Designing Arithmetic Circuits by Refinement in Ruby

Geraint Jones[1] and Mary Sheeran[2]

[1] Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, England; electronic mail: Geraint.Jones@comlab.oxford.ac.uk.
[2] Informationsbehandling, Chalmers Tekniska Högskola, S 412 96 Göteborg, Sweden; electronic mail: ms@cs.chalmers.se.

Abstract. This paper presents in some detail the systematic derivation of a static bit-level parallel algorithm to implement multiplication of integers, that is to say one which might be implemented as an electronic circuit. The circuit is well known, but the derivation shows that its design can be seen as the consequence of decisions made (and explained) in terms of the abstract algorithm. The systematic derivation serves both as an explanation of the circuit, and as a demonstration that it is correct 'by construction'. We believe that the technique is applicable to a wide range of similar algorithms.

1 Introduction

We advocate a style of 'design by calculation' for the very fine-grained parallel algorithms that are implemented as regular arrays of electronic circuits. The design of such circuits is particularly difficult because the implementation medium imposes severe constraints on what is possible and what is reasonably efficient. In consequence the details of the final implementation have a pervasive influence on the whole design process, and it is regrettable that many such designs are presented with little or no abstract justification, and solely in the low-level terms of the final detailed implementation.

These same constraints, it seems to us, make it easier to apply the systematic design methods which are being used in the development of software algorithms. Our work is much influenced by that of Bird and Meertens [2, 13], although in contrast to their use of functions we choose to use relations as our abstract programs. This same choice has been made by the authors of [1] and there are close parallels [8] between the framework in which they work and ours.

We represent by relations both abstract specifications of the desired behaviour of circuits and the components from which circuits are built. Ruby [10] is a language of operations on relations in which the combining forms correspond naturally to the ways that components are connected in our target implementations. The algebra of these combining forms is a framework in which to refine the specification of an algorithm into an implementable form.

The notion of an abstraction relation is important, as is the composition of an abstraction with the inverse of another abstraction. This is a representation changer: a device to consume one representation of some abstraction, and return another representation. In this paper we show that a multiplication circuit can be specified naturally as such a representation changer, and that the specification can be refined into a regular array of components each of which is also a representation changer. These small representation changers are the sorts of devices which would be found in a hardware designer's library of standard components, and are the parts from which the circuit would be built. The multiplier is not new; we first presented it much more informally in 1985 [14]. Here we present its derivation as a new account of how such a circuit might be designed.

As well as explaining how a design came about, a derivation constitutes a proof that the implementation meets its specification. Usually such proofs are constructed after the circuit has been designed [16, 4], and independently of the method used to produce the design [18]. With that approach the design process offers no help in the proof, nor does the verification offer any assistance in the design. Even when circuits are synthesised mechanically from parametric designs, after-the-fact verification of the synthesis functions [5] gives no guidance on their construction.

2 Ruby

We model the behaviour of circuits by binary relations, denoted in Ruby by expressions which have a form that suggests the shapes of the corresponding circuits as well as their behaviour. In this paper we confine our attention to combinational circuits, operating on simple data values. Lifting to time sequences of values, to describe the behaviour of circuits with internal state, is also a part of the interpretation of the Ruby expression. In this way we our combinational designs can be transformed into very high throughput systolic designs [14], but this lifting is not covered in this paper.

2.1 Basic Structuring Functions

The most important structuring functions are relational composition and inverse (or converse):

$$\begin{aligned} a\,(R\,;S)\,c \;&\Leftrightarrow\; \exists b.\, a\,R\,b \;\&\; b\,S\,c \\ a\,R^{-1}\,b \;&\Leftrightarrow\; b\,R\,a \end{aligned}$$

Repeated composition is denoted by exponentiation: $R^1 = R$ and $R^{n+1} = R^n \,;R$. These operators have many simply expressed properties:

$$\begin{aligned} R\,;(S\,;T) &= (R\,;S)\,;T \\ (R^{-1})^{-1} &= R \\ (R\,;S)^{-1} &= S^{-1}\,;R^{-1} \\ (R^n)^{-1} &= (R^{-1})^n \end{aligned}$$

and it is these kinds of properties that we use in deriving implementations from the specification of a required behaviour.

The pictures in this paper are drawn to a convention in which the domain of a relation is on the left and the range (or codomain) on the right, so that relational

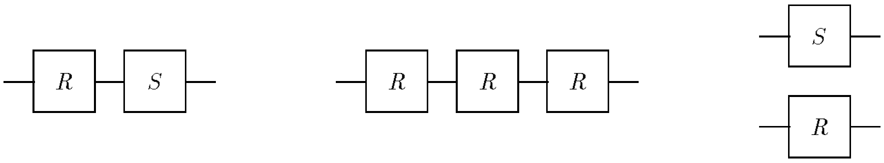

Fig. 1. $R\,;S$ and R^3 and $[R,S]$

composition is as shown in Fig. 1. Taking the inverse of a relation corresponds to flipping the picture over so that the domain and range are swapped.

The parallel composition of two circuits $[R,S]$ relates a pair of values in the domain to a pair of values in the range if and only if the components are related by the corresponding component circuits, $\langle a,b\rangle\,[R,S]\,\langle x,y\rangle \;\Leftrightarrow\; a\,R\,x \;\&\; b\,S\,y$. The identity relation id is defined by $x\,id\,y \;\Leftrightarrow\; x=y$. The functions fst and snd abbreviate frequently used instances of parallel composition: $\mathsf{fst}\,R = [R, id]$ and $\mathsf{snd}\,S = [id, S]$. Among the properties of parallel composition that we will use are that

$$\begin{aligned}
[R,S]^{-1} &= [R^{-1}, S^{-1}] \\
[R,S]\,;[T,U] &= [R\,;T, S\,;U] \\
(\mathsf{fst}\,R)^{-1} &= \mathsf{fst}(R^{-1}) \\
\mathsf{fst}\,R\,;\mathsf{fst}\,S &= \mathsf{fst}(R\,;S) \\
[R,S] &= \mathsf{fst}\,R\,;\mathsf{snd}\,S \\
&= \mathsf{snd}\,S\,;\mathsf{fst}\,R
\end{aligned}$$

2.2 Some Primitive Relations

In addition to the full identity relation id, we introduce identities on finite sets of values, including subranges of the integers. For example, $[\![0,1,2]\!]$ is the identity on the set $\{0,1,2\}$. Constant relations are identities on singleton sets. The relation $[\![a\,..\,b]\!]$ is the identity on integers in the range a to b inclusive, $x\,[\![a\,..\,b]\!]\,y \;\Leftrightarrow\; x = y \;\&\; a \le x \le b$.

The following primitive relations are useful building blocks:

$$\begin{aligned}
\langle a,b\rangle\;\pi_1\;c &\;\Leftrightarrow\; a = c \\
\langle a,b\rangle\;\pi_2\;c &\;\Leftrightarrow\; b = c \\
\langle a,b\rangle\;swp\;\langle c,d\rangle &\;\Leftrightarrow\; a = d \;\&\; b = c \\
a\;fork\;\langle b,c\rangle &\;\Leftrightarrow\; a = b = c
\end{aligned}$$

Note that the inverse of a projection is a perfectly good relation, which will occur often in our calculations: a component of its *range* is unconstrained. For the arithmetic circuits in this paper we will use:

$$\begin{aligned}
\langle a,b\rangle + c &\;\Leftrightarrow\; a + b = c \\
\langle a,b\rangle \times c &\;\Leftrightarrow\; a \times b = c
\end{aligned}$$

Note that the symbols $+$ and $\times$ are being used as the names of relations. For binary arithmetic, two other useful components are the relation $\times 2$ that relates an integer x to the even integer $2x$, and $s+ = \mathsf{snd} \times 2 \,;\, +$.

2.3 Measuring the Domain and Range of a Relation

Despite the misleading notation, beware that $R^{-1} \,;\, R$ is not always an identity relation. If f is a function from domain to range, then $f^{-1} \,;\, f$ is the identity on the range of f, that is $f = f \,;\, f^{-1} \,;\, f$. For more general relations, however, we can only assume that R is a subrelation of $R \,;\, R^{-1} \,;\, R$. That is, $R^{-1} \,;\, R$ may be bigger than the identity on the range of R. For example, the relation $+^{-1} \,;\, +$ is the identity on the integers (written int), but $+ \,;\, +^{-1}$ relates a given pair of integers to every other pair of integers that has the same sum.

It is convenient to introduce notations for the forms $R^{-1} \,;\, R$ and $R \,;\, R^{-1}$:

$$R{>} = R^{-1} \,;\, R$$
$${<}R = R \,;\, R^{-1}$$

which we read as 'R right' and 'left R' respectively. It can be shown that $R{>} = (R{>})^{-1} = {<}(R^{-1}) = ({<}(R^{-1}))^{-1}$. Some examples: $fork{>}$ is the identity on pairs whose elements are equal; $swp{>} = [id, id]$ is the identity on all pairs.

In calculations, we often make and use assertions of the form $A = A \,;\, B$, so we introduce another abbreviation: $A \vdash B \;\Leftrightarrow\; A = A \,;\, B$, which we read as '$A$ guarantees B (on the right)'. Think of $A \vdash B$ as an assertion about the range of A. Some useful properties of $\vdash$ are

$$B \vdash C \;\Rightarrow\; A \,;\, B \vdash C$$
$$B \vdash C \,;\, D \;\Rightarrow\; B \,;\, C \vdash D \,;\, C$$
$$B \vdash C \,;\, D \;\&\; D \,;\, E \vdash F \;\Rightarrow\; B \,;\, E \vdash F$$

Similarly, $\dashv$ (guarantees on the left) captures assertions about the domain of a relation: $A \dashv B \;\Leftrightarrow\; A \,;\, B = B$. Because $A \dashv B \;\Leftrightarrow\; B^{-1} \vdash A^{-1}$, the properties of $\dashv$ are dual to those of the right-handed operator.

2.4 Types

Data types in Ruby are partial equivalence relations ($pers$), that is, relations P that satisfy $P \vdash P^{-1}$. If A is a per and $A \dashv R$, then we say that R has (a possible) domain type A. Similarly, if B is a per and $R \vdash B$, say that R has range type B. A relation may have many such domain and range types. We write $R : A \sim B$ as an abbreviation for $A \dashv R \,\&\, R \vdash B$. It is convenient to have data types that are relations because then types can be manipulated and reasoned about in just the same way as any other Ruby relation. The use of pers as types been developed independently by Voermans [8].

The largest type is the full relation any for which $x \; any \; y$ for any x and y. For example, the assertion $\mathsf{snd}\; any \dashv \pi_1$ expresses the fact that the second component of the domain value of the selector π_1 is not constrained in any way: it is not connected to anything by the selector.

We are now in a position to consider what we mean by R^0. For $R : T \sim T$, R^0 must also have type $T \sim T$, and it must be both a left type and a right type of R. Accordingly, we choose R^0 to be the type T' for which $R : T' \sim T'$ and $R : T \sim T \;\Rightarrow\; T' : T \sim T$. This can be shown to be well-defined [11].

2.5 Lists

We collect data values into lists, and have structuring functions which construct relations to operate on them. The repeated parallel composition $\mathsf{map}_n\, R$ relates two n-lists if their corresponding elements are related by R, that is

$$x\,(\mathsf{map}_n\, R)\, y \;\Leftrightarrow\; \#x = \#y = n \;\;\&\;\; \forall i.\, x_i\, R\, y_i$$

Triangle, another pointwise operator, relates the ith elements of two n-lists by R^i.

$$x\,(\mathsf{tri}_n\, R)\, y \;\Leftrightarrow\; \#x = \#y = n \;\;\&\;\; \forall i.\, x_i\, R^i\, y_i$$

For example, $\mathsf{map}_n \times 2$ doubles each element of a list of n integers, while $\mathsf{tri}_n \times 2$ relates the ith element of the list, x_i, to $x_i 2^i$, the relation $(\times 2)^0$ being the identity on the integers. Figure 2 shows instances of map and triangle.

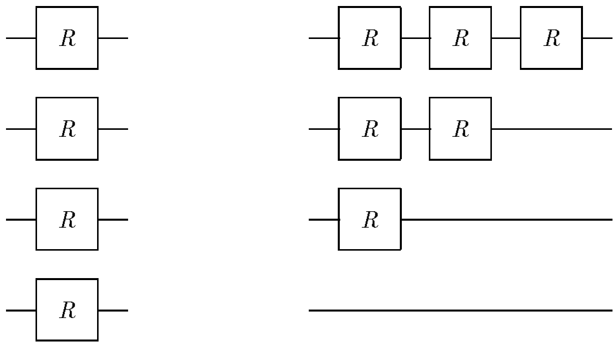

Fig. 2. $\mathsf{map}_4\, R$ and $\mathsf{tri}_4\, R$

The relation zip interleaves a pair of lists to give a list of pairs: $\langle x, y\rangle\; zip_n\; z \;\Leftrightarrow\; \#x = \#y = \#z = n \;\;\&\;\; \forall i.\, \langle x_i, y_i\rangle = z_i$. It satisfies

$$[\mathsf{tri}_n\, A, \mathsf{tri}_n\, B]\;;\; zip_n \;=\; zip_n\;;\; \mathsf{tri}_n [A, B]$$
$$[\mathsf{map}_n\, A, \mathsf{map}_n\, B]\;;\; zip_n \;=\; zip_n\;;\; \mathsf{map}_n [A, B]$$

2.6 Circuits Connected in Two Dimensions

To broaden the range of circuit forms that can be described, we introduce a new convention for pictures of pair-to-pair relations: they can be viewed as four-sided tiles, with the domain on the left and top edges, and the range on the bottom and right edges. This ordering of the labels is chosen to be consistent with the labelling of circuits drawn under the earlier linear convention.

The structuring function $\leftrightarrow$ (read 'beside') places two four-sided tiles side by side as shown in Fig. 3. The resulting relation is again pair-to-pair.

$$\langle a, \langle b, c\rangle\rangle \; R \leftrightarrow S \; \langle\langle d, e\rangle, f\rangle \;\Leftrightarrow\; \exists g.\, \langle a, b\rangle \; R \; \langle d, g\rangle \;\;\&\;\; \langle g, c\rangle \; S \; \langle e, f\rangle$$

Composing further relations around the edges of a beside can be done in various ways:

$$[A, [B, C]] \,;\, (R \leftrightarrow S) \,;\, [[D, E], F] = ([A, B] \,;\, R \,;\, \mathsf{fst}\, D) \leftrightarrow (\mathsf{snd}\, C \,;\, S \,;\, [E, F])$$

and a relation on the arc between R and S can be bracketed with R or with S.

$$(R \,;\, \mathsf{snd}\, A) \leftrightarrow S \;=\; R \leftrightarrow (\mathsf{fst}\, A \,;\, S)$$

Perhaps the simplest pair-to-pair relation is $[id, id]$. Placing two of these side by side gives $rsh = [id, id] \leftrightarrow [id, id]$, which is also shown in Fig. 3. From the properties of beside, it follows that

$$[A, [B, C]] \,;\, rsh \;=\; rsh \,;\, [[A, B], C]$$

which we will call 'right-shifting' of $[A, [B, C]]$.

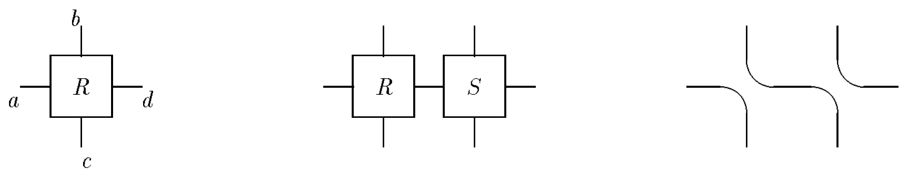

Fig. 3. $\langle a, b\rangle \; R \; \langle c, d\rangle$ and layouts suggested by $R \leftrightarrow S$ and rsh

Just as map generalises parallel composition, we can generalise the beside operator by an operator that connects a horizontal array of pair-to-pair relations:

$$\begin{aligned} &\langle a, x\rangle \; (\mathsf{row}_n\; R) \; \langle y, b\rangle \\ &\quad = \#x = \#y = n \;\;\& \\ &\qquad\quad \exists z.\, \#z = n + 1 \;\;\&\;\; a = z_0 \;\;\&\;\; (\forall i.\, \langle z_i, x_i\rangle \; R \; \langle y_i, z_{i+1}\rangle) \;\;\&\;\; z_n = b \end{aligned}$$

If $R : [A, B] \sim [C, A]$ then $\mathsf{row}_{n+1}\, R : [A, \mathsf{map}_{n+1}\, B] \sim [\mathsf{map}_{n+1}\, C, A]$, although the corresponding result does not hold for row_0. It seems that there is no 'obviously right' meaning for the row of zero width, but we have chosen here that $\langle a, b\rangle \, (\mathsf{row}_0\, R) \, \langle c, d\rangle$ if and only if $a = d$ and $b = c = \langle\,\rangle$, that is $\mathsf{row}_0\, R = \mathsf{snd}\; \mathsf{map}_0\; id \,;\, swp$.

Two properties of row used often in calculations are

$$\mathsf{snd\ map}_n\ R\ ;\ \mathsf{row}_n\ S\ ;\ \mathsf{fst\ map}_n\ T = \mathsf{row}_n(\mathsf{snd}\ R\ ;\ S\ ;\ \mathsf{fst}\ T)$$

which we call 'pushing maps into row' and

$$\mathsf{fst}\ R\ ;\ S = T\ ;\ \mathsf{snd}\ R \Rightarrow \mathsf{fst}\ R\ ;\ \mathsf{row}_{n+1}\ S\ =\ \mathsf{row}_{n+1}\ T\ ;\ \mathsf{snd}\ R$$

which is 'row induction'. Again there seems to be no right way of dealing with the empty row, which we will deal with separately if necessary.

Complex wiring relations can be built from rows of simple cells: for example, the relation $\mathsf{row}_n(\mathit{fork}{>} \leftrightarrow \mathit{swp})$ can be customised to give some useful standard wiring relations. The relation that distributes a value across a list to give a list of pairs, is

$$\begin{aligned} \mathit{dstl}_n &= \mathsf{row}_n(\mathsf{snd}\ \pi_2^{-1}\ ;\ (\mathit{fork}{>} \leftrightarrow \mathit{swp}))\ ;\ \pi_1 \\ &= \mathsf{row}_n(\mathit{fork}\ ;\ \mathsf{snd}\ \pi_1)\ ;\ \pi_1 \end{aligned}$$

From the properties of row and of fork it can be shown that $[A, \mathsf{map}_n\ B]\ ;\ \mathit{dstl}_n\ = \mathit{dstl}_n\ ;\ \mathsf{map}_n[A, B]$. Figure 4 shows an instance of dstl.

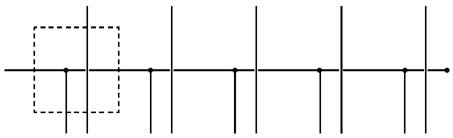

Fig. 4. $\mathit{dstl}_5\ =\ \mathsf{row}_5(\mathit{fork}\ ;\ \mathsf{snd}\ \pi_1)\ ;\ \pi_1$

Since taking the inverse of a pair-to-pair relation corresponds to flipping its picture about the bottom-left to top-right diagonal, vertical patterns can be made by flipping, making horizontal patterns and then restoring the original orientation. We define $\updownarrow$ (read 'below') and col as duals of $\leftrightarrow$ and row.

$$\begin{aligned} R \updownarrow S &= (R^{-1} \leftrightarrow S^{-1})^{-1} \\ \mathsf{col}_n\ R &= (\mathsf{row}_n\ R^{-1})^{-1} \end{aligned}$$

The properties of below and column can be derived from those of beside and row. The dual of rsh is $\mathit{lsh} = \mathit{rsh}^{-1} = [\mathit{id}, \mathit{id}] \updownarrow [\mathit{id}, \mathit{id}]$. The inverse of $\mathsf{row}_n(\mathit{fork}{>} \leftrightarrow \mathit{swp})$ is $\mathsf{col}_n(\mathit{fork}{>} \updownarrow \mathit{swp})$ and we can make a 'distribute right' wiring relation from this.

$$\begin{aligned} \mathit{dstr}_n &= \mathsf{col}_n(\mathsf{fst}\ \pi_1^{-1}\ ;\ (\mathit{fork}{>} \updownarrow \mathit{swp}))\ ;\ \pi_2 \\ &= \mathsf{col}_n(\mathit{fork}\ ;\ \mathsf{fst}\ \pi_2)\ ;\ \pi_2 \end{aligned}$$

Not all of the relations that we want to connect in these kinds of patterns are pair-to-pair. We use degenerate cases of the structuring functions already introduced to cope with those relations that do not quite have the right numbers of connections.

For example, to make a continued version of a pair-to-value relation, we use a version of row that has some of its arcs removed.

$$\mathsf{rdl}_n\, R = \mathsf{row}_n(R\,;\pi_2^{-1})\,;\pi_2$$

This operator is familiar from functional programming, where it is often called 'reduce left' or 'fold'. For example, to sum n integers:

$$sum_n = \pi_2^{-1}\,;\mathsf{fst}\,[\![0]\!]\,;\mathsf{rdl}_n\,+$$

The $[\![0]\!]$ constrains the first component of the domain of the reduction to be zero, and the inverse projection hides this from the domain of sum.

3 Abstraction and Representation

An abstraction relates concrete values to the abstract values that they represent. We say that abs is an abstraction if $abs \vdash abs{>}$. If the assertion $abs \vdash abs{>}$ holds, then $abs{>}$ is guaranteed to be a type. We call this range type of abs the abstract type. Similarly ${<}abs$ is guaranteed to be a type, and this domain type of abs is called the concrete type. This is really only a matter of convenience as this notion of abstraction is symmetric: if abs is an abstraction, so is abs^{-1}. (An earlier paper [9] of ours on refinement in Ruby incorrectly denies this.) The relations which we call abstractions are those that are called *difunctionals* [8]: all functions are abstractions, as are all inverses of functions [15].

Since we want to design arithmetic circuits, we will be considering abstractions onto the integers and subranges of the integers. The relations $+$, $\times$, $s{+}$ and sum_n are all abstractions onto the integers. The relation $\times 2$ is an abstraction not onto the integers but onto the *even* integers, that is the type $(\times 2){>}$.

In the abstraction sum_n, each element of the list of integers in the domain has the same weight. We can give the elements of a list weights that are increasing powers of two by using a triangle of $\times 2$ components. The resulting relation, which we call a *ladder*, is also an abstraction onto the integers.

$$ladder_n = \mathsf{tri}_n \times 2\,; sum_n$$

We will be constructing circuits which operate on 'voltage levels', which are the values of the type $sig = [\![high, low]\!]$. The abstraction bi, for 'bit to integer', relates abstract voltages to numbers in the set $\{0, 1\}$.

$$\begin{aligned} bi &= \{(high, 1), (low, 0)\} \\ bi{>} &= [\![0, 1]\!] \\ {<}bi &= sig \end{aligned}$$

Similarly, we can represent the numbers 0 and -1 by signals. We call the resulting 'negative bit' a *nit*, and the abstraction is called ni for 'negative bit to integer'.

$$\begin{aligned} ni &= \{(high, -1), (low, 0)\} \\ ni{>} &= [\![-1, 0]\!] \\ {<}ni &= sig \end{aligned}$$

We can also represent numbers by pairs of signals in various ways. A pair of bits of the same weight can represent numbers in $\{0,1,2\}$. This abstraction is often used so we give it the name $bb{+}$.

$$\begin{aligned} bb{+} &= [bi, bi] \,;\, + \\ (bb{+}){>} &= [\![0,1,2]\!] \end{aligned}$$

This is a redundant representation; the number 1 is represented both by the pair $\langle high, low\rangle$ and by the pair $\langle low, high\rangle$. This means that ${<}(bb{+}) \neq [sig, sig]$, but rather it is the partial equivalence relation described by

$$\begin{aligned} {<}(bb{+}) = \{ & (\langle low, low\rangle, \langle low, low\rangle), \quad (\langle high, high\rangle, \langle high, high\rangle), \\ & (\langle high, low\rangle, \langle high, low\rangle), (\langle high, low\rangle, \langle low, high\rangle), \\ & (\langle low, high\rangle, \langle low, high\rangle), (\langle low, high\rangle, \langle high, low\rangle) \quad \} \end{aligned}$$

This is the concrete type of pairs of equal weight bits.

A pair of bits, one of weight 1 and one of weight 2, can represent numbers in $\{0,1,2,3\}$. The corresponding abstraction relation is $[bi, bi] \,;\, s{+}$. We call this a 'bit step' and abbreviate it to $bbs{+}$.

$$\begin{aligned} bbs{+} &= [bi, bi] \,;\, s{+} \\ (bbs{+}){>} &= [\![0,1,2,3]\!] \\ {<}(bbs{+}) &= [sig, sig] \end{aligned}$$

A nit and a bit, each of weight one, can represent numbers in $\{-1,0,1\}$. This is a standard abstraction in arithmetic circuit design, and is usually called a *trit*.

$$\begin{aligned} ti &= [ni, bi] \,;\, + \\ ti{>} &= [\![-1,0,1]\!] \end{aligned}$$

Again, ${<}ti$ is a partial equivalence relation that is bigger than $[sig, sig]$. This is because the integer 0 is represented both by the pair $\langle high, high\rangle$ and by the pair $\langle low, low\rangle$. (We can write that assertion as $ti \,;\, [\![0]\!] = fork^{-1} \,;\, sig \,;\, any \,;\, [\![0]\!]$.)

If we restrict the elements of the lists in the domain of a ladder to be bits, we get $\mathsf{map}_n\ bi \,;\, ladder_n$, which is the well known n-bit binary least-significant bit at the left representation of numbers in the range 0 to $2^n - 1$. The form $\mathsf{map}_n\ abs \,;\, ladder_n$, where abs is an abstraction, appears so often that we abbreviate it to $\mathsf{L}_n\ abs$. We think of the n-bit binary abstraction as a 'ladder of bits'.

$$\begin{aligned} (\mathsf{L}_n\ bi){>} &= [\![0\,..\,2^n - 1]\!] \\ &\quad \text{where } \mathsf{L}_n\ abs = \mathsf{map}_n\ abs \,;\, ladder_n \end{aligned}$$

This is the type of numbers that can be represented as n-bit binary numerals. Because the abstraction is one-to-one, ${<}(\mathsf{L}_n\ bi) = \mathsf{map}_n\ sig$.

The carry-save abstraction is a ladder of pairs of equal weight bits, which we can write as $\mathsf{L}_n\ bb{+}$.

$$(\mathsf{L}_n\ bb{+}){>} = [\![0\,..\,2(2^n - 1)]\!]$$

Carry-save is a redundant abstraction both because $bb+$ is redundant, and because the value that can be represented at each step of the ladder is not constrained to be smaller than the radix (which is two).

A number of useful transformations of these abstractions can be derived by induction on the lengths of the ladders, starting from the properties of the underlying arithmetic. The sum of two equal-length ladders can be written as a ladder of sums, provided the two concrete lists are first interleaved (or zipped together):

$$[\mathsf{L}_n\, a, \mathsf{L}_n\, b] \,;+ = zip_n \,;\, \mathsf{L}_n\,([a, b]\,;+) \tag{1}$$

because of the associativity and commutativity of addition. Similarly, the product of an abstraction and a ladder is a ladder of products:

$$[a, \mathsf{L}_n\, b] \,;\times = dstl_n \,;\, \mathsf{L}_n\,([a, b]\,;\times) \tag{2}$$

$$[\mathsf{L}_n\, a, b] \,;\times = dstr_n \,;\, \mathsf{L}_n\,([a, b]\,;\times) \tag{3}$$

because multiplication distributes over addition.

To push abstractions around in circuit descriptions, we use instances of the theorems about the structuring functions, for example by row induction

$$\mathsf{fst}\, a \,;\, R \vdash \mathsf{snd}\, a{>} \quad \Rightarrow \quad \mathsf{fst}\, a \,;\, \mathsf{row}_n\, R = \mathsf{row}_n(\mathsf{fst}\, a \,;\, R \,;\, \mathsf{snd}\, a^{-1}) \,;\, \mathsf{snd}\, a$$

3.1 Representation Changers

Many design problems can naturally be cast as the implementation of representation changers. This seems particularly to be the case for arithmetic circuits. The specification of a representation changer is the composition of an abstraction relation – which relates a given concrete representation to its abstract meaning – and the inverse of another abstraction relation – which relates that abstract value to the desired concrete value. If a and b are abstractions, then $a \,;\, b^{-1}$ relates the type ${<}a$ to the type ${<}b$. Provided $a{>} = b{>}$ the translation is complete and faithful in both directions; if they differ it is faithful only between those representations which correspond to abstract values which are in both types.

We choose to think of our circuits as representation changers because simple representation changers can often be implemented directly in hardware. For example, $[bi, bi] \,;\times;\, bi^{-1}$ is just an *and* gate: that is, it is a relation between a pair of signals and a signal which relates $\langle high, high\rangle$ to $high$ and any other pair to low. Similarly, $bb+ \,;\, bbs+^{-1}$ is a standard component in the digital designer's repertoire: it is a 'half-adder', thought of as a function from two input bits to a sum and carry pair. That is, it performs addition of two one-bit binary numerals, producing a two-bit answer. It can readily be implemented in any digital circuit technology: for example as an and-gate and an exclusive-or gate, or as an eight-bit read-only store.

The representation changer $[bi, ni] \,;\, s+ \,;\, ti^{-1}$ can also be implemented by a half-adder in which the carry flows from range to domain. It is a half-adder rotated through ninety degrees, so that the domain is a pair consisting of one input and the carry out, and the range is the pair of the other input and the sum output. Notice that we abstracted away from the distinction between inputs and outputs by choosing to represent circuits by relations.

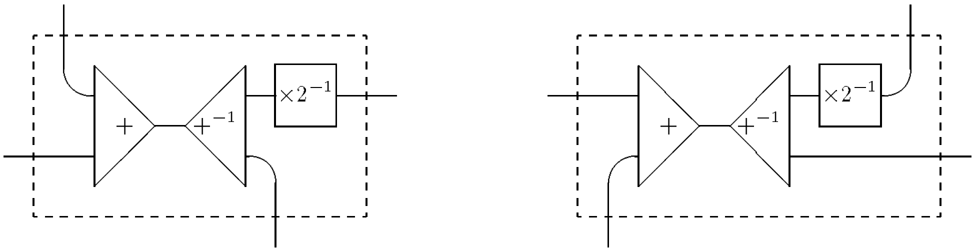

Fig. 5. a representation of a half-adder, $bb+ \,;\, bbs+^{-1}$, and of one its rotations which is equivalent to $[bi, ni] \,;\, s+ \,;\, ti^{-1}$

A 'full-adder' relates a bit and a pair of equal weight bits to a sum and carry pair, so it is $[bi, bb+] \,;\, + \,;\, bbs+^{-1}$. Just as we did for a half-adder, we can implement the representation changer $[bb+, ni] \,;\, s+ \,;\, ti^{-1}$ by a full-adder rotated so that the carry out is in the domain and the carry in is in the range. The easiest way to check such assertions about small relations is to enumerate the values which they relate.

3.2 Refinement

Representation changers that cannot immediately be implemented as hardware primitives must be re-expressed as networks of smaller representation changers. These smaller changers can then either be implemented directly in hardware or rewritten as networks of yet smaller changers of representation. This refinement proceeds hierarchically, and stops when we reach cells that we know how to implement directly in hardware.

We will need to know how to reorganise large changers of representation into networks of smaller ones. Because we want to refine to regular array circuits, we concentrate on ways of calculating with rows and columns. We have previously shown [9], using the Ruby equivalent of Horner's rule [3] for polynomial evaluation, that

$$\mathsf{snd}\, ladder_n \,;\, + \,;\, +^{-1} \,;\, [ladder_n, \times 2^n]^{-1} = \mathsf{row}_n(+ \,;\, s+^{-1}) \tag{4}$$

This allows us to calculate that for any a and b

$$\begin{aligned}
&\mathsf{snd}(\mathsf{L}_n\, a) \,;\, + \,;\, +^{-1} \,;\, [\mathsf{L}_n\, b, \times 2^n]^{-1} \\
=&\qquad \{\text{definition of } \mathsf{L}_n\} \\
&\mathsf{snd}(\mathsf{map}_n\, a \,;\, ladder_n) \,;\, + \,;\, +^{-1} \,;\, [(\mathsf{map}_n\, b \,;\, ladder_n), \times 2^n]^{-1} \\
=&\qquad \{\text{properties of inverse}\} \\
&\mathsf{snd}\,\mathsf{map}_n\, a \,;\, \mathsf{snd}\, ladder_n \,;\, + \,;\, +^{-1} \,;\, [ladder_n, \times 2^n]^{-1} \,;\, \mathsf{fst}\,\mathsf{map}_n\, b^{-1} \\
=&\qquad \{\text{ equation 4}\} \\
&\mathsf{snd}\,\mathsf{map}_n\, a \,;\, \mathsf{row}_n(+ \,;\, s+^{-1}) \,;\, \mathsf{fst}\,\mathsf{map}_n\, b^{-1} \\
=&\qquad \{\text{pushing maps into row}\} \\
&\mathsf{row}_n(\mathsf{snd}\, a \,;\, + \,;\, s+^{-1} \,;\, \mathsf{fst}\, b^{-1})
\end{aligned} \tag{5}$$

showing that a representation changer which relates a carry in and a ladder of as to a ladder of the same number of bs and a carry out can be implemented by a row of components each which relates one a and one b, connected by a carry chain.

Similarly for any a and b

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{snd}(\mathsf{L}_n\, a) \,;\, s{+} \,;\, +^{-1} \,;\, [\mathsf{L}_n\, b, \times 2^n]^{-1}} \\
= & \{\text{definition of } s{+} \text{ and } (\mathsf{L}_n\, a) \,;\, \times 2 = \mathsf{L}_n(a \,;\, \times 2)\,\} \\
& \mathsf{snd}(\mathsf{L}_n(a \,;\, \times 2)) \,;\, + \,;\, +^{-1} \,;\, [\mathsf{L}_n\, b, \times 2^n]^{-1} \\
= & \{\text{equation } 5\} \\
& \mathsf{row}_n(\mathsf{snd}(a \,;\, \times 2) \,;\, + \,;\, s{+}^{-1} \,;\, \mathsf{fst}\, b^{-1}) \\
= & \{\text{definition of } s{+}\} \\
& \mathsf{row}_n(\mathsf{snd}\, a \,;\, s{+} \,;\, s{+}^{-1} \,;\, \mathsf{fst}\, b^{-1})
\end{array} \tag{6}
$$

Taking inverses on both sides of (5) and (6) and renaming variables gives

$$[\mathsf{L}_n\, a, \times 2^n] \,;\, + \,;\, +^{-1} \,;\, \mathsf{snd}(\mathsf{L}_n\, b)^{-1} = \mathsf{col}_n(\mathsf{fst}\, a \,;\, s{+} \,;\, +^{-1} \,;\, \mathsf{snd}\, b^{-1}) \tag{7}$$

and

$$[\mathsf{L}_n\, a, \times 2^n] \,;\, + \,;\, s{+}^{-1} \,;\, \mathsf{snd}(\mathsf{L}_n\, b)^{-1} = \mathsf{col}_n(\mathsf{fst}\, a \,;\, s{+} \,;\, s{+}^{-1} \,;\, \mathsf{snd}\, b^{-1}) \tag{8}$$

These four equations guide our refinements, so that if we are aiming for a column of components, for instance, we try to massage the expression to match the left hand side either of (7) or of (8).

A useful property of the composition of $+$ and its inverse, which appears often in these expressions, is that

$$+ \,;\, +^{-1} = \mathsf{snd} +^{-1} \,;\, rsh \,;\, \mathsf{fst} + \tag{9}$$

The left-hand side relates $\langle a, b\rangle$ and $\langle c, d\rangle$ if $a + b = c + d$, and the right hand side similarly if there is some x for which $b = x + d$ and $a + x = c$, which is the same constraint. From this, it follows that

$$
\begin{array}{lll}
s{+} \,;\, s{+}^{-1} = & & \{\text{definition of } s{+}\} \\
& \multicolumn{2}{l}{\mathsf{snd} \times 2 \,;\, + \,;\, +^{-1} \,;\, \mathsf{snd} \times 2^{-1}} \\
& = & \{\text{equation } 9\} \\
& \multicolumn{2}{l}{\mathsf{snd}(\times 2 \,;\, +^{-1}) \,;\, rsh \,;\, [+, \times 2^{-1}]} \\
& = & \{\text{right-shifting, } rsh \,;\, \mathsf{snd} \times 2^{-1} = \mathsf{snd}\,\mathsf{snd} \times 2^{-1} \,;\, rsh\,\} \\
& \multicolumn{2}{l}{\mathsf{snd}(\times 2 \,;\, +^{-1} \,;\, \mathsf{snd} \times 2^{-1}) \,;\, rsh \,;\, \mathsf{fst} +} \\
& = & \{\text{arithmetic of even numbers}\} \\
& \multicolumn{2}{l}{\mathsf{snd}(+^{-1} \,;\, \mathsf{fst} \times 2) \,;\, rsh \,;\, \mathsf{fst} +} \\
& = & \{\text{right-shifting}\} \\
& \multicolumn{2}{l}{\mathsf{snd} +^{-1} \,;\, rsh \,;\, \mathsf{fst}(\mathsf{snd} \times 2 \,;\, \mathsf{fst} +)} \\
& = & \{\text{definition of } s{+}\} \\
& \multicolumn{2}{l}{\mathsf{snd} +^{-1} \,;\, rsh \,;\, \mathsf{fst}\, s{+}}
\end{array} \tag{10}
$$

4 Small Examples

As small examples of the method, we will develop some well-known adder circuits by refinement of representation changers.

To specify the addition of a carry-in bit to a binary number to give a binary number and a carry-out bit, the abstraction $[bi, \mathsf{L}_n\, bi]\,;+$ is composed with the inverse of $[\mathsf{L}_n\, bi, (bi\,;\times 2^n)]\,;+$. This matches the pattern of (5).

$$
\begin{aligned}
&[bi, \mathsf{L}_n\, bi]\,;+\,;+^{-1}\,;[\mathsf{L}_n\, bi, (bi\,;\times 2^n)]^{-1}\\
&\quad = \quad \{\text{properties of inverse}\}\\
&\qquad \mathsf{fst}\, bi\,;\mathsf{snd}\,\mathsf{L}_n\, bi\,;+\,;+^{-1}\,;[\mathsf{L}_n\, bi, \times 2^n]^{-1}\,;\mathsf{snd}\, bi^{-1}\\
&\quad = \quad \{\text{equation 5}\}\\
&\qquad \mathsf{fst}\, bi\,;\mathsf{row}_n(\mathsf{snd}\, bi\,;+\,;s{+}^{-1}\,;\mathsf{fst}\, bi^{-1})\,;\mathsf{snd}\, bi^{-1}
\end{aligned}
\tag{11}
$$

The next step is to push the abstractions at each end of the row ($\mathsf{fst}\, bi$ and $\mathsf{snd}\, bi^{-1}$) onto the internal arcs of the row so that the components of the row are then themselves in the form of representation changers. This is done using a row induction. Using the two lemmas (which can be checked by enumeration)

$$[bi, bi]\,;+ \vdash [\![0,1,2]\!]$$

and

$$[\![0,1,2]\!]\,;s{+}^{-1}\,;\mathsf{fst}\, bi \vdash \mathsf{snd}\, bi{>}$$

it can be shown that

$$\mathsf{fst}\, bi\,;(\mathsf{snd}\, bi\,;+\,;s{+}^{-1}\,;\mathsf{fst}\, bi^{-1}) \vdash \mathsf{snd}\, bi{>} \tag{12}$$

This is then used as the condition for a row induction.

$$
\begin{aligned}
&\mathsf{fst}\, bi\,;\mathsf{row}_n(\mathsf{snd}\, bi\,;+\,;s{+}^{-1}\,;\mathsf{fst}\, bi^{-1})\,;\mathsf{snd}\, bi^{-1}\\
&\quad = \quad \{\text{equation 12 and row induction}\}\\
&\qquad \mathsf{row}_n([bi, bi]\,;+\,;s{+}^{-1}\,;[bi, bi]^{-1})\,;\mathsf{snd}\,{<}bi\\
&\quad = \quad \{\text{definitions of } bb{+} = [bi, bi]\,;+ \text{ and } bbs{+} = [bi, bi]\,;s{+}\}\\
&\qquad \mathsf{row}_n(bb{+}\,;bbs{+}^{-1})\,;\mathsf{snd}\,{<}bi\\
&\quad = \quad \{\text{definition of } halfadd = bb{+}\,;bbs{+}^{-1} \text{ and } {<}bi = sig\}\\
&\qquad \mathsf{row}_n\, halfadd\,;\mathsf{snd}\, sig
\end{aligned}
$$

We can stop here because $bb{+}\,;bbs{+}^{-1}$ is just a half-adder. The $\mathsf{snd}\, sig$ on the right shows that if $n = 0$ the circuits consists of a single concrete wire – a thing that can carry a signal of type sig.

The derivation of a circuit to add a bit to a carry-save number to give a binary number and a carry-out is almost identical. This time the component is a full adder.

$$
\begin{aligned}
&[bi, \mathsf{L}_n\, bb{+}]\,;+\,;+^{-1}\,;[\mathsf{L}_n\, bi, (bi\,;\times 2^n)]^{-1}\\
&\quad = \quad \{\text{equation 5}\}\\
&\qquad \mathsf{fst}\, bi\,;\mathsf{row}_n(\mathsf{snd}\, bb{+}\,;+\,;s{+}^{-1}\,;\mathsf{fst}\, bi^{-1})\,;\mathsf{snd}\, bi^{-1}
\end{aligned}
$$

$$
\begin{aligned}
= \quad & \{\, \mathsf{fst}\ bi\ ;(\mathsf{snd}\ bb{+}\ ;+\ ;s{+}^{-1}\ ;\mathsf{fst}\ bi^{-1}) \vdash \mathsf{snd}\ bi{>} \text{ and induction} \,\} \\
& \mathsf{row}_n([bi, bb{+}]\ ;+\ ;s{+}^{-1}\ ;[bi, bi]^{-1})\ ;\mathsf{snd}\ {<}bi \\
= \quad & \{\, \text{definition of } bbs{+} = [bi, bi]\ ;s{+} \text{ and } {<}bi = sig \,\} \\
& \mathsf{row}_n([bi, bb{+}]\ ;+\ ;bbs{+}^{-1})\ ;\mathsf{snd}\ sig \\
= \quad & \{\, \text{definition of } fulladd = [bi, bb{+}]\ ;+\ ;bbs{+}^{-1} \,\} \\
& \mathsf{row}_n\ fulladd\ ;\mathsf{snd}\ sig
\end{aligned}
$$

In our third example, we relate a carry-save number and a top-carry that is a nit (rather than a bit) to a nit and a binary number. The process of refinement is the same as in the previous examples. We use one of (5) to (8) to introduce a row or column and then use either row or column induction to push the remaining abstractions inside the array.

$$
\begin{aligned}
& [\mathsf{L}_n\ bb{+}, (ni\ ;\times 2^n)]\ ;+\ ;+^{-1}\ ;[ni, \mathsf{L}_n\ bi]^{-1} \\
= \quad & \{\, \text{equation } 7 \,\} \\
& \mathsf{snd}\ ni\ ;\mathsf{col}_n(\mathsf{fst}\ bb{+}\ ;s{+}\ ;+^{-1}\ ;\mathsf{snd}\ bi^{-1})\ ;\mathsf{fst}\ ni^{-1} \\
= \quad & \{\, \mathsf{snd}\ ni{>} \dashv (\mathsf{fst}\ bb{+}\ ;s{+}\ ;+^{-1}\ ;\mathsf{snd}\ bi^{-1})\ ;\mathsf{fst}\ ni^{-1} \text{ and induction} \,\} \\
& \mathsf{snd}\ {<}ni\ ;\mathsf{col}_n([bb{+}, ni]\ ;s{+}\ ;+^{-1}\ ;[ni, bi]^{-1}) \\
= \quad & \{\, \text{definition of } ti = [ni, bi]\ ;+ \text{ and } {<}ni = sig \,\} \\
& \mathsf{snd}\ sig\ ;\mathsf{col}_n([bb{+}, ni]\ ;s{+}\ ;ti^{-1}) \\
= \quad & \{\, \text{definition of } fulladd' = [bb{+}, ni]\ ;s{+}\ ;ti^{-1} \,\} \\
& \mathsf{snd}\ sig\ ;\mathsf{col}_n\ fulladd'
\end{aligned}
$$

The column component is a full adder with a carry that flows from range to domain, so the whole circuit has an upward-flowing carry chain, as shown in Fig. 6. In the domain, the carry-save number is an input and the nit is an output; in the range, the nit is an input and the binary number an output. This choice of inputs and outputs ensures that the full adder components are driven correctly.

5 Derivation of an Array Multiplier

In each of the above examples, refinement stopped after one iteration. In more complicated examples, one would expect to have to go down through several levels of hierarchy before reaching primitives. We now demonstrate this hierarchical approach in a substantial calculation, deriving the design of an n-bit by m-bit binary multiplier.

The initial specification of the circuit is

$$[\mathsf{L}_n\ bi, \mathsf{L}_m\ bi]\ ;\times\ ;(\mathsf{L}_{n+m}\ bi)^{-1}$$

which is to say, that it should relate a pair of an n-bit and an m-bit binary number to the $(n+m)$-bit number that represents their product. From (2) and (3) about the product of an abstraction and a ladder, we have that

$$
\begin{aligned}
[\mathsf{L}_n\ bi, \mathsf{L}_m\ bi]\ ;\times &= dstl_m\ ;\mathsf{L}_m([\mathsf{L}_n\ bi, bi]\ ;\times) \\
&= dstr_n\ ;\mathsf{L}_n([bi, \mathsf{L}_m\ bi]\ ;\times)
\end{aligned}
$$

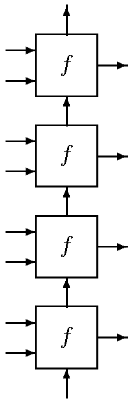

Fig. 6. $\mathsf{col}_4\ \mathit{fulladd}'$ showing constraints on data-flow in an implementation

but there seems to be no obvious next move.

Because we have a wealth of laws involving carries, we will introduce a carry in to the circuit. The carry will be represented in n-bit binary – a design decision that we have made before starting the derivation. We could have chosen a different representation, and in [9] we derive a multiplier design which arises naturally from chosing carry-save representation for the carry at this point.

To take account of the n-bit binary carry, the domain abstraction must be changed to

$$[\mathsf{L}_n\ bi, ([\mathsf{L}_n\ bi, \mathsf{L}_m\ bi]\ ;\times)]\ ;+$$

and in order to make an opportunity to apply (5) we change the range representation, so that the answer comes in two parts: the least significant m bits as a binary number, and the n most significant bits as a binary number of weight 2^m. The corresponding abstraction is

$$[\mathsf{L}_m\ bi, (\mathsf{L}_n\ bi\ ;\times 2^m)]\ ;+$$

so the specification of the circuit which we will develop is

$$\mathcal{M} = [\mathsf{L}_n\ bi, ([\mathsf{L}_n\ bi, \mathsf{L}_m\ bi]\ ;\times)]\ ;+\ ;+^{-1}\ ;[\mathsf{L}_m\ bi, (\mathsf{L}_n\ bi\ ;\times 2^m)]^{-1}$$

an interpretation of which is shown in Fig. 7.

The first step in the refinement is to rearrange the product of the two ladders of bits into a single ladder. We can then use (5) to introduce a row.

$$\begin{aligned}
\mathcal{M} &= [\mathsf{L}_n\ bi, ([\mathsf{L}_n\ bi, \mathsf{L}_m\ bi]\ ;\times)]\ ;+\ ;+^{-1}\ ;[\mathsf{L}_m\ bi, (\mathsf{L}_n\ bi\ ;\times 2^m)]^{-1} \\
&= \qquad \{\text{product of an abstraction and a ladder (equation 2)}\} \\
&\quad \mathsf{snd}\ dstl_m\ ;[\mathsf{L}_n\ bi, \mathsf{L}_m([\mathsf{L}_n\ bi, bi]\ ;\times)]\ ;+\ ;+^{-1}\ ;[\mathsf{L}_m\ bi, (\mathsf{L}_n\ bi\ ;\times 2^m)]^{-1} \\
&= \qquad \{\text{equation 5}\} \\
&\quad [\mathsf{L}_n\ bi, dstl_m]\ ;\mathsf{row}_m(\mathsf{snd}([\mathsf{L}_n\ bi, bi]\ ;\times)\ ;+\ ;s{+}^{-1}\ ;\mathsf{fst}\ bi^{-1})\ ;\mathsf{snd}(\mathsf{L}_n\ bi)^{-1}
\end{aligned}$$

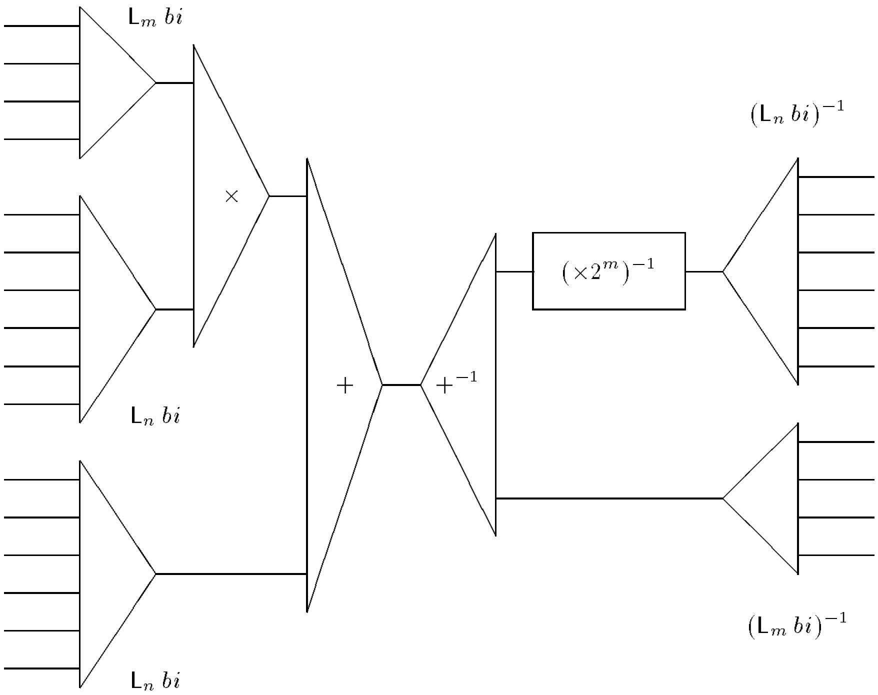

Fig. 7. $\mathcal{M} = [\mathsf{L}_n\ bi, [\mathsf{L}_n\ bi, \mathsf{L}_m\ bi] \,;\, \times] \,;\, + \,;\, +^{-1} \,;\, [\mathsf{L}_m\ bi, (\mathsf{L}_n\ bi \,;\, \times 2^m)]^{-1}$ with $m, n = 4, 6$

Next, we push the abstraction $\mathsf{L}_n\ bi$ rightwards along the row; the result is a row of representation changers. By calculating the ranges of numbers that can be represented we can show that

$$[\mathsf{L}_n\ bi, ([\mathsf{L}_n\ bi, bi] \,;\, \times)] \ \vdash\ [(\mathsf{L}_n\ bi)>, (\mathsf{L}_n\ bi)>]$$

and

$$[(\mathsf{L}_n\ bi)>, (\mathsf{L}_n\ bi)>] \,;\, + \,;\, s+^{-1} \,;\, \mathsf{fst}\ bi^{-1} \ \vdash\ \mathsf{snd}\,(\mathsf{L}_n\ bi)>$$

Combining these judgements gives

$$\mathsf{fst}(\mathsf{L}_n\ bi) \,;\, \left(\mathsf{snd}([\mathsf{L}_n\ bi, bi] \,;\, \times) \,;\, + \,;\, s+^{-1} \,;\, \mathsf{fst}\ bi^{-1}\right) \vdash \mathsf{snd}\,(\mathsf{L}_n\ bi)> \tag{13}$$

and, as in the previous examples, this can be used as the condition for a row induction.

$$\begin{aligned}
&[\mathsf{L}_n\ bi, dstl_m] \,;\, \mathsf{row}_m \left(\mathsf{snd}([\mathsf{L}_n\ bi, bi] \,;\, \times) \,;\, + \,;\, s+^{-1} \,;\, \mathsf{fst}\ bi^{-1}\right) \,;\, \mathsf{snd}(\mathsf{L}_n\ bi)^{-1} \\
&\quad = \qquad \{\,\text{row induction from equation 13}\,\} \\
&\qquad \mathsf{snd}\ dstl_m \,; \\
&\qquad\qquad \mathsf{row}_m([\mathsf{L}_n\ bi, ([\mathsf{L}_n\ bi, bi] \,;\, \times)] \,;\, + \,;\, s+^{-1} \,;\, [bi, \mathsf{L}_n\ bi]^{-1}) \,; \\
&\qquad\qquad\quad \mathsf{snd} <(\mathsf{L}_n\ bi)
\end{aligned}$$

$$
\begin{aligned}
= &\quad \{\text{naming } c_2\} \\
& \mathsf{snd}\ dstl_m \ ; \mathsf{row}_m\ c_2 \ ; \mathsf{snd} <\!(\mathsf{L}_n\ bi) \\
& \text{where } c_2 = [\mathsf{L}_n\ bi, ([\mathsf{L}_n\ bi, bi] \ ; \times)] \ ; + \ ; s{+}^{-1} \ ; [bi, \mathsf{L}_n\ bi]^{-1} \\
= &\quad \{ <\!(\mathsf{L}_n\ bi) = \mathsf{map}_n\ sig \} \\
& \mathsf{snd}\ dstl_m \ ; \mathsf{row}_m\ c_2 \ ; \mathsf{snd}\ \mathsf{map}_n\ sig
\end{aligned}
$$

So $\mathcal{M}$ can be implemented by some wiring and a row of c_2 cells; we are now free to concentrate on the design of c_2.

Because c_2 is the component of a row, it is good idea to aim to implement it as a column (rather than a row) of components. Rows of rows do not make attractive layouts, but rows of columns do. This bias towards a column influences the refinement of c_2, making it like a mirror image of the refinement so far: where we had $dstl$, we now have $dstr$; where we had a row, we now have a column.

The first steps are to rewrite $[\mathsf{L}_n\ bi, bi] \ ; \times$ as a ladder and to note that we can implement $[bi, bi] \ ; \times$ using an and gate. The resulting sum of two ladders can again be rewritten as a ladder.

$$
\begin{aligned}
c_2 &= [\mathsf{L}_n\ bi, ([\mathsf{L}_n\ bi, bi] \ ; \times)] \ ; + \ ; s{+}^{-1} \ ; [bi, \mathsf{L}_n\ bi]^{-1} \\
&= \quad \{\text{product of a ladder and an abstraction (equation 3)}\} \\
&\quad [\mathsf{L}_n\ bi, (dstr_n \ ; \mathsf{L}_n([bi, bi] \ ; \times))] \ ; + \ ; s{+}^{-1} \ ; [bi, \mathsf{L}_n\ bi]^{-1} \\
&= \quad \{ and \ ; bi = [bi, bi] \ ; \times \ ; bi^{-1} \ ; bi = [bi, bi] \ ; \times \} \\
&\quad [\mathsf{L}_n\ bi, (dstr_n \ ; \mathsf{L}_n(and \ ; bi))] \ ; + \ ; s{+}^{-1} \ ; [bi, \mathsf{L}_n\ bi]^{-1} \\
&= \quad \{\text{sum of two ladders (equation 1)}\} \\
&\quad \mathsf{snd}\ dstr_n \ ; zip_n \ ; \mathsf{L}_n([bi, and \ ; bi] \ ; +) \ ; s{+}^{-1} \ ; [bi, \mathsf{L}_n\ bi]^{-1} \\
&= \quad \{\text{definition of } bb{+} = [bi, bi] \ ; + \} \\
&\quad \mathsf{snd}\ dstr_n \ ; zip_n \ ; \mathsf{L}_n(\mathsf{snd}\ and \ ; bb{+}) \ ; s{+}^{-1} \ ; [bi, \mathsf{L}_n\ bi]^{-1}
\end{aligned}
$$

Unfortunately, the resulting pattern does not match our rule for introducing columns (8). We would like the abstraction on the left to be of the form $[\mathsf{L}_n\ a, \times 2^n];+$ but we have only a single ladder. Again we introduce a carry, this time forced to be zero, to get the required pattern. Adding zero to an integer makes no difference, and multiplying zero by 2^n gives zero, so if $a \vdash int$, it can be replaced by

$$
\begin{aligned}
a &= \pi_1^{-1} \ ; [a, [\![0]\!]] \ ; + \\
&= \pi_1^{-1} \ ; [a, ([\![0]\!] \ ; \times 2^n)] \ ; +
\end{aligned}
$$

In this case, certainly $\mathsf{L}_n(\mathsf{snd}\ and \ ; bb{+}) \vdash int$, so

$$
\begin{aligned}
& \mathsf{snd}\ dstr_n \ ; zip_n \ ; \mathsf{L}_n(\mathsf{snd}\ and \ ; bb{+}) \ ; s{+}^{-1} \ ; [bi, \mathsf{L}_n\ bi]^{-1} \\
= &\quad \{\text{zero introduction}\} \\
& \mathsf{snd}\ dstr_n \ ; zip_n \ ; \pi_1^{-1} \ ; \\
& \qquad [\mathsf{L}_n(\mathsf{snd}\ and \ ; bb{+}), ([\![0]\!] \ ; \times 2^n)] \ ; + \ ; s{+}^{-1} \ ; [bi, \mathsf{L}_n\ bi]^{-1} \\
= &\quad \{\text{equation 8}\} \\
& \mathsf{snd}\ dstr_n \ ; zip_n \ ; \pi_1{}^{-1} \ ; \mathsf{snd}\ [\![0]\!] \ ; \\
& \qquad \mathsf{col}_n(\mathsf{fst}(\mathsf{snd}\ and \ ; bb{+}) \ ; s{+} \ ; s{+}^{-1} \ ; \mathsf{snd}\ bi^{-1}) \ ; \mathsf{fst}\ bi^{-1}
\end{aligned}
$$

$$= \quad \{\text{naming } c_3\}$$
$$\mathsf{snd}\ dstr_n \ ; zip_n \ ; {\pi_1}^{-1} ; \mathsf{snd}\ [\![0]\!] \ ; \mathsf{col}_n\ c_3 \ ; \mathsf{fst}\ bi^{-1}$$
$$\text{where } c_3 = \mathsf{fst}(\mathsf{snd}\ and \ ; bb{+}) \ ; s{+} \ ; s{+}^{-1} \ ; \mathsf{snd}\ bi^{-1}$$

Now we would like to push abstractions onto the internal arcs of the column. However the most obvious strategy, of pushing bi^{-1} from right to left, does not work. It is *not* the case that $\mathsf{snd}\ bi{>} \dashv c_3 \ ; \mathsf{fst}\ bi^{-1}$. In fact since

$$\langle\langle high, \langle high, high\rangle\rangle, -1\rangle \ \ (c_3 \ ; \mathsf{fst}\ bi^{-1}) \ \ \langle low, low\rangle$$

the relevant abstract wire in the circuit can even take a negative value. However, the internal arcs are of type $ti{>} = [\![-1, 0, 1]\!]$, rather than $bi{>}$ as we had hoped.

Before pushing a trit abstraction through the column, we need to introduce $\mathsf{snd}\ ti^{-1}$ at the right. This can be done because trits and bits are related by

$$bi \ = \ {\pi_2}^{-1} \ ; [low, sig] \ ; ti$$

That is to say, a bit represents the same integer as a trit in which the wire of negative weight is held low and the positive wire is connected to the same level as the given bit.

The best way to proceed is to find a c_4 for which $c_3 \ ; \mathsf{fst}\ ti^{-1} = \mathsf{snd}\ ti^{-1} \ ; c_4$ and then use column induction.

$$\begin{aligned}
c_3 \ ; \mathsf{fst}\ ti^{-1} &= \mathsf{fst}(\mathsf{snd}\ and \ ; bb{+}) \ ; s{+} \ ; s{+}^{-1} \ ; \mathsf{snd}\ bi^{-1} \ ; \mathsf{fst}\ ti^{-1} \\
&= \quad \{ s{+} \ ; s{+}^{-1} = \mathsf{snd}\ {+}^{-1} \ ; rsh \ ; \mathsf{fst}\ s{+}, \text{ equation } 10 \} \\
&\quad \mathsf{fst}(\mathsf{snd}\ and \ ; bb{+}) \ ; \mathsf{snd}\ {+}^{-1} \ ; rsh \ ; [(s{+} \ ; ti^{-1}), bi^{-1}] \\
&= \quad \{\text{right-shifting}\} \\
&\quad \mathsf{snd}({+}^{-1} \ ; \mathsf{snd}\ bi^{-1}) \ ; rsh \ ; \mathsf{fst}(\mathsf{fst}(\mathsf{snd}\ and \ ; bb{+}) \ ; s{+} \ ; ti^{-1}) \\
&= \quad \{ \mathsf{snd}\ ni{>} \dashv \mathsf{fst}\ bb{+} \ ; s{+} \ ; ti^{-1} \} \\
&\quad \mathsf{snd}({+}^{-1} \ ; \mathsf{snd}\ bi^{-1}) \ ; rsh \ ; \mathsf{fst}([(\mathsf{snd}\ and \ ; bb{+}), ni{>}] \ ; s{+} \ ; ti^{-1}) \\
&= \quad \{\text{right-shifting, and definition of } ni{>} = ni^{-1} \ ; ni \} \\
&\quad \mathsf{snd}({+}^{-1} \ ; [ni^{-1}, bi^{-1}]) \ ; rsh \ ; \mathsf{fst}([(\mathsf{snd}\ and \ ; bb{+}), ni] \ ; s{+} \ ; ti^{-1}) \\
&= \quad \{\text{definition of } ti = [ni, bi] \ ; + \} \\
&\quad \mathsf{snd}\ ti^{-1} \ ; rsh \ ; \mathsf{fst}([(\mathsf{snd}\ and \ ; bb{+}), ni] \ ; s{+} \ ; ti^{-1}) \\
&= \quad \{ ti^{-1} \vdash \mathsf{snd}\ sig \text{ and right-shifting} \} \\
&\quad \mathsf{snd}\ ti^{-1} \ ; rsh \ ; [([(\mathsf{snd}\ and \ ; bb{+}), ni] \ ; s{+} \ ; ti^{-1}), sig] \\
&= \quad \{\text{definition of } fulladd' = [bb{+}, ni] \ ; s{+} \ ; ti^{-1} \} \\
&\quad \mathsf{snd}\ ti^{-1} \ ; rsh \ ; [(\mathsf{fst}\,\mathsf{snd}\ and \ ; fulladd'), sig] \\
&= \quad \{\text{naming } c_4\} \\
&\quad \mathsf{snd}\ ti^{-1} \ ; c_4 \\
&\quad \text{where } c_4 = rsh \ ; [(\mathsf{fst}\,\mathsf{snd}\ and \ ; fulladd'), sig]
\end{aligned}$$

The cell c_4 consists of wiring, an *and* gate and a full adder. It requires skill and experience to be able to recognise opportunities to use standard components. We expect that tables of readily-implementable relations would be a useful tool for the practising designer, just as books of data sheets have been in the past.

The introduction of trits and the above calculation both required some ingenuity, but the rest of the derivation is largely mechanical. Because c_4 need not be refined any further, we have almost completed the design of c_2. The ti^{-1} that we need for the column induction is introduced at the bottom of the column, pushed up the column, and finally implemented when it meets a zero at the top.

$$\begin{aligned}
c_2 &= \mathsf{snd}\ dstr_n \ ; zip_n \ ; {\pi_1}^{-1} \ ; \mathsf{snd}\ [\![0]\!] \ ; \mathsf{col}_n\ c_3 \ ; \mathsf{fst}\ bi^{-1} \\
&= \qquad \{ \text{bits as trits, since } bi = {\pi_2}^{-1} \ ; [low, sig] \ ; ti \} \\
&\quad \mathsf{snd}\ dstr_n \ ; zip_n \ ; {\pi_1}^{-1} \ ; \mathsf{snd}\ [\![0]\!] \ ; \mathsf{col}_n\ c_3 \ ; \mathsf{fst}(ti^{-1} \ ; [low, sig] \ ; \pi_2) \\
&= \qquad \{ \text{column induction from } c_3 \ ; \mathsf{fst}\ ti^{-1} = \mathsf{snd}\ ti^{-1} \ ; c_4 \} \\
&\quad \mathsf{snd}\ dstr_n \ ; zip_n \ ; {\pi_1}^{-1} \ ; \mathsf{snd}([\![0]\!] \ ; ti^{-1}) \ ; \mathsf{col}_n\ c_4 \ ; \mathsf{fst}([low, sig] \ ; \pi_2)
\end{aligned}$$

The final step is to eliminate the remaining ti. The representation of zero as a trit is a pair of equal signals, $[\![0]\!] \ ; ti^{-1} = [\![0]\!] \ ; any \ ; sig \ ; fork$, so

$$\begin{aligned}
{\pi_1}^{-1} \ ; \mathsf{snd}([\![0]\!] \ ; ti^{-1}) &= {\pi_1}^{-1} \ ; \mathsf{snd}(any \ ; [\![0]\!] \ ; any \ ; sig \ ; fork) \\
&= {\pi_1}^{-1} \ ; \mathsf{snd}(sig \ ; fork)
\end{aligned}$$

This completes the first implementation for the multiplier: gathering all the parts we have that

$$\begin{aligned}
\mathcal{M} &= \mathsf{snd}\ dstl_m \ ; \mathsf{row}_m\ c_2 \ ; \mathsf{snd}\ \mathsf{map}_n\ sig \\
&\qquad \text{where } c_2 = \mathsf{snd}\ dstr_n \ ; zip_n \ ; \pi_1^{-1} \ ; \mathsf{snd}(sig \ ; fork) \ ; \\
&\qquad\qquad\qquad \mathsf{col}_n\ c_4 \ ; \mathsf{fst}([low, sig] \ ; \pi_2) \\
&\qquad\qquad c_4 = rsh \ ; [\mathsf{fst}\,\mathsf{snd}\ and \ ; fulladd', sig]
\end{aligned}$$

However, if we try to draw this circuit it turns out to be far from beautiful. The instances of *dstl*, *dstr* and *zip* in particular consist of large bundles of wires which are knitted together in untidy knots.

This would not matter if we only needed to produce a netlist for input to some absolutely general automatic layout generator such as a programmer for configurable logic arrays, or a VLSI layout system. However if we want to interface to a design system using a more structured circuit description, we must rearrange the wiring to give a more satisfactory implementation. Most refinements will naturally result in circuits that need to be improved by moving wires or components about.

6 Rearrangement

A further calculation, about as long as that which has gone before, will allow the wires of the implementation of the multiplier to be 'combed out' into a regular pattern. The strategy for this part of the calculation is to represent as much as possible of the structure of the circuits as adjacent rows and columns of simple components – even the wiring – and then to interleave these rows and columns.

Because the beside and below operators distribute over each other in this way:

$$(A \updownarrow B) \leftrightarrow (C \updownarrow D) = (A \leftrightarrow C) \updownarrow (B \leftrightarrow D)$$

(a property which Richard Bird calls 'abiding' [3]) it follows by induction that adjacent columns can be combined

$$\begin{aligned}(\mathsf{col}_n\ R) \leftrightarrow (\mathsf{col}_n\ S) &= \mathsf{col}_n (R \leftrightarrow S)\\ (\mathsf{row}_n\ R) \updownarrow (\mathsf{row}_n\ S) &= \mathsf{row}_n (R \updownarrow S)\end{aligned}$$

at least when $n > 0$, and similarly rows that are below each other can be combined; and more generally $\mathsf{row}_n\ \mathsf{col}_m\ R = \mathsf{col}_m\ \mathsf{row}_n\ R$, provided $n > 0$ and $m > 0$.

The pattern

$$\begin{aligned}A \gtrless B &= (A \updownarrow swp) \leftrightarrow (swp \updownarrow B)\\ &= (A \leftrightarrow swp) \updownarrow (swp \leftrightarrow B)\end{aligned}$$

which is illustrated in Fig. 8 is one which arises often in the course of these interleavings, and has the pleasant property that

$$\begin{aligned}\mathsf{col}_n (A \gtrless B) &= \mathsf{fst}\ zip_n^{-1} \,;\, ((\mathsf{col}_n\ A) \gtrless (\mathsf{col}_n\ B)) \,;\, \mathsf{snd}\ zip_n\\ \mathsf{row}_n (A \gtrless B) &= \mathsf{snd}\ zip_n^{-1} \,;\, ((\mathsf{row}_n\ A) \gtrless (\mathsf{row}_n\ B)) \,;\, \mathsf{fst}\ zip_n\end{aligned}$$

again provided $n > 0$, which can be used right-to-left to rewrite matching rows and columns as single structures of more complex cells, possibly eliminating the large number of wire crossings indicated by zip_n.

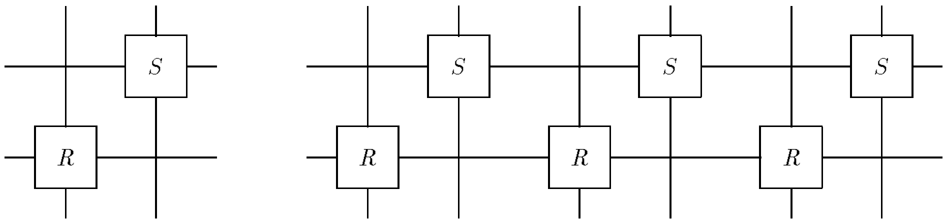

Fig. 8. $R \gtrless S$ and $\mathsf{row}_3 (R \gtrless S)$

If R and S are relations for which $R \updownarrow S = \mathsf{fst}\ swp \,;\, (S \updownarrow R) \,;\, \mathsf{snd}\ swp$, which is the case if either R or S is itself swp, then

$$\mathsf{col}_n (R \updownarrow S) = \mathsf{fst}\ zip_n \,;\, (\mathsf{col}_n\ R \updownarrow \mathsf{col}_n\ S) \,;\, \mathsf{snd}\ zip_n^{-1}$$

which can be used as a right-to-left rule to interleave two columns of largely independent cells. The dual is a statement about rows, that if $R \leftrightarrow S = \mathsf{snd}\ swp \,;\, (S \leftrightarrow R) \,;\, \mathsf{fst}\ swp$

$$\mathsf{row}_n (R \leftrightarrow S) = \mathsf{snd}\ zip_n \,;\, (\mathsf{row}_n\ R \leftrightarrow \mathsf{row}_n\ S) \,;\, \mathsf{fst}\ zip_n^{-1}$$

Not all the structure of the implementation of the multiplier yet consists of rows and columns, but we have already seen that the instances of *dstr* and *dstl* have implementations as columns and rows of simpler cells, for

$$\begin{aligned} dstr_n &= \mathsf{col}_n(fork\ ;\mathsf{fst}\ \pi_2)\ ;\pi_2 \\ dstl_n &= \mathsf{row}_n(fork\ ;\mathsf{snd}\ \pi_1)\ ;\pi_1 \end{aligned}$$

and other structures and wiring components can also be expanded into rows and columns in this way, for

$$\begin{aligned} \mathsf{map}_n\ R &= \pi_1^{-1}\ ;\mathsf{col}_n(\pi_1\ ;R\ ;\pi_2^{-1})\ ;\pi_2 \\ &= \pi_2^{-1}\ ;\mathsf{row}_n(\pi_2\ ;R\ ;\pi_1^{-1})\ ;\pi_1 \\ \mathsf{map}_n\ R\ ;fork &= \mathsf{map}_n(R\ ;fork)\ ;zip_n^{-1} \\ \mathsf{fst}\ \mathsf{map}_n\ R\ ;swp &= \mathsf{col}_n(\mathsf{fst}\ R\ ;swp) \\ \mathsf{snd}\ \mathsf{map}_n\ R\ ;swp &= \mathsf{row}_n(\mathsf{snd}\ R\ ;swp) \end{aligned}$$

and so on.

6.1 Tidying the Wiring

Although we performed this part of the calculation in about the same detail as the first part of the derivation, we are not satisfied that our presentation is yet sufficiently lucid, nor that it conveys any new insights, to justify its inclusion in full in this paper. In particular, it has proved very hard to control the size of the expressions and they are not easily read. Accordingly we will only outline the shape of the calculation, confident in our ability if challenged to explain the calculation in more detail than a human reader could want. Indeed we intend that these calculations should be explained to a machine that would check their accuracy, and machine assistance in such a detailed presentation is certainly necessary.

Since we expect that the structure of the circuit will change as we bring in more wiring from the outside, the restructuring will be done from the inside out. The component in the middle of the circuit, if we agree to disregard c_4 which we claim is in essence already implemented, begins

$$\begin{aligned} \mathsf{snd}\ dstr_n\ ;zip_n = \quad & \{\text{expanding } dstr \text{ and interleaving the columns}\} \\ & rsh\ ;[zip_n, \pi_2]\ ;\mathsf{col}_n\ c_5\ ;\pi_2 \\ & \text{where } c_5 = (\pi_1\ ;\pi_2^{-1}) \between (\mathsf{fst}\ \pi_1^{-1}\ ;(swp \updownarrow (\pi_2\ ;fork))) \end{aligned}$$

The component c_5 can be simplified, by first expressing it in terms of beside and below, and then using the properties of the projections, the abiding of beside with below, and reassociating operations with left and right shifts. Thus we arrive at

$$c_5 = [\pi_1^{-1}, \pi_2]\ ;(swp \updownarrow (\pi_2\ ;fork))\ ;[\pi_2^{-1}, lsh]$$

which, since c_5 is just a wiring relation, can be checked by showing that each side relates the same tuples in the same ways. Then by a further column induction

$$\begin{aligned} \mathsf{snd}\ dstr_n\ ;zip_n = rsh\ ;\mathsf{fst}\ zip_n\ ;\mathsf{col}_n\ c_6\ ;\pi_2 \\ \text{where } c_6 = \mathsf{fst}\ \pi_1^{-1}\ ;(swp \updownarrow (\pi_2\ ;fork))\ ;\mathsf{snd}\ lsh \end{aligned}$$

Substituting in the definition of c_2, this column can be merged with the $\mathsf{col}_n\, c_4$ which it is beside, ultimately yielding

$$\begin{aligned}c_2 = \mathsf{snd}\,\mathsf{snd}(\pi_1^{-1}\,;\mathsf{snd}(sig\,;fork))\,;\\ ((zip_n\,;\pi_2^{-1}) \leftrightarrow (\mathsf{col}_n(c_6 \leftrightarrow c_4)))\,;\\ \mathsf{fst}(\pi_2\,;\pi_2\,;[low, sig]\,;\pi_2)\end{aligned} \tag{14}$$

When this is substituted back into the implementation of $\mathcal{M}$, it appears alongside an instance of $dstl_m$ which can be expanded as a row, and merged with it:

$$\begin{aligned}\mathcal{M} &= \mathsf{snd}\ dstl_m\,;\mathsf{row}_m\, c_2\,;\mathsf{snd}\,\mathsf{map}_n\, sig\\ &= rsh\,;((\mathsf{row}_m\, c_2) \updownarrow (dstl_m\,;\pi_1^{-1}))\,;\mathsf{snd}(\pi_1\,;\mathsf{map}_n\, sig)\\ &= \qquad \{\text{expanding } dstl \text{ and row below row}\}\\ &\quad rsh\,;\mathsf{snd}\,\mathsf{map}_m\,\pi_2^{-1}\,;\mathsf{row}_m(c_2 \updownarrow ((\pi_1\,;fork) \leftrightarrow swp))\,;\mathsf{snd}(\pi_1\,;\mathsf{map}_n\, sig)\end{aligned}$$

Now note that $\mathsf{snd}\,\mathsf{fst}\,\mathsf{map}_n\, id \dashv c_2$, from which it follows that the $fork$ and swp in this expression are also n signals wide. These can be expanded and interleaved in the same way, and the resulting column combined with the $\mathsf{col}_n(c_6 \leftrightarrow c_4)$ obtained by substituting for c_2 using (14). The result of this calculation is

$$\begin{aligned}\mathcal{M} = rsh\,;[zip_n, \mathsf{map}_m(\pi_1^{-1}\,;\mathsf{snd}(sig\,;fork))]\,;\\ \mathsf{row}_m\,\mathsf{col}_n\, cell\,;\\ [\mathsf{map}_m(\pi_2\,;[low, sig]\,;\pi_2), \mathsf{map}_n(\pi_1\,;sig)]\\ \text{where } cell = \mathsf{fst}(\mathsf{snd}\, fork\,;rsh)\,;((c_6 \leftrightarrow c_4) \updownarrow swp)\end{aligned} \tag{15}$$

which is a grid of cells, surrounded by some simple wiring.

The only part of the wiring which we would prefer not to implement is the instance of zip_n which would involve $n(n-1)/2$ wire crossings. We choose to have the user of the multiplier implement this, and change the specification of our design to $\mathsf{fst}\, zip_n^{-1}\,;lsh\,;\mathcal{M}$. That is to say that we will offer the user a component which must be given one of the factors already interleaved with the carry in. The user may well want the carry in to be zero anyway, in which case the interleaving is much cheaper.

6.2 Tidying the Cell

Now consider the cell of which this grid is composed.

$$\begin{aligned}cell = \mathsf{fst}(\mathsf{snd}\, fork\,;rsh)\,;((c_6 \leftrightarrow c_4) \updownarrow swp)\\ \text{where } c_6 = \mathsf{fst}\,\pi_1^{-1}\,;(swp \updownarrow (\pi_2\,;fork))\,;\mathsf{snd}\, lsh\\ c_4 = rsh\,;[\mathsf{fst}\,\mathsf{snd}\, and\,;fulladd', sig]\end{aligned}$$

It is now necessary to simplify the wiring of this cell, and further to make sure that each of the remaining components is in our repertoire of standard components.

This means, for example, that we will not leave $fork$ in the final implementation, because $fork$ may in general involve forking a large bus of wires, involving many wire crossings. We will only be satisfied when each fork in the circuit divides a single signal wire, that is when each is an instance of $sfork = sig \,;fork$. Similar considerations apply to the other primitive wiring tiles.

A calculation about as long as that in the preceding section of the paper, but conceptually simpler in that it involves no rows and columns, shows that

$$
\begin{aligned}
cell = {} & (\mathsf{fst}\,\mathsf{fst}\,\pi_1^{-1} \,;((sswp \updownarrow (\pi_2 \,; sfork)) \updownarrow sswp) \,; \mathsf{snd}(\mathsf{snd}\, sfork \,; srsh)) \leftrightarrow \\
& \quad ((\mathsf{fst}(slsh \,; \mathsf{snd}\, and) \,; srsh \,; [fulladd', sig]) \updownarrow (sswp \leftrightarrow sswp)) \\
& \text{where } sfork = sig \,; fork \\
& \qquad sswp = [sig, sig] \,; swp \\
& \qquad slsh = [[sig, sig], sig] \,; lsh \qquad = [sig, sig] \updownarrow [sig, sig] \\
& \qquad srsh = [[sig, sig], [sig, sig]] \,; rsh = [[sig, sig], sig] \leftrightarrow [sig, sig]
\end{aligned}
$$

Figure 9 illustrates a layout for the cell, suggested by this equation, annotated with indications of which wires are necessarily inputs to or outputs from the cell.

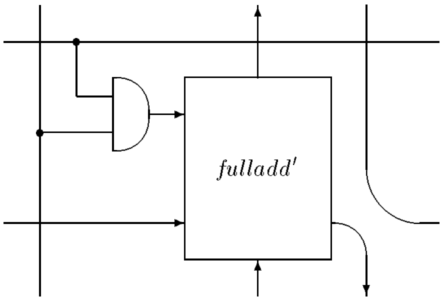

Fig. 9. the *cell* of the multiplier

This completes the development of a bit-level regular array that implements the multiplier, because substituting back into (15),

$$
\begin{aligned}
\mathsf{fst}\, zip_n^{-1} \,; lsh \,; \mathcal{M} = {} & \mathsf{snd}\,\mathsf{map}_m(\pi_1^{-1} \,; \mathsf{snd}\, sfork) \,; \\
& \quad \mathsf{row}_m\, \mathsf{col}_n\, cell \,; \\
& \qquad [\mathsf{map}_m(\pi_2 \,; [low, sig] \,; \pi_2), \mathsf{map}_n\, \pi_1]
\end{aligned}
$$

where *cell* is implemented entirely in terms of bit-operations. The layout which this suggests for $\mathsf{fst}\, zip^{-1} \,; lsh \,; \mathcal{M}$ is shown in Fig. 10.

This design is essentially the one in [14], where we presented it rather more abruptly. That paper systematically transforms the circuit into a bit-systolic array able to deliver a multiplication once on each cycle of a clock whose period is roughly the delay through a full adder. Notice that the circuit is hexagonally-connected, although we were able to derive it without needing to introduce any special constructors, nor to develop an algebra of hex-connected circuits.

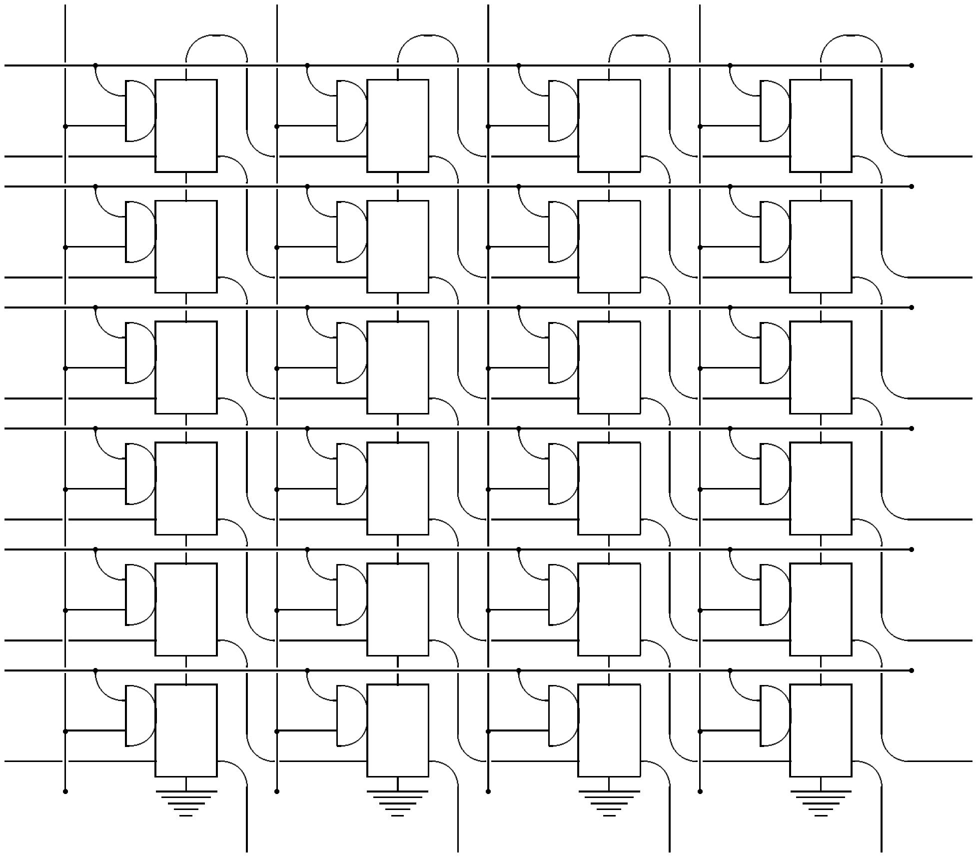

Fig. 10. fst zip^{-1} ; lsh ; $\mathcal{M}$ with $m = 4$ and $n = 6$

7 Conclusion

We have shown in some detail how the design of a complex and subtle circuit is a natural consequence of a relatively abstract specification and some simple design decisions. These decisions are ones that can largely be explained at the level of the abstract specification. The most positive thing about this calculation is that it is an argument for the correctness of the design *before the fact*. That is to say that once the design has been done there is no need to construct a separate proof of correctness.

That a derivation can be driven by high-level decisions is promising because it suggests that there is a systematic way of delineating a design space which could be explored for alternative implementations of the same specification. For example the choice of representation for the carries has driven us to this design whereas another choice would have produced the carry-save multiplier [9]; other designs are suggested by the possibility of choosing some other base (than two) to represent any of the inputs or outputs or carries. Perhaps base four might yield a design with fewer, larger cells? We hope that as more experience is gained in deriving designs in this way, better methods will emerge of aiming more directly at good implementations.

The derivation proceeds by refinement: a process of reducing the complexity of the data on the 'wires' of the circuit until only bits remain, and all of the components are implementable. We found very useful the ability to deal in a uniform way both with the abstractions such as integers and multiplication, and with concrete signal levels and gates. Distinguishing between numbers and signal levels is something which we have not done in the past, and we believe that it makes the step to 'known' hardware components easier to recognise.

As far as was possible we tried to be honest about the design process: we followed the calculation through with no particular target implementation in mind. For example, at the point where we discovered that $\mathsf{snd}\, bi{>}\not\dashv c_3 \,;\, \mathsf{fst}\, bi^{-1}$ and were driven to introduce trits into the design, we did so 'blindly'. One of us was unable to proceed with the design, and the other suggested this invention step with no global knowledge of the design, nor of the ultimate consequences of this decision.

Notice that the two wires representing this trit in the final circuit carry signals in opposite directions. That is to say, one is an input to a component and the other is an output from it. The calculations in Ruby, and the introduction of the trit data type, were performed in complete ignorance of this fact. This abstraction from inputs and outputs is a consequence of the decision to represent circuits by relations, and a valuable separation of concerns. The cost is of course that when claiming that we have implemented a primitive relation by some electronic component we must check that it has an appropriate allocation of inputs and outputs, compatible with those of the rest of the circuit of which it is a part.

We think that the first half of the derivation presented in this paper is a convincing account of the development of the algorithm which is implemented by the final circuit. It is disappointing however that the rearrangement of the circuit remains tedious and difficult; we hope that this can be improved, perhaps by a better understanding of the sort of calculation involved. It is clear, however, that mechanical assistance in checking the details of our calculations would be a great help. The very low levels of circuit design are already heavily automated, and it is one of our aims to expand the range of such mechanical assistance to higher and more abstract levels of the design process.

The techniques used here should be applicable to a range of similar arithmetic circuits, but we see no reason why they should not be extended to other fields. All that is necessary is that solutions can be constructed from components whose combinations have algebraic properties like those of the arithmetic building-blocks which we used here. We are investigating, as an example, the derivation of routing circuits implemented as arrays of binary decisions and arbiters [12].

Acknowledgement. The work reported in this paper has been supported by grants from the UK Science and Engineering Research Council, and was done while Mary Sheeran held a Royal Society of Edinburgh BP Research Fellowship at the Department of Computing Science of the University of Glasgow.

References

1. R. C. Backhouse, P. J. de Bruin, P. Hoogendijk, G. Malcolm, E. Voermans and J. C. S. P. van der Woude, *A relational theory of datatypes*, unpublished.
2. Richard Bird, *An Introduction to the Theory of Lists*, in M. Broy (ed.), *Logic of programming and calculi of discrete design*, NATO advanced study institutes, Series F: Computer and systems sciences, Springer-Verlag, 1987. pp. 5–42.
3. Richard Bird, *Lectures on constructive functional programming*, in M. Broy (ed.), *Constructive methods in computing science*, NATO advanced study institutes, Series F: Computer and systems sciences, Springer-Verlag, 1989.
4. D. Borrione and A. Salem, *Proving an on-line multiplier with OBJ and TACHE: a practical experience*, in [6].
5. S-K. Chin, *Verified synthesis functions for negabinary arithmetic hardware*, in [6].
6. L. J. M. Claesen, *Applied formal methods for correct VLSI design*, North-Holland, 1989.
7. Rogardt Heldal, Carsten Kehler Holst, Phil Wadler (eds.), *Functional programming*, Glasgow 1991, Springer-Verlag (Workshops in Computing), 1992.
8. Graham Hutton and Ed Voermans, *A calculational theory of pers as types*, in [7].
9. Geraint Jones and Mary Sheeran, *Relations and Refinement in Circuit Design* in Carroll Morgan and J. C. P. Woodcock (eds.), Proceedings of the 3rd Refinement Workshop, 9–11 January 1990, Springer-Verlag (Workshops in Computing), 1991. pp. 133–152.
10. Geraint Jones and Mary Sheeran, *Circuit design in Ruby*, in Jørgen Staunstrup (ed.), *Formal methods for VLSI design*, North-Holland, 1990. pp. 13–70.
11. Geraint Jones and Mary Sheeran, *A certain loss of identity*, in John Launchbury and Patrick Sansom (eds.), *Functional Programming, Glasgow 1992*, Springer-Verlag (Workshops in Computing), 1992. pp. 113–121.
12. Mark B. Josephs, Rudolph H. Mak, Jan Timen Udding, Tom Verhoeff and Jelio T. Yantchev, High-level design of an asynchronous packet routing chip, in [17].
13. L. G. L. T. Meertens, *Constructing a calculus of programs*, in J. L. A. van de Snepscheut (ed.), *Mathematics of program construction*, Springer-Verlag LNCS 375, 1989.
14. Mary Sheeran, *Designing regular array architectures using higher-order functions*, in J.-P. Jouannaud (ed.), *Functional programming languages and computer architecture*, Springer-Verlag LNCS 201, 1985. pp. 220–237.
15. Mary Sheeran, *A note on abstraction in Ruby*, in [7].
16. M. Simonis, *Formal verification of multipliers* in [6].
17. Jørgen Staunstrup and R. Sharp (eds.), *Designing Correct Circuits*, Lyngby 1992, IFIP Transactions A, Volume 5, North-Holland, 1992.
18. D. Verkest, L. Claesen and H. De Man, *A proof of the non-restoring division algorithm and its implementation on the Cathedral-II ALU*, in [17].