDR L)))
      0))

(DEFN PERMUTATION (A B)
  (IF (LISTP A)
      (AND (MEMBER (CAR A) B)
           (PERMUTATION (CDR A) (DELETE (CAR A) B)))
      (NOT (LISTP B))))

(DEFN SETP (L)
  (IF (LISTP L)
      (AND (NOT (MEMBER (CAR L) (CDR L)))
           (SETP (CDR L)))
      T))

(DEFN SUBBAGP (X Y)
  (IF (LISTP X)
      (IF (MEMBER (CAR X) Y)
          (SUBBAGP (CDR X) (DELETE (CAR X) Y))
          F)
      T))
```

3. Arithmetic

This theory is the most developed and best tested. I've been quite pleased with its performance in proofs higher up in the hardware library. The theory is about the arithmetic functions introduced by `BOOT-STRAP`: `PLUS`, `DIFFERENCE`, `TIMES`, `REMAINDER` and `QUOTIENT`. Three other defined functions, `EXP`, `LOG` and `GCD` are included.

```
(DEFN EXP (I J)
  (IF (ZEROP J)
      1
      (TIMES I (EXP I (SUB1 J)))))
```

```
(DEFN LOG (BASE N)
  (IF (LESSP BASE 2)
      0
      (IF (ZEROP N)
          0
          (ADD1 (LOG BASE (QUOTIENT N BASE))))))

(DEFN GCD (X Y)
  (IF (ZEROP X)
      (FIX Y)
      (IF (ZEROP Y)
          X
          (IF (LESSP X Y)
              (GCD  X (DIFFERENCE Y X))
              (GCD (DIFFERENCE X Y) Y)))))
```

The first part of the devolopment of arithmetic is devoted to rules which canonicalize terms involving `PLUS` and `DIFFERENCE`. Rewrite rules canonicalize terms involving these two functions into the difference of two sums. For instance, `DIFF-DIFF-ARG1` canonicalizes a `DIFFERENCE` term whose first argument is a `DIFFERENCE`.

```
(LEMMA DIFF-DIFF-ARG1 (REWRITE)
    (EQUAL (DIFFERENCE (DIFFERENCE X Y) Z)
           (DIFFERENCE X (PLUS Y Z))))
```

The `META` lemma `CORRECTNESS-OF-CANCEL-DIFFERENCE` permits execution of a procedure which cancels like terms in the difference of two sums. This method has been effective in dealing with complicated compositions of `PLUS` and `DIFFERENCE` terms. In all, three related `META` procedures are installed in arithmetic: `CANCEL-DIFFERENCE`, `CANCEL-PLUS` (Boyer and Moore's original example), and `CANCEL-LESSP-PLUS` which cancels addends which appear on both sides of a `LESSP`.

The lemmas about `TIMES`, `REMAINDER` and `QUOTIENT` have been distilled from *A Computational Logic*, the proof of FM8501, and the proof of Kit. I have taken some pains to omit crocks. i.e., to generalize rules whenever possible.

The facts described about `EXP` and `LOG` are simple. I stole `LOG` from Bishop Brock's proof of a floating point adder. I have not used these facts yet, but have included them because I foresee using them.

4. Integers

Facts about the following functions occur in the deftheory `INTEGERS`. The representation of integers uses the shell `MINUS` which is defined during `BOOT-STRAP`. An integer is either a number, or is a `MINUS` object whose argument isn't `0`. Therefore there is only one representation for zero in the integers. The function `INTEGERP` recognizes an integer. The function `FIX-INT` coerces a non-integer to `0`. The functions `IZEROP`, `ILESSP`, `ILEQ`, `IPLUS`, `IDIFFERENCE` and `ITIMES` are the integer versions of arithmetic operations. The function `INEG` negates an integer, and `IABS` returns the absolute value of an integer.

```
(DEFN INTEGERP (X)
  (OR (NUMBERP X)
      (AND (NEGATIVEP X)
           (NOT (ZEROP (NEGATIVE-GUTS X))))))

(DEFN FIX-INT (X)
  (IF (INTEGERP X) X 0))

(DEFN IZEROP (I)
  (IF (INTEGERP I)
      (EQUAL I 0)
      T))
```

```
(DEFN ILESSP (I J)
  (IF (NEGATIVEP I)
      (IF (NEGATIVEP J)
          (LESSP (NEGATIVE-GUTS J) (NEGATIVE-GUTS I))
          T)
      (IF (NEGATIVEP J)
          F
          (LESSP I J))))

(DEFN ILEQ (I J)
  (OR (ILESSP I J)
      (EQUAL I J)))

(DEFN IPLUS (X Y)
  (IF (NEGATIVEP X)
      (IF (NEGATIVEP Y)
          (IF (AND (ZEROP (NEGATIVE-GUTS X))
                   (ZEROP (NEGATIVE-GUTS Y)))
              0
              (MINUS (PLUS (NEGATIVE-GUTS X)
                           (NEGATIVE-GUTS Y))))
          (IF (LESSP Y (NEGATIVE-GUTS X))
              (MINUS (DIFFERENCE (NEGATIVE-GUTS X) Y))
              (DIFFERENCE Y (NEGATIVE-GUTS X))))
      (IF (NEGATIVEP Y)
          (IF (LESSP X (NEGATIVE-GUTS Y))
              (MINUS (DIFFERENCE (NEGATIVE-GUTS Y) X))
              (DIFFERENCE X (NEGATIVE-GUTS Y)))
          (PLUS X Y))))

(DEFN INEG (X)
  (IF (NEGATIVEP X)
      (NEGATIVE-GUTS X)
      (IF (ZEROP X)
          0
          (MINUS X))))

(DEFN IDIFFERENCE (X Y)
  (IPLUS X (INEG Y)))

(DEFN IABS (I)
  (IF (NEGATIVEP I)
      (NEGATIVE-GUTS I)
      (FIX I)))

(DEFN ITIMES (I J)
  (IF (NEGATIVEP I)
      (IF (NEGATIVEP J)
          (TIMES (NEGATIVE-GUTS I) (NEGATIVE-GUTS J))
          (FIX-INT (MINUS (TIMES (NEGATIVE-GUTS I) J))))
      (IF (NEGATIVEP J)
          (FIX-INT (MINUS (TIMES I (NEGATIVE-GUTS J))))
          (TIMES I J))))
```

There are several different ways to define integer division. The standard approach derives from from the following theorem.

Division Theorem.

$\forall\ i, j, j \neq 0,\ \ \exists!\ q, r \text{ such that } \ i = jq + r,\ \ 0 \leq r < |j|.$

The functions `IQUOTIENT` and `IREMAINDER` are intended to compute q and r. This fact is proved in the integer library.

```
(DEFN IQUOTIENT (I J)
  (IF (IZEROP J)
      0
      (IF (NEGATIVEP I)
          (IF (NEGATIVEP J)
              (IF (EQUAL (REMAINDER (NEGATIVE-GUTS I)
                                    (NEGATIVE-GUTS J))
                         0)
                  (QUOTIENT (NEGATIVE-GUTS I) (NEGATIVE-GUTS J))
                  (ADD1 (QUOTIENT (NEGATIVE-GUTS I)
                                  (NEGATIVE-GUTS J))))
              (IF (EQUAL (REMAINDER (NEGATIVE-GUTS I) J) 0)
                  (FIX-INT (MINUS (QUOTIENT (NEGATIVE-GUTS I) J)))
                  (FIX-INT (MINUS (ADD1 (QUOTIENT (NEGATIVE-GUTS I)
                                                  J))))))
          (IF (NEGATIVEP J)
              (FIX-INT (MINUS (QUOTIENT I (NEGATIVE-GUTS J))))
              (QUOTIENT I J)))))

(DEFN IREMAINDER (I J)
  (IDIFFERENCE (FIX-INT I) (ITIMES J (IQUOTIENT I J))))
```

It turns out that in computer arithmetic, notions of division other than that given by the division theorem are used. Two in particular, called *truncate towards negative infinity* and *truncate towards zero* are common. We include their definitions in the integer library, and prove the necessary theorems to establish their correctness.

Division Theorem (truncate towards negative infinity variant).

$\forall\ i,j,\ j \neq 0,\ \ \exists!\ q,r$ *such that* $\ i = jq + r,$ *and*

$$\begin{array}{ll} 0 \le r < j, & \textit{for } j > 0, \\ j < r \le 0, & \textit{for } j < 0. \end{array}$$

In this version, the integer quotient of two integers is the integer floor of the real quotient of the integers. The remainder has the sign of the divisor. The functions `IDIV` and `IMOD` compute *q* and `R`.

```
(DEFN IDIV (I J)
  (IF (IZEROP J)
      0
      (IF (NEGATIVEP I)
          (IF (NEGATIVEP J)
              (QUOTIENT (NEGATIVE-GUTS I) (NEGATIVE-GUTS J))
              (IF (EQUAL (REMAINDER (NEGATIVE-GUTS I) J) 0)
                  (FIX-INT (MINUS (QUOTIENT (NEGATIVE-GUTS I) J)))
                  (FIX-INT (MINUS (ADD1 (QUOTIENT (NEGATIVE-GUTS I)
                                                  J))))))
          (IF (NEGATIVEP J)
              (IF (EQUAL (REMAINDER I (NEGATIVE-GUTS J)) 0)
                  (FIX-INT (MINUS (QUOTIENT I (NEGATIVE-GUTS J))))
                  (FIX-INT
                      (MINUS (ADD1 (QUOTIENT I (NEGATIVE-GUTS J))))))
              (QUOTIENT I J)))))

(DEFN IMOD (I J)
  (IDIFFERENCE (FIX-INT I) (ITIMES J (IDIV I J))))
```

Division Theorem (truncate towards zero variant).

$\forall\ i,j,\ j \neq 0,\ \ \exists!\ q,r$ *such that* $\ i = jq + r,$ *and*

$$\begin{array}{ll} 0 \le r < |j|, & \textit{for } i \ge 0, \\ -|j| < r \le 0, & \textit{for } i < 0. \end{array}$$

In this version, the integer quotient of two integers is the integer floor of the real quotient of the integers, if the real quotient is positive. If the real quotient is negative, the integer quotient is the integer ceiling of the real quotient. The remainder has the sign of the dividend. The functions `IQUO` and `IREM` compute *q* and *r*.

```
(DEFN IQUO (I J)
  (IF (IZEROP J)
      0
      (IF (NEGATIVEP I)
          (IF (NEGATIVEP J)
              (QUOTIENT (NEGATIVE-GUTS I) (NEGATIVE-GUTS J))
              (FIX-INT (MINUS (QUOTIENT (NEGATIVE-GUTS I) J))))
          (IF (NEGATIVEP J)
              (FIX-INT (MINUS (QUOTIENT I (NEGATIVE-GUTS J))))
              (QUOTIENT I J)))))

(DEFN IREM (I J)
  (IDIFFERENCE (FIX-INT I) (ITIMES J (IQUO I J))))
```

5. Lists

The theory of lists is about the following functions. `APPEND` appends two lists. `FIRSTN` returns the first *n* elements of a list, up to the length of the list. `INIT` initializes a list to a given length. `LASTCDR` `CDR`s down a list until the first non-`CONS` is found. `LENGTH` return the length of a list. `PLIST` coerces the last `CDR` of a list to be `NIL`. `PLISTP` recognizes a list whose last `CDR` is `NIL`. `RESTN` is like Lisp's `NTHCDR`. It `CDR`s down a list *n* times. `REVERSE` reverses a list. Lemmas about these functions are grouped in the deftheory `LISTS`.

```
(DEFN APPEND (A B)
  (IF (LISTP A)
      (CONS (CAR A) (APPEND (CDR A) B))
      B))

(DEFN FIRSTN (N L)
  (IF (LISTP L)
      (IF (ZEROP N)
          NIL
          (CONS (CAR L) (FIRSTN (SUB1 N) (CDR L))))
      NIL))

(DEFN INIT (VAL LENGTH)
  (IF (ZEROP LENGTH)
      NIL
      (CONS VAL (INIT VAL (SUB1 LENGTH)))))

(DEFN LASTCDR (L)
  (IF (LISTP L)
      (LASTCDR (CDR L))
      L))

(DEFN LENGTH (L)
  (IF (LISTP L)
      (ADD1 (LENGTH (CDR L)))
      0))

(DEFN PLIST (L)
  (IF (LISTP L)
      (CONS (CAR L) (PLIST (CDR L)))
      NIL))

(DEFN PLISTP (L)
  (IF (LISTP L)
      (PLISTP (CDR L))
      (EQUAL L NIL)))
```

```
(DEFN RESTN (N L)
  (IF (LISTP L)
      (IF (ZEROP N)
          L
          (RESTN (SUB1 N) (CDR L)))
      L))

(DEFN REVERSE (L)
  (IF (LISTP L)
      (APPEND (REVERSE (CDR L)) (LIST (CAR L)))
      NIL))
```

6. Vectors

The theory of vectors provides direct access to elements of lists, and to contiguous sublists of lists. All access uses zero-based indices. GET returns the *n*th element of a list. GETS returns the sublist of a list beginning at location *n* for length *k*. PUT updates a list by placing an item at the *n*th position in a list. PUTS lays down an entire sublist at a given location in a list. The deftheory VECTORS includes facts about these functions, plus the theory LISTS.

These functions can be defined recursively, instead of appealing to the primitive list functions. The resulting definitions are cleaner looking. I chose these non-recursive definitions since I found that the proofs of properties of these functions are simpler when defined in this manner.

```
(DEFN GET (N L)
  (CAR (RESTN N L)))

(DEFN GETS (N K L)
  (FIRSTN K (RESTN N L)))

(DEFN PUT (N V L)
  (IF (LESSP N (LENGTH L))
      (APPEND (FIRSTN N L)
              (CONS V (RESTN (ADD1 N) L)))
      L))

(DEFN PUTS (N S L)
  (IF (LESSP N (LENGTH L))
      (IF (LEQ (PLUS N (LENGTH S)) (LENGTH L))
          (APPEND (FIRSTN N L)
                  (APPEND S (RESTN (PLUS N (LENGTH S)) L)))
          (APPEND (FIRSTN N L)
                  (APPEND (FIRSTN (DIFFERENCE (LENGTH L) N) S)
                          (LASTCDR L))))
      L))
```

7. Bit Vectors

The theory of bit vectors is much like the theory of vectors, since a bit vector is roughly a proper list whose elements are restricted to TRUE and FALSE. The shell BV defines bit vectors.

```
(ADD-SHELL BV BV-NIL BVP
           ((BV-BIT (ONE-OF FALSEP TRUEP) FALSE)
            (BV-VEC (ONE-OF BVP)          BV-NIL)))
```

BV-NILP recognizes an object which is either (BV-NIL) or not a bit vector. BV-FIX coerces an object which is not a bit vector to (BV-NIL). The following functions are analogous to those defined for the theory of lists. BV-APPEND appends two bit vectors. BV-FIRSTN returns a bit vector containing the firstn *n* bits of a bit vector. BV-INIT initializes a bit vector to a given length. BV-LENGTH returns the length of a bit vector. BV-RESTN does *n* BV-VECs on a bit vector.

```
(DEFN BV-NILP (A)
  (IF (BVP A) (EQUAL A (BV-NIL)) T))

(DEFN BV-FIX (X)
  (IF (BVP X) X (BV-NIL)))

(DEFN BV-APPEND (A B)
  (IF (BV-NILP A)
      (BV-FIX B)
      (BV (BV-BIT A) (BV-APPEND (BV-VEC A) B))))

(DEFN BV-FIRSTN (N BV)
  (IF (BV-NILP BV)
      (BV-NIL)
      (IF (ZEROP N)
          (BV-NIL)
          (BV (BV-BIT BV) (BV-FIRSTN (SUB1 N) (BV-VEC BV))))))

(DEFN BV-INIT (BIT N)
  (IF (ZEROP N)
      (BV-NIL)
      (BV BIT (BV-INIT BIT (SUB1 N)))))

(DEFN BV-LENGTH (BV)
  (IF (BV-NILP BV)
      0
      (ADD1 (BV-LENGTH (BV-VEC BV)))))

(DEFN BV-RESTN (N BV)
  (IF (BV-NILP BV)
      (BV-FIX BV)
      (IF (ZEROP N)
          (BV-FIX BV)
          (BV-RESTN (SUB1 N) (BV-VEC BV)))))
```

The following functions provide direct access to parts of bit vectors. All access uses zero-based indices. `BV-GET` returns the *n*th bit if a bit vector. `BV-GETS` returns the sub-bit-vector of a bit vector beginning at location *n* for length *k*. `BV-PUT` updates a bit vector by placing a bit at the *n*th position in a bit vector. `BV-PUTS` lays down an entire bit vector at a given location in another bit vector. The deftheory `BIT-VECTORS` includes facts about all the functions described in this section.

```
(DEFN BV-GET (N BV)
  (BV-BIT (BV-RESTN N BV)))

(DEFN BV-GETS (N K BV)
  (BV-FIRSTN K (BV-RESTN N BV)))

(DEFN BV-PUT (N V BV)
  (IF (LESSP N (BV-LENGTH BV))
      (BV-APPEND (BV-FIRSTN N BV)
                 (BV V (BV-RESTN (ADD1 N) BV)))
      (BV-FIX BV)))

(DEFN BV-PUTS (N S BV)
  (IF (LESSP N (BV-LENGTH BV))
      (IF (LEQ (PLUS N (BV-LENGTH S)) (BV-LENGTH BV))
          (BV-APPEND (BV-FIRSTN N BV)
                     (BV-APPEND S
                                (BV-RESTN (PLUS N (BV-LENGTH S)) BV)))
          (BV-APPEND (BV-FIRSTN N BV)
                     (BV-FIRSTN (DIFFERENCE (BV-LENGTH BV) N) S)))
      (BV-FIX BV)))
```

8. Bit Vector Operations

This theory builds on the primitive bit vector functions by defining several common operations on bits and bit vectors. Facts about these functions are in the deftheory `BV-OPERATIONS`. First, here are the bit operations. These are identical to some Boyer-Moore functions, but are defined separately to get greater control in enabling and disabling.

```
(DEFN BOOLP (X)
  (OR (TRUEP X) (FALSEP X)))

(DEFN B-AND (X Y)
  (AND X Y))

(DEFN B-EQV (X Y)
  (IF X (IF Y T F) (IF Y F T)))

(DEFN B-NAND (X Y)
  (IF X (IF Y F T) T))

(DEFN B-NOR (X Y)
  (IF X F (IF Y F T)))

(DEFN B-NOT (X)
  (IF X F T))

(DEFN B-OR (X Y)
  (OR X Y))

(DEFN B-XOR (X Y)
  (IF X (IF Y F T) (IF Y T F)))
```

The following functions take one or more bit vectors as arguments. `BV-ALL-TRUEP` determines if every bit in a bit vector is `TRUE`. `BV-ALL-FALSEP` determines if every bit in a bit vector is `FALSE`. `BV-AND` *and*s two bit vectors together. `BV-INCR` performs an operation whose natural number interpretation is that of incrementing the value of the bit vector by `1`. `BV-LAST-CARRY` returns the last carry computed by a `BV-INCR` operation. `BV-NOT` returns a bit vector, every bit of which is the negation of the corresponding bit in the argument. `BV-OR` *or*s two bit vectors together, and `BV-XOR` *xor*s two bit vectors.

```
(DEFN BV-ALL-TRUEP (BV)
  (IF (BV-NILP BV)
      T
      (AND (BV-BIT BV)
           (BV-ALL-TRUEP (BV-VEC BV)))))

(DEFN BV-ALL-FALSEP (BV)
  (IF (BV-NILP BV)
      T
      (AND (NOT (BV-BIT BV))
           (BV-ALL-FALSEP (BV-VEC BV)))))

(DEFN BV-AND (BV1 BV2)
  (IF (BV-NILP BV1)
      (BV-NIL)
      (BV (B-AND (BV-BIT BV1) (BV-BIT BV2))
          (BV-AND (BV-VEC BV1) (BV-VEC BV2)))))

(DEFN BV-INCR (C BV)
  (IF (BV-NILP BV)
      (BV-NIL)
      (BV (B-XOR C (BV-BIT BV))
          (BV-INCR (B-AND C (BV-BIT BV))
                   (BV-VEC BV)))))
```

```
(DEFN BV-LAST-CARRY (C BV)
  (IF (BV-NILP BV)
      C
      (BV-LAST-CARRY (B-AND C (BV-BIT BV))
                     (BV-VEC BV))))

(DEFN BV-NOT (BV)
  (IF (BV-NILP BV)
      (BV-NIL)
      (BV (B-NOT (BV-BIT BV))
          (BV-NOT (BV-VEC BV)))))

(DEFN BV-OR (BV1 BV2)
  (IF (BV-NILP BV1)
      (BV-NIL)
      (BV (B-OR (BV-BIT BV1) (BV-BIT BV2))
          (BV-OR (BV-VEC BV1) (BV-VEC BV2)))))

(DEFN BV-XOR (BV1 BV2)
  (IF (BV-NILP BV1)
      (BV-NIL)
      (BV (B-XOR (BV-BIT BV1) (BV-BIT BV2))
          (BV-XOR (BV-VEC BV1) (BV-VEC BV2)))))
```

For the shift and rotate operations, we use *left* and *right* to correspond to the way `BV` expressions are read, and not to signify an interpretation of bits as high order or low order. In the bit vector `(BV T (BV F (BV-NIL)))`, `T` is to the left of `F`. A function which defines a hardware shift-left operation usually means a shift to the high-order end. Such a function can call either the primitive shift-left or shift-right operation defined here, depending on how bit vectors are interpreted.

`BV-LSHIFT` shifts all but the first bit of a bit vector one bit to the left, and makes a given bit the last bit. `BV-RSHIFT` shifts all but the last bit of a bit vector one bit to the right, and makes a given bit the first bit. `BV-LSHIFTN` performs *n* left shifts, and `BV-RSHIFTN` preforms *n* right shifts. `BV-ROTATE-LEFT` rotates the bits of a bit vector *n* times to the left, and `BV-ROTATE-RIGHT` rotates the bits of a bit vector *n* times to the right.

```
(DEFN BV-LSHIFT (BIT BV)
  (IF (BV-NILP BV)
      (BV-FIX BV)
      (BV-APPEND (BV-VEC BV)
                 (BV BIT (BV-NIL)))))

(DEFN BV-RSHIFT (BIT BV)
  (IF (BV-NILP BV)
      (BV-FIX BV)
      (BV BIT
          (BV-FIRSTN (SUB1 (BV-LENGTH BV)) BV))))

(DEFN BV-LSHIFTN (N BIT BV)
  (IF (ZEROP N)
      (BV-FIX BV)
      (BV-LSHIFTN (SUB1 N) BIT (BV-LSHIFT BIT BV))))

(DEFN BV-RSHIFTN (N BIT BV)
  (IF (ZEROP N)
      (BV-FIX BV)
      (BV-RSHIFTN (SUB1 N) BIT (BV-RSHIFT BIT BV))))

(DEFN BV-ROTATE-LEFT (N BV)
  (IF (ZEROP N)
      (BV-FIX BV)
      (BV-ROTATE-LEFT (SUB1 N)
                      (BV-LSHIFT (BV-BIT BV) BV))))
```

```
(DEFN BV-ROTATE-RIGHT (N BV)
  (IF (ZEROP N)
      (BV-FIX BV)
      (BV-ROTATE-RIGHT (SUB1 N)
                       (BV-RSHIFT (BV-GET (SUB1 (BV-LENGTH BV)) BV)
                                  BV))))
```

9. The Natural Number Interpretation of Bit Vectors

The two mapping functions BV-TO-NAT and NAT-TO-BV define the natural number interpretation of bit vectors. These are the only two definitions which occur in this theory. The deftheory BV-TO-NATS contain facts about the natural number interpretation of many bit vector operations.

```
(DEFN BV-TO-NAT (X)
  (IF (BV-NILP X)
      0
      (PLUS (IF (TRUEP (BV-BIT X)) 1 0)
            (TIMES 2 (BV-TO-NAT (BV-VEC X))))))

(DEFN NAT-TO-BV (N LENGTH)
  (IF (ZEROP LENGTH)
      (BV-NIL)
      (BV (IF (ZEROP (REMAINDER N 2)) F T)
          (NAT-TO-BV (QUOTIENT N 2) (SUB1 LENGTH)))))
```

10. The Integer Interpretation of Bit Vectors

These functions describe the 2's complement representation of integers in bit vectors. Facts about the functions mentioned here are in the deftheory BV-TO-INTS. TC-REPRESENTABLE-INTEGER recognizes an integer which can be represented in 2's complement notation in a given bit vector length. BV-TO-INT maps the 2's complement representation of an integer to the integer, and INT-TO-BV maps an integer to its 2's complement representation.

```
(DEFN TC-REPRESENTABLE-INTEGER (I N)
  (IF (ZEROP N)
      F
      (IF (NEGATIVEP I)
          (LEQ (NEGATIVE-GUTS I) (EXP 2 (SUB1 N)))
          (LESSP (FIX I) (EXP 2 (SUB1 N))))))

(DEFN BV-TO-INT (BV)
  (IF (BV-GET (SUB1 (BV-LENGTH BV)) BV)
      (MINUS (BV-TO-NAT (BV-INCR T (BV-NOT BV))))
      (BV-TO-NAT BV)))

(DEFN INT-TO-BV (I LENGTH)
  (IF (NEGATIVEP I)
      (BV-INCR T (BV-NOT (NAT-TO-BV (NEGATIVE-GUTS I) LENGTH)))
      (NAT-TO-BV I LENGTH)))
```

11. How to Use the Library

You must load Matt Kaufmann's `DEFTHEORY` package into nqthm, since the library has some `DEFTHEORY` events. My personal copy of this is in "/usr/home/bevier/nqthm/theory". To get the whole library, note the file /usr/home/bevier/libs/bv-to-int. To get less, note one of the other libraries in /usr/home/bevier/libs/. Noting a library gets you everything which precedes it. The stack of libraries is as follows.

```
BV-TO-INT
BV-TO-NAT
BV-OPERATIONS
BIT-VECTORS
VECTORS
INTEGERS
ARITHMETIC
SETS-AND-BAGS
```

You may browse the corresponding event file to examine the contents of a library. There is currently very little documentation in the event files.

When you note one of these libraries, all non-recursive definitions and all crock lemmas (i.e., ones which I've proved to get important results) are globally disabled. All rewrite lemmas which are part of a theory, and all recursive functions are globally enabled. Therefore, after noting a library, you should be all set to start proving. You need not use `DEFTHEORY`.

Once J Moore's library package is available, I'll work on making this library available in that paradigm.

Acknowledgements

[This work was supported in part at Computational Logic, Inc., by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency or the U.S. Government.]

A Library For Hardware Verification

Table of Contents