2

BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic

Bernard Serpette
Jean Vuillemin
Jean-Claude Hervé

May 1989

Publication Notes

Bernard Serpette is with the Institut National de Recherche en Informatique et Automatique, 78150 Rocquencourt, France.

This report will also appear in French as an INRIA report.

Abstract

We describe a C package for arbitrary-precision integer arithmetic that is portable, yet efficient. Making the package run fast on a given computer involves re-writing a small kernel of our package in native assembly language. We provide such assembly code for VAX, 68020 and NS instruction sets, with good benchmarks. This package serves as a foundation for two arithmetic packages written in higher level languages LeLisp [1] and Modula2+.

This package is available to non-commercial users.

Résumé

Nous décrivons un module C d'arithmétique entière à précision arbitraire, portable, mais néanmoins efficace. La rapidité d'exécution de ce module est assujettie à la réécriture en langage machine du noyau du module. Nous fournissons les codes assembleurs pour les machines VAX, Mips, 68020 et NS. Ce module sert comme base pour deux bibliothèques arithmétiques écrites en langages de haut niveau LeLisp [2] et Modula2+.

Ce module est du domaine public pour toute utilisation non commerciale.

[1] LeLisp is a registered trademark of INRIA.
[2] LeLisp est une marque déposée par l'INRIA

Keywords

arithmetic, arbitrary length integer arithmetic

Acknowledgements

We are thankful to the following people, for their contribution to this package: Patrice Bertin, Hans Boehm, Jerôme Chailloux, Michel Gangnet, François Morain, David Salesin, and Mark Shand.

Contents

1 Introduction

Developing an arbitrary-precision arithmetic package that is both efficient and portable brings up two main problems:

- If an arithmetic package is written in a high-level language (such as languages Modula2+, LeLisp , C,. . .), the compiled code will typically be 4 to 10 times slower than carefully hand-crafted machine code.
- Many important arithmetic computations run an order of magnitude faster when all forms of storage allocation are removed from the inner loop, storing intermediate and final results exclusively in the memory area used by input variables. Yet some advanced form of automatic storage management and garbage collection must be part of any useful arithmetic package; furthermore, the number allocator must blend well with native storage allocation in the host language.

To satisfy these conflicting requirements, we have organized our software in two layers:

1. A layer called `Bn`, in which every operation deals with unsigned integers, allocates no storage, and returns results in place of the first argument passed to the routine.
2. A layer called `Bz`, implemented on top of `Bn`, which implements signed arithmetic operations and allocates storage for the results, in a straightforward manner.

Any high-level language $\mathcal{L}$ capable of interfacing directly with the language C can do so directly with `Bn` and `Bz`. When language $\mathcal{L}$ possesses its own storage allocator (possibly with garbage collection, as in LeLisp and Modula2+), it is best to rewrite directly in $\mathcal{L}$ the (small) storage allocation code of the `Bz` package. In this way, allocating and freeing numbers is directly handled in $\mathcal{L}$.

For speed reasons, the package `Bn` itself is structured in two layers:

- The kernel `KerN`, which contains code for the time-critical low-level operations.
- The rest of `Bn`, whose C code calls the kernel.

The kernel `KerN` is written in C for portability and documentation, and can be compiled as such. However, to obtain a truly efficient implementation on a given machine, `KerN` must be directly written in assembly code, which we provide for VAX, 68020, and NS instruction sets. `KerN` is small indeed: 325 to 475 lines of C code, and 500 to 700 lines of VAX assembly code.

The non-kernel part of `Bn` is written in C and compiled directly. The distinction between kernel and non-kernel operations is defined so that the time penalty for running the mixture of assembly and C code, as opposed to pure assembly code, is less than 20% on typical benchmarks. The knowledge of which procedures are in `KerN` is only important for someone

who attempts to port the package on a new machine and who is not satisfied with the speed of the C implementation.

Finally, we point out that Bn has proved to be a sound basis on which to develop other specialized packages, such as rational, polynomial or modular arithmetic. Having full control over the exact storage representation of numbers is the key to truly fast implementations of such extensions to the basic package.

Details on how to obtain this package are in appendix B.

2 Number representation

Using radix b positional notation, an integer $N \in \mathbf{N}$ can be written as:

$$\mathtt{N} = \sum_{0 \le i < nl} n_i \mathbf{b}^i \equiv {}_b[n_0 \cdots n_{nl-1}].$$

In this equation:

- $\mathbf{b} > 1$ is the *base*.
- For all i, $0 \le i < nl$, n_i are the *digits* of N written in base $\mathbf{b}$, such that $n_i \in \mathbf{B}_b = [0..\mathbf{b} - 1]$.
- The *length* nl of N is any integer greater than or equal to the number $\lceil log_b(N + 1) \rceil$ of significant digits of N written in base $\mathbf{b}$.

In the implementation, the base $\mathbf{b}$ is the largest power of two (typically 16 or 32) such that a digit fits in a memory word, and the instruction set supports base $\mathbf{b}$ unsigned extended arithmetic. The digit bit length is parameterized in the global C constant

$$\texttt{BN\_DIGIT\_SIZE} = \log_2(\mathbf{b}).$$

In `Bn`, an integer $\mathtt{N} \in \mathbf{N}$ is passed as a pair `(n,nl)` where:

1. `n` is a pointer inside an array of consecutive digits,
2. `nl` is the number (length) of digits from `n` in base $\mathbf{b}$.

Thus, for $0 \le i < nl$, n_i is the content of memory location $n + i$. We use below the notation:

$$\mathtt{N} = \sum_{0 \le i < nl} n_i \mathbf{b}^i \equiv (\mathtt{n}, \mathtt{nl}) \equiv {}_b[n_0 \cdots n_{nl-1}].$$

for the integer N.

In `Bz`, an integer is passed as a pointer to an array of consecutive digits; the header of this array contains the size (number of digits) of the array, and the sign of the number, defined as:

$$sgn(z) = \begin{cases} -1 \text{ if } z < 0; \\ 0 \text{ if } z = 0; \\ 1 \text{ if } z > 0; \end{cases}$$

The implementation defines the following C types:

```
typedef unsigned int   BigNumDigit; /* A single 32 bits digit */
typedef BigNumDigit    BigNumCarry; /* Either 0 or 1 */
typedef BigNumDigit *  BigNum;      /* Entity seen by the user */
typedef int            BigNumCmp;   /* = -1, 0, or 1 */
typedef int            Boolean;
```

Note that for the 16 bit implementation, the type `BigNumDigit` must be cast to a C `unsigned short`.

3 In place operations: Bn

We classify the procedures in Bn according to their underlying mathematical operations.

3.1 Initialization

Procedure 1 *Initialization* `BnnInit`*:*

- **C header***:*

  ```
  void BnnInit()
  ```

- **Inputs***: none*
- **Output***: none*
- **Side Effect***: initializes Bn.*

Note: `BnnInit` ∉ `KerN`.

Procedure 2 *Closing* `BnnClose`*:*

- **C header***:*

  ```
  void BnnClose()
  ```

- **Inputs***: none*
- **Output***: none*
- **Side Effect***: closes Bn.*

Note: `BnnClose` ∉ `KerN`.

3.2 Addition

Procedure 3 *Increment* `BnnAddCarry`*:*

- **C header***:*

```
BigNumCarry BnnAddCarry(n, nl, c)
    BigNum n;
    unsigned nl;
    BigNumCarry c;
```

- **Inputs***:*
  1. `N` $\equiv$ `(n,nl)` $\equiv {}_b[n_0 \cdots n_{nl-1}]$, *the integer to be incremented.*
  2. `c` $\in \mathbf{B}_2$, *the carry in.*
- **Invariant***:* `N` + `c` $= S \equiv {}_b[s_0 \cdots s_{nl-1} s_{nl}]$.
- **Output***: the carry out* $s_{nl} \in \mathbf{B}_2$.
- **Side Effect***: the* nl *least significant digits of* S *are stored back in memory locations* n *through* $n + nl - 1$ *as a* **side effect** *on the first argument* `n` *of the procedure:*

$$ {}_b[s_0 \cdots s_{nl-1}] \Rightarrow \texttt{(n,nl)}. $$

Note: `BnnAddCarry` $\in$ `KerN`; the output is 1 iff `N` $\equiv \mathbf{b}^{nl} - 1$ and `c` = 1; `nl` can be equal to zero, in which case the carry out has the same value as the carry in.

Procedure 4 *Addition* `BnnAdd`*:*

- **C header:**

```
BigNumCarry BnnAdd(m, ml, n, nl, c)
    BigNum m, n;
    unsigned ml, nl;
    BigNumCarry c;
```

- **Inputs***:*
  1. `M` $\equiv$ `(m,ml)` $\equiv {}_b[m_0 \cdots m_{ml-1}]$, *the addend;*
  2. `N` $\equiv$ `(n,nl)` $\equiv {}_b[n_0 \cdots n_{nl-1}]$, *the augend, no longer than the addend:* $nl \leq ml$.
  3. `c` $\in \mathbf{B}_2$, *the carry in.*
- **Invariant***:* `N` + `M` + `c` $= S \equiv {}_b[s_0 \cdots s_{ml-1} s_{ml}]$.
- **Output***: the carry out* $s_{ml} \in \mathbf{B}_2$.
- **Side Effect***:* ${}_b[s_0 \cdots s_{ml-1}] \Rightarrow$ `(m,ml)`.

Note: `BnnAdd` $\in$ `KerN`; it is possible to call `BnnAdd` with `m` = `n`; `nl` can be equal to zero, in which case the operation `BnnAdd` is equivalent to `BnnAddCarry(m, ml, c)`.

3.3 Subtraction

Procedure 5 *Additive Inverse* `BnnComplement`*:*

- **C header***:*

```
void BnnComplement(n, nl)
    BigNum n;
    unsigned nl;
```

- **Input***:* `N` $\equiv$ `(n,nl)` $\equiv {}_b[n_0 \cdots n_{nl-1}]$.
- **Invariant***:* `N` + $\overline{N}$ = $\mathbf{b}^{nl} - 1$; $\overline{N} \equiv {}_b[\overline{n_0} \cdots \overline{n_{nl-1}}]$, *with* $\overline{n_k} = \mathbf{b} - n_k - 1$.
- **Output***: none*
- **Side Effect***:* ${}_b[\overline{n_0} \cdots \overline{n_{nl-1}}] \Rightarrow$ `(n,nl)`.

Note: `BnnComplement` $\in$ `KerN`; does nothing when $nl = 0$.

Procedure 6 *Decrement* `BnnSubtractBorrow`*:*

- **C header:**

```
BigNumCarry BnnSubtractBorrow(n, nl, br)
    BigNum n;
    unsigned nl;
    BigNumCarry br;
```

- **Inputs***:*
  1. `N` $\equiv$ `(n,nl)` $\equiv {}_b[n_0 \cdots n_{nl-1}]$, *the integer to be decremented.*
  2. `br` $\in \mathbf{B}_2$, *the borrow in.*
- **Invariant***:* `N` + $\mathbf{b}^{nl}$ + `br` $- 1 = S \equiv {}_b[s_0 \cdots s_{nl-1} s_{nl}]$.
- **Output***: the borrow out* $s_{nl} \in \mathbf{B}_2$.
- **Side Effect***:* ${}_b[s_0 \cdots s_{nl-1}] \Rightarrow$ `(n,nl)`.

Note: `BnnSubtractBorrow` $\in$ `KerN`; the output is 0 iff `N` = `br` = 0; `nl` can be equal to zero, in which case the carry out has the same value as the carry in.

Procedure 7 *Subtraction* `BnnSubtract`*:*

- **C header:**

```
BigNumCarry BnnSubtract(m, ml, n, nl, br)
    BigNum m, n;
    unsigned ml, nl;
    BigNumCarry br;
```

- **Inputs***:*
  1. M ≡ (m,ml) ≡ $_b[m_0 \cdots m_{ml-1}]$, *the positive term;*
  2. N ≡ (n,nl) ≡ $_b[n_0 \cdots n_{nl-1}]$, *the negative term, no longer than the positive:* nl ≤ ml.
  3. br ∈ $\mathbf{B}_2$, *the borrow in.*
- **Invariant***:* M + $\mathbf{b}^{ml}$ − N + br − 1 = S ≡ $_b[s_0 \cdots s_{ml-1} s_{ml}]$.
- **Output***: the borrow out* $s_{ml} \in \mathbf{B}_2$.
- **Side Effect***:* $_b[s_0 \cdots s_{ml-1}] \Rightarrow$ (m,ml).

Note: BnnSubtract ∈ KerN; the output s_{ml} is 0 iff N + br ≤ M; it is possible to call BnnSubtract with m = n; nl can be equal to zero, in which case the operation BnnSubtract is equivalent to BnnSubtractBorrow(m, ml, br).

3.4 Multiplication

Procedure 8 *Multiplication by a digit* BnnMultiplyDigit*:*

- **C header:**

```
BigNumCarry BnnMultiplyDigit(p, pl, m, ml, d)
    BigNum p, m;
    unsigned pl, ml;
    BigNumDigit d;
```

- **Inputs***:*
  1. j ≡ (p,pl) ≡ $_b[p_0 \cdots p_{pl-1}]$, *the sum;*
  2. M ≡ (m,ml) ≡ $_b[m_0 \cdots m_{ml-1}]$, *the multiplier, shorter than the sum:* pl > ml.
  3. D ≡ *the multiplicand digit* d.
- **Invariant***:* j + M × D = R ≡ $_b[r_0 \cdots r_{pl-1} r_{pl}]$.
- **Output***: the carry out* $r_{pl} \in \mathbf{B}_2$.
- **Side Effect***:* $_b[r_0 \cdots r_{pl-1}] \Rightarrow$ (p,pl).

Note: `BnnMultiplyDigit` ∈ `KerN`; `ml` can be equal to zero, in which case the carry out is zero and no side effect is performed; it is possible to call `BnnMultiplyDigit` with `p` ≡ `m`; the digit `d` can be any digit of jor `M`.

Procedure 9 *Long Multiplication* `BnnMultiply`*:*

- **C header:**

```
BigNumCarry BnnMultiply (p, pl, m, ml, n, nl)
    BigNum p, m, n;
    unsigned pl, ml, nl;
```

- **Inputs***:*

  1. j ≡ `(p,pl)` ≡ $_b[p_0 \cdots p_{pl-1}]$, *the sum;*
  2. `M` ≡ `(m,ml)` ≡ $_b[m_0 \cdots m_{ml-1}]$, *the multiplier;*
  3. `N` ≡ `(n,nl)` ≡ $_b[n_0 \cdots n_{nl-1}]$, *the multiplicand.*

  The length of operands must be such that `pl` ≥ `nl` + `ml` *and* `ml` ≥ `nl`*; this last condition is imposed by speed requirements.*

- **Invariant***:* $j + \mathtt{M} \times \mathtt{N} = R \equiv {}_b[r_0 \cdots r_{pl-1} r_{pl}]$.

- **Output***: the carry out* $r_{pl} \in \mathbf{B}_2$.

- **Side Effect***:* $_b[r_0 \cdots r_{pl-1}] \Rightarrow$ `(p,pl)`.

Note: `In some implementations BnnMultiply` ∉ `KerN`; it is possible to call `BnnMultiply` with `n` = `m` if in addition $nl = ml$ special squaring code is used that is considerably faster than normal multiplication; when `nl` = 0 there is no side effect and the carry out is 0.

Procedure 10 *Multiply by a power of 2* `BnnShiftLeft`*:*

- **C header***:*

```
BigNumDigit BnnShiftLeft(m, ml, nbits)
    BigNum m;
    unsigned ml, nbits;
```

- **Inputs***:*

  1. `M` ≡ `(m,ml)` ≡ $_b[m_0 \cdots m_{ml-1}]$, *the integer to be shifted left.*
  2. `nbits` ∈ **N**, *the shift amount,* 0 ≤ `nbits` < $\log_2(\mathbf{b})$.

- **Invariant***:* $\texttt{M} \times 2^{\texttt{nbits}} = S \equiv {}_b[s_0 \cdots s_{ml-1} s_{ml}]$.
- **Output***: the digit s_{ml} shifted out.*
- **Side Effects***:* ${}_b[s_0 \cdots s_{ml-1}] \Rightarrow \texttt{(m,ml)}$.

Note: `BnnShiftLeft` $\in$ `KerN`; if `ml` = 0 then $s_{ml} = 0$.

3.5 Division

Procedure 11 *Division by a digit* `BnnDivideDigit`*:*

- **C header:**

```
BigNumDigit BnnDivideDigit(q, n, nl, d)
    BigNum q, n;
    unsigned nl;
    BigNumDigit d;
```

- **Inputs***:*
  1. $J \equiv$ `(q,nl-1)` $\equiv {}_b[q_0 \cdots q_{nl-2}]$,.
  2. `N` $\equiv$ `(n,nl)` $\equiv {}_b[n_0 \cdots n_{nl-1}]$, *the dividend.*
  3. `D` $\equiv$ *the divisor digit* `d`*, whose value must be greater than that of the most significant digit of* `N`*:* `d` $> n_{nl-1}$.
- **Invariant***:* `N` = `D` $\times J +$ `R`, $0 \leq$ `R` $<$ `D`*, with* $J \equiv {}_b[q_0 \cdots q_{nl-2}]$, `R` $\equiv {}_b[r_0]$.
- **Output***: the remaining digit R.*
- **Side Effect***:* ${}_b[q_0 \cdots q_{nl-2}] \Rightarrow \texttt{(q,nl-1)}$.

Note: `BnnDivideDigit` $\in$ `KerN` stores the `nl` -1 digits of the quotient in J and returns the remainder.

Procedure 12 *Long Division* `BnnDivide`*:*

- **C header:**

```
void BnnDivide (n, nl, d, dl)
    BigNum n, d;
    unsigned nl, dl;
```

- **Inputs***:*

1. `N` ≡ `(n,nl)` ≡ ${}_b[n_0 \cdots n_{nl-1}]$, *the dividend.*
2. `D` ≡ `(d,dl)`, *the divisor, shorter than the dividend:* `dl` < `nl`.

The most significant digit of the divisor must be greater than that of the dividend:

$$d_{dl-1} > n_{nl-1}.$$

Without this condition, we could not guarantee that both quotient and remainder will exactly fit in the storage allocated to `N`.

- **Invariant***:* `N` = `D` × J + `R`, $0 \leq$ `R` < `D`, *with* $J \equiv {}_b[q_0 \cdots q_{nl-dl-1}]$, `R` ≡ ${}_b[r_0 \cdots r_{dl-1}]$.
- **Output***: none*
- **Side Effect***:* ${}_b[r_0 \cdots r_{dl-1}] \Rightarrow$ `(n,dl)`; ${}_b[q_0 \cdots q_{nl-dl-1}] \Rightarrow$ `(n+dl,nl-dl)`.

Note: `BnnDivide` ∉ `KerN`, replaces the $nl - dl$ most significant digits of `N` by the quotient, and the dl least significant digits by the remainder.

Procedure 13 *Divide by a power of 2* `BnnShiftRight`*:*

- **C header***:*

```
BigNumDigit BnnShiftRight(m, ml, nbits)
    BigNum m;
    unsigned ml, nbits;
```

- **Inputs***:*
  1. `M` ≡ `(m,ml)` ≡ ${}_b[m_0 \cdots m_{ml-1}]$, *the integer to be shifted right.*
  2. `nbits` ∈ **N**, *the shift amount,* 0 ≤`nbits`< $\log_2(\mathbf{b})$.
- **Invariant***:* `M` = $2^{\texttt{nbits}} \times S + R \times 2^{\texttt{nbits}-BN\_DIGIT\_SIZE} = 2^{\texttt{nbits}} \times (S + R \times \mathbf{b}^{-1})$ *with* $S \equiv {}_b[s_0 \cdots s_{ml-1}]$, $0 \leq R < \mathbf{b}$.
- **Output***: the shifted out digit* R.
- **Side Effect***:* ${}_b[s_0 \cdots s_{ml-1}] \Rightarrow$ `(m,ml)` .

Note: `BnnShiftRight` ∈ `KerN`; if `ml` = 0 then R = 0.

The next two procedures are used in the normalization step of long division.

Procedure 14 `BnnNumLeadingZeroBitsInDigit`*:*

- **C header***:*

```
unsigned BnnNumLeadingZeroBitsInDigit(d)
    BigNumDigit d;
```

- **Input***: digit* d.
- **Invariant***:* $\frac{\mathbf{b}}{2} < 2^k \times (\mathsf{d} + 1) \leq \mathbf{b}$.
- **Output***:* k.
- **Side Effect***: none*

Note: `BnnNumLeadingZeroBitsInDigit` $\in$ `KerN`. The output is the number of most significant bits equal to zero in d.

Procedure 15 `BnnIsDigitNormalized` *:*

- **C header***:*

```
Boolean BnnIsDigitNormalized(d)
    BigNumDigit d;
```

- **Input***: digit* d.
- **Output***: the predicate* $\frac{\mathbf{b}}{2} \leq \mathsf{d} < \mathbf{b}$.
- **Side Effect***: none.*

Note: `BnnIsDigitNormalized` $\in$ `KerN`.

3.6 Comparisons

Procedure 16 *Test for zero digit* `BnnIsDigitZero`*:*

- **C header***:*

```
Boolean BnnIsDigitZero(d)
    BigNumDigit d;
```

- **Input***: digit* d.
- **Output***: the predicate* (d = 0).
- **Side Effect***: none*

Note: `BnnIsDigitZero` $\in$ `KerN`.

Procedure 17 *Test for zero number* `BnnIsZero`*:*

- **C header***:*

```
Boolean BnnIsZero (n, nl)
    BigNum n;
    unsigned nl;
```

- **Input***:* `N` $\equiv$ `(n,nl)` $\equiv {}_b[n_0 \cdots n_{nl-1}]$.
- **Output***: the predicate* (`N` = 0).
- **Side Effect***: none*

Note: `BnnIsZero` $\notin$ `KerN`.

Procedure 18 *Digit comparison* `BnnCompareDigits`*:*

- **C header***:*

```
BigNumCmp BnnCompareDigits(c, d)
    BigNumDigit c, d;
```

- **Inputs***: digits* `c` *and* `d`.
- **Output***:* sgn(`c` $-$ `d`) = $\{-1, 0, 1\}$.
- **Side Effect***: none*

Note: `BnnCompareDigits` $\in$ `KerN`.

Procedure 19 *Number comparison* `BnnCompare`*:*

- **C header***:*

```
BigNumCmp BnnCompare (m, ml, n, nl)
    BigNum m, n;
    unsigned ml, nl;
```

- **Inputs***:*
  1. `M` $\equiv$ `(m,ml)` $\equiv {}_b[m_0 \cdots m_{ml-1}]$.
  2. `N` $\equiv$ `(n,nl)` $\equiv {}_b[n_0 \cdots n_{nl-1}]$.

- **Output**: $sgn(\mathtt{M} - \mathtt{N}) = \{-1, 0, 1\}$.
- **Side Effect**: *none*

Note: `BnnCompare` $\notin$ `KerN`.

Procedure 20 `BnnIsDigitOdd` :

- **C header**:

```
Boolean BnnIsDigitOdd(d)
    BigNumDigit d;
```

- **Input**: *digit* d.
- **Output**: *the predicate* (d mod 2 = 1).
- **Side Effect**: *none.*

Note: `BnnIsDigitOdd` $\notin$ `KerN`.

Procedure 21 `BnnNumDigits` :

- **C header**:

```
unsigned BnnNumDigits(n, nl)
    BigNum n;
    unsigned nl;
```

- **Input**: `N` $\equiv$ `(n,nl)`.
- **Invariant**: $ln = \lceil \log_{\mathbf{b}}(\mathtt{N} + 1) \rceil$, *if* $\mathtt{N} \neq 0$ *and* 1 *otherwise.*
- **Output**: ln.
- **Side Effect**: *none.*

Note: `BnnNumDigits` $\in$ `KerN`; ln is the number of significant digits of `N`.

3.7 Logical operations

Procedure 22 `BnnAndDigits` :

- **C header**:

```
void BnnAndDigits(n, d)
    BigNum n;
    BigNumDigit d;
```

- **Inputs**
  1. `N` ≡ `(n,1)`.
  2. `D` ≡ *digit* `d`.
- **Invariant**: *let* $n\&d$ *be the bitwise logical* AND *of* `N` *and* D.
- **Output**: *none*
- **Side Effect**: $n\&d \Rightarrow$ `(n,1)`.

Note: `BnnAndDigits` ∈ `KerN`.

Procedure 23 `BnnOrDigits` :

- **C header**:

```
void BnnOrDigits(n, d)
    BigNum n;
    BigNumDigit d;
```

- **Inputs**
  1. `N` ≡ `(n,1)`.
  2. `D` ≡ *digit* `d`.
- **Invariant**: *let* $n \vee d$ *be the bitwise logical* OR *of* `N` *and* D.
- **Output**: *none*
- **Side Effect**: $n \vee d \Rightarrow$ `(n,1)`.

Note: `BnnOrDigits` ∈ `KerN`.

Procedure 24 `BnnXorDigits` :

- **C header**:

```
void BnnXorDigits(n, d)
    BigNum n;
    BigNumDigit d;
```

- **Inputs**
  1. `N` ≡ `(n,1)`.
  2. `D` ≡ *digit* `d`.
- **Invariant**: *let* $n \oplus d$ *be the bitwise* EXCLUSIVE-OR *of* `N` *and* D.
- **Output**: *none*
- **Side Effect**: $n \oplus d \Rightarrow$ `(n,1)`.

Note: `BnnXorDigits` ∈ `KerN`.

3.8 Assignments

The following functions permit direct manipulation of the representation.

Procedure 25 `BnnSetToZero` :

- **C header**:

```
void BnnSetToZero(n, nl)
    BigNum n;
    unsigned nl;
```

- **Input**: `N` ≡ `(n,nl)` ≡ $_b[n_0 \cdots n_{nl-1}]$.
- **Output**: *none*
- **Side Effect**: $0 \Rightarrow$ `(n,nl)`.

Note: `BnnSetToZero` ∈ `KerN`; if `nl` = 0 then no side effect is performed.

Procedure 26 `BnnSetDigit` :

- **C header**:

```
void BnnSetDigit(n, d)
    BigNum n;
    BigNumDigit d;
```

- **Inputs**
  1. `N` ≡ `(n,1)`.
  2. `d` *is an integer* $0 \leq$ `d` $<$ **b**.

- **Output***: none*
- **Side Effect***:* `d` $\Rightarrow$ `(n,1)`.

Note: `BnnSetDigit` $\in$ `KerN`.

Procedure 27 `BnnAssign` *:*

- **C header***:*

```
void BnnAssign(m, n, nl)
    BigNum m, n;
    unsigned nl;
```

- **Inputs**
  1. `M` $\equiv$ `(m,nl)` $\equiv$ $_b[m_0 \cdots m_{nl-1}]$.
  2. `N` $\equiv$ `(n,nl)` $\equiv$ $_b[n_0 \cdots n_{nl-1}]$.
- **Output***: none*
- **Side Effect***:* `N` $\Rightarrow$ `(m,nl)`.

Note: `BnnAssign` $\in$ `KerN`; all kinds of overlapping are possible; no side effect when `nl` = 0.

3.9 Conversion to a small integer

In most languages, it is not possible to represent, as the value of an ordinary integer, a full size (e.g., 32 bit) digit. The following predicate specifies which digits can be directly represented by an integer in the target language. The binary length of such numbers is less than the package constant: `BN_WORD_SIZE`. (In LeLisp , `BN_WORD_SIZE`=15, for example).

Procedure 28 `BnnDoesDigitFitInWord` *:*

- **C header***:*

```
Boolean BnnDoesDigitFitInWord(d)
    BigNumDigit d;
```

- **Input***:* `D` $\equiv$ *digit* `d`.
- **Output***: the predicate* $\log_2(\texttt{D}) \leq$ `BN_WORD_SIZE`.
- **Side Effect***: none.*

Note: `BnnDoesDigitFitInWord` ∈ `KerN`.

Procedure 29 `BnnGetDigit` *:*

- **C header***:*

```
BigNumDigit BnnGetDigit(n)
    BigNum n;
```

- **Input***:* `N` ≡ `(n,1)` ≡ n_0*, such that* $\log_2(n_0) \leq$ `BN_WORD_SIZE`.
- **Output***: the digit* n_0 *as a regular C unsigned integer.*
- **Side Effect***: none*

Note: `BnnGetDigit` ∈ `KerN`; this function does not test whether n_0 actually fits in a word.

4 Storage allocating operations: Bz

The layer Bz is conceptually simpler than Bn. A number $z \in Z$ is represented by a pointer to an array containing the sign and the digits of the base **b** representation of z. Procedures in Bz allocate storage for their results.

4.1 Initialization Operations

Procedure 30 *Initialization* `BzInit`*:*

- **C header***:*

```
void BzInit()
```

- **Inputs***: none*
- **Output***: none*
- **Side Effect***: initializes Bz and Bn.*

Procedure 31 *Closing* `BzClose`*:*

- **C header***:*

```
void BzClose()
```

- **Inputs***: none*
- **Output***: none*
- **Side Effect***: closes Bz and Bn.*

4.2 Storage operations

Procedure 32 *Allocate* `BzCreate`*:*

- **C header***:*

```
BigZ BzCreate (size)
     unsigned size;
```

- **Output***: a number having* `size` *digits.*

Procedure 33 *Dispose* `BzFree`*:*

- **C header***:*

```
void BzFree (z)
     BigZ z;
```

- **Output***: none*
- **Side Effect***: Frees the storage occupied by* z.

Procedure 34 *Dispose* `BzFreeString`*:*

- **C header***:*

```
void BzFreeString (s)
     char *s;
```

- **Output***: none*
- **Side Effect***: Frees the storage occupied by* s *(previously allocated by* `BzToString`*).*

Procedure 35 *Physical copy* `BzCopy`*:*

- **C header***:*

```
BigZ BzCopy (z)
     BigZ z;
```

- **Output***:* `BzCopy(z)` $= z$.

Procedure 36 *Size* `BzNumDigits`*:*

- **C header***:*

```
unsigned BzNumDigits (z)
     BigZ z;
```

- **Output***: the number of significant digits of* z.

Procedure 37 *Size* `BzGetSize`*:*

- **C header***:*

```
unsigned BzGetSize (z)
     BigZ z;
```

- **Output***: the number of allocated digits of* z.

4.3 Arithmetic Operations

Procedure 38 *Absolute value* `BzAbs`

- **C header***:*

```
BigZ BzAbs (z)
     BigZ z;
```

- **Output***:* `BzAbs(z)`$= |z| = sgn(z) \times z$.

Procedure 39 *Sign* `BzGetSign`*:*

- **C header***:*

```
BigNumCmp BzGetSign (z)
     BigZ z;
```

- **Output***:* `BzGetSign(z)` $= sgn(z) = \{-1, 0, 1\}$.

Procedure 40 *Arithmetic opposite* `BzNegate`*:*

- **C header***:*

```
BigZ BzNegate (z)
     BigZ z;
```

- **Output***:* `BzNegate(z)` $= -z$.

Procedure 41 *Comparison* `BzCompare`*:*

- **C header***:*

```
BigNumCmp BzCompare (y, z)
     BigZ y, z;
```

- **Output***:* `BzCompare(y,z)` $= sgn(y - z) = \{-1, 0, 1\}$.

Procedure 42 *Addition* `BzAdd`*:*

- **C header***:*

```
BigZ BzAdd (y, z)
     BigZ y, z;
```

- **Output**: `BzAdd(y,z)` = $y + z$.

Procedure 43 *Subtraction* `BzSubtract`:

- **C header**:

```
BigZ BzSubtract (y, z)
     BigZ y, z;
```

- **Output**: `BzSubtract(y,z)` = $y - z$.

Procedure 44 *Multiplication* `BzMultiply`:

- **C header**:

```
BigZ BzMultiply (y, z)
     BigZ y, z;
```

- **Output**: `BzMultiply(y,z)` = $y \times z$.

Procedure 45 *Quotient* `BzDiv`:

- **C header**:

```
BigZ BzDiv (y, z)
     BigZ y, z;
```

- **Output**: `BzDiv(y,z)` = $y \div z$.
  Returns $NULL$ *if* $z = 0$.
  Returns $floor(y/z)$ *if* $z > 0$
  otherwise returns $ceil(y/z)$
  where $/$ *is the real numbers division.*

Procedure 46 *Modulo* `BzMod`:

- **C header**:

```
BigZ BzMod (y, z)
     BigZ y, z;
```

- **Output**: `BzMod(y,z)` = $y \bmod z$.

Procedure 47 *Division* `BzDivide` :

- **C header***:*

```
BigZ BzDivide (y, z, r)
     BigZ y, z, *r;
```

- **Output***: the quotient* $y \div z$*.*
  Returns $NULL$ *if* $z = 0$*.*
  Returns $floor(y/z)$ *if* $z > 0$
  otherwise returns $ceil(y/z)$
  where $/$ *is the real numbers division.*

- **Side Effect***: assigns the modulo to* r *such that* $0 \leq r < abs(z)$

4.4 Read and Print

Procedure 48 *Write in base* b `BzToString`*:*

- **C header***:*

```
char* BzToString (z, b)
     BigZ z;
     unsigned b;
```

- **Output***:* `BzToString(z,b)` *is the string representing* z *in base* b*, with* $2 \leq b \leq 16$*.*

Procedure 49 *Read in base* b `BzFromString`*:*

- **C header***:*

```
BigZ BzFromString (s, b)
     char *s;
     unsigned b;
```

- **Output***:* `BzFromString(s,b)` *is the number represented by the string* s*, in base* b *with* $2 \leq b \leq 16$*, .*

4.5 Conversions

Procedure 50 `BzFromInteger`*:*

- **C header***:*

```
BigZ BzFromInteger (i)
     int i;
```

- **Output***: a number equal to* `i`.

Procedure 51 `BzToInteger`*:*

- **C header***:*

```
int BzToInteger (z)
     BigZ z;
```

- **Output***: an integer equal to* `z` *iff* `-MAXINT` $<$ `z` $\leq$ `MAXINT`*, otherwise returns* `-MAXINT`.

Procedure 52 `BzFromBigNum`*:*

- **C header***:*

```
BigZ BzFromBigNum (n, nl)
     BigNum n;
     unsigned nl;
```

- **Inputs***:* `N` $\equiv$ `(n,nl)` $\equiv$ $_b[n_0 \cdots n_{nl-1}]$.
- **Output***: a number equal to* `N`.

Procedure 53 `BzToBigNum`*:*

- **C header***:*

```
BigNum BzToBigNum (z, nl)
     BigZ z;
     unsigned * nl;
```

- **Output***: a number* `N` *equal to* `z` *iff* `z` ≥ 0*, otherwise returns* $NULL$.
- **Side Effect***: assigns the length of* `N` *to* `nl`.

5 Bibliography

[Knuth] D. E. Knuth, The Art of Computer Programming, vol. 2, Seminumerical Algorithms. Addison Wesley, 1981.

A What is in the package?

```
   Documentation Files:
      doc/bn.tex       - This document in LaTeX format
      doc/bnf.tex      - Document BigNum in French and LaTeX format

   C Include Files:
      h/BigZ.h         - Types and structures for clients of BigZ
      h/BigNum.h       - Types and structures for clients of BigNum

   C Source Code:
      c/bz.c           - BigZ implementation
      c/bn.c           - BigNum implementation ("non-kernel" routines)
      c/KerN.c         - BigNum implementation ("kernel" routines)
      c/bztest.c       - Test program for verifying BigZ implementation
      c/testKerN.c     - Test program for verifying KerN implementation

   Assembly-Language Source Code:
      s/vaxKerN.s      - VAX implementation of KerN
      s/68KerN.s       - 68020 implementation of KerN (MIT syntax)
      s/68KerN.mot.s   - 68020 implementation of KerN (Motorola syntax)
      s/nsKerN.s       - NS implementation of KerN

   Other Files:
      Makefile         - Compiles source code, creates test programs

In order to build or modify the current version of the package,
the following commands are provided:
      make vax         - to use vax assembly code
      make 68K         - to use 68020 assembly code
      make ns          - to use NS assembly code
      make C16         - to use C code with 16 bit digits
      make C32         - to use C code with 32 bit digits
      make             - to use the default version (C32)

One of these commands products the following files:
      BigNum.a         - BigNum library
      bztest           - Test program executable for BigZ
      testKerN         - Test program executable for KerN

If you have the tools LaTeX, makeindex and aptex, type:
      make doc         - to build the Postscript files of the documents
```

B How to obtain the package?

This document and the source code of the BigNum package bear the marking "Copyright Digital Equipment Corporation & INRIA 1989" This documentation and the source code of the BigNum package may be ordered either by postal or electronic mail from:

Librarian
Digital PRL
85, Avenue Victor Hugo
92563 Rueil Malmaison Cedex France
(doc-server@prl.dec.com)
(or decprl::doc-server)

or INRIA
Bernard Serpette
Domaine de Voluceau
78150 Rocquencourt France
(serpette@inria.inria.fr)

The source code will be sent through electronic mail.

This documentation, and the source code of the BigNum package may be reproduced and distributed freely to non commercial usage provided that the following conditions are respected:

- Digital PRL or INRIA should be notified of the copy.
- The original Copyright notice should not be removed from the documentation or from the source code under any circumstances.
- Any work using the BigNum package should state explicitly the use of such package, and its origin by including the following sentence: *This work uses the BigNum package developed jointly by INRIA and Digital PRL.*
- If any modification is applied to the BigNum package, explicit statements should identify the fact that such modifications have been made, by whom, and where. These statements should not be removed in any further distribution.
- Any work using extensively the BigNum package should be freely distributed under conditions similar to the distribution of the BigNum package.

INRIA and Digital Equipment Corporation make no representations, express or implicit, with

respect to this documentation or the software it describes, including without limitations, any implied warranties of merchandability or fitness for a particular purpose, all of which are expressly disclaimed. INRIA and Digital Equipment Corporation or subsequent distributors shall in no event be liable for any indirect, incidental or consequential damages.

Index

PRL Research Reports

outsideorderinfo

Research Report 1: *Incremental Computation of Planar Maps*. Michel Gangnet, Jean-Claude Hervé, Thierry Pudet, and Jean-Manuel Van Thong. May 1989.

Research Report 2: *BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic*. Bernard Serpette, Jean Vuillemin, and Jean-Claude Hervé. May 1989.

Research Report 3: *Introduction to Programmable Active Memories*. Patrice Bertin, Didier Roncin, and Jean Vuillemin. June 1989.

Research Report 4: *Compiling Pattern Matching by Term Decomposition*. Laurence Puel and Ascánder Suárez. January 1990.

Research Report 5: *The WAM: A (Real) Tutorial*. Hassan Aït-Kaci. January 1990.

Research Report 6: *Binary Periodic Synchronizing Sequences*. Marcin Skubiszewski. May 1991.

Research Report 7: *The Siphon: Managing Distant Replicated Repositories*. Francis J. Prusker and Edward P. Wobber. May 1991.

Research Report 8: *Constructive Logics. Part I: A Tutorial on Proof Systems and Typed λ-Calculi*. Jean Gallier. May 1991.

Research Report 9: *Constructive Logics. Part II: Linear Logic and Proof Nets*. Jean Gallier. May 1991.

Research Report 10: *Pattern Matching in Order-Sorted Languages*. Delia Kesner. May 1991.

Research Report 11: *Towards a Meaning of LIFE*. Hassan Aït-Kaci and Andreas Podelski. May 1991.

Research Report 12: *Residuation and Guarded Rules for Constraint Logic Programming*. Gert Smolka. May 1991.

Research Report 13: *Functions as Passive Constraints in LIFE*. Hassan Aït-Kaci and Andreas Podelski. May 1991.

Research Report 14: *Automatic Motion Planning for Complex Articulated Bodies*. Jérôme Barraquand. May 1991.