Design and Proof of Multipliers by Correctness-Preserving Transformation*

Carlos Delgado Kloos
Depto. Ingeniería de Sistemas Telemáticos
Universidad Politécnica de Madrid
email: cdk@dit.upm.es

Walter Dosch Bernhard Möller
Institut für Mathematik
Universität Augsburg
email: {dosch,moeller}@uni-augsburg.de

Abstract

Transformational development allows one to design systems and simultaneously prove them correct. We present transformational developments of multiplier circuits from a common specification. Careful choice of the notation (a functional language with polymorphic and dependent higher-order (sub)types) and of the foundations for the transformations (some lemmas over the data domains, embeddings of functions into more general ones, and use of the unfold/fold strategy) allow highlighting the design decisions in a systematic way.

1 Introduction

In the area of software engineering, program design is understood as a development process in which a problem-oriented specification ("what is to be done") is stepwise refined to an efficient algorithm ("how is it done"). In the rigorous approach of transformational programming the final program is correct by construction, since the transitions between subsequent versions follow formal, semantics-preserving rules. Thus, transformational development methodologically integrates two processes, viz. the *design* and the *correctness proof* of a program.

In the present paper we employ the transformational technique to derive descriptions of digital multipliers in a formal way from their common specification. The aim in this paper is not the development of new multipliers, but showing how the basic parallel and serial versions can systematically be derived by transformational reasoning. To this end, we view a synchronous digital circuit as a "frozen" algorithm, the primitive components of which are realized by electronic devices. The formal derivation of trustworthy circuits also gives insight into the algorithmic principles underlying the hardware design. In particular, the derivation disentangles the design decisions concerning

- the representation of numbers by digit sequences;
- the use of fixed word lengths;
- the effects of binary coding;
- the tradeoff between space and time.

For lack of space we cannot show the complete transformational developments. Rather, we exhibit the major design decisions and give important intermediate versions of the algorithms arising during the derivation. In the Appendix one sample development is presented in more detail.

*This research has been partially carried out in the project "Modelado funcional de sistemas distribuidos / Funktionale Modellierung verteilter Systeme" within the programme "Acciones integradas hispano-alemanas", supported by the Spanish CICYT and the German DAAD. The help of M. Russling in preparing the manuscript is gratefully acknowledged.

2 Formalizing the Problem

In this section we specify multiplication as an operation on a concrete representation of natural numbers by digit sequences.

2.1 The Notation

We use concepts from classical mathematics and modern functional languages such as Miranda or Standard ML as well as from FunMath ([1], [5]). In particular, we employ higher-order, polymorphic, and dependent types, including subtypes.

A higher-order type is a function type in which at least one of the component types (for the domain and/or the argument) is again a function type. In a dependent type, some component type depends on a value from another type. In a polymorphic type, some component type depends on another type. Dependent types capture many intuitive notions prop-

erly, as pointed out in [4]. A subtype denotes a subset of the set associated with its supertype. The subtype inherits all operations from its supertype, although it may not be closed under them.

For every object we specify its type, using the membership operation from set theory, and its value, by a defining expression. For functions not used in prefix notation we indicate the places of the operands with squares (□). The boxes may contain numbers that relate the arguments in the application of such a function to their types in the function declaration.

Dependent types may be introduced in two different ways. When using Curry-notation, the polymorphic identity function is defined as

$\forall A \in \tau.\ \mathsf{id} \in\ A \to\ A$

$\mathsf{id}\, x = x$

where τ is the universe of all types. In applying id, the type A need not be mentioned explicitly; the function symbol is thus overloaded. Alternatively we may use Church-notation

$\mathsf{id}_{\square} \square\ \in\ \tau \ni A \to\ A \to\ A$

$\mathsf{id}_A\, x = x$

where the type A is an argument of the defined function.

2.2 Data Structures

The set of natural numbers is $\mathcal{N} = \{0, 1, 2, \ldots\}$. We are concerned with the multiplication of natural numbers represented as sequences of digits in the *radix* (or *positional*) number system over a *basis* $p \geq 2$. The constructor for the type of p-adic digits is

$\aleph\ \in\ \mathcal{N}\ \to\ \mathcal{P}\mathcal{N}$

$\aleph\ p = \{n \in \mathcal{N}.\ 0 \leq n < p\}$

where $\mathcal{P}$ is the powerset constructor. $\aleph\ 10$ is the set of *decimal digits* and $\aleph\ 2$ the set of *binary digits* or *bits*, which we identify with the boolean values.

We define *sequences* or *vectors* of a fixed length $k \in \mathcal{N}$ as functions from the interval $\aleph\ k$ into the component type A; hence indexing a sequence is just function application. The constructor for the type of vectors

$\square^{\square} \in\ \tau\ \to\ \mathcal{N}\ \to\ \tau$

$A^k = (\aleph\ k) \to\ A$

is simply denoted by superscripting. By this definition A^0 is a singleton set, since $\aleph\ 0 = \emptyset$; its only element $\diamond$ plays the rôle of the empty sequence. In denoting sequences we distinguish the respective identifiers by vector arrows on top.

Functions are extended elementwise from component types to vector types. E.g., for dyadic functions we define:

$\forall A, B, C \in \tau.^{\boxed{1}}_{\boxed{2}} \in\ \mathcal{N} \ni k\ \to$
$\quad (A \to B \to C) \to (A^k \to B^k \to C^k)$

$\overset{k}{f}\, \vec{s}\, \vec{t}\, i = f(\vec{s}\, i)(\vec{t}\, i)$

Considering an element $a \in A$ as a nullary function, we denote by $\overset{k}{a}$ the sequence of k elements a.

2.3 Representation

In using sequences of digits as representations for natural numbers one has to distinguish the *low* index carrying the least significant digit. We decide this to be 0. The operation $\vec{s} \triangleright a$ appends the element a to the sequence $\vec{s}$ at the low end. Concatenation of sequences $\vec{s}$ and $\vec{t}$ placing $\vec{s}$ at the high and $\vec{t}$ at the low side is denoted by $\vec{s} \rhd \vec{t}$. By $\vec{s}_{i:j}$ we denote the subsequence of $\vec{s}$ that results from restricting $\vec{s}$ to the interval from i inclusively to j exclusively and shifting the indices to start with 0.

The function code converts a bounded natural number into a vector of k digits. It uses the operations $\div$ of integer division and $\dagger$ of integer remainder:

$\forall p \geq 2.\ \mathsf{code}_{\square}\, \square\ \in\ \mathcal{N} \ni k \to \aleph(p^k) \to (\aleph\ p)^k$

$\mathsf{code}_0\, 0 = \diamond$

$\mathsf{code}_{k+1}\, n = (\mathsf{code}_k\, (n \div p)) \triangleright (n \dagger p)$

If necessary, the representation is filled up with zeros at the high side. The type of code depends on two values $p, k \in \mathcal{N}$. We have chosen Curry-notation for p and Church-notation for k, in order to have p implicit and k explicit in each function application.

Conversely, the function deco converts a k digit sequence $\vec{s}$ into the natural number $\Sigma i \in (\aleph\ k).\ (\vec{s}\, i) \cdot p^i$ following Horner's scheme of polynomial evaluation:

$\forall p \geq 2.\ \mathsf{deco}_{\square}\, \square\ \in\ \mathcal{N} \ni k \to\ (\aleph\ p)^k \to\ \aleph(p^k)$

$\mathsf{deco}_0\, \diamond = 0$

$\mathsf{deco}_{k+1}\, (\vec{s} \triangleright x) = (\mathsf{deco}_k\, \vec{s}) \cdot p + x$

Do not confuse $(\aleph\ p)^k$ with $\aleph(p^k)$; a superscript indicates arithmetic exponentiation if the base is a number and a vector type if the base is a type.

2.4 Specification

Let $\square \cdot \square\ \in\ \mathcal{N} \to \mathcal{N} \to \mathcal{N}$ be the multiplication function for natural numbers. From it we want to develop a version for numbers of bounded size:

$\forall p \geq 2.\ {}^{\boxed{3}}_{\boxed{1}}\mathsf{mult}_{\boxed{2}}^{\boxed{4}}\ \in\ \mathcal{N} \ni k \to \mathcal{N} \ni l \to$
$\quad \aleph(p^k) \to \aleph(p^l) \to \aleph(p^{k+l})$

$x\ {}_k\mathsf{mult}_l\ y = x \cdot y$

Within the bounds, the result coincides with the usual multiplication. The bounds on the arguments provide important assertions and allow considerable simplification. During the transformational development the multiplication $\cdot$ over the naturals may still occur as an

auxiliary operation which will be eliminated at the end of the development.

The function mult then induces a multiplier function vmult on digit vectors:

$\forall p \geq 2.\ \boxed{3}\,{}_{\boxed{1}}\mathsf{vmult}_{\boxed{2}}\,\boxed{4} \in \mathcal{N} \ni k \to \mathcal{N} \ni l \to (\aleph\ p)^k \to (\aleph\ p)^l \to (\aleph\ p)^{k+l}$

$\vec{x}\ {}_k\mathsf{vmult}_l\ \vec{y} = \mathsf{code}_{k+l}\,((\mathsf{deco}_k\ \vec{x})\ {}_k\mathsf{mult}_l\,(\mathsf{deco}_l\ \vec{y}))$

For the following developments we need an adder function vadd on digit vectors, specified analogously:

$\forall p \geq 2.\ \boxed{3}\,{}_{\boxed{1}}\mathsf{vadd}_{\boxed{2}}\,\boxed{4} \in \mathcal{N} \ni k \to \mathcal{N} \ni l \to (\aleph\ p)^k \to (\aleph\ p)^l \to (\aleph\ p)^{max(k,l)+1}$

$\vec{x}\ {}_k\mathsf{vadd}_l\ \vec{y} = \mathsf{code}_{max(k,l)+1}\,((\mathsf{deco}_k\ \vec{x}) + (\mathsf{deco}_l\ \vec{y}))$

For transformational developments of adders see for instance [2] or [3].

We also need a function digmu multiplying a vector with a digit:

$\forall p \geq 2.\ \boxed{2}\,{}_{\boxed{1}}\mathsf{digmu}\,\boxed{3} \in \mathcal{N} \ni k \to (\aleph\ p)^k \to \aleph\ p \to (\aleph\ p)^{k+1}$

$\vec{x}\ {}_k\mathsf{digmu}\ a = \mathsf{code}_{k+1}\,((\mathsf{deco}_k\ \vec{x})\cdot a)$

3 First Transformation Steps

In this section we collect important laws for the further derivations and obtain a first directly recursive version of the multiplication function.

3.1 Algebraic Laws

Transformational development extensively uses the algebraic properties of the underlying data structures. Here, these are the natural numbers and digit sequences. We start with a few fundamental properties of integer division and remainder:

Lemma 1 $\forall p \geq 1.\ \forall x, y \in \mathcal{N}.$

1. $y = (y \div p)\cdot p + (y \dagger p)$
2. $x \dagger p = 0 \Rightarrow$
   $(x + y) \dagger p = y \dagger p$
   $(x + y) \div p = x \div p + y \div p$

Next we give some properties of $\div$ and $\dagger$ with respect to powers p^k of the basis p of the number system used:

Lemma 2 $\forall p \geq 1.\ \forall x, m, n \in \mathcal{N}.$

1. $(x \div p^m) \div p^n = x \div p^{m+n}$
2. $(x \dagger p^m) \dagger p^n = x \dagger p^{min(m,n)}$
3. $(x \dagger p^m) \div p^n = (x \div p^n) \dagger p^{max(0,m-n)}$
4. $(x \div p^m) \dagger p^n = (x \dagger p^{m+n}) \div p^m$

Note that 4. is a special case of 3. (read from right to left).

The operations code_k and deco_k are inverse mappings:

Lemma 3 $\forall p \geq 2.\ \forall k \in \mathcal{N}.$

1. $\forall n \in \aleph(p^k).\ \mathsf{deco}_k\,(\mathsf{code}_k\ n) = n$
2. $\forall \vec{s} \in (\aleph\ p)^k.\ \mathsf{code}_k\,(\mathsf{deco}_k\ \vec{s}) = \vec{s}$

Moreover, we have the decomposition properties

Lemma 4 $\forall p \geq 2.\ \forall k, l \in \mathcal{N}.$

1. $\forall x \in \aleph(p^k).\ \forall y \in \aleph(p^l).$
   $\mathsf{code}_{k+l}\,(x\cdot p^l + y) = (\mathsf{code}_k\ x) \rhd (\mathsf{code}_l\ y)$
2. $\forall \vec{s} \in \aleph(p)^k.\ \forall \vec{t} \in \aleph(p)^l.$
   $\mathsf{deco}_{k+l}\,(\vec{s} \rhd \vec{t}) = (\mathsf{deco}_k\ \vec{s})\cdot p^l + (\mathsf{deco}_l\ \vec{t})$

3.2 Multiplication by Repeated Addition

As a first major design decision, we implement multiplication by successive additions. We use the above algebraic properties to transform the function mult from Section 2.4. First we *unfold*, i.e., we substitute an application of mult by its instantiated body. Now we perform a case analysis on the argument l. If $l = 0$, we immediately obtain

$x\ {}_k\mathsf{mult}_0\ y = 0$

Otherwise we apply the divide-and-conquer strategy for the multiplicator y. Since we aim at a logarithmic complexity, we use the decomposition property from Lemma 1.1, obtaining

$x\cdot y = x\cdot(y \div p)\cdot p + x\cdot(y \dagger p)$

Now we can *fold* (i.e., replace an expression by a suitable function application) with the definition of mult and obtain the recursion equation

$x\ {}_k\mathsf{mult}_{l+1}\ y = (x\ {}_k\mathsf{mult}_l\,(y \div p))\cdot p + x\cdot(y \dagger p)$

An initial call $x\ {}_k\mathsf{mult}_l\ y$ leads to exactly l recursive calls. Note that this recursion does not terminate early but continues adding even when the repeated divisions lead to trivial summands only.

4 The Parallel Multiplier

As a case study in the design of a combinational circuit, we now develop a function describing a parallel multiplier.

4.1 Product Generation Digit by Digit

The function for the parallel multiplier has to describe a recursion into the breadth of the network. In every incarnation it should yield one additional digit of the product. Thus, we head for a linear but non-tail recursion where the pending operation consists in appending a product digit. To this end, we introduce the generalization

$\forall p \geq 2.\ \boxed{3}\,{}_{\boxed{1}}\mathsf{pmul}_{\boxed{2}}\,\boxed{4}\,\boxed{5} \in \mathcal{N} \ni k \to \mathcal{N} \ni l \to \aleph(p^k) \to \aleph(p^l) \to \aleph(p^k) \to \aleph(p^{k+l})$

$x\ {}_k\mathsf{pmul}_l\ y\ z = (x\ {}_k\mathsf{mult}_l\ y) + z$

of mult where the additional parameter z accumulates the partial sums. In particular, with

$x\ {}_k\mathsf{pmul}_l\ y\ 0 = x\ {}_k\mathsf{mult}_l\ y$

we regain the original task. Using unfold/fold transformations and algebraic simplifications we obtain the direct recursion

$x\ {}_k\mathsf{pmul}_0\ y\ z = z$

$x\ {}_k\mathsf{pmul}_{l+1}\ y\ z =$

$\quad (x\ {}_k\mathsf{pmul}_l\ (y \div p)\ (r \div p)) \cdot p + (r \dagger p)$

where $r \in \aleph(p^{k+1})$

$\quad r = x \cdot (y \dagger p) + z$

4.2 Multiplication of Digit Vectors

Analogously to Section 2.4, we may convert the function pmul to work over digit vectors:

$\forall p \geq 2.\ {}^{3}_{1}\mathsf{vpmul}_{2}{}^{4}\ {}^{5} \in \mathcal{N} \ni k \to \mathcal{N} \ni l \to$

$\quad (\aleph\ p)^k \to (\aleph\ p)^l \to (\aleph\ p)^k \to (\aleph\ p)^{k+l}$

$\vec{x}\ {}_k\mathsf{vpmul}_0\ \vec{y}\ \vec{z} = \vec{z}$

$\vec{x}\ {}_k\mathsf{vpmul}_{l+1}\ (\vec{y} \triangleright y)\ \vec{z} = (\vec{x}\ {}_k\mathsf{vpmul}_l\ \vec{y}\ \vec{r}) \triangleright r$

where $\vec{r} \triangleright r \in (\aleph\ p)^{k+1}$

$\quad \overset{1}{0} \triangleright (\vec{r} \triangleright r) = (\vec{x}\ {}_k\mathsf{digmu}\ y)\ {}_{k+1}\mathsf{vadd}_k\ \vec{z}$

Note that this non-tail-recursive function generates the product digit by digit starting with the least significant position.

4.3 Specialization to Bit Vectors

For technological reasons, in digital electronics presently the binary number system is taken. Therefore we specialize our development for $p = 2$. This allows us to implement the formation of a partial sum by a multiple and-gate with the least significant digit of the multiplicator:

$\vec{x}\ {}_k\mathsf{digmu}\ a = \overset{1}{0} \triangleright (\vec{x} \overset{k}{\wedge} \overset{k}{a})$

The leading zero allows simplification of the vector addition resulting in

$\vec{x}\ {}_k\mathsf{vpmul}_0\ \vec{y}\ \vec{z} = \vec{z}$

$\vec{x}\ {}_k\mathsf{vpmul}_{l+1}\ (\vec{y} \triangleright y)\ \vec{z} = (\vec{x}\ {}_k\mathsf{vpmul}_l\ \vec{y}\ \vec{r}) \triangleright r$

where $\vec{r} \triangleright r \in (\aleph\ p)^{k+1}$

$\quad \vec{r} \triangleright r = (\vec{x} \overset{k}{\wedge} \overset{k}{y})\ {}_k\mathsf{vadd}_k\ \vec{z}$

When visualizing this function as a digital circuit, we get a recursion into the breadth of the network, where every building block adds up one partial sum to the product, see Fig. 1.

In the i-th incarnation, the i-th partial sum with the multiplicator digit $\vec{y}\,i$ is formed, added to the current intermediate product, the least significant digit of which is the i-th result digit.

The function vpmul terminates after exactly l calls, since the size of the multiplicator is bounded by l. We can therefore completely unwind the recursion which results in a purely combinational network.

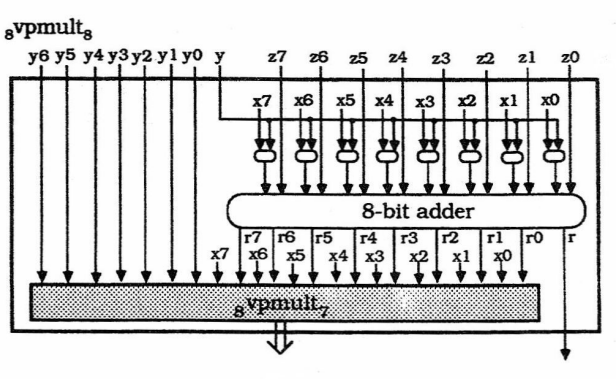

Figure 1: Parallel multiplier as a recursive network

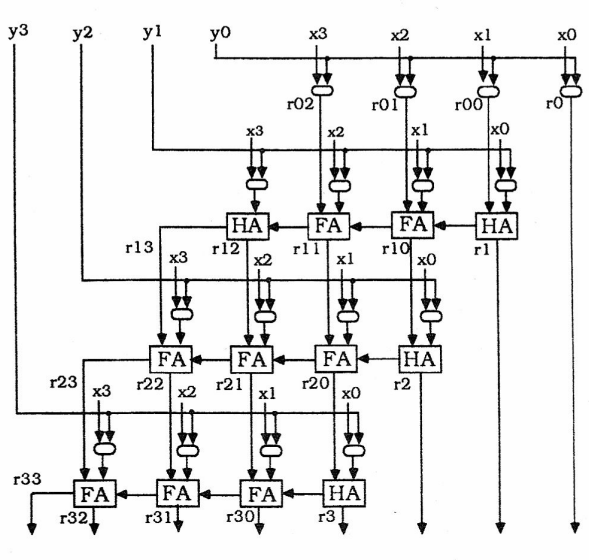

Figure 2: 4-bit parallel multiplier

For example, for $k = l = 4$ we obtain the 4-bit parallel multiplier:

$\vec{x}\ {}_4\mathsf{vpmul}_4\ \vec{y}\ \overset{4}{0} = \vec{r}_3 \triangleright r_3 \triangleright r_2 \triangleright r_1 \triangleright r_0$

where $\vec{r}_3 \triangleright r_3 = (\vec{x} \overset{4}{\wedge} \overbrace{\vec{y}3}^{4})\ {}_4\mathsf{vadd}_4\ \vec{r}_2$

where $\vec{r}_2 \triangleright r_2 = (\vec{x} \overset{4}{\wedge} \overbrace{\vec{y}2}^{4})\ {}_4\mathsf{vadd}_4\ \vec{r}_1$

where $\vec{r}_1 \triangleright r_1 = (\vec{x} \overset{4}{\wedge} \overbrace{\vec{y}1}^{4})\ {}_4\mathsf{vadd}_4\ \vec{r}_0$

where $\vec{r}_0 \triangleright r_0 = (\vec{x} \overset{4}{\wedge} \overbrace{\vec{y}0}^{4})\ {}_4\mathsf{vadd}_4\ \overset{4}{0}$

As the argument $\vec{z}$ is $\overset{4}{0}$, there are many additions with 0 that can be eliminated upon expanding vadd. The accordingly simplified network is shown in Fig. 2.

Observe that the recursive function leads to a regularly structured network by cascading the addition of the partial sums. This systematics is no longer obvious when the recursion is unrolled into a non-recursive expression. It is completely lost if one attempts to minimize the resulting boolean expression. This shows that recursion is a fundamental concept for designing regularly structured networks.

5 The Serial Multiplier

In this section we derive the circuit for the serial multiplier from the specification. This time we head for a tail recursion corresponding to an imperative program on register variables. Finally, we again specialize the result to binary digits.

5.1 Tail-Recursive Solution

We start again from the recursion for mult in Section 3.2. To transform it to tail recursion, we introduce an accumulator z for the partial sums together with a shift factor i for the product:

$$\forall p \geq 2.\ {}^{4}_{1}\mathsf{mul}^{\ 3}_{2\ 5\ 6} \in \mathcal{N} \ni k \to \mathcal{N} \ni l \to \mathcal{N} \ni i \to \aleph(p^k) \to \aleph(p^l) \to \aleph(p^{k+i}) \to \aleph(p^{k+l+i})$$

$$x\ {}_k\mathsf{mul}^i_l\ y\ z = (x\ {}_k\mathsf{mult}_l\ y)\cdot p^i + z$$

The original problem can be retrieved by the specialization

$$x\ {}_k\mathsf{mul}^0_l\ y\ 0 = x\ {}_k\mathsf{mult}_l\ y$$

We can derive a direct recursion for pmul using again unfold/fold transformations and algebraic simplifications, obtaining

$$x\ {}_k\mathsf{mul}^i_0\ y\ z = z$$

$$x\ {}_k\mathsf{mul}^i_{l+1}\ y\ z = x\ {}_k\mathsf{mul}^{i+1}_l\ (y \div p)\ (x\cdot(y \dagger p)\cdot p^i + z)$$

5.2 Merging Parameters

A multiplier mimicking the behaviour of mul would use three registers for the parameters x, y, and z. However, the number of digits needed to represent both y and z is invariantly $k+l+i$. So we can merge y and z into a single parameter q which will fit into a register of size $k+l+i$. At each stage the lower l digits of q represent y while the remaining ones represent z. Formally, this can be introduced using the generalization

$$\forall p \geq 2.\ {}^{4}_{1}\mathsf{mu}^{\ 3}_{2\ 5} \in \mathcal{N} \ni k \to \mathcal{N} \ni l \to \mathcal{N} \ni i \to \aleph(p^k) \to \aleph(p^{k+l+i}) \to \aleph(p^{k+l+i})$$

$$x\ {}_k\mathsf{mu}^i_l\ q = x\ {}_k\mathsf{mul}^i_l\ (q \dagger p^l)\ (q \div p^l)$$

With the specialization

$$y \in \aleph(p^l) \Rightarrow x\ {}_k\mathsf{mu}^0_l\ y = x\ {}_k\mathsf{mult}_l\ y$$

we retrieve the original problem.

Applying again unfold/fold transformations and algebraic simplifications, notably those from Lemma 2, we derive the direct recursion

$$x\ {}_k\mathsf{mu}^i_0\ q = q$$

$$x\ {}_k\mathsf{mu}^i_{l+1}\ q = x\ {}_k\mathsf{mu}^{i+1}_l\ ((x\cdot(q \dagger p)\cdot p^{i+l+1} + q) \div p)$$

5.3 Multiplication of Digit Vectors

According to the specification in Section 2.3, we now transfer the multiplication function mu from natural numbers to digit vectors. The induced function vmu is specified by

$$\forall p \geq 2.\ {}^{4}_{1}\mathsf{vmu}^{\ 3}_{2\ 5} \in \mathcal{N} \ni k \to \mathcal{N} \ni l \to \mathcal{N} \ni i \to (\aleph\ p)^k \to (\aleph\ p)^{k+l+i} \to (\aleph\ p)^{k+l+i}$$

$$\vec{x}\ {}_k\mathsf{vmu}^i_l\ \vec{q} = \mathsf{code}_{k+l+i}\ ((\mathsf{deco}_k\ \vec{x})\ {}_k\mathsf{mu}^i_l\ (\mathsf{deco}_{k+l+i}\ \vec{q}))$$

Employing again unfold/fold transformations and algebraic simplifications, in particular using Lemma 4, we obtain:

$$\vec{x}\ {}_k\mathsf{vmu}^i_0\ \vec{q} = \vec{q}$$

$$\vec{x}\ {}_k\mathsf{vmu}^i_{l+1}\ (\vec{q} \triangleright q) = \vec{x}\ {}_k\mathsf{vmu}^{i+1}_l\ ((\vec{x}\ {}_k\mathsf{digmu}\ q)\ {}_{k+1}\mathsf{vadd}_k\ \vec{q}_{i+l:k+i+l}) \triangleright \vec{q}_{0:i+l}$$

5.4 Specialization to Bit Vectors

Analogously to Section 4.3, we now specialize the function vmu to binary digits. Although the description in functional notation already perfectly describes the circuit, we transform it further into imperative form. Performing at the same time a backward substitution of the embedding calls we obtain

$\vec{x}\ {}_k\mathsf{vmult}_l\ \vec{y} =$
$\mathsf{begin\ var}\ MD \in (\aleph\ p)^k; \mathsf{var}\ AC \in (\aleph\ p)^{k+l};$
$MD := \vec{x};\ AC := \overset{k}{0} \triangleright \vec{y};$
$\mathsf{from}\ l\ \mathsf{downto}\ 1\ \mathsf{do}$
$AC := ((MD \overset{k}{\wedge} \overbrace{AC\ 0}^{k})\ {}_k\mathsf{vadd}_k\ AC_{l:k+l}) \triangleright AC_{1:l};$
AC
end

For $k = l = 4$ we obtain the 4-bit serial multiplier the block diagram of which is shown in Fig. 3.

In the circuit, the program variables MD and AC for binary words of lengths k and $k+l$ correspond to the (static) multiplicand register and the accumulator. Initially, the multiplicand is loaded into the MD-register and the multiplicator into the right half of the accumulator. In each execution of the loop body the bits of the multiplicand are *and*ed with the least significant bit of the multiplicator and the result is added to the left part of the accumulator. After that, the accumulator is shifted to the right, dragging a possible

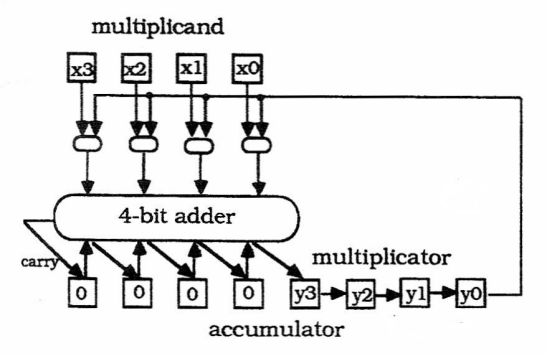

Figure 3: 4-bit serial multiplier, initial stage

carry into the most significant position. The network is operated by a control (not shown in Fig. 3) corresponding to the for-loop in the algorithm.

In summary, the network of the serial multiplier has essentially been obtained by the transition from general linear recursion to iteration. This allows the reuse of the circuit in different loop executions while keeping the entire network static.

6 Conclusion

We have used a *formal language* (based on a functional style notation and the concepts of polymorphic and dependent higher-order (sub)types) together with a *formal method* (that of transformational development) in order to describe, design, and prove circuits correct.

In this case study concerning multipliers, we essentially have used properties of the data domain combined with unfold/fold transformations. The most creative part of the inductive reasoning consists in finding the right embeddings. When aiming at a serial circuit, the goal is to obtain a tail-recursive function, whereas for a parallel combinational network one needs a pending operation in the recursion.

With this case study, others already done and more to come, we want to show the appropriateness of the transformational technique for hardware design. Note that it even handles parametrized systems without additional effort.

Appendix: A Sample Derivation

To show more details of the transformational technique, we give the derivation steps for the development of the function mu from the function mul in Section 5.2. We proceed by induction on l. It is straightforward to see that $x \,{}_k\mathsf{mu}_0^i\, q = 0$. Moreover

$$
\begin{aligned}
& x \,{}_k\mathsf{mu}_{l+1}^i\, q \\
= \;& \{\text{unfolding of } \mathsf{mu}\} \\
& x \,{}_k\mathsf{mul}_{l+1}^i\, (q \dagger p^{l+1})(q \div p^{l+1}) \\
= \;& \{\text{unfolding of } \mathsf{mul}\} \\
& x \,{}_k\mathsf{mul}_{l}^{i+1}\, ((q \dagger p^{l+1}) \div p) \\
& \qquad (x \cdot ((q \dagger p^{l+1}) \dagger p) \cdot p^i + (q \div p^{l+1})) \\
= \;& \{\text{Lemma 2}\} \\
& x \,{}_k\mathsf{mul}_{l}^{i+1}\, ((q \div p) \dagger p^l) \\
& \qquad (x \cdot (q \dagger p) \cdot p^i + (q \div p^{l+1})) \\
= \;& \{\div \text{ is inverse to } \cdot\} \\
& x \,{}_k\mathsf{mul}_{l}^{i+1}\, ((q \div p) \dagger p^l) \\
& \quad (((x \cdot (q \dagger p) \cdot p^{i+l+1}) \div p^{l+1}) + (q \div p^{l+1})) \\
= \;& \{\text{Lemma 1}\} \\
& x \,{}_k\mathsf{mul}_{l}^{i+1}\, ((q \div p) \dagger p^l) \\
& \qquad ((x \cdot (q \dagger p) \cdot p^{i+l+1} + q) \div p^{l+1}) \\
= \;& \{\text{Lemma 2}\} \\
& x \,{}_k\mathsf{mul}_{l}^{i+1}\, ((q \div p) \dagger p^l) \\
& \qquad (((x \cdot (q \dagger p) \cdot p^{i+l+1} + q) \div p) \div p^l) \\
= \;& \{\text{Lemmas 1 and 2}\} \\
& x \,{}_k\mathsf{mul}_{l}^{i+1}\, (((x \cdot (q \dagger p) \cdot p^{i+l+1} + q) \div p) \dagger p^l) \\
& \qquad (((x \cdot (q \dagger p) \cdot p^{i+l+1} + q) \div p) \div p^l) \\
= \;& \{\text{folding of } \mathsf{mu}\} \\
& x \,{}_k\mathsf{mu}_{l}^{i+1}\, ((x \cdot (q \dagger p) \cdot p^{i+l+1} + q) \div p)
\end{aligned}
$$

References

[1] R.T. Boute: *Declarative languages — still a long way to go*, in: D. Borrione, R. Waxman (eds.): *Computer hardware description languages and their applications*, Proc. IFIP WG 10.2 Symposium, Marseille, 22–24 April 1991, 165–192

[2] C. Delgado Kloos, W. Dosch: *Transformational development of circuit descriptions for binary adders*, in: M. Wirsing, M. Broy (eds.): *Methods of programming, LNCS* **544**, Berlin: Springer 1991, 217–237

[3] C. Delgado Kloos, W. Dosch: *Efficient circuits as implementations of non-strict functions*, in: G. Jones, M. Sheeran (eds.): *Designing correct circuits*, Proc. Workshop, Oxford, Sept. 1990, London: Springer 1991, 212–230

[4] F.K. Hanna, N. Daeche, M. Longley: *Specification and verification using dependent types*, IEEE Trans. Software Eng. **16**:9, 1990, 949–964

[5] Mark Seutter (ed.): *Glass: A system description language and its environment — Introduction and user manual*, ESPRIT project 881 Forfun, May 1990