Deriving bit-serial circuits in Ruby

Geraint Jones[a] and Mary Sheeran[b]

[a]Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, England; Electronic mail: *gj@comlab.oxford.ac.uk*

[b]Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland; Electronic mail: *ms@dcs.glasgow.ac.uk*

Abstract

The action of bit-serial arithmetic circuits is often explained in purely pictorial terms. In contrast, this paper describes an attempt to deal with the systematic development of bit-serial arithmetic circuits within a mathematical framework which we have previously used to develop parallel circuits. A well-known bit-serial adder is formally shown to implement the specification of an adder, without any recourse to detailed arguments about snapshots or specific arguments about sequences of inputs.

1 DESIGN BY CALCULATION

Ruby [1] is a language of relations and functions that supports a style of development by calculation. A circuit is represented by a relation between the signals at its terminals, and forms of circuit are captured by functions which operate on those relations.

For example, if the second terminal of a two-terminal component R is connected to the first of a similar component S, they behave as $R\,;S$, defined by $x\ (R\,;S)\ y \Leftrightarrow \exists z.\, x\ R\ z\ \&\ z\ S\ y$. At another extreme, the parallel composition of two circuits with no connection between them is described by $[R, S]$ where $\langle x_1, x_2\rangle\ [R, S]\ \langle y_1, y_2\rangle \Leftrightarrow x_1\ R\ y_1\ \&\ x_2\ S\ y_2$. The converse of a relation, defined by $x\ R^{-1}\ y \Leftrightarrow y\ R\ x$, represents a circuit which is like R but with its left and right connections swapped.

The composition operators can easily be shown to satisfy many laws such as $[P, Q]\,;[R, S] = [P\,;R, Q\,;S]$, which is true for all P, Q, R, and S. Similarly, $[R, S]^{-1} = [R^{-1}, S^{-1}]$ and $(R;S)^{-1} = S^{-1};R^{-1}$. These and many similar laws can be used for what is essentially algebraic manipulation of expressions describing circuits, in the course of which there is little need to consider the details of what the circuit does.

Ruby relations can be used to describe more or less abstract views of a circuit, and in particular it is possible to describe within the one framework the process of refining what might be thought of as a specification into something that might be an implementation. For example, if *add* is the relation between a pair of natural numbers and the natural number which is their sum, and if *bin* is the relation between a tuple of bits and the natural number of which they are a binary least-significant bit first representation, then $[bin, bin]\,; add\,; bin^{-1}$ describes a 'circuit' which relates two such tuples of bits to a similar tuple of bits representing their sum.

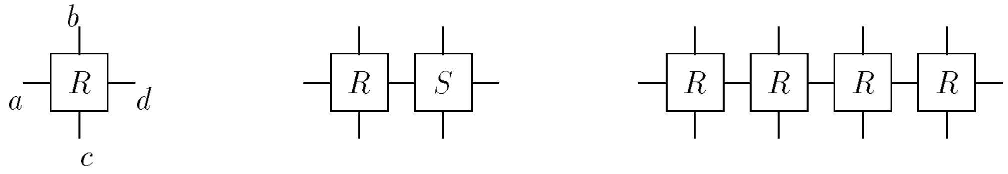

Figure 1: $\langle a, b\rangle\ R\ \langle c, d\rangle$ and layouts suggested by $R \leftrightarrow S$, and $\mathsf{row}\, R$

2 NOTATION USED IN THIS PAPER

We will need a number of specific circuit constructors in the calculations of this paper. These are chosen from a large collection of common forms discussed in more detail in, for example, reference [1]. One reason for having such a great number is that there is a rich collection of equations relating them, which makes calculation easier.

Many of the forms are abbreviations, which capture common patterns, for example we write $R \setminus S$, read 'R conjugated by S', for $S^{-1}\,;R\,;S$. We write $\mathsf{fst}\, R$ for the parallel composition of R and the identity relation, and $\mathsf{snd}\, S$ for the identity relation in parallel with S. It follows that $\mathsf{fst}\, R ; \mathsf{snd}\, S = [R, S] = \mathsf{snd}\, S ; \mathsf{fst}\, R$ and so on. Distinguish between fst and snd and the particular pieces of wiring represented by the projection relations π_1, π_2, which are defined by $\langle x, y\rangle\ \pi_1\ x$ and $\langle x, y\rangle\ \pi_2\ y$.

We write R^n for the repeated composition of n copies of R, so that $R^1 = R$, and $R^{n+1} = R^n\,;R$. Repeated parallel composition is captured by $\mathsf{map}\, R$, a relation between equal-length lists, relating two lists if corresponding elements of the lists are related by R.

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

Those relations which are equivalence relations, that is T for which $T = T^{-1} = T^2$, serve the rôle of types in our calculations. A constant is just a type with only one equivalence class: that is a type k for which $k = k\,;\mathit{any}\,;k$ where any is the universal type, the relation for which $x\ \mathit{any}\ y$ for any x and y. A type T for which $x\ T\ y$ only if $x = y$, is just a data constraint. In particular, for any number n we write n for the identity relation on lists of that length, so for example 2 is the type of all pairs. Constraints capture observations such as $C\,;R = C\,;S$, that two circuits, R and S, have the same behaviour so long as the signals in their domain meet some constraint, C.

We will need to deal specially with some relations between pairs and pairs. If R relates pairs to pairs, we will often want to divide its domain signals between the left and top, and the range signals between the bottom and right, as shown in figure 1. For such relations we define $R \leftrightarrow S$, read 'R beside S', by

$$\langle a, \langle b, c\rangle\rangle\ (R \leftrightarrow S)\ \langle\langle x, y\rangle, z\rangle \quad\Leftrightarrow\quad \exists p.\, \langle a, b\rangle\ R\ \langle x, p\rangle \ \ \&\ \ \langle p, c\rangle\ S\ \langle y, z\rangle$$

which suggests the layout also shown in figure 1. Complementary to this is $R \updownarrow S$, read 'R below S' and defined by $R \updownarrow S = (R^{-1} \leftrightarrow S^{-1})^{-1}$. The generalisation of beside-composition to lists is a row of components, defined by

$$\begin{aligned}&\langle a, bs\rangle\ \mathsf{row}\, R\ \langle ys, z\rangle \\ &\qquad\Leftrightarrow\quad \#bs = \#ys \ \ \&\ \ (\exists ps.\, a = ps_0 \ \ \&\ \ \forall i.\, \langle ps_i, bs_i\rangle\ R\ \langle ys_i, ps_{i+1}\rangle \ \ \&\ \ ps_{\#bs} = z)\end{aligned}$$

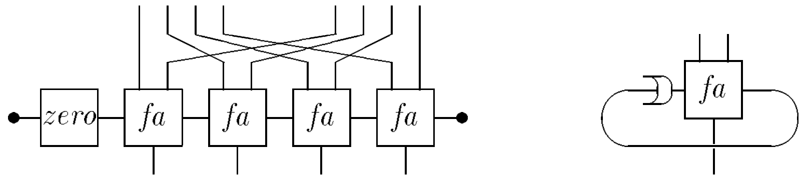

Figure 2: an instance of the bit-parallel adder $zip \mathbin{;} \pi_2^{-1} \mathbin{;} \mathsf{fst}\, zero \mathbin{;} \mathsf{row}\, fa \mathbin{;} \pi_1$, and the bit-serial adder $\mu(\mathsf{fst}\, \mathcal{D} \mathbin{;} fa)$ which we derive from it

and an instance of its layout is also shown in figure 1.

We will also need an interleaving relation, *zip*, which relates a pair of equal length lists to a list of the same length each element of which is a pair, defined by

$$\langle x, y\rangle \; zip \; z \quad \Leftrightarrow \quad \#x = \#y = \#z \;\;\&\;\; \forall i.\, \langle x_i, y_i\rangle = z_i$$

We can use this to represent the interleaving of two buses of wires.

3 A BIT-PARALLEL ADDER

We have previously [2] shown several calculations which start from specifications consisting of representation relations and abstract arithmetic operations, and leading to expressions which describe bit-parallel circuits. Here we will just quote the result of such a calculation.

Using the abstract adder, *add*, which relates two natural numbers to their sum, and the representation relation *bin*, which relates a tuple of bits to the number which it represents, we would like to specify that an adder circuit should implement the relation $[bin, bin] \mathbin{;} add \mathbin{;} bin^{-1}$. This relates a pair of tuples of bits to the tuple of bits which represents the sum of the numbers represented by the pair.

This specification is not readily implementable as a circuit because its interface is not of a fixed size: a circuit would have to have a particular width of input and of output. Let N be the identity on N-tuples, then $A = [N \mathbin{;} bin, N \mathbin{;} bin] \mathbin{;} add \mathbin{;} (N \mathbin{;} bin)^{-1}$ is a circuit which takes lists of bits of width N and produces an output of width N.

Since only numbers less than 2^N are represented by $N \mathbin{;} bin$, this relation only adds pairs of inputs which represent numbers that have a sum less than 2^N. It is therefore not a total function from all pairs of N-lists, but all our circuits eventually implement total functions – they produce *some* output for every input.

Let sum_{2^N} be the identity relation on pairs of number which sum to less than 2^N, then $sum'_{2^N} = sum_{2^N} \setminus [N \mathbin{;} bin, N \mathbin{;} bin]^{-1}$ is the constraint which we need, being the identity on the N-bit representations of pairs which sum to less than 2^N. Certainly, $A = sum'_{2^N} \mathbin{;} A$, but we can also refine $sum'_{2^N} \mathbin{;} A$ into an implementation

$$A \;\; = \;\; sum'_{2^N} \mathbin{;} zip \mathbin{;} \pi_2^{-1} \mathbin{;} \mathsf{fst}\, zero \mathbin{;} \mathsf{row}\, fa \mathbin{;} \pi_1 \mathbin{;} N \tag{1}$$

where fa is a full-adder. The term $zip \mathbin{;} \pi_2^{-1} \mathbin{;} \mathsf{fst}\, zero \mathbin{;} \mathsf{row}\, fa \mathbin{;} \pi_1 \mathbin{;} N$ is the circuit itself, which is a total function on all pairs of N-bit representations, and has a fixed size. Its form suggests the layout shown in figure 2.

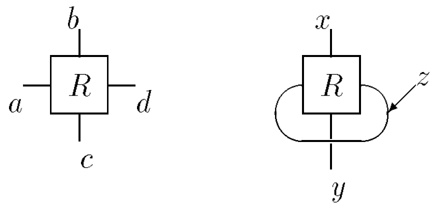

Figure 3: Layouts suggested by $\langle a, b\rangle\, R\, \langle c, d\rangle$ and $x\,(\mu\, R)\, y$, defined to be $\exists z.\, \langle z, x\rangle\, R\, \langle y, z\rangle$

4 TIME

We argue [3] that Ruby supports simple reasoning about sequential circuits. Suppose R describes a 'stateless' (combinational) circuit, and that it is provided with a succession of (potentially different) inputs at a succession of times. If the circuit is allowed sufficient time to settle into a stable state before being clocked again, that state is described by the relation R. That is to say, the whole 'clocked' or 'sequential' behaviour of the circuit is described by $\mathsf{map}\, R$, seen as a relation between two time-sequences.

We can give a similar account of each of the operators which appear in Ruby expressions, giving them a new interpretation which combines the sequential behaviours of the component expressions to yield the sequential behaviour of the whole. For example, the parallel composition of R and S is a relation between two time-sequences of pairs: at any time, the pair in the domain is related to the pair in the range at that time if their first components would have been related by the R at that time, and their second by S. That is to say, the sequences related by the clocked interpretation of the parallel composition of two circuits are obtained by zipping together those related by the components.

It can be proved, by induction over the language, that any expression E in Ruby in which each of the components is stateless is also itself stateless, that is that the sequential interpretation of E is $\mathsf{map}\, E$. In order to be able to talk about circuits which are not stateless we introduce the primitive circuit $\mathcal{D}$, read 'delay', defined by $x\,\mathcal{D}\, y \Leftrightarrow \forall t.\, x_t = y_{t+1}$. That is to say, the signal in the range of $\mathcal{D}$ at any time is the same as the signal in its domain at the previous time. It might be implemented by a latch, register, or delay element taking the domain signal as input, and returning the range signal as output. Similarly $\mathcal{D}^{-1}$, the 'anti-delay', predicts its domain signal in the range. Although it might seem unrealistic, it could be implemented by a latch operating in the other direction.

Although many of the Ruby operators have complicated interpretations as operations on sequential behaviours, reference [3] shows that any well-typed equation which is true in the simpler combinational interpretation of Ruby remains true in the new sequential interpretation. This is the case even if the variables in the equation are allowed to range over all (Ruby-definable) sequential circuits, even ones that are not stateless. This means that we can often prove things about sequential circuits just by the much simpler expedient of showing that the corresponding thing is true of combinational circuits.

Of course, only equations which make sense in the combinational interpretation are amenable to this approach. There are a few things which we need to know and that are particular to sequential circuits, and for these we have to resort to checking their validity directly in the sequential interpretation. For example, if k is a constant circuit, which

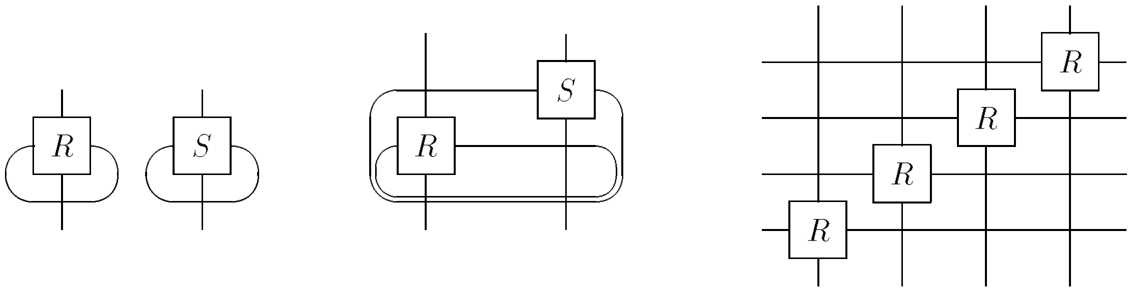

Figure 4: $[\mu R, \mu S]$ compared to $\mu([R, S] \setminus zip^{-1})$, and an instance of $\mathsf{map}\, R \setminus zip^{-1}$

holds its connections at the same value at all times, then

$$k \;=\; k \,;\, \mathcal{D} \tag{2}$$

which must be an equation about sequential circuits, since $\mathcal{D}$ has no other interpretation.

The sequential interpretation of equation 1 describes a combinational circuit which implements a word-serial, bit-parallel adder; but it has a long ripple-propagation path along the carry chain, and elsewhere [3, 4] we have pipelined such circuits, using Leiserson's retiming regime [5]. From the pipelining calculation we would derive requirements on the data-skew of the input, and consequences for the data-skew of the output.

5 FEEDBACK

Any bit-serial design must have an internal state which contains the carry forward from one bit's calculation, and a feedback mechanism which delivers it to the next bit's calculation. We can deal with the state using $\mathcal{D}$, and we introduce a new construction to deal with feedback,

$$x\,(\mu R)\,y \;\;\Leftrightarrow\;\; \exists z.\, \langle z, x\rangle \, R \, \langle y, z\rangle$$

the definition of which suggests the layout shown in figure 3.

The feedback constructor has a rich collection of simple equations which relate it to other parts of the language, for example

$$A \,;\, \mu R \,;\, B \;=\; \mu(\mathsf{snd}\, A \,;\, R \,;\, \mathsf{fst}\, B) \tag{3}$$

$$\mu(\mathsf{fst}\, P \,;\, R) \;=\; \mu(R \,;\, \mathsf{snd}\, P) \tag{4}$$

which explain, respectively, the interface of μR, and how the representation of the 'state' on the feedback wire may be changed. Notice that the truth of equation 4 does not depend on any properties of P: it need not be a function, nor the inverse of a function, nor onto, nor total. This one equation, therefore, captures many different kinds of 'change of state' transformations.

An unused feedback loop can always be eliminated

$$\mu(\mathsf{fst}\, any \,;\, R \,;\, \mathsf{snd}\, any) \;=\; \pi_2^{-1} \,;\, R \,;\, \pi_1 \tag{5}$$

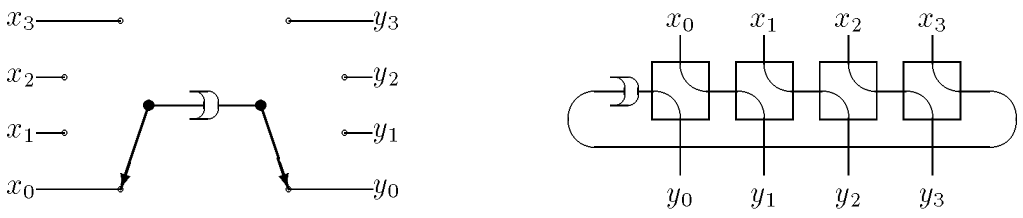

Figure 5: $\mathcal{D} \setminus \underline{\mathit{bundle}}$ and the corresponding $\mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{row}\,2)$

and feedback loops can be composed in series and parallel, $\mu\, R\,;\,\mu\, S \;=\; \mu(S \updownarrow R)$ and $[\mu\, R, \mu\, S] = \mu([R, S] \setminus \mathit{zip}^{-1})$. The latter equation is illustrated in figure 4, and generalises to arbitrarily wide parallel compositions, because $\mathsf{map}\,\mu\, R \;=\; \mu(\mathsf{map}\, R \setminus \mathit{zip}^{-1})$. The kernel of the right-hand side suggests the layout shown at the right of figure 4, and $\mu(\mathsf{map}\, R \setminus \mathit{zip}^{-1})$ would be obtained by joining corresponding wires on the left and right.

6 SLOWING

So far, all the circuits have been operating at the same rate. We can also describe circuits which have regions running on clocks with different frequencies [4]. An N-slow circuit [5] is one in which every instance of $\mathcal{D}$ has been replaced by $\mathcal{D}^N$. The N is a constant in the whole of this paper, but we could deal with different values of N by decorating a couple of the constructors with the scaling factor to which they relate.

In order to explain slowing in Ruby, we will introduce a new primitive which describes the demultiplexing of a sequence of values into N separate signals.

$$x\ \underline{\mathit{bundle}}\ y \quad \Leftrightarrow \quad x_{Nt+i} = y_{t,i}$$

Each y_i is a sequence delivered, or sampled, at $1/N$th the rate of x. We underline $\underline{\mathit{bundle}}$ as a reminder that – like $\mathcal{D}$ – it has only a sequential interpretation. That means, for example, that in equations like

$$[\underline{\mathit{bundle}}, \underline{\mathit{bundle}}]\,;\, \mathit{zip} \quad = \quad 2\,;\, \underline{\mathit{bundle}} \tag{6}$$

the only possible reading is the sequential one.

It can be proved, by an induction over all expressions in the language, that an N-slow version of any Ruby-definable circuit R behaves like N independent copies of R with the inputs being demultiplexed and the outputs being interleaved.

$$\mathsf{slow}\, R \quad = \quad (\mathsf{map}\, R) \setminus \underline{\mathit{bundle}}^{-1} \tag{7}$$

This shows that although $\mathsf{slow}\, R$ was defined by a manipulation of the text of expressions, it depends only on the meaning of the expression, so that if $R = S$ then $\mathsf{slow}\, R = \mathsf{slow}\, S$. Since slowing was defined by a substitution, it is immediate that it distributes through all the operators, so $\mathsf{slow}(R\,;S) = \mathsf{slow}\, R\,; \mathsf{slow}\, S$, and $\mathsf{slow}[R, S] = [\mathsf{slow}\, R, \mathsf{slow}\, S]$, and so on. Moreover, $\mathsf{slow}\, R = R$ for any stateless R, and $\mathsf{slow}\,\mathcal{D} = \mathcal{D}^N$.

Since $\mathcal{D}$ is itself Ruby-definable, $\mathcal{D}^N = \mathsf{slow}\,\mathcal{D} = (\mathsf{map}\,\mathcal{D}) \setminus \underline{\mathit{bundle}}^{-1}$ by equation 7. A similar result, which we will need in this paper, describes a way of implementing isolated

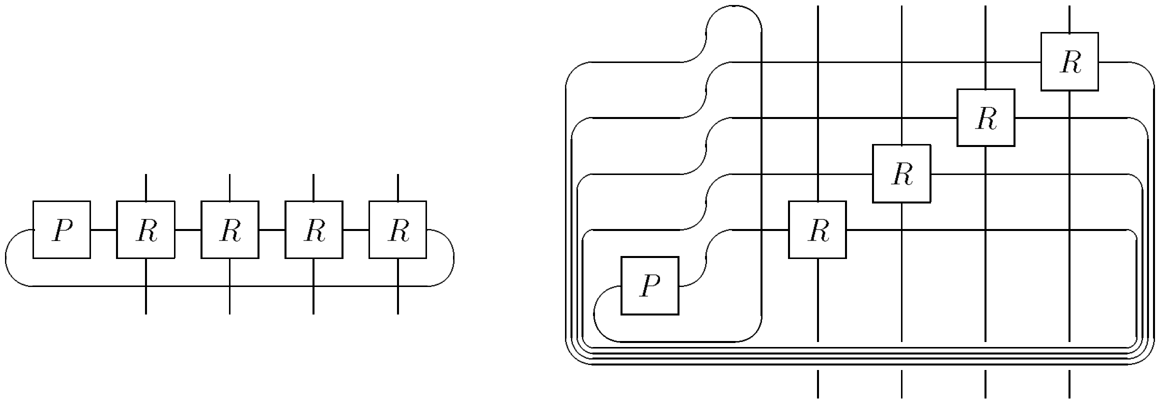

Figure 6: instances of $\mu(\mathsf{fst}\,P\,;\mathsf{row}\,R)$ and $\mu(\mathsf{fst}\,\mu(\mathsf{fst}\,P\,;\mathsf{row}\,2)\,;(\mathsf{map}\,R \setminus zip^{-1}))$

single delays – ones which are not a part of a $\mathcal{D}^N$ – in circuits that are otherwise slow. Consider $\mathcal{D} \setminus \underline{bundle}$, which can be read as a delay sandwiched between a multiplexer and a demultiplexer, as illustrated in figure 5. It is the relation given by

$$x\,(\mathcal{D} \setminus \underline{bundle})\,y \quad\Leftrightarrow\quad \begin{cases} x_{t,i} = y_{t,i+i}, & \text{if } 0 \le i < N-1 \\ x_{t,N-1} = y_{t+1,0} \end{cases}$$

and from this it can be proved that

$$\mathcal{D} \setminus \underline{bundle} \quad=\quad (\mu(\mathsf{fst}\,\mathcal{D}\,;\mathsf{row}\,2)) \setminus N \tag{8}$$

The construction $\mathsf{row}\,2$ on the right-hand side is a common idiom: the 2 is just two wires in parallel, and a row of them is a shift, which is made into a cyclic shift by the μ, and the signal that is carried around is delayed, as illustrated in figure 5.

7 SLOW COMPONENTS WITH BOTH STATE AND FEEDBACK

A bit-serial arithmetic circuit will be implemented by a bit-operation clocked once per bit, and slowed to the speed at which N-bit words are presented. So we will be interested in folding word-level circuits, ones with $\mathsf{row}\,R$ in them, to get forms like $\mu(\ldots\mathsf{slow}\,R\ldots)$ which implement the same operation with bit-level components R.

A general result about feedback and row, which will prove useful in this is that

$$\mu(\mathsf{fst}\,P\,;\mathsf{row}\,R) \quad=\quad \mu(\mathsf{fst}\,\mu(\mathsf{fst}\,P\,;\mathsf{row}\,2)\,;(\mathsf{map}\,R \setminus zip^{-1})) \tag{9}$$

The right-hand side suggests the tangled layout shown in figure 6. Notice that this result, which we quote here without proof, is one which can be checked in the simple combinational interpretation of the language, and then promoted to the sequential interpretation in which it would otherwise be much more complicated to prove.

Armed with this result, and earlier observations, we proceed to expand a candidate bit-level circuit to see what word-level operation it would implement

$$
\begin{aligned}
& \mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{slow}\,R) \\
= & \{\text{ equation 7 and } \mu\,R = \mu(R \setminus 2)\,\} \\
& \mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{map}\,R \setminus (2\,;\,\underline{bundle})^{-1}) \\
= & \{\text{ equation 6 }\} \\
& \mu([\mathcal{D}\,;\,\underline{bundle}, \underline{bundle}]\,;\,(\mathsf{map}\,R \setminus zip^{-1})\,;\,[\underline{bundle}^{-1}, \underline{bundle}^{-1}]) \\
= & \{\text{ equation 4 }\} \\
& \mu([\mathcal{D} \setminus \underline{bundle}, \underline{bundle}]\,;\,(\mathsf{map}\,R \setminus zip^{-1})\,;\,\mathsf{fst}\,\underline{bundle}^{-1}) \\
= & \{\text{ equation 8 }\} \\
& \mu([\mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{row}\,2) \setminus N, \underline{bundle}]\,;\,(\mathsf{map}\,R \setminus zip^{-1})\,;\,\mathsf{fst}\,\underline{bundle}^{-1}) \\
= & \{\text{ equation 4 }\} \\
& \mu([\mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{row}\,2)\,;\,N, \underline{bundle}]\,;\,(\mathsf{map}\,R \setminus zip^{-1})\,;\,[\underline{bundle}^{-1}, N]) \\
= & \{\text{ since } zip \text{ relates only pairs of lists of equal length }\} \\
& \mu([\mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{row}\,2), \underline{bundle}]\,;\,(\mathsf{map}\,R \setminus zip^{-1})\,;\,\mathsf{fst}\,\underline{bundle}^{-1}) \\
= & \{\text{ equation 3 }\} \\
& (\mu(\mathsf{fst}\,\mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{row}\,2)\,;\,(\mathsf{map}\,R \setminus zip^{-1}))) \setminus \underline{bundle}^{-1} \\
= & \{\text{ equation 9 }\} \\
& (\mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{row}\,R)) \setminus \underline{bundle}^{-1}
\end{aligned}
$$

and then conjugating on both sides with $\underline{bundle}$, since $N = \underline{bundle}^{-1}\,;\,\underline{bundle}$ we obtain

$$
(\mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{slow}\,R)) \setminus \underline{bundle} \quad = \quad (\mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{row}\,R)) \setminus N \tag{10}
$$

which we will use to replace the word-level operation, involving $\mathsf{row}\,R$, on the right-hand side by the bit-serial operation, $\mu(\mathsf{fst}\,\mathcal{D}\,;\,\mathsf{slow}\,R)$, on the left.

8 LIMITING THE CARRY-AROUND

It remains only to show that word-level addition can be implemented by a circuit with feedback, of the form $\mu(\mathsf{fst}\,\mathcal{D};\mathsf{row}\,R)$. The intuition behind this step of the development is that such a circuit will do the right thing – provided only that the input is so constrained that the state being fed back can be guaranteed always to be just that constant *zero* which is required in equation 1. Let b be any constraint strong enough that

$$
\mathsf{snd}\,b\,;\,\mathsf{row}\,R \quad = \quad \mathsf{snd}\,b\,;\,\mathsf{row}\,R\,;\,\mathsf{snd}\,k \tag{11}
$$

for some constant k. Then

$$
\begin{aligned}
& b\,;\,\pi_2^{-1}\,;\,\mathsf{fst}\,k\,;\,\mathsf{row}\,R\,;\,\pi_1 \\
= & \{\text{ equation 5 and equation 3 }\} \\
& \mu([any\,;\,k, b]\,;\,\mathsf{row}\,R\,;\,\mathsf{snd}\,any) \\
= & \{\text{ equation 11 }\} \\
& \mu([any\,;\,k, b]\,;\,\mathsf{row}\,R\,;\,\mathsf{snd}(k\,;\,any))
\end{aligned}
$$

$= \{$ equation 4 $\}$

$\quad \mu([k\ ;\ any\ ;\ any\ ;\ k, b]\ ;\ \mathsf{row}\ R)$

$= \{$ since any is a type, k is a constant $\}$

$\quad \mu([k, b]\ ;\ \mathsf{row}\ R)$

$= \{$ equation 2 $\}$

$\quad \mu([k\ ;\ \mathcal{D}, b]\ ;\ \mathsf{row}\ R)$

$= \{$ equation 4 $\}$

$\quad \mu([\mathcal{D}, b]\ ;\ \mathsf{row}\ R\ ;\ \mathsf{snd}\ k)$

$= \{$ equation 11 $\}$

$\quad \mu([\mathcal{D}, b]\ ;\ \mathsf{row}\ R)$

$= \{$ equation 3 $\}$

$$b\ ;\ \mu(\mathsf{fst}\ \mathcal{D}\ ;\ \mathsf{row}\ R) \tag{12}$$

which has just the form required to apply equation 10.

9 ASSEMBLING THE RESULTS

To complete the development of the adder we need a constraint on the input to meet the hypothesis in equation 11 with $k = zero$. We will take $b = sum'_{2^N-1} \setminus zip$, although with a bit more work we could have got a slightly better result.

$$\begin{aligned}
sum'_{2^N-1}\ ;\ A \quad &= \{ \text{equation 1, since } sum'_x\ ;\ sum'_y = sum'_{\min(x,y)} \} \\
&\quad sum'_{2^N-1}\ ;\ zip\ ;\ \pi_2^{-1}\ ;\ \mathsf{fst}\ zero\ ;\ \mathsf{row}\ fa\ ;\ \pi_1\ ;\ N \\
&= \{ \text{since } sum'_x = zip\ ;\ zip^{-1}\ ;\ sum'_x \} \\
&\quad zip\ ;\ b\ ;\ \pi_2^{-1}\ ;\ \mathsf{snd}\ zero\ ;\ \mathsf{row}\ fa\ ;\ \pi_1\ ;\ N \\
&\quad \text{where } b = sum'_{2^N-1} \setminus zip \\
&= \{ \text{equation 12, since } \mathsf{snd}\ b\ ;\ \mathsf{row}\ fa = \mathsf{snd}\ b\ ;\ \mathsf{row}\ fa\ ;\ \mathsf{snd}\ zero \} \\
&\quad zip\ ;\ b\ ;\ \mu(\mathsf{fst}\ \mathcal{D}\ ;\ \mathsf{row}\ fa)\ ;\ N \\
&= \{ \text{equation 10, since } b = b\ ;\ N \} \\
&\quad sum'_{2^N-1}\ ;\ zip\ ;\ (\mu(\mathsf{fst}\ \mathcal{D}\ ;\ \mathsf{slow}\ fa)) \setminus \underline{bundle} \\
&= \{ \text{since } fa \text{ is stateless, so } fa = \mathsf{slow}\ fa \} \\
&\quad sum'_{2^N-1}\ ;\ zip\ ;\ (\mu(\mathsf{fst}\ \mathcal{D}\ ;\ fa)) \setminus \underline{bundle} \\
&= \{ \text{equation 3, using } fa = \mathsf{snd}\ 2\ ;\ fa \} \\
&\quad sum'_{2^N-1}\ ;\ zip\ ;\ \underline{bundle}^{-1}\ ;\ 2\ ;\ \mu(\mathsf{fst}\ \mathcal{D}\ ;\ fa)\ ;\ \underline{bundle} \\
&= \{ \text{equation 6, and } sum'_x = sum'_x\ ;\ zip\ ;\ zip^{-1} \} \\
&\quad sum'_{2^N-1}\ ;\ [\underline{bundle}^{-1}, \underline{bundle}^{-1}]\ ;\ \mu(\mathsf{fst}\ \mathcal{D}\ ;\ fa)\ ;\ \underline{bundle}
\end{aligned}$$

and composing on both sides with $[\underline{bundle}, \underline{bundle}]$ on the left and $\underline{bundle}^{-1}$ on the right

$$\begin{aligned}
&sum''_{2^N-1}\ ;\ [\underline{bundle}\ ;\ bin, \underline{bundle}\ ;\ bin]\ ;\ add\ ;\ (\underline{bundle}\ ;\ bin)^{-1} \\
&\quad = \quad sum''_{2^N-1}\ ;\ \mu(\mathsf{fst}\ \mathcal{D}\ ;\ fa) \\
&\quad\quad\ \ \text{where } sum''_x = sum_x \setminus [(\underline{bundle}\ ;\ bin)^{-1}, (\underline{bundle}\ ;\ bin)^{-1}]
\end{aligned}$$

The term $\mu(\mathsf{fst}\,\mathcal{D}\,;fa)$ is the actual circuit, illustrated in figure 2, consisting of a stateless full-adder with a single one-bit latch on a feedback loop which takes the carry signal. The left-hand side of the equation shows that all three numbers are represented by $\underline{bundle}\,;bin$, that is as N-bit binary numbers given bit-serially with least-significant bit first. Finally, the sum''_{2^N-1} is a constraint which describes the way that input must be supplied: correct operation is guaranteed if the input represents pairs which each sum to less than 2^N-1.

10 CONCLUSION

We have shown how we can deal with the systematic development of a bit-serial adder in the Ruby notation which had previously been used only to develop parallel circuits. The development of the adder itself makes no recourse to arguments about when inputs are provided nor when states are recorded.

Although almost all of the results quoted in this paper are used in the development of the adder, few of them are specific to this circuit. The bulk of the results could equally appear in many other developments. Indeed of all the equations, only those in which *add* or *fa* appear involve facts particular to the adder.

The material in this paper about time and delays was originally developed to deal with pipelining of bit-parallel circuits; that dealing with feedback was for calculating with state-machines; the work on $\underline{bundle}$ and slowing was originally for treating circuits in which only a proportion of the components are active on each cycle. By combining techniques from these three areas we have been able to deal with bit-serial circuits. There is scope for taking this work further, for example although $\underline{bundle}\,;bin$ describes only least-significant bit first representations, we can perform similar calculations with $\underline{bundle}\,;reverse\,;bin$, to deal with most-significant bit first representations.

This gives us some confidence that we could go on to develop a general theory of serialisation, making it possible to reason precisely about the balance between the time and space efficiency of circuits which we can know with great confidence to be otherwise identical in behaviour.

REFERENCES

1 G. Jones and M. Sheeran, *Circuit design in Ruby*, in [6]. 13–70.

2 G. Jones and M. Sheeran, *Relations and refinement in circuit design*, in [7]. 133–152.

3 G. Jones and M. Sheeran, *Timeless truths about sequential circuits*, in [8]. 245–259.

4 M. Sheeran, *Retiming and slowdown in Ruby*, in [9]. 289–308.

5 C. E. Leiserson and J. B. Saxe, *Retiming Synchronous Circuitry*, Tech. Report 13, Digital Systems Research Center, Palo Alto, California, 1986.

6 Jørgen Staunstrup (ed.), *Formal methods for VLSI design*, North-Holland, 1990.

7 C. C. Morgan, J. C. P. Woodcock (eds.), *3rd Refinement Workshop*, (Proc BCS FACS workshop, Hursley, January 1990) Springer 1991.

8 S. K. Tewksbury, B. W. Dickinson and S. C. Schwartz (eds.), *Concurrent computations: algorithms, architecture and technology*, Plenum Press, New York, 1988.

9 G. J. Milne (ed.), *The fusion of hardware design and verification*, North-Holland, 1988.