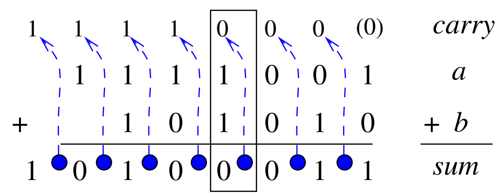
# M. Methodology

## M.1 Implementing Addition

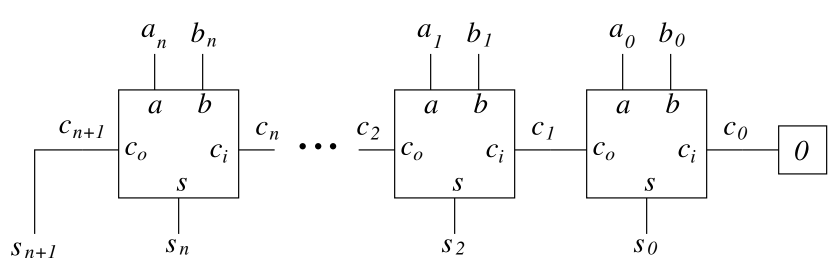
The PVS source file `K.pvs` illustrates basic concepts of *implementation verification* using binary addition as an example.

### M.1.1 Review of Binary Addition

You might never have learned, or may not recall, how binary addition works. Section 2.5 of *Induction, Recursion and Programming* describes this in detail. More briefly, binary addition is done in the same way as decimal addition, column by column. The only difference is that the *base* is 2 rather than 10. Each column is summed and if the column-sum exceeds a single digit the leading 1 is carried to the next column.




*Implementing Binary Addition* The goal is to describe implemention of this algorithm in boolean logic, using operators '·' for logical *and*. '+' for logical *or*, and $\overline{x}$ for *not*. Such an implementation would have identical components for each column.




The carry bit $c_{i+1}$ is 1 whenever two or more of the inputs are 1s, that is,

$$c_{i+1} = \mathit{majority}(a_i, b_i, c_i) \stackrel{\text{def}}{=} a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$

The sum bit $s_i$ is 1 when an odd number of inputs are 1s, that is,

$$s_i = \mathit{parity}(a_i, b_i, c_i) \stackrel{\text{def}}{=} a_i \oplus b_i \oplus c_i$$

where '$\oplus$' stands for *exclusive-or*,

$$x \oplus y \stackrel{\text{def}}{=} x \cdot \overline{y} \; + \; \overline{x} \cdot y$$

### M.1.2 Implementation Verification

Recall from the *Terminology* notes, that *verification* is described as the process of determining whether an implementation *satisfies* specification. In this example, the specification is, “add two natural *numbers*,” and the implementation is to perform binary addition on two stings of binary digits, or *numerals*. So the key detail added in this implementation is the representation of *numbers* by binary numerals.

$$
\begin{tikzcd}
\mathit{number}^2 \arrow[r, "+"] \arrow[d, shift right, "\rho"'] & \mathit{number} \arrow[d, shift right, "\rho"'] \\
\mathsf{numeral}^2 \arrow[r, "\mathsf{adder}"'] \arrow[u, shift right, "\alpha"'] & \mathsf{numeral} \arrow[u, shift right, "\alpha"']
\end{tikzcd}
$$

The functions $\rho$ and $\alpha$ relate numbers and numerals. Given an number $n$ and a numeral $N = d_k \cdots d_1\, d_0$, the abstraction function $\alpha$ in the diagram is defined

$$\alpha[\![d_0\, d_1 \cdots d_k]\!] \quad = \quad \sum_{i=0}^{k} 2^i \widetilde{d_i}$$

On the right-hand side, digit $d_i$ has been decorated $\widetilde{d_i}$ because it is being interpreted as a number rather than a symbol: $\widetilde{\mathsf{0}} \leftrightarrow 0$ and $\widetilde{\mathsf{1}} \leftrightarrow 1$. We do not need to define a representation function ($\rho$, see Note 1) because the form of our correctness statement is

$$\text{For all } X, Y \in \mathsf{numeral},\ \alpha[\![\mathsf{adder}(X, Y)]\!] = \alpha[\![X]\!] + \alpha[\![Y]\!]$$

In words, “All *representable* numbers are added correctly.”

$$
\begin{tikzcd}
\alpha(X),\ \alpha(Y) \arrow[r] & \alpha(X) + \alpha(Y) \\
X,\ Y \arrow[r] \arrow[u] & \mathsf{adder}(X, Y) \arrow[u]
\end{tikzcd}
$$

### M.1.3 Formulation in PVS

Numerals are modeled as inductively defined *boolean lists*.

```
boolist: DATATYPE
 BEGIN
  null: null?
  cons (first: bool, rest:boolist):cons?
 END boolist
```

The primitive `bool` type is used to model binary digits (bits) so that PVS’s logical operations `AND`, `OR`, `NOT`, `XOR`, etc. may be used to formulate the `majority` and `parity` functions defined earlier. A `boolist` represents a binary numeral whose leading digit is the least significant bit. For example, the binary numeral 1011 is expressed as

```
cons(true, cons(true, cons(false, cons(true, null))))
```

The abstraction function $\alpha$ can then be easily defined recursively as

```
VAL(l:boolist): RECURSIVE nat =
  CASES l OF
    null: 0,
    cons(b, tl): (IF b THEN 1 ELSE 0 ENDIF) + 2 * VAL(tl)
  ENDCASES
  MEASURE l by <<
```

### M.1.4 Notes

1. Let '$\div$' stand for *integer quotient*. The representation function $\rho$ would be

$$\rho(n) = d_k \cdots d_1\, d_0 \text{ where } k \geq (\log_2 n) \text{ and } d_i = \begin{cases} \mathtt{0} & \text{if } (n \div 2^i) \text{ is even} \\ \mathtt{1} & \text{if } (n \div 2^i) \text{ is odd} \end{cases}$$

2. The `INC` example recursively traverses a `boolist`.

```
INC(l:boolist): RECURSIVE boolist =
  CASES l OF
    null: cons(true, null),
    cons(b, tl): IF b
                 THEN cons(false, INC(tl))
                 ELSE cons(true, tl)
                 ENDIF
  ENDCASES
  MEASURE l by <<
```

   Whether this suggest temporal ("bit-serial") or geometric ("bit-parallel") iteration open to interpretation. It depends on what the recursion is intended to model.

3. The suggested `ADD` function,

```
ADD(l1, l2: boolist, c:bool): RECURSIVE boolist = ...
```

   does not require the `boolist` arguments to be the same length. This is a bit simpler to deal with, but one would ordinarily expect to see an $N$-bit adder, for some fixed constant $N$.