# Verifying adder circuits using powerlists

William Adams*
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712–1188
USA
e-mail: `will@cs.utexas.edu`

March 29, 1994

**Abstract**

We define the ripple-carry and the carry-lookahead adder circuits in the powerlist notation and we use the powerlist algebra to prove that these circuits correctly implement addition for natural numbers represented as bit vectors.

## 0 Introduction

As hardware designs increase in complexity it is less possible to reason informally about their behaviour, or to exhaustively test all possible behaviours. Several researchers have used formal systems for hardware verification, such as the Boyer-Moore logic [2], HOL [1], Nuprl [5] and Ruby [3]. We propose the use of a new data structure, the *powerlist*, for circuit verification. We show how powerlists may be used to express circuits and reason about their correctness.

The powerlist data structure has been recently introduced by Misra [6]. It provides a notation for compactly expressing synchronous parallel computations in a functional programming style and an algebra within which properties of such computations can be proven. The notation allows a computation on a given powerlist to be expressed in terms of computations on two components, each of which is half the size of the original list. Recursion is used to express "divide and conquer" algorithms.

Synchronous digital circuits are often regular, in the sense that a computation is performed in terms of several disjoint, similar, subcomputations. Thus

*This material is based in part upon work supported by National Science Foundation Award CCR-9111912

powerlists appear to be well-suited to expressing the functionality and reasoning about the behaviour of such circuits. We have used powerlists to define and verify a couple of standard adder circuits, the ripple-carry and the carry-lookahead adders.

The powerlist notation allows us to write succinct definitions of these circuits. Also, the powerlist algebra allows short, equational proofs of all the results that we need to verify these circuits. We have provided detailed proofs of our results; the steps in these proofs are sufficiently small that there is a good possibility of providing automated support for the generation of similar proofs[1].

Our main result is a proof that the ripple-carry and the carry-lookahead adder circuits compute the same function. We also give a proof that the ripple-carry adder circuit correctly implements addition on natural numbers. The two results together show that the carry-lookahead adder circuit also correctly implements addition.

## 1 Powerlists

Misra defines a data structure called the *powerlist*. Given a set $B$ of elements, called *scalars*, the set of powerlists over $B$ is the set of linear lists over $B$ whose length is a power of 2. The smallest powerlist is thus a list containing a single element $x$ of $B$, which we write $\langle x \rangle$. We call such a powerlist a *singleton*. There are two constructors which allow us to construct new powerlists from powerlists $p$ and $q$, where $p$ and $q$ are of equal length.

- $p \mid q$ (read $p$ *tie* $q$ ) is the powerlist formed by concatenating $p$ and $q$.
- $p \bowtie q$ (read $p$ *zip* $q$ ) is the powerlist formed by interleaving the elements of $p$ and $q$.

**Note** We use the following conventions for variable names throughout this paper: $a$, $b$, $c$, $d$, $e$, $f$, $g$, $h$, $x$, $y$ and $z$ represent scalars (elements of $B$); $p$, $q$, $r$, $s$, $t$, $u$, $v$, $w$ represent powerlists. When giving examples of powerlists we enclose the elements in angle brackets.

The following are examples of powerlists.

$$\langle a \rangle$$

$$\langle e \;\, f \rangle$$

$$\langle b \;\, d \;\, f \;\, h \rangle$$

$$\langle a \;\, b \;\, c \;\, d \;\, e \;\, f \;\, g \;\, h \rangle$$

[1]Researchers at SUNY Albany, led by Deepak Kapur, have already mechanically generated proofs for many of the results given in Misra's original powerlist paper.

The following illustrate the definition of tie and zip.

$$\langle a\ b\ c\ d\rangle \mid \langle e\ f\ g\ h\rangle \ = \ \langle a\ b\ c\ d\ e\ f\ g\ h\rangle$$

$$\langle a\ b\ c\ d\rangle \bowtie \langle e\ f\ g\ h\rangle \ = \ \langle a\ e\ b\ f\ c\ g\ d\ h\rangle$$

$$\langle a\rangle \mid \langle b\rangle \ = \ \langle a\ b\rangle \ = \ \langle a\rangle \bowtie \langle b\rangle$$

It is clear that if we confine ourselves to the singleton constructor $\langle\cdot\rangle$ and the constructors $\mid$ and $\bowtie$ on equal length lists that we can construct all lists of length $2^n$ for $n \geq 0$ and no others. For a powerlist of length $2^n$ we call $n$ the *loglen* of $p$, which we write $lgl.p$. We use the notation $p_i$ to denote element $i$ in the powerlist $p$, for $0 \leq i < 2^{lgl.p}$.

If we take equality of powerlists to be equivalent to equality of linear lists (that is, element-by-element equality) then it is a straightforward matter to show the definitions of $\langle\cdot\rangle$, $\mid$ and $\bowtie$ satisfy the following axioms.

**Powerlist axioms**

| | |
|---|---|
| (Singleton) | $\langle x\rangle \ = \ \langle y\rangle \ \equiv \ x \ = \ y$ |
| (Pair) | $\langle x\rangle \mid \langle y\rangle \ = \ \langle x\rangle \bowtie \langle y\rangle$ |
| (Tie) | $p \mid q \ = \ r \mid s \ \equiv \ p \ = \ r \ \wedge \ q \ = \ s$ |
| (Zip) | $p \bowtie q \ = \ r \bowtie s \ \equiv \ p \ = \ r \ \wedge \ q \ = \ s$ |
| (Commutativity) | $(p \bowtie q) \mid (r \bowtie s) \ = \ (p \mid r) \bowtie (q \mid s)$ |

More generally, we can define a *powerlist algebra* over $B$ to be a tuple $(P, \langle\cdot\rangle, \mid, \bowtie, lgl)$ such that

- $P$ is a set of powerlists
- $\langle\cdot\rangle$ is a constructor $B \longrightarrow P$
- $\mid$ and $\bowtie$ are constructors $P \times P \longrightarrow P$
- $lgl$ is a function $P \longrightarrow \mathbf{N}$

which satisfy the following rules

- $\langle x\rangle$ is defined for all $x \in B$
- $lgl.p$ is defined for all $p \in P$
- $p \mid q$ and $p \bowtie q$ are defined iff $lgl.p = lgl.q$
- $lgl.p = 0$ iff $p = \langle x\rangle$ for some $x \in B$
- $lgl.(p \mid q) = lgl.(p \bowtie q) = lgl.p + 1$
- $lgl.p > 0$ iff $p = q \mid r$ or $p = q \bowtie r$ for some $q, r \in P$

- $\langle\cdot\rangle$, $|$ and $\bowtie$ satisfy the powerlist axioms

Misra's powerlists are one example of such a powerlist algebra, which we call the *standard model*. Kornerup [4] has another example where $|$ and $\bowtie$ are defined differently, using the Gray-code and inverse Gray-code permutations.

We refer to the standard model to provide an operational interpretation of the functions that we define, though our proofs are entirely within the powerlist algebra and so are valid for all models.

From the axioms we can prove the following.

**Theorem 0 (Dual decomposition)** *For any non-singleton powerlist $p$ there are unique powerlists $r$, $s$, $u$ and $v$ such that $p = r \mid s$ and $p = u \bowtie v$.*

**Note** Proofs omitted from the main body of the paper are in the appendix.

## 2 *last* and the shift operator

As our first example of a powerlist function we define the function *last*. We use a pattern-matching style of function definition. The period represents function application.

**Definition 0**

$$\begin{aligned} last.\langle x\rangle &= x \\ last.(p \mid q) &= last.q \end{aligned}$$

Note that because of Theorem 0, this function is defined on all powerlists: any powerlist is a singleton or is the tie of two powerlists.

The interpretation of *last* in the standard model should be clear: *last* returns the last (rightmost) element of the powerlist. A result of this definition is the following result.

**Lemma 1** $\quad last.(p \bowtie q) = last.q$

We could equally well have chosen to define *last* in terms of $\bowtie$ and proven a result about *last* applied to $p \mid q$.

**Notation** Where convenient we write $\overline{p}$ for $last.p$.

The infix shift operator, written $\rightarrow$ takes a scalar as its left argument and a powerlist as its right argument and returns a powerlist of the same size. In the standard model the intended effect of $x \rightarrow p$ is a powerlist $q$ of the same size as $p$ where $q_0 = x$ and $q_i = p_{i-1}$ for $1 \leq i < 2^{lgl.q}$. So, for example, we expect

$$a \rightarrow \langle e\ f\ g\ h\rangle = \langle a\ e\ f\ g\rangle$$

The following definition realises this description.

**Definition 2**

$$\begin{aligned} x \rightarrow \langle y \rangle &= \langle x \rangle \\ x \rightarrow (p \bowtie q) &= (x \rightarrow q) \bowtie p \end{aligned}$$

We have a result about $\rightarrow$ in terms of $|$.

**Lemma 3** $\quad x \rightarrow (p \mid q) \;=\; (x \rightarrow p) \mid (\overline{p} \rightarrow q)$

## 3 Pointwise operators

We can define powerlist functions in terms of operators on the scalar set $B$. One common way to do this is to coerce operators on $B$ to be operators on powerlists over $B$. The simplest way to do this is so that

$$\langle a\ b\ c\ d \rangle \star \langle e\ f\ g\ h \rangle \quad = \quad \langle a \star e \quad b \star f \quad c \star g \quad d \star h \rangle$$

The definition of this coercion is straightforward.

**Definition 4** *For $\star$ a binary operator on $B$, we coerce $\star$ to be a binary operator on $P$ by*

$$\begin{aligned} \langle x \rangle \star \langle y \rangle &= \langle x \star y \rangle \\ (p \bowtie q) \star (r \bowtie s) &= (p \star r) \bowtie (q \star s) \end{aligned}$$

We call such an operator $\star$ a *pointwise operator* on powerlists. The following result is easily proven.

**Lemma 5**

$$(p \mid q) \star (r \mid s) \quad = \quad (p \star r) \mid (q \star s)$$

We generalize the definition of pointwise operators to $n$-ary operators in the obvious way. Note that $p \star q$ is defined iff $lgl.p = lgl.q$ and that $lgl.(p \star q) = lgl.p$. A simple induction establishes that commutativity, associativity and distributivity properties of binary operators on $B$ are carried over to the corresponding pointwise operators on powerlists. We can also show that

$$\begin{aligned} last.p \star last.q &= last.(p \star q) \\ (x \rightarrow p) \star (y \rightarrow q) &= (x \star y) \rightarrow (p \star q) \end{aligned}$$

We call functions and operators on powerlists which commute with pointwise operators in this way *positional*.

## 4 Shifted operators

For $\star$ a binary operator on $B$, we define *shift* $\star$, written $\vec{\star}$, so that

$$\langle a\ b\ c\ d\rangle \vec{\star} \langle e\ f\ g\ h\rangle \quad = \quad \langle e \quad a\star f \quad b\star g \quad c\star h\rangle$$

The operator $\vec{\star}$ is similar to the pointwise version of the $\star$ operator except that elements from the first powerlist are shifted one place to the right before $\star$ is applied pointwise.

**Definition 6** *For $\star$ a binary operator on $B$, we define $\vec{\star}$ on $P$ by*

$$\begin{aligned}\langle x\rangle \vec{\star} \langle y\rangle &= \langle y\rangle \\ (p \bowtie q)\vec{\star}(r \bowtie s) &= (q \vec{\star} r) \bowtie (p \star s)\end{aligned}$$

Note that, as with the pointwise version of $\star$, $p \vec{\star} q$ is defined iff $lgl.p = lgl.q$ and that $lgl.(p \vec{\star} q) = lgl.p$.

We have the following quasi-associativity results.

**Lemma 7** *For associative $\star$,*

$$\begin{aligned}(p \vec{\star} q) \star r &= p \vec{\star} (q \star r) \\ (p \star q) \vec{\star} r &= p \vec{\star} (q \vec{\star} r)\end{aligned}$$

## 5 Prefix computation

If $p$ is a powerlist over scalar set $B$ and $\star$ is an associative binary operator on $B$ then we define the *prefix computation* of $p$ with respect to $\star$ in the standard model to be a powerlist $w$ of the same size as $p$ such that $w_i = p_0 \star \cdots \star p_i$. So, for example, if $p = \langle a\ b\ c\ d\rangle$, the prefix computation of $p$ with respect to $\star$ is the powerlist

$$\langle a \quad a\star b \quad a\star b\star c \quad a\star b\star c\star d\rangle$$

We can express the conditions on the prefix computation in the following way.

$$\begin{aligned}w_0 &= p_0 \\ w_i &= w_{i-1} \star p_i \quad 1 \le i < 2^{lgl.p}\end{aligned}$$

This is equivalent to the powerlist expression

$$w \quad = \quad w \vec{\star} p$$

We have the following result.

**Lemma 8** *For $\star$ an associative binary operator on $B$, the equation*

$$w \quad = \quad w \vec{\star} p$$

*has a unique solution in $w$ for any $p$.*

This justifies defining of the prefix sum operator as follows.

**Definition 9 (Prefix computation)** *For $\star$ an associative binary operator on $B$, the prefix computation of powerlist $p$ with respect to $\star$, written $\boxed{\star}p$, is defined by*

$$w = \boxed{\star}p \quad \equiv \quad w = w \vec{\star} p$$

For our convenience in defining the carry-lookahead adder we define another operator on powerlists which combines the shift and prefix computation operators.

**Definition 10 (Shift-prefix)** *The shift-prefix computation of $x$ and $p$ with respect to $\star$, written $x \overset{\star}{\mapsto} p$, is defined by*

$$x \overset{\star}{\mapsto} p \quad = \quad \boxed{\star}(x \rightarrow p)$$

The following lemma gives a direct definition of $\overset{\star}{\mapsto}$.

**Lemma 11**

$$\begin{array}{rcl} x \overset{\star}{\mapsto} \langle y \rangle & = & \langle x \rangle \\ x \overset{\star}{\mapsto} (p \mid q) & = & u \mid v \\ & & \text{where} \quad u \; = \; x \overset{\star}{\mapsto} p \\ & & \qquad\quad\; v \; = \; (\overline{u} \star \overline{p}) \overset{\star}{\mapsto} q \end{array}$$

## 6 Using powerlists to describe circuits

The circuit descriptions that we give are algorithmic, in that they indicate which intermediate results the circuit is to calculate and how it is to combine these to reach the desired result. (We do not address the question of translating these algorithms to hardware.) The powerlist notation allows for a compact representation of circuits. The style of definition we use is similar to that used by Hunt and Brock [2], except they used Lisp lists where we use powerlists and they provided mappings down to the hardware level.

The input and output values of circuits are given in terms of values from the set $\{0, 1\}$, which we call *bits*. We regard this set as a subset of the natural numbers, so we allow ordinary arithmetic operations on it. We represent input and output registers as powerlists of bits, which we call *bit vectors*. When we are using registers to represent natural numbers we assume that bits go left to right from low-order to high-order within the corresponding bit vectors.

As an example of the style in which we present circuits, we define two circuits to compute the parity of the bits in a register. This is a special case of a more general problem: given a powerlist $p$ and an associative binary scalar operator $\star$, compute $p_0 \star \ldots \star p_{N-1}$ where $N$ is the length of $p$ (so $N = 2^{lgl.p}$). We call this computation the *reduction* of $p$ with respect to $\star$. Parity is reduction of a bit vector with respect to the operator $\oplus$, which is addition modulo 2 (or, equivalently, logical exclusive-or if we regard 0 and 1 as representing *false* and *true* repectively). As a first (inefficient) way of doing this we consider the function *iterpar* defined by the following.

$$\begin{array}{rcl} iterpar.b.\langle x\rangle & = & b \oplus x \\ iterpar.b.(p \mid q) & = & iterpar.c.q \\ & & \text{where} \quad c \;\; = \;\; iterpar.b.p \end{array}$$

Function *iterpar* takes two arguments, a single bit and a bit vector and returns a single bit representing the parity of all the bits of its arguments. Thus $iterpar.0.p$ returns the parity of $p$. We interpret this function as describing a circuit in which we can identify particular wires as carrying the input, output and intermediate values which are generated as we unwind the recursive calls. The function describes not a single circuit, but a family of circuits, one for each register width $2^n$, for $n \geq 0$. The recursive nature of the definition indicates that we should be able to identify circuits of width $2^{n-1}$ within a circuit of width $2^n$.

It should be clear that *iterpar* describes an iterative circuit which accumulates its result by checking the bits of the input register left to right, checking each bit's parity against that of the previously seen bits. This is a rather unnatural definition to give in the powerlist notation, but it is close to the Lisp-style definitions used by Hunt and Brock.

As an obvious improvement (in both the circuit delay and the number of gates required) we propose the circuit *parapar* which uses concurrent operations on the bits of the input registers.

$$\begin{array}{rcl} parapar.\langle x\rangle & = & x \\ parapar.(p \bowtie q) & = & parapar.r \\ & & \text{where} \quad r \;\; = \;\; p \oplus q \end{array}$$

Function *parapar* takes as input a single bit vector and uses a tree computation to calculate the result. Here again we regard this as describing a circuit in which we can identify wires holding the intermediate results used. As with *iterpar*, *parapar* describes a family of circuits of width $2^n$. Though (we claim) $iterpar.0$ (the *iterpar* circuit with the initial input bit fixed at 0) and *parapar* define the same function, when we interpret their definitions as circuits we see that they describe different circuits.

To verify these circuits, we first need a way to state their correctness, so we must define the parity of a powerlist. From what we have previously defined we could use the following.

$$parity.p \quad = \quad last.\boxed{\oplus}p$$

The verification task is then to prove that $iterpar.0$ and *parapar* are both functionally the same as *parity*. Since to prove the three equalities

$$iterpar.0 \quad = \quad parity$$

$$\begin{aligned} parapar &= parity \\ iterpar.0 &= parapar \end{aligned}$$

we need only prove any two of them, we are free to choose which two to prove. Use this freedom when, rather than attempting to prove directly that the carry-lookahead adder correctly implements addition, we prove this by showing it equivalent to the ripple-carry adder, whose correctness is easier to establish directly.

## 7 The ripple-carry adder

The ripple-carry adder is the simplest of the adder circuits to describe and explain. The basic element of the circuit is the *full-adder*. This is circuit element with three bit inputs, $x$, $y$ and $b$ ($x$ and $y$ are the *local inputs* and $b$ is the *carry-in*) and two bit outputs, $z$ and $c$ ($z$ is the *local output* and $c$ is the *carry-out*). The operation of the full-adder is illustrated in figure 0. The output $z$ is 1 if one or three of the input bits are 1 and is 0 otherwise; the output $c$ is 1 if two or more of the input bits are 1 and is 0 otherwise (here $\div$ represents integer division).

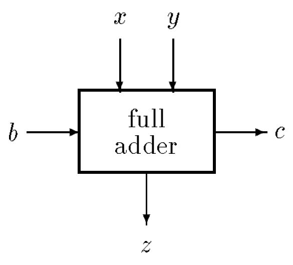
The idea is that $x$ and $y$ are bit values at corresponding positions in two registers, and $b$ is the carry generated by adding bits at lower order positions; the result of the local addition is the bit $z$ which represents the bit in the output register at the position corresponding to $x$ and $y$, and a carry bit $c$ which is passed on to the higher order bits. We regard a full-adder as a ripple-carry adder of width 1. We join two ripple-carry adders of width $2^n$ to give one of width $2^{n+1}$ as shown in figure 1. In the figure the input and output registers are shown with their least significant bits on the left.

The function $\mathsf{rc}$ takes as its inputs a single bit $b$ representing the carry-in and equal-length bit vectors $p$ and $q$ representing the input registers. The output of $\mathsf{rc}$ is a pair $(t, d)$ where $t$ is a bit vector of the same size as the input bit vectors representing the output register and $d$ is a single bit representing the carry-out (overflow).

**Definition 12 (Ripple-carry adder)**

$$\begin{aligned} \mathsf{rc}.b.\langle x\rangle.\langle y\rangle &= (\langle (x+y+b) \bmod 2\rangle, (x+y+b) \div 2) \\ \mathsf{rc}.b.(p \mid q).(r \mid s) &= (t, d) \\ &\quad \text{where } \begin{aligned}[t] t &= u \mid v \\ (u, c) &= \mathsf{rc}.b.p.r \\ (v, d) &= \mathsf{rc}.c.q.s \end{aligned} \end{aligned}$$

We show in Appendix A.5 that the ripple-carry adder defined in this way correctly implements addition.



$$z = (x + y + b) \bmod 2$$
$$c = (x + y + b) \div 2$$

Figure 0: The full-adder circuit element

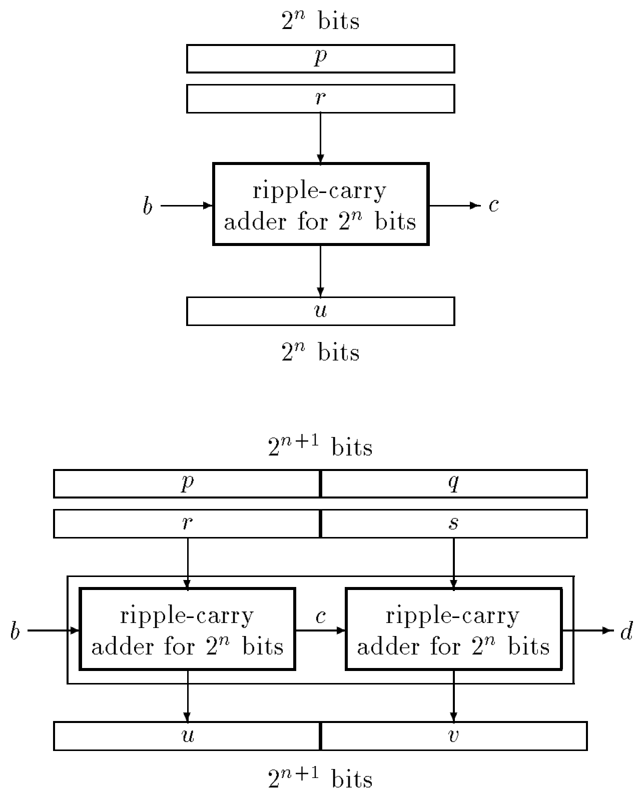

2^n bits
p
r
b
ripple-carry adder for 2^n bits
c
u
2^n bits
2^{n+1} bits
p
q
r
s
b
ripple-carry adder for 2^n bits
c
ripple-carry adder for 2^n bits
d
u
v
2^{n+1} bits

Figure 1: Ripple-carry adders of widths $2^n$ and $2^{n+1}$

It is well known that the ripple-carry adder is not the most efficient circuit for adding natural numbers represented as bit vectors. We next describe the carry-lookahead adder which is known to be more efficient. However, the correctness of the carry-lookahead adder is not nearly as apparent as the correctness of the ripple-carry adder, but using the powerlist algebra we prove that the two circuits compute the same function.

## 8 The carry-lookahead adder

The inefficiency of the ripple-carry adder circuit arises from a linear data dependence between the computations at each bit position: the computation at any position requires as one of its inputs a carry-in, which is the carry-out from the position to its left (the carry bits are said to *ripple* from left to right, which gives the circuit its name). The strategy used in the carry-lookahead adder is to first calculate the carry-in to each position using a prefix computation (this can be done in time logarithmic in the width of the input registers). Once this has been done the local result for all bit positions can be calculated concurrently using only local computations.

To explain the carry-lookahead adder we first note that in the ripple-carry adder, if the input bits $p_i$ and $q_i$ at position $i$ are both 1 then the carry-out from that position is 1 regardless of the carry-in (in this case we say position $i$ *generates* a carry). Similarly, if $p_i$ and $q_i$ are both 0 then the carry-out from position $i$ is 0 regardless of the carry-in (position $i$ *stops* a carry). If one of $p_i$ and $q_i$ is 1 and the other is 0 then the sum of these bits is 1 and the carry-out from position $i$ is the same as the carry-in (position $i$ *propagates* a carry).

We define a binary operator $\bullet$ to calculate whether two bits at a given position generate, stop or propagate a carry.

**Definition 13**

$$x \bullet y \;=\; \begin{cases} x & \text{if} \quad x = y \\ \pi & \text{if} \quad x \neq y \end{cases}$$

Here outputs 1 and 0 represent generate and stop respectively (these are given by the appropriate carry value) and output $\pi$ represents propagate. Pointwise application of $\bullet$ to input bit vectors $p$ and $q$ gives us the initial carry-out vector.

We use the example in figure 2 as we continue the explanation of the operation of the carry-lookahead adder. The example and the new operators introduced in it are explained in the following paragraphs.

The first three lines are the input bit vectors $p$ and $q$ and the carry-in bit $b$. The next line shows the initial carry-out vector $r$ which, as explained above, is given by $p \bullet q$.

The initial carry-in values for each bit position are now given by the vector $b \rightarrow r$, which is shown in the fifth line. The value $\overline{r}$ is shifted out of the vector by

|  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $p$ |  | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |  |  |
| $q$ |  | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |  |  |
| $b$ | 0 |  |  |  |  |  |  |  |  |  |  |
| $r$ |  | 0 | $\pi$ | 1 | $\pi$ | $\pi$ | 0 | 1 | $\pi$ |  | $p \bullet q$ |
| $b \longrightarrow r$ |  | 0 | 0 | $\pi$ | 1 | $\pi$ | $\pi$ | 0 | 1 |  |  |
| $\overline{r}$ |  |  |  |  |  |  |  |  |  | $\pi$ |  |
| $s$ |  | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |  | $b \overset{\star}{\mapsto} r$ |
| $t$ |  | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |  | $s \odot r$ |
| $d$ |  |  |  |  |  |  |  |  |  | 1 | $\overline{s} \star \overline{r}$ |

Figure 2: Example showing the stages in calculating the outputs $t$ and $d$ using carry-lookahead for inputs $p$, $q$ and $b$

this operation. From the above remarks we can conclude that a position that has a non-$\pi$ value in $b \longrightarrow r$ has the correct carry-in value. Any position that has a $\pi$ carry-in value gets the rightmost non-$\pi$ carry-in value to its left as its final carry-in value. We can represent the computation of the final carry-in values $s$ as a prefix computation with respect to the binary operator $\star$ defined as follows.

**Definition 14**

$$x \star y \;=\; \begin{cases} y & \text{if} \quad y \neq \pi \\ x & \text{if} \quad y = \pi \end{cases}$$

Note that $\star$ is associative, so prefix computation with respect to $\star$ is well-defined. The value of $s$ is thus $\boxed{\star}(b \longrightarrow r)$ or $b \overset{\star}{\mapsto} r$.

To compute the vector of local outputs $t$ we note that if $r_i \neq \pi$ then the contribution of $p_i$ and $q_i$ to $t_i$ is 0, so $t_i$ is the value of the carry-in $s_i$. If $r_i = \pi$ then the contribution of $p_i$ and $q_i$ to $t_i$ is 1, so the local result is the inverse of $s_i$. We define a binary operator $\odot$ which gives this result when applied pointwise to $s$ and $r$.

**Definition 15**

$$\begin{array}{rcl} x \odot y & = & \begin{cases} x & \text{if} \quad y \neq \pi \\ \neg x & \text{if} \quad y = \pi \end{cases} \\ & & \begin{array}{rrcl} \text{where} & \neg 0 & = & 1 \\ & \neg 1 & = & 0 \\ & \neg \pi & = & \pi \end{array} \end{array}$$

The value of $t$ is thus $s \odot r$.

To determine the carry-out $d$ of this operation we notice that $d$ is the value of the carry-out from the the rightmost position of the bit vectors, or alternatively the carry-in to a position just to the right of the rightmost position. If we consider the value that a prefix computation for carry-in bits including this position would yield we see that $d$ is given by $\overline{s} \star \overline{r}$.

Combining the informal arguments above we now propose the following definition of an addition circuit. We use the same input and output formats for $\mathsf{cl}$ as we used for $\mathsf{rc}$.

**Definition 16 (Carry-lookahead adder)**

$$\begin{array}{rcllll} \mathsf{cl}.b.p.q & = & (t, d) & & & \\ & & \text{where} & t & = & s \odot r \\ & & & d & = & \overline{s} \star \overline{r} \\ & & & r & = & p \bullet q \\ & & & s & = & b \overset{\star}{\mapsto} r \end{array}$$

It can be seen from this definition that all operations used in the calculation of the results of the carry-lookahead adder, apart from the shift-prefix computation used to evaluate $s$, are pointwise (and thus can be performed for all bit positions in parallel). Shift-prefix can be evaluated on input registers of width $2^n$ in time proportional to $n$. Hence the overall running time is logarithmic in the width of the input registers. In contrast, the running time for the ripple-carry adder circuit is linear in the width of the input registers.

**Note** The function $\mathsf{cl}$ is defined for $b \in \{0, 1, \pi\}$ and for $p$ and $q$ powerlists over the same set. However, in the theorem below we restrict the inputs to come from the set $\{0, 1\}$. One consequence of the theorem is that in this case the output of $\mathsf{cl}$ does not contain the value $\pi$ (since $\mathsf{rc}$ cannot generate such a value in its output).

**Theorem 1 (Equivalence of the adder circuits)** *For all $p$, $q$, $b$, where $p$ and $q$ are bit vectors of equal size and $b \in \{0, 1\}$,*

$$\mathsf{rc}.b.p.q \quad = \quad \mathsf{cl}.b.p.q$$

**Proof** We show that $\mathsf{cl}$ satisfies the equations defining $\mathsf{rc}$. That is, we use Definition 16 to calculate $\mathsf{cl}.b.p.q$ for singleton $p$ and $q$ and for non-singleton $p$ and $q$ and we show in each case that the result can be put in the form of Definition 12. The result then follows by a simple structural induction.

For $\mathsf{cl}.b.\langle x\rangle.\langle y\rangle$ we get the following. The format of the proofs is explained in Appendix A.0.

$$\begin{array}{ll} & r \\ = & \qquad \{ \text{ Definition 16 } \} \\ & \langle x\rangle \bullet \langle y\rangle \\ = & \qquad \{ \bullet \text{ pointwise } \} \\ & \langle x \bullet y\rangle \end{array}$$

$$\begin{array}{ll}
 & s \\
= & \quad \{\ \text{Definition 16}\ \} \\
 & b \overset{\star}{\mapsto} r \\
= & \quad \{\ \text{value of } r\text{, above}\ \} \\
 & b \overset{\star}{\mapsto} \langle x \bullet y \rangle \\
= & \quad \{\ \text{Lemma 11}\ \} \\
 & \langle b \rangle
\end{array}$$

$$\begin{array}{ll}
 & t \\
= & \quad \{\ \text{Definition 16}\ \} \\
 & s \odot r \\
= & \quad \{\ \text{values of } s \text{ and } r\text{, above}\ \} \\
 & \langle b \rangle \odot \langle x \bullet y \rangle \\
= & \quad \{\ \odot \text{ pointwise}\ \} \\
 & \langle b \odot (x \bullet y) \rangle
\end{array}$$

$$\begin{array}{ll}
 & d \\
= & \quad \{\ \text{Definition 16}\ \} \\
 & \overline{s} \star \overline{r} \\
= & \quad \{\ \text{values of } s \text{ and } r\text{, above}\ \} \\
 & \mathit{last}.\langle b \rangle \star \mathit{last}.\langle x \bullet y \rangle \\
= & \quad \{\ \text{Definition 0, twice}\ \} \\
 & b \star (x \bullet y)
\end{array}$$

Since we can show

$$\begin{array}{rcl}
b \odot (x \bullet y) & = & (x + y + b) \bmod 2 \\
b \star (x \bullet y) & = & (x + y + b) \div 2
\end{array}$$

for all $x$, $y$ and $b$ taking values from $\{0, 1\}$, we have

$$\mathsf{cl}.b.\langle x \rangle.\langle y \rangle \quad = \quad (\langle (x + y + b) \bmod 2 \rangle, (x + y + b) \div 2)$$

which matches the base case of Definition 12.

We now need to express $\mathsf{cl}.b.(p \mid q).(p' \mid q')$ in terms of $\mathsf{cl}.b_p.p.p'$ and $\mathsf{cl}.b_q.q.q'$ for appropriate values of $b_p$ and $b_q$ to match the recursive case of Definition 12. We have the following from Definition 16.

$$\begin{array}{lrcl}
 & \mathsf{cl}.b.(p \mid q).(p' \mid q') & = & (t, d) \\
 & & & \text{where} \quad \begin{array}{rcl} t & = & s \odot r \\ d & = & \overline{s} \star \overline{r} \\ r & = & (p \mid q) \bullet (p' \mid q') \\ s & = & b \overset{\star}{\mapsto} r \end{array}
\end{array} \qquad (0)$$

$$(1)\qquad \begin{array}{rcl} \mathsf{cl}.b_p.p.p' & = & (t_p, d_p) \\ & & \text{where } \begin{array}{rcl} t_p & = & s_p \odot r_p \\ d_p & = & \overline{s_p} \star \overline{r_p} \\ r_p & = & p \bullet p' \\ s_p & = & b_p \overset{\star}{\mapsto} r_p \end{array} \end{array}$$

$$(2)\qquad \begin{array}{rcl} \mathsf{cl}.b_q.q.q' & = & (t_q, d_q) \\ & & \text{where } \begin{array}{rcl} t_q & = & s_q \odot r_q \\ d_q & = & \overline{s_q} \star \overline{r_q} \\ r_q & = & q \bullet q' \\ s_q & = & b_q \overset{\star}{\mapsto} r_q \end{array} \end{array}$$

We now express the intermediate results for $\mathsf{cl}.b.(p \mid q).(p' \mid q')$ in terms of the intermediate results for $\mathsf{cl}.b_p.p.p'$ and $\mathsf{cl}.b_q.q.q'$. In doing so we choose appropriate values for $b_p$ and $b_q$.

$$\begin{array}{cl} & r \\ = & \quad \{ \text{ (0) } \} \\ & (p \mid q) \bullet (p' \mid q') \\ = & \quad \{ \bullet \text{ pointwise } \} \\ & (p \bullet p') \mid (q \bullet q') \\ = & \quad \{ \text{ (1) ; (2) } \} \\ & r_p \mid r_q \end{array}$$

$$\begin{array}{cl} & s \\ = & \quad \{ \text{ (0) } \} \\ & b \overset{\star}{\mapsto} r \\ = & \quad \{ \text{ value of } r\text{, above } \} \\ & b \overset{\star}{\mapsto} (r_p \mid r_q) \end{array}$$

Using Lemma 11 we calculate $u$ and $v$ so that $s = u \mid v$.

$$\begin{array}{cl} & u \\ = & \quad \{ \text{ Lemma 11 } \} \\ & b \overset{\star}{\mapsto} r_p \\ = & \quad \{ \text{ choose } b_p = b\text{, (1) } \} \\ & s_p \end{array}$$

$$\begin{array}{cl} & v \\ = & \quad \{ \text{ Lemma 11 } \} \\ & (\overline{u} \star \overline{r_p}) \overset{\star}{\mapsto} r_q \\ = & \quad \{ \text{ value of } u\text{, above } \} \end{array}$$

$$\begin{array}{ll}
 & (\overline{s_p} \star \overline{r_p}) \overset{\star}{\mapsto} r_q \\
= & \qquad \{ \ (1) \ \} \\
 & d_p \overset{\star}{\mapsto} r_q \\
= & \qquad \{ \ \text{choose } b_q = d_p, \ (2) \ \} \\
 & s_q
\end{array}$$

Thus if we choose $b_p = b$ and $b_q = d_p$ we get $s = s_p \mid s_q$.

$$\begin{array}{ll}
 & t \\
= & \qquad \{ \ (0) \ \} \\
 & s \odot r \\
= & \qquad \{ \ \text{values of } s \text{ and } r, \text{ above} \ \} \\
 & (s_p \mid s_q) \odot (r_p \mid r_q) \\
= & \qquad \{ \ \odot \text{ pointwise} \ \} \\
 & (s_p \odot r_p) \mid (s_q \odot r_q) \\
= & \qquad \{ \ (1); \ (2) \ \} \\
 & t_p \mid t_q
\end{array}$$

$$\begin{array}{ll}
 & d \\
= & \qquad \{ \ (0) \ \} \\
 & \overline{s} \star \overline{r} \\
= & \qquad \{ \ \text{values of } s \text{ and } r, \text{ above} \ \} \\
 & \mathit{last}.(s_p \mid s_q) \star \mathit{last}.(r_p \mid r_q) \\
= & \qquad \{ \ \text{Definition 0, twice} \ \} \\
 & \overline{s_q} \star \overline{r_q} \\
= & \qquad \{ \ (2) \ \} \\
 & d_q
\end{array}$$

Combining the above results we get

$$\begin{array}{rcllcl}
\mathsf{cl}.b.(p \mid q).(p' \mid q') & = & (t, d_q) & & & \\
 & & \text{where} & t & = & t_p \mid t_q \\
 & & & (t_p, d_p) & = & \mathsf{cl}.b.p.p' \\
 & & & (t_q, d_q) & = & \mathsf{cl}.d_p.q.q'
\end{array}$$

which is precisely the form of the recursive case of Definition 16.
(End of proof)

## 9 Conclusions

Using powerlists we have shown that two adder circuits with dissimilar algorithms, one with running time linear in the input register width and the other with logarithmic running time, compute the same input-output function. This result is, of course, well known. However, our proof of this result (Theorem 1) is surprisingly compact. We believe this compactness is a strong argument in favour of using the powerlist notation for describing and verifying circuits.

A further result is that the prefix computation can be expressed as the unique solution of a fixpoint equation, and that we can derive its properties from this equation. Since the prefix computation plays such and important rôle in digital design, we expect the results in Appendix A.4 leading to the proof of Lemma 11 to be of use as we attempt to specify and verify other circuits.

We plan to further explore the use of powerlists for expressing and verifying digital circuits in future work.

### Acknowledgements

Jay Misra suggested that I look at the possibilities for the use of his powerlist notation for circuit specification and verification. This work was completed thanks to Jay's continual support and encouragement. Other colleagues at UT Austin, notably Al Carruth, Warren Hunt and Jacob Kornerup, made several suggestions for improving the presentation of the material.

## A Appendix: additional proofs

### A.0 Proof format

Where we give calculational proofs we use the format from Dijkstra and Scholten [0]. So, for example, the proof of the identity

$$(\langle a\rangle \bowtie \langle b\rangle) \bowtie (\langle c\rangle \bowtie \langle d\rangle) \quad = \quad (\langle a\rangle \mid \langle c\rangle) \mid (\langle b\rangle \mid \langle d\rangle)$$

is presented as a chain of equalities in the following format.

$$\begin{array}{ll} & (\langle a\rangle \bowtie \langle b\rangle) \bowtie (\langle c\rangle \bowtie \langle d\rangle) \\ = & \qquad \{\text{ Pair axiom, twice }\} \\ & (\langle a\rangle \mid \langle b\rangle) \bowtie (\langle c\rangle \mid \langle d\rangle) \\ = & \qquad \{\text{ Commutativity axiom }\} \\ & (\langle a\rangle \bowtie \langle c\rangle) \mid (\langle b\rangle \bowtie \langle d\rangle) \\ = & \qquad \{\text{ Pair axiom, twice }\} \\ & (\langle a\rangle \mid \langle c\rangle) \mid (\langle b\rangle \mid \langle d\rangle) \end{array}$$

The lines beginning with equality symbols contain justifications (in curly braces) for the claimed equality between the term above and the term below.

### A.1 Dual decomposition

**Theorem 0 (Dual decomposition)** *For any non-singleton powerlist $p$ there are unique powerlists $r$, $s$, $u$ and $v$ such that $p = r \mid s$ and $p = u \bowtie v$.*

**Proof** We note that the uniqueness of $r$, $s$, $u$ and $v$ is guaranteed by the Tie and Zip axioms and that, from powerlist algebra rules, $p$ is not a singleton iff $p = r \mid s$ for some $r$ and $s$ or $p = u \bowtie v$ for some $u$ and $v$. We assume that $p$ is non-singleton and $p = r \mid s$ and show that $p = u \bowtie v$ for some $u$ and $v$. A corresponding proof shows the converse. The proof is by induction of the structure of $r$ and $s$.

Basis: $r$ and $s$ are singletons.

$$\begin{array}{ll} & p \\ = & \qquad \{\ \text{given}\ \} \\ & r \mid s \\ = & \qquad \{\ r \text{ and } s \text{ singletons, Pair axiom}\ \} \\ & r \bowtie s \end{array}$$

So if $u = r$ and $v = s$ then $p = u \bowtie v$.

Induction step: $r$ and $s$ are non-singleton. We assume as the induction hypothesis that $r = r' \bowtie r''$ and $s = s' \bowtie s''$ for some $r'$, $r''$, $s'$ and $s''$.

$$\begin{array}{ll} & p \\ = & \qquad \{\ \text{given}\ \} \\ & r \mid s \\ = & \qquad \{\ \text{induction hypothesis}\ \} \\ & (r' \bowtie r'') \mid (s' \bowtie s'') \\ = & \qquad \{\ \text{Commutativity axiom}\ \} \\ & (r' \mid s') \bowtie (r'' \mid s'') \end{array}$$

So if $u = r' \mid s'$ and $v = r'' \mid s''$ then $p = u \bowtie v$.
(End of proof)

This proof makes use of the Pair and Commutativity axioms, which are the only axioms that involve both $\mid$ and $\bowtie$. The form of this proof is one that arises repeatedly in proving results about powerlists.

## A.2 Results for *last* and the shift operator

We give first a couple of results which are useful in the bases of inductive proofs of results involving $\rightarrow$ and *last*. The proofs come directly from Definitions 2 and 0.

**Lemma 17** *For any scalar $x$ and for any singletons $p$ and $q$,*

$$\begin{array}{rcl} x \rightarrow p & = & \langle x \rangle \\ \overline{p} \rightarrow q & = & p \end{array}$$

Lemmas 1 and 3 use the same form of induction as used above to prove Theorem 0. We show only the proof of Lemma 3.

**Lemma 3** $\quad x \rightarrow (p \mid q) \; = \; (x \rightarrow p) \mid (\overline{p} \rightarrow q)$

**Proof** We use induction of the structure of the right-hand argument to $\rightarrow$.
Basis: $p$ and $q$ are singletons.

$$\begin{array}{ll}
 & x \rightarrow (p \mid q) \\
= & \quad \{ \text{ Pair axiom } \} \\
 & x \rightarrow (p \bowtie q) \\
= & \quad \{ \text{ Definition 2 } \} \\
 & (x \rightarrow q) \bowtie p \\
= & \quad \{ \text{ Lemma 17 } \} \\
 & \langle x \rangle \bowtie p \\
= & \quad \{ \text{ Pair axiom } \} \\
 & \langle x \rangle \mid p \\
= & \quad \{ \text{ Lemma 17, twice } \} \\
 & (x \rightarrow p) \mid (\overline{p} \rightarrow q)
\end{array}$$

Induction step: $p$ and $q$ are non-singleton. We assume $p = r \bowtie s$ and $q = u \bowtie v$ and, as the induction hypothesis, that $x \rightarrow (s \mid v) = (x \rightarrow s) \mid (\overline{s} \rightarrow v)$.

$$\begin{array}{ll}
 & x \rightarrow (p \mid q) \\
= & \quad \{ \text{ value of } p \text{ and } q \ \} \\
 & x \rightarrow ((r \bowtie s) \mid (u \bowtie v)) \\
= & \quad \{ \text{ Commutativity axiom } \} \\
 & x \rightarrow ((r \mid u) \bowtie (s \mid v)) \\
= & \quad \{ \text{ Definition 2 } \} \\
 & (x \rightarrow (s \mid v)) \bowtie (r \mid u) \\
= & \quad \{ \text{ induction hypothesis } \} \\
 & ((x \rightarrow s) \mid (\overline{s} \rightarrow v)) \bowtie (r \mid u) \\
= & \quad \{ \text{ Commutativity axiom } \} \\
 & ((x \rightarrow s) \bowtie r) \mid ((\overline{s} \rightarrow v) \bowtie u) \\
= & \quad \{ \text{ Definition 2, twice } \} \\
 & (x \rightarrow (r \bowtie s)) \mid (\overline{s} \rightarrow (u \bowtie v)) \\
= & \quad \{ \text{ Lemma 1 } \} \\
 & (x \rightarrow (r \bowtie s)) \mid (\mathit{last}.(r \bowtie s) \rightarrow (u \bowtie v)) \\
= & \quad \{ \text{ value of } p \text{ and } q \ \} \\
 & (x \rightarrow p) \mid (\overline{p} \rightarrow q)
\end{array}$$

(End of proof)

## A.3 Results for shifted operators

We first prove a result giving $\vec{\star}$ in terms of $\mid$.

**Lemma 18**

$$(p \mid q) \,\vec{\star}\, (r \mid s) \quad = \quad (p \,\vec{\star}\, r) \mid ((\overline{p} \rightarrow q) \star s)$$

**Proof** We use induction on the arguments.

Basis: $p$, $q$, $r$, $s$ are singletons.

$$\begin{array}{ll}
 & (p \mid q) \vec{\star} (r \mid s) \\
= & \quad \{ \text{ Pair axiom, twice } \} \\
 & (p \bowtie q) \vec{\star} (r \bowtie s) \\
= & \quad \{ \text{ Definition 6 } \} \\
 & (q \vec{\star} r) \bowtie (p \star s) \\
= & \quad \{ \text{ Definition 6 } \} \\
 & r \bowtie (p \star s) \\
= & \quad \{ \text{ Pair axiom } \} \\
 & r \mid (p \star s) \\
= & \quad \{ \text{ Definition 6; Lemma 17 } \} \\
 & (p \vec{\star} r) \mid ((\overline{p} \rightarrow q) \star s)
\end{array}$$

Induction step: $p$, $q$, $r$, $s$ are non-singleton. We assume that $p = p' \bowtie p''$, $q = q' \bowtie q''$, $r = r' \bowtie r''$, $s = s' \bowtie s''$ and, as the induction hypothesis,

$$(p'' \mid q'') \vec{\star} (r' \mid s') \quad = \quad (p'' \vec{\star} r') \mid ((\overline{p''} \rightarrow q'') \star s')$$

We then have

$$\begin{array}{ll}
 & (p \mid q) \vec{\star} (r \mid s) \\
= & \quad \{ \text{ value of } p,\ q,\ r,\ s \ \} \\
 & ((p' \bowtie p'') \mid (q' \bowtie q'')) \vec{\star} ((r' \bowtie r'') \mid (s' \bowtie s'')) \\
= & \quad \{ \text{ Commutativity axiom, twice } \} \\
 & ((p' \mid q') \bowtie (p'' \mid q'')) \vec{\star} ((r' \mid s') \bowtie (r'' \mid s'')) \\
= & \quad \{ \text{ Definition 6 } \} \\
 & ((p'' \mid q'') \vec{\star} (r' \mid s')) \bowtie ((p' \mid q') \star (r'' \mid s'')) \\
= & \quad \{ \text{ induction hypothesis; } \star \text{ pointwise } \} \\
 & ((p'' \vec{\star} r') \mid ((\overline{p''} \rightarrow q'') \star s')) \bowtie ((p' \star r'') \mid (q' \star s'')) \\
= & \quad \{ \text{ Commutativity axiom } \} \\
 & ((p'' \vec{\star} r') \bowtie (p' \star r'')) \mid (((\overline{p''} \rightarrow q'') \star s') \bowtie (q' \star s'')) \\
= & \quad \{ \text{ Definition 6; } \star \text{ pointwise } \} \\
 & ((p' \bowtie p'') \vec{\star} (r' \bowtie r'')) \mid (((\overline{p''} \rightarrow q'') \bowtie q') \star (s' \bowtie s'')) \\
= & \quad \{ \text{ Definition 2; value of } p,\ r,\ s \ \} \\
 & (p \vec{\star} r) \mid ((\overline{p''} \rightarrow (q' \bowtie q'')) \star s) \\
= & \quad \{ \text{ Lemma 1; value of q } \} \\
 & (p \vec{\star} r) \mid ((\mathit{last}.(p' \bowtie p'') \rightarrow q) \star s) \\
= & \quad \{ \text{ value of } p \ \} \\
 & (p \vec{\star} r) \mid ((\overline{p} \rightarrow q) \star s)
\end{array}$$

(End of proof)

The following lemma further illustrates the close connection between $\vec{\star}$ and $\rightarrow$. The proof is a straightforward induction and is omitted.

**Lemma 19** $\quad p \vec{\star} (x \rightarrow q) = x \rightarrow (p \star q)$

Proof for Lemma 7 is also a straightforward induction and is likewise omitted.

## A.4 Results for prefix computation and shift-prefix

The following lemma was used to justify the use of a fixpoint equation in Definition 9.

**Lemma 8** *For $\star$ an associative binary operator on $B$, the equation*

$$w \quad = \quad w \vec{\star} p$$

*has a unique solution in $w$ for any $p$.*

**Proof** We use structural induction on $p$. Note that, from the definition of $\vec{\star}$, any solution in $w$ to the equation $w = w \vec{\star} p$ must be such that $lgl.w = lgl.p$.

Basis: $p$ is a singleton. From Definition 6, for any singleton $w$

$$w \vec{\star} p \quad = \quad p$$

So we have

$$w = w \vec{\star} p \quad \equiv \quad w = p$$

and thus $w$ is uniquely specified.

Induction step: $p$ is non-singleton, say $p = r \bowtie s$. We assume that for any $t$ such that $lgl.t = lgl.r$ the fixpoint equation

$$w' \quad = \quad w' \vec{\star} t$$

has a unique solution in $w'$. Since any solution to the fixpoint equation for $w$ must be non-singleton, we can write such a solution as $u \bowtie v$. We now calculate conditions on $u$ and $v$.

$$\begin{array}{ll}
 & u \bowtie v \\
= & \qquad \{ \text{ fixpoint equation for } w \text{ } \} \\
 & (u \bowtie v) \vec{\star} p \\
= & \qquad \{ \text{ value of } p \text{ } \} \\
 & (u \bowtie v) \vec{\star} (r \bowtie s) \\
= & \qquad \{ \text{ Definition 6 } \} \\
 & (v \vec{\star} r) \bowtie (u \star s)
\end{array}$$

So from the Zip axiom

(3) $$u \quad = \quad v \vec{\star} r$$

(4) $$v \quad = \quad u \star s$$

This gives

$$\begin{array}{ll} & u \\ = & \quad \{ \text{ Substitute from (4) in (3) } \} \\ & (u \star s) \vec{\star} r \\ = & \quad \{ \text{ Lemma 7, } \star \text{ associative } \} \\ & u \vec{\star} (s \vec{\star} r) \end{array}$$

Since $lgl.(s \vec{\star} r) = lgl.r$, from the induction hypothesis there is a unique $u$ satisfying the equation

$$u \quad = \quad u \vec{\star} (s \vec{\star} r)$$

Using this value of $u$, (4) thus gives a unique $v$ and so $w$ is uniquely determined. (End of proof)

For an operator $\star$ having a left identity $l$ (so $l \star x = x$ for all $x \in B$), we can show from the definitions that

$$p \vec{\star} q \quad = \quad (l \rightarrow p) \star q$$

We can thus regard the following lemma as a generalization of Lemma 8.

**Lemma 20** *For $\star$ a binary operator on $B$, the equation*

$$w \quad = \quad (x \rightarrow w) \star p$$

*has a unique solution in $w$ for any $x$ and $p$.*

**Proof** We use structural induction on $p$. Note that from the definition of a pointwise operator, any solution $w$ to the equation must be such that $llen.w = llen.p$.
Basis: $p$ is a singleton. For singleton $w$ we have, from Lemma 17,

$$w = (x \rightarrow w) \star p \quad \equiv \quad w = \langle x \rangle \star p$$

so there is a unique solution in $w$ to the equation.
Induction step: $p$ is non-singleton, say $p = r \mid s$. We assume that for any $x$ the fixpoint equations

$$\begin{array}{rcl} w' & = & (x \rightarrow w') \star r \\ w'' & = & (x \rightarrow w'') \star s \end{array}$$

have unique solutions in $w'$ and $w''$. Since any solution to the fixpoint equation for $w$ must be non-singleton, we can write such a solution as $u \mid v$. We now calculate conditions on $u$ and $v$.

$$
\begin{array}{ll}
 & u \mid v \\
= & \quad \{ \text{ fixpoint equation for } w \ \} \\
 & (x \rightarrow (u \mid v)) \star p \\
= & \quad \{ \text{ Lemma 3; value of } p \ \} \\
 & ((x \rightarrow u) \mid (\overline{u} \rightarrow v)) \star (r \mid s) \\
= & \quad \{ \ \star \text{ pointwise } \} \\
 & ((x \rightarrow u) \star r) \mid ((\overline{u} \rightarrow v) \star s)
\end{array}
$$

So from the Tie axiom

$$u \ = \ (x \rightarrow u) \star r \tag{5}$$

$$v \ = \ (\overline{u} \rightarrow v) \star s \tag{6}$$

From the induction hypothesis (5) has a unique solution in $u$. Using this solution, the induction hypothesis gives us that (6) has a unique solution in $v$, so $w$ is uniquely determined.
(End of proof)

**Note** In Lemma 20 we do not require that the operator $\star$ be associative and the proof does not require this. It is possible to prove Lemma 8 using $\mid$ instead of $\bowtie$ as the constructor in the induction step and appealing to Lemmas 18 and 20. We could thus remove the restriction that $\star$ be associative from Lemma 8. We choose not to do this since the main purpose of Lemma 8 is to establish the validity of Definition 9, and prefix computation as it is normally understood is only defined with respect to associative operators.

Our next goal is to prove Lemma 11. We first need to prove some results about the prefix computation.

We use the notation $[x\,\star]$ for the unary function $(\lambda y :: x{\star}y)$ on elements of $B$ (so $[x\,\star].y = x \star y$). This function applies pointwise to powerlists. The following lemma (and its generalization to the case when $z$ is replaced by a powerlist) is easily proven.

**Lemma 21** *For associative* $\star$*,*

$$[[x\,\star].y\,\star].z \quad = \quad [x\,\star].([y\,\star].z)$$

**Note** Throughout this section all prefix computations and shift-prefix computations are with respect to $\star$, a given associative binary operator, so we omit this operator from our notation and write $\sqcup p$ and $x \mapsto p$ for $\boxed{\star}p$ and $x \overset{\star}{\mapsto} p$.

The first result for the prefix computation operator is an alternative way of defining it.

**Lemma 22**

$$
\begin{array}{rcl}
\sqcup\langle x\rangle & = & \langle x\rangle \\
\sqcup(p \mid q) & = & u \mid v \\
 & & \text{where} \quad u \ = \ \sqcup p \\
 & & \qquad\quad\;\; v \ = \ [\overline{u}\,\star].\sqcup q
\end{array}
$$

We prove this lemma in conjuction with the following one, which we can regard as a generalization of Definition 9.

**Lemma 23** *For associative $\star$,*

$$w = (x \rightharpoondown w) \star p \quad \equiv \quad w = [x\, \star].\bigsqcup p$$

**Proof** We prove the lemmas together with a single structural induction. Note that for Lemma 23 it is sufficient to prove

$$w = [x\, \star].\bigsqcup p \quad \Rightarrow \quad w = (x \rightharpoondown w) \star p$$

since the converse then follows from Lemma 20.

Basis: We need to show

$$\begin{array}{llcl} (7) & \bigsqcup\langle x\rangle & = & \langle x\rangle \\ (8) & w = [x\, \star].\bigsqcup\langle y\rangle & \Rightarrow & w = (x \rightharpoondown w) \star \langle y\rangle \end{array}$$

Proof of (7): Follows immediately from Definitions 9 and 6.

Proof of (8): Suppose that $w = [x\, \star].\bigsqcup\langle y\rangle$. We have

$$\begin{array}{ll} & w \\ = & \quad \{ \text{ assumption and (7) } \} \\ & [x\, \star].\langle y\rangle \\ = & \quad \{ \ [x\, \star] \text{ pointwise } \} \\ & \langle x \star y\rangle \end{array}$$

So we get

$$\begin{array}{ll} & (x \rightharpoondown w) \star \langle y\rangle \\ = & \quad \{ \text{ Lemma 17, } w \text{ a singleton (above) } \} \\ & \langle x\rangle \star \langle y\rangle \\ = & \quad \{ \ \star \text{ pointwise } \} \\ & \langle x \star y\rangle \\ = & \quad \{ \text{ above } \} \\ & w \end{array}$$

Induction step: We assume that for some $p$ and $q$ such that $lgl.p = lgl.q$ we have, for all $w$ and $x$,

$$\begin{array}{lrcl} (9) & w = (x \rightharpoondown w) \star p & \equiv & w = [x\, \star].\bigsqcup p \\ (10) & w = (x \rightharpoondown w) \star q & \equiv & w = [x\, \star].\bigsqcup q \end{array}$$

We show

$$\begin{array}{lrcl} & \bigsqcup(p \mid q) & = & u \mid v \\ (11) & & & \text{where} \quad u \ = \ \bigsqcup p \\ & & & \phantom{\text{where}} \quad v \ = \ [\overline{u}\, \star].\bigsqcup q \end{array}$$

(12) $$w = [x\,\star].\bigsqcup(p \mid q) \quad \Rightarrow \quad w = (x \rightarrow w) \star (p \mid q)$$

Proof of (11): Suppose $w = \bigsqcup(p \mid q)$. From Definition 9 we have that $w$ is not a singleton, so let $w = u \mid v$.

$$\begin{array}{ll} & u \mid v \\ = & \qquad \{ \text{ Definition 9 } \} \\ & ((u \mid v)) \vec{\star} (p \mid q) \\ = & \qquad \{ \text{ Lemma 18 } \} \\ & (u \vec{\star} p) \mid ((\overline{u} \rightarrow v) \star q) \end{array}$$

So from the Tie axiom

$$\begin{array}{rcl} u & = & u \vec{\star} p \\ v & = & (\overline{u} \rightarrow v) \star q \end{array}$$

From Definition 9 we get

$$u \quad = \quad \bigsqcup p$$

and (10) gives

$$v \quad = \quad [\overline{u}\,\star].\bigsqcup q$$

Proof of (12): Suppose $w = [x\,\star].\bigsqcup(p \mid q)$. Taking $u$ and $v$ satisfying (11) we get

$$\begin{array}{ll} & w \\ = & \qquad \{ \text{ assumption and (11) } \} \\ & [x\,\star].(u \mid v) \\ = & \qquad \{ \; [x\,\star] \text{ pointwise } \} \\ & [x\,\star].u \mid [x\,\star].v \end{array}$$

So we have

(13) $$\begin{array}{rcl} w & = & r \mid s \\ & & \text{where} \quad \begin{array}{rcl} r & = & [x\,\star].u \\ s & = & [x\,\star].v \end{array} \end{array}$$

From (13) and (11) we get

$$r \quad = \quad [x\,\star].\bigsqcup p$$

So from (9)

(14) $$r \quad = \quad (x \rightarrow r) \star p$$

Also

$$
\begin{array}{ll}
 & s \\
= & \quad \{\ (13) \text{ and } (11)\ \} \\
 & [x\,\star].([\overline{u}\,\star].\bigsqcup q) \\
= & \quad \{\ \text{Lemma 21}\ \} \\
 & [[x\,\star].\overline{u}\,\star].\bigsqcup q \\
= & \quad \{\ [x\,\star] \text{ pointwise, } \mathit{last} \text{ positional}\ \} \\
 & [\mathit{last}.([x\,\star].u)\,\star].\bigsqcup q \\
= & \quad \{\ (13)\ \} \\
 & [\overline{r}\,\star].\bigsqcup q
\end{array}
$$

So from (10)

(15) $$s \ = \ (\overline{r} \rightarrow s) \star q$$

Thus

$$
\begin{array}{ll}
 & (x \rightarrow w) \star (p \mid q) \\
= & \quad \{\ (13)\ \} \\
 & (x \rightarrow (r \mid s)) \star (p \mid q) \\
= & \quad \{\ \text{Lemma 3}\ \} \\
 & ((x \rightarrow r) \mid (\overline{r} \rightarrow s)) \star (p \mid q) \\
= & \quad \{\ \star \text{ pointwise}\ \} \\
 & (((x \rightarrow r) \star p) \mid ((\overline{r} \rightarrow s) \star q)) \\
= & \quad \{\ (14);\ (15)\ \} \\
 & r \mid s \\
= & \quad \{\ (13)\ \} \\
 & w
\end{array}
$$

(End of proof)

**Note** Lemma 22 gives an iterative algorithm for the prefix computation, which has running time linear in the length of the input list. We can derive other algorithms for the prefix computation which have running time logarithmic in the length of the input list. These algorithms use $\bowtie$ to deconstruct the list in the recursive case. See [6] for details.

The following lemma shows how $[x\,\star]$ distributes over $\mapsto$.

**Lemma 24** $\quad [x\,\star].(y \mapsto p) = (x \star y) \mapsto p$

**Proof** Let $w = (x \star y) \mapsto p$, so from Definition 10, $w = \bigsqcup((x \star y) \rightarrow p)$

$$
\begin{array}{ll}
 & w \\
= & \quad \{\ \text{Definition 9}\ \} \\
 & w \,\vec{\star}\, ((x \star y) \rightarrow p) \\
= & \quad \{\ \text{Lemma 19}\ \} \\
 & (x \star y) \rightarrow (w \star p) \\
= & \quad \{\ \star \text{ pointwise, } \rightarrow \text{ positional}\ \} \\
 & (x \rightarrow w) \star (y \rightarrow p)
\end{array}
$$

So from Lemma 23

$$w \quad = \quad [x\,\star].\bigsqcup(y \rightarrow p)$$

From Definition 10 this gives

$$w \quad = \quad [x\,\star].(y \mapsto p)$$

(End of proof)

We now are able to prove Lemma 11.

**Lemma 11**

$$\begin{array}{rcl} x \mapsto \langle y\rangle & = & \langle x\rangle \\ x \mapsto (p \mid q) & = & u \mid v \\ & & \text{where} \quad \begin{array}{rcl} u & = & x \mapsto p \\ v & = & (\overline{u} \star \overline{p}) \mapsto q \end{array} \end{array}$$

**Proof**

$$\begin{array}{ll} & x \mapsto \langle y\rangle \\ = & \quad \{\text{ Definition 10 }\} \\ & \bigsqcup(x \rightarrow \langle y\rangle) \\ = & \quad \{\text{ Definition 2 }\} \\ & \bigsqcup\langle x\rangle \\ = & \quad \{\text{ Lemma 22 }\} \\ & \langle x\rangle \end{array}$$

$$\begin{array}{ll} & x \mapsto (p \mid q) \\ = & \quad \{\text{ Definition 10 }\} \\ & \bigsqcup(x \rightarrow (p \mid q)) \\ = & \quad \{\text{ Lemma 3 }\} \\ & \bigsqcup((x \rightarrow p) \mid (\overline{p} \rightarrow q)) \end{array}$$

So by Lemma 22

$$x \mapsto (p \mid q) \quad = \quad u \mid v$$

where

$$\begin{array}{rcl} u & = & \bigsqcup(x \rightarrow p) \\ v & = & [\overline{u}\,\star].\bigsqcup(\overline{p} \rightarrow q) \end{array}$$

Definition 10 then gives

$$\begin{array}{rcl} u & = & x \mapsto p \\ v & = & [\overline{u}\,\star].(\overline{p} \mapsto q) \end{array}$$

and Lemma 24 gives

$$v \quad = \quad (\overline{u} \star \overline{p}) \mapsto q$$

(End of proof)

## A.5 Correctness of the ripple-carry adder

In this section we need some properties of mod and $\div$ on natural numbers. We take them to be defined as follows (all variables are natural numbers with $m > 1$). We give both of these operators the same binding power as $*$, the multiplication operator.

**Definition 25**

$$\begin{aligned} x \div m &= \lfloor x/m \rfloor \\ x \bmod m &= x - m * (x \div m) \end{aligned}$$

The following results can be proven from these definitions. We omit the proofs.

**Lemma 26**

**0.** $0 \leq x \bmod m < m$

**1.** $x < m \quad \equiv \quad x \bmod m = x$

**2.** $x < m^2 \quad \equiv \quad x \div m < m$

**3.** $(x \bmod m + y) \bmod m = (x + y) \bmod m$

**4.** $x + y \div m = (x * m + y) \div m$

**5.** $(x \div m) \div m = x \div m^2$

**6.** $x \bmod m + ((x \div m) \bmod m) * m = x \bmod m^2$

**7.** $(x \bmod m^2) \bmod m = x \bmod m$

**8.** $(x \bmod m^2) \div m = (x \div m) \bmod m$

To state the correctness of the ripple-carry adder we define a function $bv$ which convert natural numbers to bit vector representations and another function $vp$ to convert the output of the the function rc to the corresponding natural numbers.

Function $bv$ takes two natural numbers as arguments. The value of $bv.n.x$ in the standard model is a bit vector of length $2^n$ representing $x \bmod 2^{2^n}$ in the encoding assumed in our definition of rc.

**Definition 27**

$$\begin{aligned} bv.0.x &= \langle x \bmod 2 \rangle \\ bv.(n+1).x &= bv.n.(x \bmod 2^{2^n}) \mid bv.n.(x \div 2^{2^n}) \qquad \text{for } n \geq 0 \end{aligned}$$

The proof of the following lemma is straightforward and is omitted.

**Lemma 28** $\quad lgl.(bv.n.x) = n$

We also have the following result.

**Lemma 29** $\quad bv.n.x = bv.n.(x \bmod 2^{2^n})$

**Proof** We use inducton on $n$.
Basis: Immediate from Definition 27 and Lemma 26.3 since $2 = 2^{2^0}$.
Induction step: We define $m = 2^{2^n}$, so $m^2 = 2^{2^{n+1}}$, and our induction hypothesis is, for any $x$,

$$bv.n.x \quad = \quad bv.n.(x \bmod m)$$

We want to show

$$bv.(n+1).x \quad = \quad bn.(n+1).(x \bmod m^2)$$

$$\begin{array}{ll}
 & bv.(n+1).(x \bmod m^2) \\
= & \qquad \{ \text{ Definition 27 } \} \\
 & bv.n.((x \bmod m^2) \bmod m) \mid bv.n.((x \bmod m^2) \div m) \\
= & \qquad \{ \text{ Lemma 26.7; Lemma 26.8 } \} \\
 & bv.n.(x \bmod m) \mid bv.n.((x \div m) \bmod m) \\
= & \qquad \{ \text{ induction hypothesis, with } x := x \div m \ \} \\
 & bv.n.(x \bmod m) \mid bv.n.(x \div m) \\
= & \qquad \{ \text{ Definition 27 } \} \\
 & bv.(n+1).x
\end{array}$$

(End of proof)

The following lemma is the main result needed for the ripple-carry adder correctness proof.

**Lemma 30** *For any $x$, $y$ and $b$, $0 \le x < 2^{2^n}$, $0 \le y < 2^{2^n}$, $b \in \{0,1\}$,*

$$\mathsf{rc}.b.(bv.n.x).(bv.n.y) \quad = \quad (bv.n.(x+y+b), (x+y+b) \div 2^{2^n})$$

**Proof** Note first that for $x$, $y$ and $b$ in the ranges given, $x+y+b < 2^{1+2^n}$, so $(x+y+b) \div 2^{2^n}) \in \{0,1\}$, as required. The proof of the theorem is by induction on $n$.
Basis: Note that for $n = 0$, $x < 2$ and $y < 2$.

$$\begin{array}{ll}
 & \mathsf{rc}.b.(bv.0.x).(bv.0.y) \\
= & \qquad \{ \text{ Definition 27 } \} \\
 & \mathsf{rc}.b.\langle x \bmod 2\rangle.\langle y \bmod 2\rangle \\
= & \qquad \{ \text{ Lemma 26.1, } x < 2 \text{ and } y < 2 \ \} \\
 & \mathsf{rc}.b.\langle x\rangle.\langle y\rangle \\
= & \qquad \{ \text{ Definition 12 } \} \\
 & (\langle (x+y+b) \bmod 2\rangle, (x+y+b) \div 2) \\
= & \qquad \{ \text{ Definition 27; } 2^0 = 1 \ \} \\
 & (bv.0.(x+y+b), (x+y+b) \div 2^{2^0})
\end{array}$$

Induction step: We define $m = 2^{2^n}$, so $m^2 = 2^{2^{n+1}}$, and we assume that for all $x$, $y$, $0 \leq x < m$ and $0 \leq y < m$, and for $b \in \{0,1\}$,

$$\mathsf{rc}.b.(bv.n.x).(bv.n.y) \quad = \quad (bv.n.(x+y+b), (x+y+b) \div m)$$

We have to show

$$\begin{aligned} &\mathsf{rc}.b.(bv.(n+1).x).(bv.(n+1).y) \quad = \\ &\qquad (bv.(n+1).(x+y+b), (x+y+b) \div m^2) \end{aligned} \tag{16}$$

for $x$ and $y$ in the range $0 \leq x < m^2$ and $0 \leq y < m^2$. Note that from Definition 27 we have

$$\begin{aligned} bv.(n+1).x &= bv.n.(x \bmod m) \mid bv.n.(x \div m) \\ bv.(n+1).y &= bv.n.(y \bmod m) \mid bv.n.(y \div m) \end{aligned}$$

So from Definition 12 we have

$$\mathsf{rc}.b.(bv.(n+1).x).(bv.(n+1).y) \quad = \quad (t, d) \tag{17}$$

where

$$t \quad = \quad u \mid v \tag{18}$$
$$(u, c) \quad = \quad \mathsf{rc}.b.(bv.n.(x \bmod m)).(bv.n.(y \bmod m)) \tag{19}$$
$$(v, d) \quad = \quad \mathsf{rc}.c.(bv.n.(x \div m)).(bv.n.(y \div m)) \tag{20}$$

We have, from Lemma 26.0, $x \bmod m < m$ and $y \bmod m < m$; also, from Lemma 26.2, $x \div m < m$ and $y \div m < m$. So we apply the induction hypothesis to (19) and (20) to get

$$u \quad = \quad bv.n.(x \bmod m + y \bmod m + b) \tag{21}$$
$$c \quad = \quad (x \bmod m + y \bmod m + b) \div m \tag{22}$$
$$v \quad = \quad bv.n.(x \div m + y \div m + c) \tag{23}$$
$$d \quad = \quad (x \div m + y \div m + c) \div m \tag{24}$$

We now have

$$\begin{aligned} & u \\ = & \qquad \{ \text{ (21); Lemma 29 } \} \\ & bv.n.((x \bmod m + y \bmod m + b) \bmod m) \\ = & \qquad \{ \text{ Lemma 26.3, twice } \} \\ & bv.n.((x+y+b) \bmod m) \end{aligned}$$

and

$$
\begin{array}{ll}
& x \div m + y \div m + c \\
= & \qquad \{ \text{ (22) } \} \\
& x \div m + y \div m + (x \bmod m + y \bmod m + b) \div m \\
= & \qquad \{ \text{ Lemma 26.4, twice } \} \\
& ((x \div m) * m + (y \div m) * m + x \bmod m + y \bmod m + b) \div m \\
= & \qquad \{ \text{ Lemma 26.1, twice (since } x \div m < m \text{ and } y \div m < m \text{) } \} \\
& (((x \div m) \bmod m) * m + ((y \div m) \bmod m) * m + \\
& \qquad x \bmod m + y \bmod m + b) \div m \\
= & \qquad \{ \text{ rearrange } \} \\
& (x \bmod m + ((x \div m) \bmod m) * m + \\
& \qquad y \bmod m + ((y \div m) \bmod m) * m + b) \div m \\
= & \qquad \{ \text{ Lemma 26.6, twice } \} \\
& (x \bmod m^2 + y \bmod m^2 + b) \div m \\
= & \qquad \{ \text{ Lemma 26.1, twice (since } x < m^2 \text{ and } y < m^2 \text{) } \} \\
& (x + y + b) \div m
\end{array}
$$

Substituting in (23) gives

$$v \quad = \quad bv.n.((x + y + b) \div m)$$

Also

$$
\begin{array}{ll}
& d \\
= & \qquad \{ \text{ substituting in (24) from above } \} \\
& ((x + y + b) \div m) \div m \\
= & \qquad \{ \text{ Lemma 26.5 } \} \\
& (x + y + b) \div m^2
\end{array}
$$

So far we have the following

$$
\begin{array}{lrcl}
(25) & u & = & bv.n.((x + y + b) \bmod m) \\
(26) & v & = & bv.n.((x + y + b) \div m) \\
(27) & d & = & (x + y + b) \div m^2
\end{array}
$$

We now get

$$
\begin{array}{ll}
& t \\
= & \qquad \{ \text{ substitute from (25), (26) in (18) } \} \\
& bv.n.((x + y + b) \bmod m) \mid bv.n.((x + y + b) \div m) \\
= & \qquad \{ \text{ Definition 27 } \} \\
& bv.(n + 1).(x + y + b)
\end{array}
$$

This result, together with (17) and (27) gives (16), as required.
(End of proof)

We now define the function *vp* which returns the natural number corresponding to the result of a call to `rc`. We first define auxiliary functions *expand* and *vv*.

The idea of the function *expand* is that *expand.n.p* returns the value of the the bit vector $p$, where $p$ is regarded as a number in base $2^{2^n}$.

**Definition 31**

$$\begin{array}{rcl} expand.n.\langle x\rangle & = & x \\ expand.n.(p \bowtie q) & = & expand.(n+1).p + expand.(n+1).q * 2^{2^n} \end{array}$$

Function $vv$ is defined in terms of *expand*.

**Definition 32** $\quad vv = expand.0$

The following lemma can be proven by structural induction.

**Lemma 33** $\quad expand.n.(p \mid q) = expand.n.p + expand.n.q * 2^{2^{n+lgl.p}}$

We need only the following corollary to this lemma, which follows immediately from Definition 32.

**Lemma 33′** $\quad vv.(p \mid q) = vv.p + vv.q * 2^{2^{lgl.p}}$

We now show that $vv$ is the inverse of $bv.n$ for a subset of the natural numbers.

**Lemma 34** $\quad vv.(bv.n.x) = x \bmod 2^{2^n}$

**Proof** We use induction on $n$.
Basis:

$$\begin{array}{cl} & vv.(bv.0.x) \\ = & \qquad \{\text{ Definition 32; Definition 27 }\} \\ & expand.0.\langle x \bmod 2\rangle \\ = & \qquad \{\text{ Definition 31 }\} \\ & x \bmod 2 \\ = & \qquad \{\ 2^0 = 1\ \} \\ & x \bmod 2^{2^0} \end{array}$$

Induction step: Assume the result holds for $n$.

$$\begin{array}{cl} & vv.(bv.(n+1).x) \\ = & \qquad \{\text{ Definition 27 }\} \\ & vv.(bv.n.(x \bmod 2^{2^n}) \mid bv.n.(x \div 2^{2^n})) \\ = & \qquad \{\text{ Lemma 29 }\} \\ & vv.(bv.n.x) \mid bv.n.(x \div 2^{2^n})) \\ = & \qquad \{\text{ Lemma 33′; Lemma 28 }\} \\ & vv.(bv.n.x) + vv.(bv.n.(x \div 2^{2^n})) * 2^{2^n} \\ = & \qquad \{\text{ induction hypothesis, twice }\} \\ & x \bmod 2^{2^n} + ((x \div 2^{2^n}) \bmod 2^{2^n}) * 2^{2^n} \\ = & \qquad \{\text{ Lemma 26.6 }\} \\ & x \bmod 2^{2^{n+1}} \end{array}$$

(End of proof)

We now define function $vp$.

**Definition 35** $\quad vp.(t,d) = vv.t + d * 2^{2^{lgl.t}}$

Finally, we can state and prove that $\mathsf{rc}$ correctly implements addition.

**Theorem 2 (Correctness of the ripple-carry adder)** *For any $x$, $y$ and $b$, $0 \le x < 2^{2^n}$, $0 \le y < 2^{2^n}$, $b \in \{0,1\}$,*

$$vp.(\mathsf{rc}.b.(bv.n.x).(bv.n.y)) \quad = \quad x+y+b$$

**Proof** We first note that we have $x+y+b < 2^{1+2^n} \le 2^{2^{n+1}}$. We let $m = 2^{2^n}$, so $m^2 = 2^{2^{n+1}}$, as before. From the above and Lemma 26.2 we have

$$x+y+b \quad < \quad m^2 \tag{28}$$
$$(x+y+b) \div m \quad < \quad m \tag{29}$$

So we calculate

$$\begin{array}{ll}
 & vp.(\mathsf{rc}.b.(bv.n.x).(bv.n.y)) \\
= & \qquad \{ \text{ Lemma 30 } \} \\
 & vp.(bv.n.(x+y+b), (x+y+b) \div m) \\
= & \qquad \{ \text{ Definition 35; Lemma 28 } \} \\
 & vv.(bv.n.(x+y+b)) + ((x+y+b) \div m) * m \\
= & \qquad \{ \text{ Lemma 34; Lemma 26.1, (29) } \} \\
 & (x+y+b) \bmod m + (((x+y+b) \div m) \bmod m) * m \\
= & \qquad \{ \text{ Lemma 26.6 } \} \\
 & (x+y+b) \bmod m^2 \\
= & \qquad \{ \text{ Lemma 26.1, (28) } \} \\
 & x+y+b
\end{array}$$

(End of proof)

## References

[0] E W Dijkstra and C S Scholten. *Predicate calculus and program semantics*. Springer-Verlag, 1990.

[1] M J C Gordon. *Why higher-order logic is a good formalism for specifying and verifying hardware*. Technical report 77, University of Cambridge Computer Laboratory, 1985.

[2] Warren A Hunt, Jr and Bishop C Brock. *Verification of a bit-slice ALU*. Technical report 49, Computational Logic Inc, 1989.

[3] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal methods for VLSI design*. North-Holland, 1990.

[4] Jacob Kornerup. *Mapping powerlists onto hypercubes*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 1994. In preparation.

[5] Miriam Leeser. Using Nuprl for the verification and synthesis of hardware. *Philosopical Transactions of the Royal Society*, A **339**, 49–68, 1992.

[6] Jayadev Misra. Powerlist: a structure for parallel recursion (preliminary version). In A W Roscoe, editor, *A classical mind: essays in honour of C A R Hoare*, pages 295–316. Prentice Hall International, 1994.