Denotationally Correct Computer Arithmetic

Atticus Kuhn

2023-07-16 11:03 GMT-8

Abstract

In this paper, we will give an introduction to basic category theory. We then will use knowledge of category theory, as Agda (see [1]), to prove the formal correctness of machine algorithms for arithmetic over bits. Our goal is to make the most elegant and precise specification of binary arithmetic for proofs of correctness.

We will specifically be focusing on conversions between numbers and binary representations, as well as addition and multiplication of binary representations.

Contents

1 Introduction

The goal of this paper is to use the techniques of Denotational Design (see [2]), where we specify the meaning (or "denotation") as elegantly as possible for the ease of proof. This paper will use the example of computer binary arithmetic. We will specify the correctness of binary arithmetic using tools from category theory, and then prove that our definitions satisfy the specification. A background in category theory will be helpful, but the reader will not need to know category theory, as we will give a basic introduction in Section 2.

2 Basic Introduction to Category Theory

Most of the information on category theory comes from [4], and it is recommended that the reader look there for further information on category theory.

Definition 1. A **category** consists of a set of **objects** and a set of **arrows** (called "morphisms") between objects. We use the notation $f : A \to B$ to write that f is an arrow from A to B. Each arrow points from one object to another object. The objects and arrows must satisfy 4 properties:

1. **identity** For all objects A, there exists an arrow, $Id_A : A \to A$.
2. **composition**: For all arrows $f : B \to C$ and all arrows $g : A \to B$, there exists arrow $f \circ g : A \to C$.
3. **identity cancellation**: For all arrows $f : A \to B$, $f \circ Id_A = Id_B \circ f = f$.
4. **associativity**: For all arrows $f : C \to D$, $g : B \to C$, $h : A \to B$, $f \circ (g \circ h) = (f \circ g) \circ h$.

Example. There are many examples of categories that the reader might perhaps already know:

1. **Set**: The objects are sets and the arrows are functions between sets.
2. **Rng**: The objects are rings and the arrows are ring homomorphisms.
3. **Grp**: The objects are groups and the arrows are group homomorphisms.
4. $\leq$: The objects are natural numbers and the arrows are proofs that $a \leq b$.

One tool that is useful in category theory is the idea of a commutivity diagram, which allows us to compactly and elegently represent equations.

Definition 2. A **commutivity diagram** is a type of picture that represents a category by drawing the objects as points and arrows as directed arrows between objects, having the property that all directed paths with the same start and end must represent the same composition. For example, consider Figure 1. It represents the same information as Equation 1. Commutativity diagrams are popular in category theory.

$$id_b \circ f = f \qquad id_b \circ g = g \qquad g \circ f = h \tag{1}$$

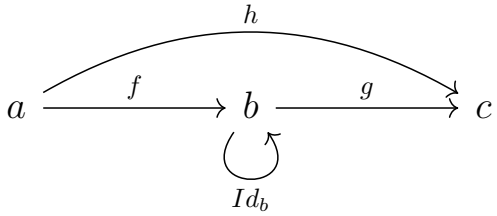

Figure 1: A Sample Commutivity Diagram

2.1 Functors

Functors are like the homomorphisms of categories, i.e. maps that respect certain properties of a category.

Definition 3. Given categories C and D, we say that $T : C \to D$ is a **functor** from C to D if T is a function from C to D. At risk of an abuse of notation, we say that T is a function on both the objects and the arrows of C. We must have that T satisfies some additional properties:

1. For any object c in C, $T(Id_c) = Id_{T(c)}$
2. For all arrows $f : B \to C$ and $g : A \to B$ in C, $T(f \circ g) = T(f) \circ T(g)$

Example. **Forgetful Functors**: One example of a functor is called a forgetful functor, which maps from a more structured category to a less structured category by simply "forgetting" some of the information in the structure. Consider, for example, the map from **Rng** $\to$ **Grp**, or the map from **Grp** $\to$ **Set**.

Another functor is called the **identity functor**, which just sends every object and arrow to itself. We write the identity functor as $\mathbf{I}_C$.

Example. Consider the category N whose objects are sets of natural numbers and whose arrows are functions on natural numbers, and consider $\mathbb{B}^n$ as a category whose objects are sets of n bit binary strings and whose arrows are functions on binary strings. A function from N to $\mathbb{B}^n$ could be to convert each natural number to its binary representation, and convert each function on natural numbers to the corresponding number on binary representations. This functor will be of interest to us later.

Definition 4. Given functors $S, T : C \to B$, a **natural transformation** $\tau : S \to T$ is a function which assigns to each object $c \in C$ an arrow $\tau_c : S(c) \to T(c)$ in B in such a way such that for every arrow $f : c \to c'$ in C, we have that Figure 2 commutes.

We may say that natural transformations are to functors as functors are to categories.

Definition 5. Given functors $S, T : C \to B$, we say there is a **natural isomorphism**, written as $S \cong T$, if there is a two-sided natural transformation between S and T. If S and T are naturally isomorphic, we may regard them as being the same for all purposes.

Using functors, we can say if two categories are "the same".

Definition 6. Given categories C and D, we say that C and D are equivalent categories if there exist functors $T : C \to D$ and $S : D \to C$ such that $S \circ T \cong \mathbf{I}_C$ and $T \circ S \cong \mathbf{I}_D$.

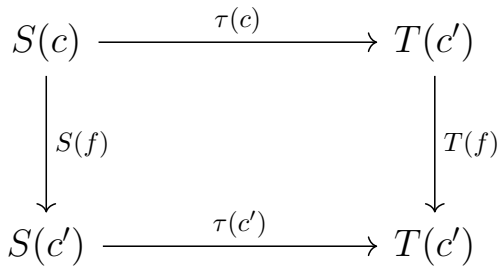

Figure 2: A Commutativity Diagram for Natural Transformations

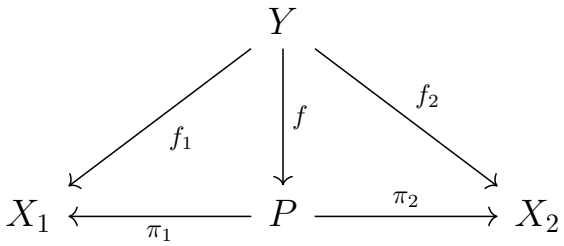

Figure 3: A Commutivity Diagram Representing Products

A property of categories that we will be using is called "products", which you may think of as being like a pair or a cartesian product.

Definition 7. Given a category C and objects X_1, X_2 in C, we call an object p in C the **product** of X_1 and X_2 (written $X_1 \times X_2$) if there exist arrows $\pi_1 : P \to X_1, \pi_2 : P \to X_2$ (called *projections*) such that for all objects Y in C and all arrows $f_1 : Y \to X_1$ and $f_2 : Y \to X_2$, there exists a unique arrow $f : Y \to P$ such that Figure 3 commutes.

We now give some examples of products.

Example. In the category of sets, the product corresponds to the Cartesian product. In the category of groups, the product corresponds to the direct sum.

Definition 8. Given a category C, we say that C is a **monoidal category** if it has a bifunctor $\otimes : C \times C \to C$ which is associatiative and has a two-sided inverse $1 \in C$. We have that equation 2 holds.

$$U \otimes (V \otimes W) \cong (U \otimes V) \otimes W \tag{2}$$
$$U \otimes 1 \cong U \tag{3}$$
$$1 \otimes U \cong U \tag{4}$$

Definition 9. Given a category C, we say that C is a **Cartesian category** if it is a monoidal category where the monoid operation is a product and the identity is the initial object.

This ends our discussion of Category Theory.

3 Preliminaries

We will be working with numbers and bits. Let N be a Cartesian category representing our numbers. We will assume that there are bifunctors on N called

$$+_N : N \times N \to N \qquad \cdot_N : N \times N \to N.$$

Let $\mathbb{B}^n$ be a Cartesian category of n-bit binary representations. We will assume functors in equation 5.

$$\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \tag{5}$$
$$\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \tag{6}$$
$$\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \tag{7}$$
$$if : \mathbb{B} \times A \times A \to A \tag{8}$$

We will assume the following properties in equation 9.

$$\forall A \in \mathbb{B} \quad \forall c \in S \quad if(A, c, c) = c \tag{9}$$
$$\forall A \in \mathbb{B} \quad \forall b, c \in S \quad \forall f : S \to T \quad f(if(A, b, c)) = if(A, f(b), f(c)) \tag{10}$$
$$\forall A, B \in \mathbb{B} \quad \forall a, b, c \in S \quad if(A, if(B, a, b), c) = if(A \wedge B, a, if(A, b, c)) \tag{11}$$
$$\forall A, B \in \mathbb{B} \quad \forall a, b, c \in S \quad if(A, a, if(B, b, c)) = if(A \oplus B, if(A, a, b), c) \tag{12}$$

To go between numbers and bits, we will assume there is a homomorphism

$$\%2 : N \twoheadrightarrow Bit$$

such that for all $A, B \in N$

$$\%2(1_N) = 1_{\mathbb{B}} \qquad \%2(0_N) = 0_{\mathbb{B}} \qquad \%2(A +_N B) = \%2(A) \otimes \%2(B).$$

You can think of %2 as just returning the last bit of a number (1 if the number is odd and 0 if the number is even).

We will represent binary numbers in Agda as lists of bits, where the least significant bit is on the left (little endian encoding).

As an additional preliminary, we expect the reader to be familiar with common bitwise operations, including $\cdot \oplus \cdot$, $\cdot \vee \cdot$, and $\cdot \wedge \cdot$ (see table 1).

$\cdot \oplus \cdot$	$\cdot \vee \cdot$	$\cdot \wedge \cdot$
$0 \oplus 0 = 0$	$0 \vee 0 = 0$	$0 \wedge 0 = 0$
$0 \oplus 1 = 1$	$0 \vee 1 = 1$	$0 \wedge 1 = 0$
$1 \oplus 0 = 1$	$1 \vee 0 = 1$	$1 \wedge 0 = 0$
$1 \oplus 1 = 0$	$1 \vee 0 = 1$	$1 \wedge 1 = 1$

Table 1: $\cdot \oplus \cdot$, $\cdot \vee \cdot$, and $\cdot \wedge \cdot$

```
NatToLittleEndian : N ⇸ Bits b
NatToLittleEndian {0} = %2
NatToLittleEndian {suc b} = %2 △ (NatToLittleEndian {b} ∘ /2)

bitToNat : Bit ⇸ N
bitToNat = if ∘ const  (p1 △ p0)

LittleEndianToNat : Bits b ⇸ N
LittleEndianToNat {0} = bitToNat
LittleEndianToNat {suc b} = add ∘ (bitToNat ⊗ (*2 ∘ LittleEndianToNat {b}))
```

Figure 4: Agda Code for Converting a Number to Binary

4 Converting Between Numbers and Bits

We need to be able to convert between numbers and bits. This is not too difficult, as to get the n^{th} bit, we bit-shift over n places and then get the last bit. We might write this as

$$N to \mathbb{B}^n(A) = [A\%2, \frac{A}{2}\%2, \frac{A}{4}\%2, \ldots, \frac{A}{2^{n-1}}\%2]. \tag{13}$$

$$\mathbb{B}^n toN([b_1, b_2, \ldots, b_n]) = b_1 + 2b_2 + 4b_3 + \cdots + 2^{n-1}b_n \tag{14}$$

In Agda, we would write equation 13 as Figure 4.

See Figures 5 and 6 for an example of these conversions. In order for $N to \mathbb{B}^n$ and $\mathbb{B}^n toN$ to

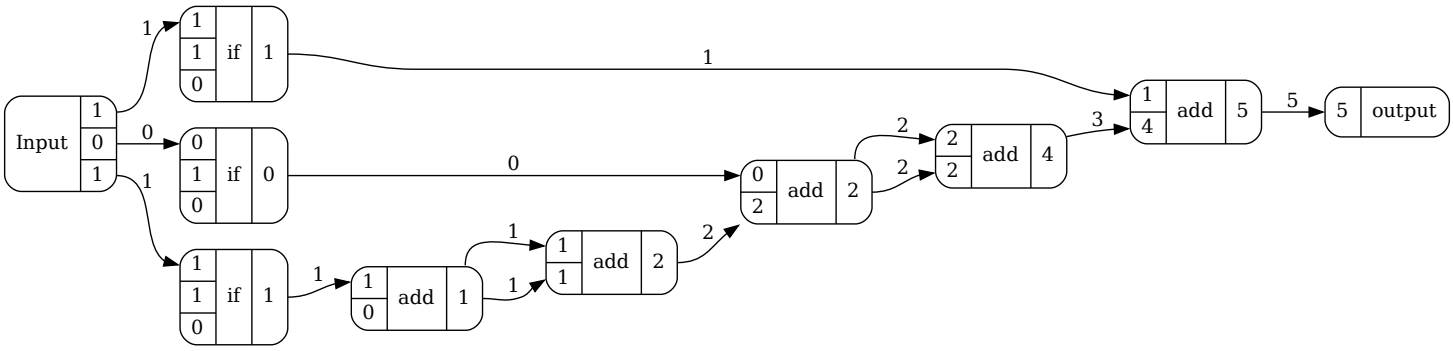

Figure 5: An Example showing $\mathbb{B}^3 toN(101) = 5$

be correct, we would need that diagram 7 commutes.

In equational form, diagram 7 is equivalent to equation 15.

$$N to \mathbb{B}^n \circ \mathbb{B}^n toN \equiv id_{\mathbb{B}^n} \qquad \mathbb{B}^n toN \circ N to \mathbb{B}^n \equiv id_N \tag{15}$$

In Agda, we would write equation 15 as Figure 8. In category theory, diagram 7 shows there

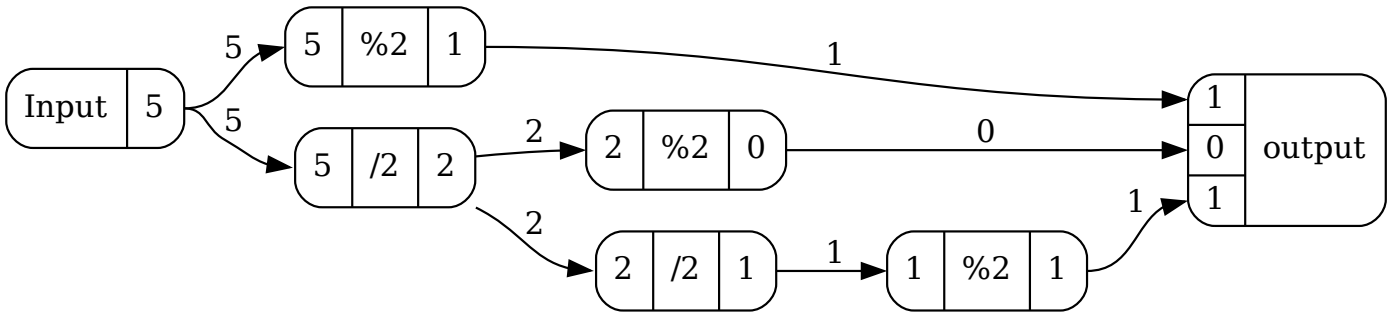

Figure 6: An Example showing $Nto\mathbb{B}^3(5) = 101$

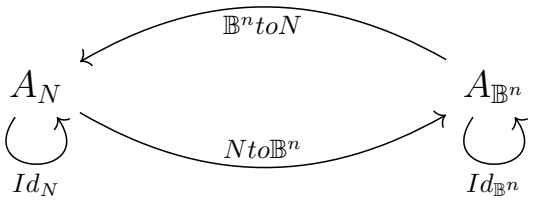

Figure 7: Correctness Specification for Conversions Between Numbers and their Respective Binary Representations

exists an equivalence of categories between N and $\mathbb{B}^n$. Equation 15 is true, but the proof is too long to be contained here. You can see the proof at [3].

5 Addition

5.1 Correctness of Addition

Before we can discuss addition, we need to make sure we know what we mean by correct addition. Addition on binary numbers is correct if and only if diagram 9 commutes. Diagram 9 in equational form is equivalent to equation 16.

$$\text{For all } A, B \in \mathbb{B}^n \qquad \mathbb{B}^n toN(A +_{\mathbb{B}^n} B) = \mathbb{B}^n toN(A) +_N \mathbb{B}^n toN(B). \tag{16}$$

In Agda, we would write the same specification as Figure 10.

```
inverses : NatToLittleEndian {b} ∘ LittleEndianToNat {b} ≈ id
inverses = {! !}
```

Figure 8: An Agda Specification that Conversions are Inverses

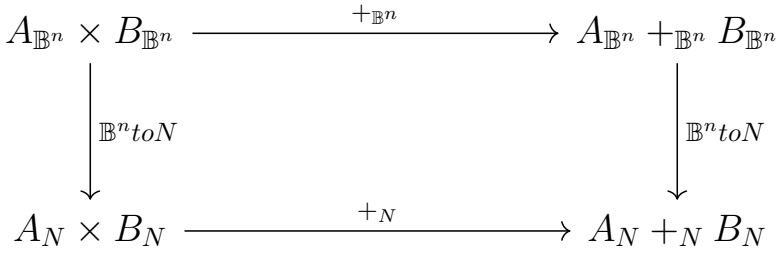

Figure 9: Correctness Specification for Addition on Binary Representations

```
RCAspecification : LittleEndianToNat {suc b} ∘ RippleAdd {b} ≈ add ∘ twice (LittleEndianToNat {b})
RCAspecification = {! !}
```

Figure 10: Specification of Addition in Agda

5.2 Defining Addition

Before we can define an adder, we need to define half-adders and full-adders. Half-adders and Full-adders are tools from electronics for adding two or three bits at a time, respectively. A half-adder adds two bits with a carry, so for example

$$0 +_H 0 = (0, 0) \qquad 0 +_H 1 = 1 +_H 0 = (1, 0) \qquad 1 +_H 1 = (0, 1)$$

where, for convention, we say the left bit is the sum and the right bit is the carry. We might write $\cdot +_H \cdot$ as equation 17.

$$\forall A, B \in \mathbb{B}^1 \qquad A +_H B = (A \oplus B, A \wedge B) \tag{17}$$

In Agda, we would write equation 17 as Figure 11.The half-adder is correct if and only if diagram 14 commutes. In equational form, diagram 14 is equivalent to equation 18.

$$\text{For all } A, B \in \mathbb{B}^1 \qquad \mathbb{B}^2 toN(A +_H B) = \mathbb{B}^1 toN(A) +_N \mathbb{B}^1 toN(B) \tag{18}$$

We will now prove equation 18

```
halfAdder : Bit × Bit ⇸ Bit × Bit
halfAdder = xor △ and
```

Figure 11: A half-adder in Agda

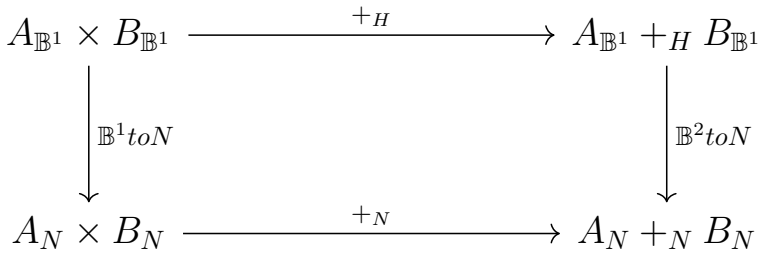

Figure 12: Correctness Specification of a Half-adder

```
halfAdderSpecification : LittleEndianToNat {1} ∘ halfAdder ≈ add ∘ twice bitToNat
```

Figure 13: A Proof of our half-adder specification written in Agda

Proof. We prove equation 18 using the algebraic laws introduced in equation 9.

$$\begin{aligned}
if(A,1,0) + if(B,1,0) &= if(A, 1 + if(B,1,0), 0 + if(B,1,0)) \\
&= if(A, if(B,2,1), if(B,1,0)) \\
&= if(A \wedge B, 2, if(A, 1, if(B,1,0))) \\
&= if(A \wedge B, 2, if(A \oplus B, if(A,1,1), 0)) \\
&= if(A \wedge B, 2, if(A \oplus B, 1, 0)) \\
&= if(false, 3, if(A \wedge B, 2, if(A \oplus B, 1, 0))) \\
&= if((A \wedge B) \wedge (A \oplus B), 3, if(A \wedge B, 2, if(A \oplus B, 1, 0))) \\
&= if(A \wedge B, if(A \oplus B, 3, 2), if(A \oplus B, 1, 0)) \\
&= if(A \wedge B, 2 + if(A \oplus B, 1, 0), 0 + if(A \oplus B, 1, 0)) \\
&= if(A \wedge B, 2, 0) + if(A \oplus B, 1, 0)
\end{aligned}$$

□

In Agda, we would write the same proof as Figure 13. To see an example of a half-adder, look at figure 14.

A full-adder is very similar to a half-adder, except a full-adder adds 3 bits and returns a sum bit and a carry bit. For example, see equation 19.

$$+_F (0,0,0) = (0,0) \tag{19}$$
$$+_F (1,0,0) = +_F(0,1,0) = +_F(0,0,1) = (1,0) \tag{20}$$
$$+_F (0,1,1) = +_F(1,0,1) = +_F(1,1,0) = (0,1) \tag{21}$$
$$+_F (1,1,1) = (1,1) \tag{22}$$

A full-adder is correct if and only if diagram 18 commutes. In equational form, diagram 18 is equivalent to equation 23.

$$\text{for all } A,B,C \in \mathbb{B}^1 \qquad \mathbb{B}^2 toN(+_F(A,B,C)) = \mathbb{B}^1 toN(A) + \mathbb{B}^1 toN(B) + \mathbb{B}^1 toN(C) \tag{23}$$

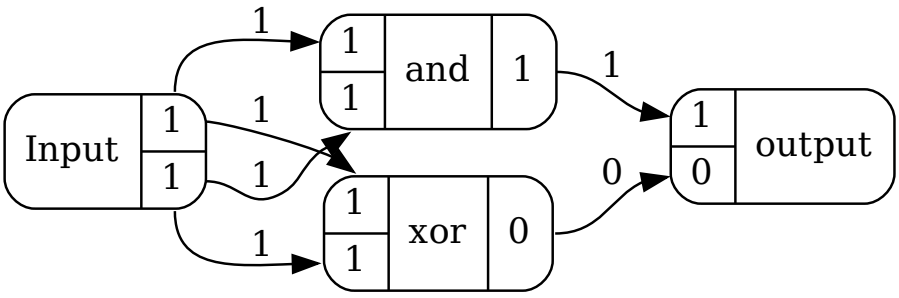

Figure 14: A Half Adder in bit Operations

$$\begin{array}{ccc} A_{\mathbb{B}^1} \times B_{\mathbb{B}^1} \times C_{\mathbb{B}^1} & \xrightarrow{+_F} & +_F(A_{\mathbb{B}^1}, B_{\mathbb{B}^1}, C_{\mathbb{B}^1}) \\ \downarrow{\scriptstyle \mathbb{B}^1 toN} & & \downarrow{\scriptstyle \mathbb{B}^2 toN} \\ A_N \times B_N \times C_N & \xrightarrow{+_N} & A_N +_N B_N +_N C_N \end{array}$$

Figure 15: Correctness Specification of a Full-adder

In Agda, we would write equation 23 as Figure 16.

We will now prove equation 23.

Proof.

$$\begin{aligned} if(A,1,0) + if(B,1.0) + if(C,1,0) &= if(A \wedge B, 2, 0) + if(A \oplus B, 1, 0) + if(C, 1, 0) \\ &= if(A \wedge B, 2, 0) + if((A \oplus B) \wedge C, 2, 0) \\ &+ if(A \oplus B \oplus C, 1, 0) \\ &= if(A \wedge B \vee A \oplus B \wedge C, 2, 0) + if(A \oplus B \oplus C, 1, 0) \end{aligned}$$

□

One example of an explicitly written full-adder is equation 24.

$$+_F(A, B, C) = (A \oplus B \oplus C, AB \vee (A \oplus B)C) \tag{24}$$

In Agda, we may write equation 24 as Figure 17. To see an example of a full-adder, look at

```
fullAdderSpecification : LittleEndianToNat {1} ∘ fullAdder ≈ add ∘ second add ∘ (bitToNat ⊗ twice bitToNat)
```

Figure 16: A Full-Adder Specification in Agda

```
fullAdder : Bit × Bit × Bit ↛ Bit × Bit
fullAdder = second or ∘ inAssoc′ (first halfAdder) ∘ second halfAdder
```

Figure 17: A full-adder in Agda

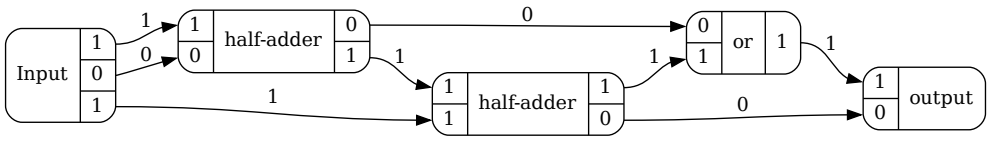

Figure 18: A Full Adder in Bit Operations

figure 18.

The reason we care about full-adders is that full-adders are the building blocks which we use to build up an n bit adder (in general, to write an n bit adder, we use n full-adders).

Using a full-adder, we may define a ripple-carry adder (RCA). An RCA is similar to the addition taught in school. See table 2. One example of an RCA is Figure 20.

$$\begin{aligned}&(a_0, a_1, a_2, \ldots, a_n) +^{c_0}_{\mathbb{B}^n} (b_0, b_1, b_2, \ldots, b_n) = (r_0, (a_1, a_2, \ldots, a_n) +^{c_1}_{\mathbb{B}^{n-1}} (b_1, b_2, \ldots, b_n))\\ &\text{where}\\ &(r_0, c_1) = +_F(a_0, b_0, c_0)\end{aligned} \tag{25}$$

To see an example of a 3-bit RCA, look at figure 19. In Agda, we could write this as Figure 19.

5.3 Proof of Correctness

We will now prove the correctness of our addition on binary numbers. We need to show equation 26.

$$\text{For all } A, B \in \mathbb{B}^n \quad c \in \mathbb{B}^1 \qquad \mathbb{B}^{n+1}toN(A +^{c}_{\mathbb{B}^n} B) = \mathbb{B}^n toN(A) +_N \mathbb{B}^n toN(B) +_N \mathbb{B}^1 toN(c) \tag{26}$$

Proof. We will prove by induction on n.

Base Case: $n = 0$ follows from a simple computation.

^{1}	^{1}1	^{1}0	1
+	1	1	1
1	1	0	0

Table 2: Grade-School Addition

```
RCA : Bit × List b (Bit × Bit) ↛ Bits (suc b)
RCA {0} = fullAdder
RCA {suc n} = second (RCA {n}) ∘ inAssoc′ (first fullAdder)

RippleAdd : Bits b × Bits b ↛ Bits (suc b)
RippleAdd {b} = RCA {b} ∘ const (bit0) ∘ zip {b}
```

Figure 19: A Ripple-Carry Adder in Agda

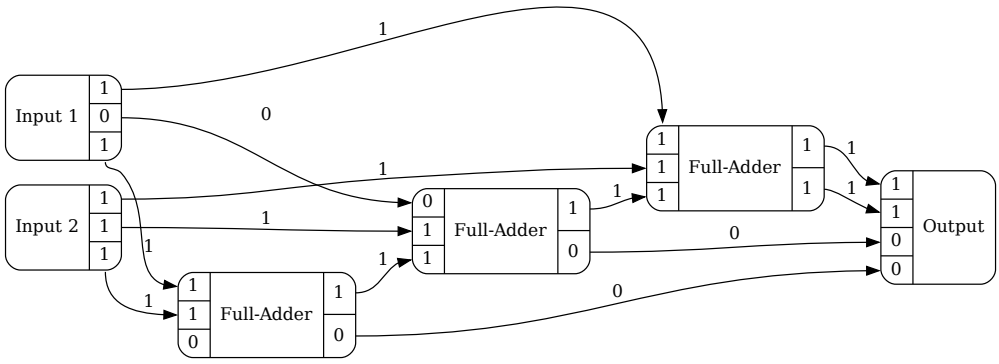

Figure 20: A 3-bit ripple carry adder

Inductive case: $n = n + 1$

$$
\begin{aligned}
&\mathbb{B}^{n+1}toN(A +^{c}_{\mathbb{B}^n} B)\\
&= if(a_0 \oplus b_0 \oplus c, 1, 0) +_N 2\mathbb{B}^{n}toN([a_1, \ldots, a_{n-1}] +^{a_b b_0 \vee (a_0 \oplus b_0)c}_{\mathbb{B}^{n-1}} [b_1, \ldots, b_{n-1}])\\
&= if(a_0 \oplus b_0 \oplus c, 1, 0) +_N 2(\mathbb{B}^{n-1}toN([a_1, \ldots, a_{n-1}])\\
&+_N \mathbb{B}^{n-1}toN([b_1, \ldots, b_{n-1}])) +_N if(a_b b_0 \vee (a_0 \oplus b_0)c, 1, 0)\\
&= if(a_0 \oplus b_0 \oplus c, 1, 0) +_N if(a_b b_0 \vee (a_0 \oplus b_0)c, 1, 0)\\
&+_N 2\mathbb{B}^{n-1}toN([a_1, \ldots, a_{n-1}]) +_N \mathbb{B}^{n-1}toN([b_1, \ldots, b_{n-1}])\\
&= if(a_0, 1, 0) +_N if(b_0, 1, 0) +_N if(c, 1, 0) +_N 2 * \mathbb{B}^{n-1}toN([a_1, \ldots, a_{n-1}])\\
&+_N \mathbb{B}^{n-1}toN([b_1, \ldots, b_{n-1}])\\
&= (if(a_0, 1, 0) +_N 2\mathbb{B}^{n-1}toN([a_1, \ldots, a_{n-1}]))\\
&+_N (if(b_0, 1, 0) +_N 2\mathbb{B}^{n-1}toN([b_1, \ldots, b_{n-1}])) +_N if(c, 1, 0)\\
&= \mathbb{B}^{n}toN(A) +_N \mathbb{B}^{n}toN(B) +_N \mathbb{B}^{1}toN(c)
\end{aligned}
$$

□

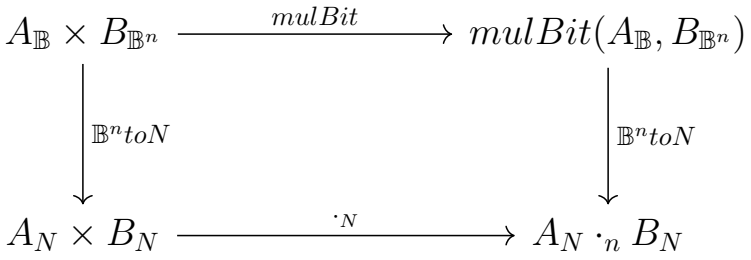

Figure 21: Correctness Specification of $mulBit$

6 Multiplication

We will now look at binary multiplication, which will seem very similar to addition. The first step we need to take is to ensure that we have a correctness specification for multiplication. We say a multiplication function $\cdot_{\mathbb{B}^{m,n}}$ is correct if it satisfies Equation 27.

$$\text{For all } A \in \mathbb{B}^m \quad B \in \mathbb{B}^n \qquad \mathbb{B}^{m+n}toN(A \cdot_{\mathbb{B}^{m,n}} B) = \mathbb{B}^m(A) \cdot_N \mathbb{B}^n toN(B) \tag{27}$$

We first need multiplication of a number by a single bit. We will say multiplication by a single bit is correct if Figure 21 commutes.

Figure 21 is equivalent to equation 28.

$$\text{For all } A \in \mathbb{B}^1 \qquad B \in \mathbb{B}^n \qquad \mathbb{B}^n toN(mulBit(A, B)) = \mathbb{B}^1 toN(A) \cdot_N \mathbb{B}^n toN(B) \tag{28}$$

An example instance of multiplication on a single bit as equation 29. We may alternatively implement a bit multiplier as equation 31, although equation 29 is easier to prove.

$$mulBit : \mathbb{B}^1 \times \mathbb{B}^n \to \mathbb{B}^n \tag{29}$$
$$mulBit(a\ ,\ [b_0, b_1, \ldots, b_{n-1}]) = [a \wedge b_0, a \wedge b_1, \ldots, a \wedge b_{n-1}] \tag{30}$$

$$mulBit : \mathbb{B}^1 \times \mathbb{B}^n \to \mathbb{B}^n \tag{31}$$
$$mulBit(a\ ,\ B) = if(a, B, 0) \tag{32}$$

We will now prove that equation 29 satisfies equation 28.

Proof. Induct on n.

$$\begin{aligned}
&\mathbb{B}^n toN(mulBit(A, [b_0, b_1, \ldots, b_{n-1}])) \\
&= if(A \wedge b_0, 1, 0) + 2\mathbb{B}^{n-1}toN(mulBit(A, [b_1, b_2, \ldots, b_{n-1}])) \\
&= if(A \wedge b_0, 1, 0) + 2(\mathbb{B}^1 toN(A) \cdot_N \mathbb{B}^{n-1}toN([b_1, b_2, \ldots, b_{n-1}])) \\
&= if(A, 1, 0) \cdot_N if(b_0, 1, 0) + 2(\mathbb{B}^1 toN(A) \cdot_N \mathbb{B}^{n-1}toN([b_1, b_2, \ldots, b_{n-1}])) \\
&= \mathbb{B}^1 toN(A) \cdot_N \mathbb{B}^n toN(B).
\end{aligned}$$

□

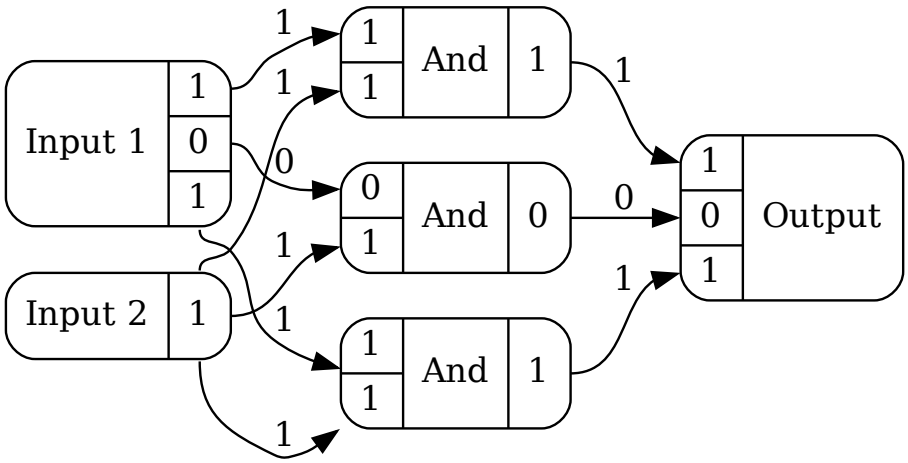

Figure 22: Diagram of a Bit Multiplier

```
mulDigit : Bit × Bits b ↛ Bits b
mulDigit {0} = and
mulDigit {suc n} = (and ⊗ mulDigit {n}) ∘ transpose ∘ first dup
```

Figure 23: A Bit Multiplier in Agda

To see an example of a bit muliplier in a diagram, look at Figure 22. In Agda, we would write this code as in Figure 23. We would write our specification as Figure 24.

For our multiplier, we also need the ability to shift over bits. Shifting provides an example of the difference between operational and denotational thinking. The operational meaning is that shifting appends n 0-bits to the end of a number. The denotational meaning is that shifting multiplies a binary representation by 2^n. We would write the denotational specification in Agda as Figure 25. Using this specification, we would then implement a shifter in Agda as Figure 26. We will leave the proof of correctness of Figure 26 as an exercise to the reader (hint: induct on the amount of shifting).

Using a bit multiplier, we can define a **shift and add** multiplier. See Table 3 for an example of shift-and-add multiplication.

In mathematical notation, we would write an add-and-shift multiplier as Equation 33.

```
mulDigitSpec : LittleEndianToNat {b} ∘ mulDigit {b} ≈ mul ∘ (bitToNat ⊗ LittleEndianToNat {b})
```

Figure 24: A Bit Multiplier Specification in Agda

```
«spec : (a b : ℕ) → LittleEndianToNat {b + a} ∘ « {a} {b} ≈ (2^ b) ∘ LittleEndianToNat {a}
«spec = {! !}
```

Figure 25: The Denotational Specifcation for Shifting

```
« : Bits a ⇾ Bits (b + a)
« {b = 0} = id
« {b = suc b} = (const bit0) ∘ « {b = b}
```

Figure 26: An Implmentation of Shift in Agda

				1	0	1	1
			×	1	1	1	0
				0	0	0	0
			1	0	1	1	
		1	0	1	1		
+	1	0	1	1			
1	0	0	1	1	0	1	0

Table 3: An Example shift-and-add multiplication

$$\text{For all } b_0, b_1, \ldots, b_{n-1} \quad A \in \mathbb{B}^m \quad [b_0, b_1, \ldots, b_{n-1}] \cdot_{\mathbb{B}^{n,m}} A = mulBit(b_0, A) + ([b_1, \ldots, b_{n-1}] \cdot_{\mathbb{B}^{n-1,m}} A) \ll \tag{33}$$

In Agda, we would write a shift-and-add multiplier as Figure 27. We will now proof Equation 33 satisfies Equation 27.

Proof. We will induct on n.

$$\begin{aligned}
&\mathbb{B}^{m+n} toN([b_0, b_1, \ldots, b_{n-1}] \cdot_{\mathbb{B}^{n,m}} [a_0, a_1, \ldots, a_{m-1}]) \\
&= \mathbb{B}^m toN(mulBit(b_0, [a_0, a_1, \ldots, a_{m-1}])) + 2\mathbb{B}^{m+n-1} toN([, b_1, \ldots, b_{n-1}] \cdot_{\mathbb{B}^{n,m}} [a_0, a_1, \ldots, a_{m-1}]) \\
&= \mathbb{B}^1 toN(b_0)\mathbb{B}^m toN(A) + 2\mathbb{B}^{m+n-1} toN([, b_1, \ldots, b_{n-1}] \cdot_{\mathbb{B}^{n,m}} [a_0, a_1, \ldots, a_{m-1}]) \\
&= \mathbb{B}^1 toN(b_0)\mathbb{B}^m toN(A) + 2\mathbb{B}^{n-1}([a_1, \ldots, a_{n-1}])\mathbb{B}^m toN(A) \\
&= \mathbb{B}^n(B)\mathbb{B}^m toN(A)
\end{aligned}$$

□

```
shiftAndAdd : Bits a × Bits b ↛ Bits (suc (a + b))
shiftAndAdd {0} {b} = («' {0} {suc b} ∘ bit0 ∘ !)
shiftAndAdd {suc a} {b} = RippleAdd {suc a + b} ∘ (« {b} {suc a} ∘ mulDigit {b} ⊗ shiftAndAdd {a} {b})
  ∘ transpose ∘ second dup
```

Figure 27: A Shift-And-Add Multiplier in Agda

7 Future Work

We will now describe future work for the technique of Denotational Design. There are of course binary subtractors and binary dividers to specificy and prove their correctness. One interesting avenue is to prove the correctness of fast adders such as the Brent-Kung adder. Such specification is of greater value because Brent-Kung adders are in use in actual hardware, and the correctness of parallel adders is less apparent.

Acknowledgements

The author would like to ackowledge Conal Elliott for his invaluable feedback and insight. The author would like to thank Simon Rubenstein-Salzedo for organizing the Euler Circle. The author would like to thank Abhy Devalapura for his wisdom and advice.

References

[1] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In *Theorem proving in higher order logics. 22nd international conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 73–78. Berlin: Springer, 2009.

[2] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN international conference on functional programming, ICFP '97, Amsterdam, Netherlands, June 9–11, 1997*, pages 263–273. New York, NY: Association for Computing Machinery (ACM), 1997.

[3] Atticus Kuhn. Parallel algorithms. https://github.com/AtticusKuhn/parallel-algorithms, 2023.

[4] Saunders Mac Lane. *Categories for the working mathematician.*, volume 5 of *Grad. Texts Math.* New York, NY: Springer, 2nd ed edition, 1998.