# Contents

# Chapter 1

# Circuit design in Ruby

G. Jones[1] and M. Sheeran[2]

## 1.1 Introduction

The business of making integrated circuits is a peculiarly difficult engineering task largely because it spans such a wide range of levels of abstraction. Figure 1.1 suggests the sorts of names that might be used to describe these layers. A good design process must necessarily insulate each step of the design – as far as is possible – from the concerns that are most readily understood and tackled at other levels. As used here, the word *designing* simply means making progress down the spectrum from specification to implementation.

On the whole, we are not going to be concerned with details of implementation like gates and transistors. So to us the word *circuit* means no more than the sort of algorithm that might well be implemented by something like an integrated circuit or a collection of them. Designing circuits is the business of filling in the gap in the figure between specification capture and circuit fabrication.

That means, for example, that we will be interested in highly parallel algorithms with a more or less static structure; also that we are going to be concerned to minimize expensive sorts of communication, like broadcasting, and in emphasizing locality. We will not, however, be at all concerned with the details of any fabrication technology; nor in such concerns as how many transistors can be fabricated on a single chip.

[1] Oxford University Computing Laboratory, 11 Keble Road, Oxford OX1 3QD, England
[2] Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland

Vague specification
Non-executable specification — capture
Abstract algorithm
Word-level algorithm
Bit-level algorithm
Gates
Transistors — fabrication
Rectangles
Silicon

Figure 1.1: the design process

It is quite common for the design process to be realized by a number of steps passing from one level to a lower level, where the designer 'invents' the implementation at the next level and then proves – after the fact – that this new design does indeed implement the specification at the preceding level. These *proof* steps would be upwards in the figure.

On the other hand, some steps of the design process consist of a purely mechanical transformation of specification into implementation: for example, the turning of a set of masks into a pattern on silicon. These can be thought of as *compilation* steps, and require no ingenuity or invention. Indeed, they are usually tedious and are best left to be done by a machine.

We tend to use of the word *calculation* to mean stages in the design process which proceed forwards, without large inventive leaps and subsequent proofs of correctness, but yet without necessarily being purely mechanical. Ruby is a framework in which to do these calculations.

### 1.1.1 The shape of this chapter

This chapter introduces Ruby, and a calculational style in which to explore the design of digital signal processing circuits and similar devices and algorithms.

The fundamental idea introduced by §1.2 is that circuits are built from parts by a process of *composition*, and that this composition has simple mathematical properties like those of the composition of functions and the composition of relations. That gives us a way of describing circuits which are composed 'sequentially' by being connected to each other, and in contrast §1.3 describes a 'parallel' composition of parts which operate independently on structured inputs: lists and tuples of signals.

The reality is that most circuit structures lie somewhere between these two extremes, being only partly interconnected and partly independently parallel. There are particular patterns of interconnection that arise often, and which make sense under the constraints on the feasibility of layout. In the subsequent sections such patterns of interconnection are discussed in Ruby, and we explore the mathematical properties of circuits which are composed in these ways. The idea is to become familiar with these properties so that they can be used to explore design choices in developing a circuit.

One of the important characteristics of Ruby is that it is no more difficult to deal with sequential circuits – ones with internal state – than it is to deal with combinational circuits. The design is done by the same sort of calculation in both cases, in the same sort of algebra. In §1.6 the interpretation of Ruby expressions as sequential circuits is introduced. We then go on to discuss manipulations like pipelining and data-skewing which are specific to sequential circuits.

By this point you should have enough Ruby to be able to tackle a sizeable example, and §1.7 does just that. It outlines a path through the design of a systolic correlator, from initial decisions about how to represent the interface to the consequences this has for the final circuit, and ultimately to optimizations of timing performance.

All of the circuit forms discussed up to that point are essentially iterative, but §1.8 shows that the same sorts of ideas are applicable to circuits like the 'butterfly' network that are naturally described in a recursive fashion. Butterfly networks are common in many sorts of circuits, but are perhaps most familiar from implementations of the 'fast Fourier transform'. Finally §1.9 shows that the butterfly implementation of the FFT can be discovered by calculation from the specification in just the same way as we did with the correlator.

### 1.1.2 The rôle of pictures

Throughout this chapter there are pictures that are intended to motivate the discussion that goes with them. At least for some people, the pictures help with an intuitive understanding of what an equation means, or of why a calculation proceeds in a certain way.

These 'abstract floorplans' share many of the important properties of the circuits with which a designer would be concerned. Principal amongst these is *locality*: whether or not components are connected to components that are near to them. Because the costs which dominate in circuit construction are those of communication, locality is very important. Similarly, the extent to which connections cross over each other shows up, and this is another cost in circuit construction.

Beware, however, of reasoning about the pictures. In particular, notice that quite often the pictures are just particular instances of the equations and calculation which they portray. There is a danger in trying to reason from the example in the picture that you might generalize a result which happens only to be true for the particular instance.

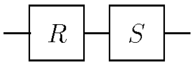



Figure 1.2: the composition $R\,;S$

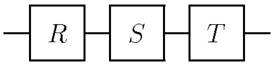



Figure 1.3: composition is associative: $(R\,;S)\,;T = R\,;S\,;T = R\,;(S\,;T)$

## 1.2 Composition and inverse

The central idea in Ruby is that of putting two circuits together to co-operate with each other. The composition of two circuits $R$ and $S$ will be written $R;S$ and you should have in mind the idea of a circuit in which connections are made from one side of $R$ to the other side of $S$, rather as shown in figure 1.2. The 'left-hand' edge of a circuit will be called its *domain* and the 'right-hand' edge its *range*. The domain of $R\,;S$ is that of $R$, its range is that of $S$, and the range of $R$ is connected to the domain of $S$. The fact that the domain appears on the left of the picture and the range on the right is of course just a convention, but will be one to which we adhere much of the time.

You should think of a circuit as being a relation which holds between the signals on its connections with the outside world. This is the relation which it enforces by arranging that the values of its outputs are consistent with those of its inputs. Beware the temptation to think of the domain signals as inputs and the range signals as outputs. The division between domain and range is a purely geographical one, and there may be inputs and outputs on either side of one of our circuits.

Composition of relations is defined by $x\,(R\,;S)\,z \iff \exists y.\, x\,R\,y\ \&\ y\,S\,z$. This is an associative operation, which is to say that the order in which the components of figure 1.3 are assembled cannot affect the meaning of the circuit. Associativity gives us an excuse for leaving out the parentheses in, for example, $R\,;S\,;T$.

Later when we come to deal with sequential circuits it will prove necessary to change our idea of what a signal is, and indeed of what a circuit is; but the things that stay the same are the laws about circuit forms. So, for example, composition will stay associative. It will pay to start as we mean to go on. You should try to avoid reasoning about the data – $x$, $y$, $z$ in this example – and as far as possible, you should try to avoid reasoning about the specific component circuits – $R$, $S$, $T$ here – concentrating instead on the combining forms, like composition, and the laws which they obey. The associative law – that $(R\,;S)\,;T = R\,;(S\,;T)$ for any $R$, $S$ and $T$ – is not about the data on which the circuit operates, nor is it about any particular circuit components $R$, $S$ and $T$; rather it is a general statement about ';'.

### 1.2.1 Repeated composition

Where it makes sense to plug two components of the same kind together, we can talk about the $n$-times repeated composition of $R$, written $R^n$, and which is defined by $R^1 = R$, and $R^{n+1} = R\,;R^n$.

Since we know that composition is associative, we already know many things about repeated composition, for example $R^{n+1} = R^n\,;R$, and $R^{m+n} = R^m\,;R^n$, and so on. The proofs can be done by induction on $n$, and notice that the proof depends only on the associativity of composition, so there is no need to appeal to the meaning of $R$ at all.

Suppose we assume for some $R$ and $S$ that $R\,;S = S\,;R$, then a simple induction shows that $R\,;S^n = S^n\,;R$. A similar induction shows that on the same assumption $(R\,;S)^n = R^n\,;S^n$.

Now, notice that although $R\,;S^n = S^n\,;R$ and $(R\,;S)^n = R^n\,;S^n$ are only proved in case $R$ and $S$ commute (in case $R\,;S = S\,;R$), and so are statements about $R$ and $S$; nevertheless the theorems 'If $R\,;S = S\,;R$ then $R\,;S^n = S^n\,;R$' and 'If $R\,;S = S\,;R$ then $(R\,;S)^n = R^n\,;S^n$' are – just like the associative law – statements about composition.

These statements about composition can be added to the collection of laws that we are accumulating, and which will eventually constitute a body of knowledge about the *forms* of all circuits.

### 1.2.2 Inverse

The inverse of a circuit is its left-to-right reflection: the connections on the left of $R$ appear on the right of $R^{-1}$ and vice versa. Since reflecting twice gets us back the original circuit, we require that for any circuit $R$ the inverse of its inverse should be the same circuit: $(R^{-1})^{-1} = R$. Inverse is often called converse, and is defined by $x\,R^{-1}\,y \iff y\,R\,x$.

Consider the relation which represents negation of Booleans: $\mathsf{T}\ \mathit{not}\ \mathsf{F}$ and $\mathsf{F}\ \mathit{not}\ \mathsf{T}$. Its inverse $\mathit{not}^{-1}$ is of course the same relation. That means that $\mathit{not}^{-1} = \mathit{not}$, so you should be aware that 'reflecting the picture' of *not* leaves it unchanged. In particular, there is no idea that the inputs and outputs of a gate have been exchanged: the sense in which $\mathit{not}^{-1} = \mathit{not}$ abstracts from such concerns as which signals are inputs and which are outputs.

The next question to ask is how inverse interacts with composition. As shown in figure 1.4 you can reflect the picture of a composition by swapping the components and reflecting each of them, so keeping corresponding edges connected. Stated as a Ruby law, we have that $(R\,;S)^{-1} = (S^{-1})\,;(R^{-1})$ for any $R$ and $S$.

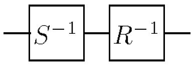



Figure 1.4: the inverse of a composition: $(R\,;S)^{-1} = (S^{-1})\,;(R^{-1})$

It follows that $(R^n)^{-1} = (R^{-1})^n$, so as a notational convenience we define $R^{-n} = (R^{-1})^n$. Be careful not to let this representational recklessness lead you astray – the induction that proves $R^{m+n} = R^m\,;R^n$ does so for positive $n$. Notice that since we know nothing in general about $R\,;R^{-1}$, it is not necessarily the case that $R^{m-n} = R^m\,;R^{-n}$. In this respect at least it can be argued that it was a mistake to christen the reflection of a circuit its 'inverse' with all the connotations that has.

### 1.2.3 Identity and types

Having defined $R^n$ for positive and negative $n$, it is natural to seek an interpretation for $R^0$ which fits neatly into the remaining gap. The time has come to be honest about types. (Perhaps, since we are talking about 'circuits' and not relations, it might have been better to talk of '*interfaces*' rather than types.)

As far as will concern us, a type is just a collection of values. In the world of relations, the things that look like types are equivalence relations: an equivalence relation determines a set of equivalence classes, and the set of equivalence classes determines the equivalence relation. To describe what it is to be a type, we have to encode in Ruby the properties that make a relation be an equivalence relation.

An equivalence relation is one which is reflexive, symmetric and transitive: symmetry is just that $R = R^{-1}$; transitivity of a reflexive relation is just that $R = R^2$ (you can show that any transitive relation contains its square, but you need reflexivity to show containment of the relation in the square). So any $R$ for which $R = R^{-1} = R^2$ (and so $R = R^n$ for all non-zero $n$) is an equivalence on all the relevant values; such an $R$ we call a *type*. It will clearly do no harm to our intuition to define $R^0 = R$ so long as $R$ is a type.

A type $D$ is a type of the domain (not, notice, *the* type of the domain) of a circuit $C$ if $C = D\,;C$, and similarly a type $R$ is a type of its range if $C = C\,;R$. Sometimes this will be written as $C : D \rightarrow R$, and we speak of $D \rightarrow R$ as a type for $C$. When speaking of a circuit $C$, we ought really to quote a particular $D$ and $R$, and speak of the circuit $C : D \rightarrow R$. Composition makes much more sense if you only ever compose circuits with matching types, say $P : A \rightarrow B$ and $Q : B \rightarrow C$, in which case $(P\,;Q) : A \rightarrow C$, as you can check from the definition.

In that case, it makes sense to talk of repeated composition only when you

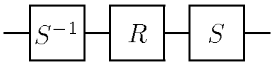



Figure 1.5: the conjugation $R \setminus S$ of $R$ by $S$

have in mind a *homogeneous* component $R : T \rightarrow T$, in which case $R^n : T \rightarrow T$ for all positive and negative $n$. For that reason it is intuitively safe to define $R^0$ to be $T$, because then certainly $R^{n+0} = R^n = R^n ; T = R^n ; R^0$ and similarly for $R^{0+n}$ and for negative powers. Be careful with this definition: strictly speaking it does not make sense to talk about $R^0$ without making explicit which particular type you have in mind for $R$ – remember that the type is not unique – and a pedant would insist on saying that it is $(R : T \rightarrow T)^0$ that is defined to be $T$.

Of course the reason for being sloppy is the usual one: that it does not often matter all that much exactly what choice you make, and when it does the right choice is obvious. You should beware, however, of any intuition that leads you to expect $R^0$ necessarily to be the identity of composition. That would be an $\iota$ for which $\iota ; Q = Q = Q ; \iota$ for all $Q$.

### 1.2.4 Conjugation

A circuit and its inverse often appear together bracketing another component, as in figure 1.5. The conjugation of $R$ by $S$ is defined by $R \setminus S = S^{-1} ; R ; S$.

A conjugation like $R \setminus S$ behaves rather like the inside $R$ part, and you can think of $S$ as a 'translation' (in almost any of the many different senses of that word) applied to the language in which you communicate with $R$. For example, $(R \setminus S)^{-1} = R^{-1} \setminus S$, so the inverse of a conjugation of $R$ is the same conjugation of the inverse of $R$.

Conjugations can be composed in the obvious way, because $(R \setminus S) \setminus T = R \setminus (S ; T)$. It is almost the case that the composition of two conjugations is the conjugation of the compositions. If $R ; R^{-1}$ is a type of the range of $S$ and the domain of $T$, then

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

Such an $R$ is called a *representation* of the type $R ; R^{-1}$, and $R^{-1}$ is an *abstraction* to that type.

The idea behind these names is that we start from some operation defined on some abstract objects, say negation of Booleans: $\mathsf{T}$ *not* $\mathsf{F}$ and $\mathsf{F}$ *not* $\mathsf{T}$. Suppose that we choose to represent Booleans by bits: each bit value is given

$$\begin{tikzcd}
(rep \,;\, rep^{-1}) \bullet \arrow[r, "R"] & \bullet\, (rep \,;\, rep^{-1}) \arrow[d, "rep"] \\
\bullet \arrow[u, "rep^{-1}"] \arrow[r, "R \setminus rep"] & \bullet
\end{tikzcd}$$

Figure 1.6: $R \setminus rep$ implements $R : (rep \,;\, rep^{-1}) \rightarrow (rep \,;\, rep^{-1})$

a meaning by the representation: say $\mathsf{F}\; rep\; 0$ and $\mathsf{T}\; rep\; 1$. You can calculate that $rep \,;\, rep^{-1}$ is the identity relation on the Booleans. Then $not \setminus rep$ is an implementation of the abstract negation operation, which if it is given a representation of a bit relates it to a representation of its negation. You can calculate that $0\,(not \setminus rep)\,1$ and $1\,(not \setminus rep)\,0$.

The condition that $rep\,;rep^{-1}$ be a type is just the requirement that if you take any abstract object of that type, then *rep* will give at least one concrete representative of that object, and that any representative given by *rep* will be a representative of that same abstract object. That is to say, *rep* gives a faithful representation of the type.

## 1.3 Lists and tuples

In order to talk about operations on collections of data, we need some structured types like lists and tuples. We are going to blur the distinction that people usually make between lists and tuples: there will be no difference between a triple of things that happen to have a common type, and a list of things that have that type and which happens to be three elements long. This makes some things easier, and some things – like mechanical type deduction – tremendously difficult.

### 1.3.1 Parallel composition

The fundamental operation that builds circuits that operate on lists is *parallel composition*. The parallel composition $[R, S]$ of $R$ and $S$ is a circuit whose domain and range are pairs of signals – as shown in figure 1.7 – which has an $R$ component operating on the first elements of the pairs, and an $S$ component independent of it operating on the second elements. That is to say $\langle x, y\rangle\,[R, S]\,\langle u, v\rangle$ exactly when both $x\,R\,u$ and $y\,S\,v$. Notice the following convention, to which the pictures in this chapter adhere: that the first component of a list of signals is on the left if you stand in the range of a circuit, looking towards

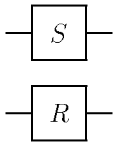


Figure 1.7: the parallel composition $[R, S]$

the domain. Since the range is on the right of figure 1.7, and the domain on the left, the first component is at the bottom and the last at the top.

The independence of the components of a parallel composition is reflected in the way that composition and inverse distribute over it, as $[R, S] \,;[T, U] = [R \,; T, S \,; U]$ and $[R, S]^{-1} = [R^{-1}, S^{-1}]$. Beware the apparently misleading punctuation in $[R \,; T, S \,; U]$, which can only mean $[(R \,; T), (S \,; U)]$.

### 1.3.2 Pairs and projections

The parallel composition of a circuit with an identity arises so often that we abbreviate the forms $\mathsf{fst}\, R = [R, \iota]$, read 'first $R$', and $\mathsf{snd}\, R = [\iota, R]$ read 'second $R$'. As you can calculate from the definition, a first and a second commute with each other, $\mathsf{fst}\, R ; \mathsf{snd}\, S = \mathsf{snd}\, S ; \mathsf{fst}\, R$, and each distributes over composition and inverse, for $\mathsf{fst}(R \,; S) = \mathsf{fst}\, R \,; \mathsf{fst}\, S$ and $(\mathsf{fst}\, R)^{-1} = \mathsf{fst}\, R^{-1}$, and similarly for second.

We will also often need to extract just one component of a pair, which we do with projections: $\pi_1$ projects a pair onto its first component, $\langle x, y\rangle \; \pi_1 \; x$, and $\pi_2$ onto its second, $\langle x, y\rangle \; \pi_2 \; y$. The inverse of a projection, of course, injects its domain into a pair (with an arbitrary other component) so that ${\pi_1}^{-1} \,; \pi_1 = \iota$ is the universal type, as is ${\pi_2}^{-1} \,; \pi_2$. The projections give us a way of talking about things that behave like pairs, and so of talking about operations on more than one argument. They satisfy $[R, S] \,; \pi_1 = \mathsf{snd}\, S \,; \pi_1 \,; R$ and $[R, S] \,; \pi_2 = \mathsf{fst}\, R \,; \pi_2 \,; S$.

Beware that although $\mathsf{fst}\, R \,; \pi_1 = \pi_1 \,; R$, in general $[R, S] \,; \pi_1 \neq \pi_1 \,; R$ for every $S$. If $S$ is the empty relation then so is $[R, S]$, and so is any composition in which it appears, but $\pi_1 \,; R$ need not be empty. So it is the case that $(\mathsf{fst}\, R) \setminus \pi_1 = R$, but $[R, S] \setminus \pi_1 \neq R$.

### 1.3.3 Types for lists

Without going into any further formality, the notion of the parallel composition of two circuits extends naturally to the composition of any number of components. In particular, there is the parallel composition of one component alone, written $[R]$. The domain signal of this circuit is a singleton list, as is its range signal. Writing 1 for the type $[\iota]$ of singletons, $[R] : 1 \to 1$.

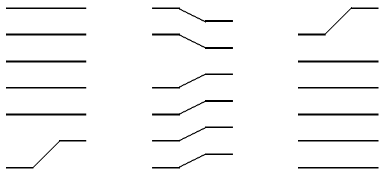


Figure 1.8: instances of $apl$, $app$, and $apr^{-1}$

In a picture it is hard to imagine there being any difference between an element of some type and a list of signals that consists of just one signal of that given type. The formalization of this similarity is that a list of one thing can be used to represent that thing, and any thing can be used to represent a list consisting of that thing alone. The abstraction that relates signals to singleton signals is here written $[\text{-}] : \iota \rightarrow 1$, which you can read as 'singleton' (or 'gift-wrap'). It is a representation for any type, which is to say that $[\text{-}] \, ; [\text{-}]^{-1} = \iota$, and it is an abstraction to singletons, which is to say that $[\text{-}]^{-1} \, ; [\text{-}] = 1$.

In order to talk about longer lists, more constants ('plumbing circuits') are needed. The thing that corresponds to appending lists is the $app$ circuit for which

$$\begin{aligned}&[R_0, R_1, \ldots, R_i, R_{i+1}, R_{i+2}, \ldots, R_n] \\ &\quad = \quad [[R_0, R_1, \ldots, R_i], [R_{i+1}, R_{i+2}, \ldots, R_n]] \setminus app\end{aligned}$$

for any $n$ and $i \leq n$. In terms of relations on signals, $app$ relates a pair of lists to the list obtained by concatenating them; but note that all the names in this equation are names of circuits, rather than signals.

You may be more familiar with the idea of building up lists one component at a time: $apl = \mathsf{fst}\,[\text{-}] \, ; app$, read 'append left' or by some people 'cons', is a circuit which adds one signal to the left of a list; and $apr = \mathsf{snd}\,[\text{-}] \, ; app$, read 'append right' or 'snoc', adds a signal to the right-hand end of a list. For example, $[R, [S, T]] \, ; apl = apl \, ; [R, S, T]$ and $[[R, S], T] \, ; apr = apr \, ; [R, S, T]$ so $[R, [S, T]] \, ; apl \, ; apr^{-1} = apl \, ; apr^{-1} \, ; [[R, S], T]$. Notice that $[R, [S, T]]$ and $[[R, S], T]$ are different circuits, each relating pairs, and each is different from $[R, S, T]$ which relates triples.

Structures with one component have type $1 = [\iota]$. Those with $n + 1$ components have one more component than those of length $n$, and can be written as any one of $n{+}1 = \mathsf{snd}\, n \setminus apl = [1, n] \setminus app = [n, 1] \setminus app = \mathsf{fst}\, n \setminus apr$. Neither $apl \, ; apl^{-1}$ nor $apl^{-1} \, ; apl$ is the identity, but each is a type: $apl^{-1} \, ; apl$ is the type of lists of at least one element, and $apl \, ; apl^{-1}$ that of pairs with a list in the second element; similarly for $apr$. In order to complete the picture, the natural definition for $0 = [\,]$, the parallel composition of no component

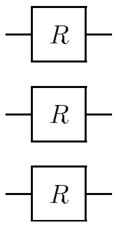



Figure 1.9: an instance of $\mathsf{map}\,R$

circuits, makes it behave like the identity relation on zero length lists.

### 1.3.4 Map

The parallel compositions that can be written with the notation above are all circuits of a given, fixed width. So, for example, $[R, R]$ takes pairs of signals in the domain and in the range. The 'map' construction generalizes this, so that $\mathsf{map}\,R$ is the parallel composition of any number of $R$ circuits. That means that $1 \,;\, \mathsf{map}\,R = [R] = \mathsf{map}\,R \,;\, 1$ and $2 \,;\, \mathsf{map}\,R = [R, R] = \mathsf{map}\,R \,;\, 2$, and so on. Figure 1.9 shows an instance of $\mathsf{map}\,R$; this is an example of a picture from which it would be dangerous to generalize, because it has many properties not shared by, for example, $(\mathsf{map}\,R) \setminus 137$, or $(\mathsf{map}\,R) \setminus 1$.

By induction on the width you can show that $\mathsf{map}$ distributes over sequential composition and inverse in the same way that parallel composition does, so $\mathsf{map}(R \,;\, S) = (\mathsf{map}\,R) \,;\, (\mathsf{map}\,S)$ and $\mathsf{map}(R^{-1}) = (\mathsf{map}\,R)^{-1}$. It follows that if $T$ is a type, then so is $\mathsf{map}\,T$, in fact it is the type of all lists of signals of type $T$ and in particular $\mathsf{map}\,\iota$ is the type of lists.

There are a great number of things that you might want to be told about $\mathsf{map}$ to be sure that we were all thinking of the same circuit structure, amongst them perhaps

$$\begin{array}{rcl}
1 \,;\, \mathsf{map}\,R & = & [R] \\
n \,;\, \mathsf{map}\,R & = & \mathsf{map}\,R \,;\, n \\
[\,\text{-}\,] \,;\, \mathsf{map}\,R & = & R \,;\, [\,\text{-}\,] \\
app \,;\, \mathsf{map}\,R & = & \mathsf{map}\,\mathsf{map}\,R \,;\, app \\
 & = & [\mathsf{map}\,R, \mathsf{map}\,R] \,;\, app \\
apl \,;\, \mathsf{map}\,R & = & [R, \mathsf{map}\,R] \,;\, apl \\
apr \,;\, \mathsf{map}\,R & = & [\mathsf{map}\,R, R] \,;\, apr
\end{array}$$

but it is clear that some of these are consequences of the others. In fact, you

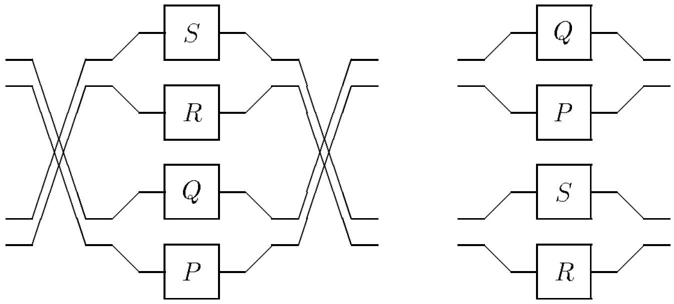



Figure 1.10: $[[P, Q], [R, S]] \setminus rev$ and the equivalent $[[R, S], [P, Q]]$

ought to be convinced by being told only that

$$\begin{aligned} [\text{-}] ; \mathsf{map}\, R &= R ; [\text{-}] \\ [m, n] ; app ; \mathsf{map}\, R &= [\mathsf{map}\, R, \mathsf{map}\, R] ; [m, n] ; app \\ (\mathsf{map}\, R)^{-1} &= \mathsf{map}(R^{-1}) \end{aligned}$$

and perhaps some equations involving 0. These equations for all $m$ and $n$ are more than enough to define $\mathsf{map}$.

### 1.3.5 Reverse

There is clearly a connection between left- and right-handed views of lists. Since any finite list could have been built up in either way, every time you say something about $apl$ you are saying something about $apr$, and every time you say something about both there is a danger of saying something inconsistent about them.

To make it easier to talk about the mirror-similarity of $apl$ and $apr$, we use a piece of plumbing $rev : \mathsf{map}\, \iota \rightarrow \mathsf{map}\, \iota$, read 'reverse', which flips a list over. In particular, $swap = rev \setminus 2$ exchanges two signals, so $rev$ satisfies $rev ; [R, S] = [S, R] ; rev$. For other lengths of list it can be defined by $0 ; rev = 0$ and $[\text{-}] ; rev = [\text{-}]$ and $app ; rev = [rev, rev] ; rev ; app$.

Immediately from these definitions, you can calculate that $apl ; rev = rev ; \mathsf{fst}\, rev ; apr$ and $apr ; rev = rev ; \mathsf{snd}\, rev ; apl$. By induction on the length of the list, $rev = rev^{-1}$ and $rev ; \mathsf{map}\, R = \mathsf{map}\, R ; rev$ and so on.

Because $rev$ reverses a list of signals, and because the convention about drawing circuits puts the beginning of a list at the bottom of the picture, the end of the list at the top, you might expect that a picture of $R \setminus rev$ would be the same as that of $R$ but flipped over about a horizontal axis. This is almost right, but be careful: suppose $R$ were a circuit operating on lists of lists, then the outermost list would be reversed but the elements in that list remain in

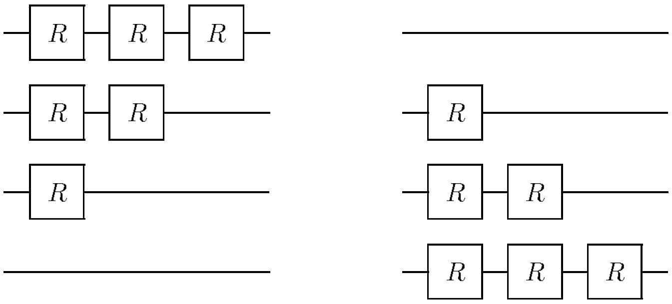



Figure 1.11: an instance of $\mathsf{tri}\,R$, and an instance of $\mathsf{irt}\,R$

the same order, so for example $[[P,Q],[R,S]] = [[R,S],[P,Q]] \setminus \mathit{rev}$ as can be seen in figure 1.10.

Reverse is the first example in Ruby of something that you might need to describe, but which it would be unpleasant to have to implement in a finished circuit. Any non-trivial instance of reverse drawn as a planar picture has to contain crossing wires: $\mathit{rev} \setminus 2$ contains exactly one crossing, $\mathit{rev} \setminus n$ contains $n(n-1)/2$. In the course of designing a circuit it will usually be an aim to eliminate instances of $\mathit{rev}$, and if that can not be done at least to reduce their width.

### 1.3.6 Triangle

A useful construction that is closely related to $\mathsf{map}$, but which seems to be almost peculiar to the sorts of programs that arise in describing circuits, is the triangle. Instances of this are illustrated in figure 1.11. For any homogeneous $R : T \rightarrow T$, the triangle $\mathsf{tri}\,R : \mathsf{map}\,T \rightarrow \mathsf{map}\,T$ relates lists of signals, just as $\mathsf{map}$ would, but it relates the $i$-th components according to $R^i$. For example, if $(\times 2)$ is the circuit which doubles a number, $\mathsf{tri}(\times 2)$ relates a list of zeroes and ones to a list of those bits weighted by the powers of two which they would represent in a binary number with its least significant bit first. Similarly, $(\mathsf{tri}(\times 2)) \setminus \mathit{rev}$ would give them the weights appropriate to having the most significant bit first. The form $(\mathsf{tri}\,R) \setminus \mathit{rev}$ appears so often that we really need a name for it; in this chapter we adopt Robin Sharp's notation for it, which is $\mathsf{irt}\,R$.

It should suffice to tell you that $\mathit{apl}\,;\mathsf{tri}\,R = [R^0, \mathsf{map}\,R\,;\mathsf{tri}\,R]\,;\mathit{apl}$, that $(\mathsf{tri}\,R)^{-1} = \mathsf{tri}(R^{-1})$, and that $0\,;\mathsf{tri}\,R = 0\,;\mathsf{map}\,R$, but of course there is a rich collection of laws that can be derived from these and the properties of other operators by induction over lists. Among the more useful ones are those that relate triangles and maps of the same components such as $(\mathsf{tri}\,R)^n = \mathsf{tri}(R^n)$ and $\mathsf{tri}\,R\,;\mathsf{map}\,R = \mathsf{map}\,R\,;\mathsf{tri}\,R$ and those about commuting circuits, for

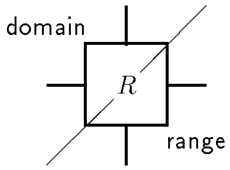



Figure 1.12: a different picture of $R : 2 \rightarrow 2$

if $R \mathbin{;} S = S \mathbin{;} R$ you can show that $\mathsf{tri}\, R \mathbin{;} \mathsf{tri}\, S = \mathsf{tri}(R \mathbin{;} S)$ and for types $\mathsf{tri}(R^0) = \mathsf{map}(R^0)$ and so $\mathsf{tri}\, R : \mathsf{map}\, R^0 \rightarrow \mathsf{map}\, R^0$.

Decomposing a triangle with $apr$ instead of $apl$, or decomposing a reversed triangle with $apl$, you can show by induction on $n$ that $\mathsf{fst}\, n \mathbin{;} apr \mathbin{;} \mathsf{tri}\, R = [n \mathbin{;} \mathsf{tri}\, R, R^n] \mathbin{;} apr$ and it turns out that this configuration appears often: an $R^n$ will arise somewhere where by writing $(\mathsf{tri}\, R) \setminus apr^{-1}$ or $(\mathsf{irt}\, R) \setminus apl^{-1}$ you can avoid mentioning $n$ altogether.

## 1.4 Rows and columns

There are fundamental reasons for being interested in circuits which can be laid out in a little more than two dimensions. Any very large system with many connections has to be more or less flat. This means that there are good reasons to be interested in two-dimensional meshes. This section is almost entirely about circuits $R : 2 \rightarrow 2$ with a pair of signals in the domain and a pair of signals in the range. To make pictures of networks of these circuits lie naturally on the page, they are drawn by following a new convention illustrated in figure 1.12. The first component of the domain lies to the left, the second component above the circuit; the first component of the range lies below the circuit, and the second component to its right. This keeps the order of the signals on the paper consistent with the convention given earlier for drawing circuits that operate on pairs.

### 1.4.1 Beside and below

There are two ways of connecting these square tiles that seem natural: as shown in figure 1.13. In each case the wires are grouped so that both $R \leftrightarrow S$, read '$R$ beside $S$', and $R \updownarrow S$, read '$R$ below $S$', are also of type $2 \rightarrow 2$; indeed $R \leftrightarrow S : \mathsf{snd}\, 2 \rightarrow \mathsf{fst}\, 2$ and $R \updownarrow S : \mathsf{fst}\, 2 \rightarrow \mathsf{snd}\, 2$.

It is clear that circuits made by beside and below are not new ones, in the sense that we already know enough wiring to be able to duplicate their function without any new primitives. Reading off the picture in figure 1.14,

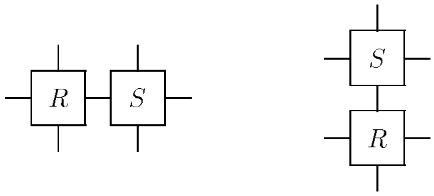
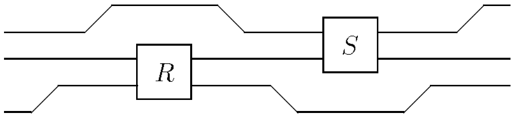



Figure 1.13: $R$ beside $S$ making $R \leftrightarrow S$, and $R$ below $S$ making $R \updownarrow S$




$apl\, apr^{-1}\, \mathsf{fst}\, R \quad apr\, apl^{-1}\, \mathsf{snd}\, S \quad apl\, apr^{-1}$

Figure 1.14: another circuit layout which implements $R \leftrightarrow S$

the definitions of beside and below would appear to be straightforward

$$\begin{aligned} R \leftrightarrow S &= apl \,;((\mathsf{fst}\, R) \setminus apr) \,;((\mathsf{snd}\, S) \setminus apl) \,; apr^{-1} \\ R \updownarrow S &= (R^{-1} \leftrightarrow S^{-1})^{-1} \\ &= apr \,;((\mathsf{snd}\, S) \setminus apl) \,;((\mathsf{fst}\, R) \setminus apr) \,; apl^{-1} \end{aligned}$$

Unfortunately, these definitions would not be very useful: this is an example of recklessly generalizing from a picture and getting more than one bargains for.

One of the properties we expect of beside and below is that $(A \leftrightarrow B) \updownarrow (C \leftrightarrow D) = (A \updownarrow C) \leftrightarrow (B \updownarrow D)$. Richard Bird [Bird88] describes this property by saying that 'beside *abides with* below', a contraction of above-and-besides. It turns out that the operators defined above do not abide: the abides property of beside and below depends on the component circuits being of type $2 \rightarrow 2$. Better definitions would therefore be given by

$$\begin{aligned} R \leftrightarrow S &= apl \,;((\mathsf{fst}(R \setminus 2)) \setminus apr) \,;((\mathsf{snd}(S \setminus 2)) \setminus apl) \,; apr^{-1} \\ R \updownarrow S &= apr \,;((\mathsf{snd}(S \setminus 2)) \setminus apl) \,;((\mathsf{fst}(R \setminus 2)) \setminus apr) \,; apl^{-1} \end{aligned}$$

which is indeed an abiding pair of operators.

Particularly useful in discussing pair-to-pair circuits will be the above and beside of two identity components. Let $rsh = \iota \leftrightarrow \iota$, for 'right shift', and $lsh = \iota \updownarrow \iota$, for 'left shift'; clearly $rsh = lsh^{-1}$ and $lsh = rsh^{-1}$. Since $\iota \leftrightarrow \iota = apl \,; 3 \,; apr^{-1}$ you can see that $[a,[b,c]] \,; rsh = rsh \,; [[a,b],c]$, whence the name; and dually $lsh = apr \,; 3 \,; apl^{-1}$ and $[[a,b],c] \,; lsh = lsh \,; [a,[b,c]]$.

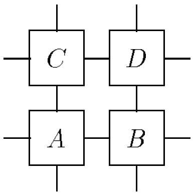


Figure 1.15: $(A \leftrightarrow B) \updownarrow (C \leftrightarrow D) = (A \updownarrow C) \leftrightarrow (B \updownarrow D)$

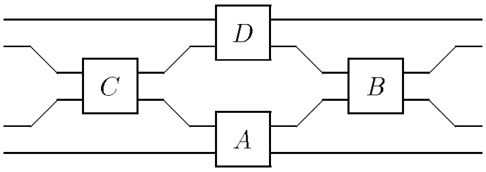


Figure 1.16: $([\iota, C, \iota] \setminus mid^{-1}) \,;\, [A, D] \,;\, ([\iota, B, \iota] \setminus mid^{-1})$

The definitions of beside and below can be recast in terms of left and right shifts, for $R \leftrightarrow S = rsh \,;\, \mathsf{fst}\, R \,;\, lsh \,;\, \mathsf{snd}\, S \,;\, rsh$ and by duality $R \updownarrow S = lsh \,;\, \mathsf{snd}\, S \,;\, rsh \,;\, \mathsf{fst}\, R \,;\, lsh$.

The easiest proof that beside abides with below goes by showing that each of $(A \leftrightarrow B) \updownarrow (C \leftrightarrow D)$ and $(A \updownarrow C) \leftrightarrow (B \updownarrow D)$ is equal to the same symmetrical form; for

$$(A \leftrightarrow B) \updownarrow (C \leftrightarrow D) \quad = \quad ([\iota, C, \iota] \setminus mid^{-1}) \,;\, [A, D] \,;\, ([\iota, B, \iota] \setminus mid^{-1})$$

where $mid$ is the unique $mid : [2, 2] \rightarrow [\iota, 2, \iota]$ for which $[[a, b], [c, d]] \,;\, mid = mid \,;\, [a, [b, c], d]$. A circuit layout suggestive of this symmetrical form is shown in figure 1.16. You can then show by taking inverses throughout that

$$(A \updownarrow C) \leftrightarrow (B \updownarrow D) \quad = \quad ([\iota, C, \iota] \setminus mid^{-1}) \,;\, [A, D] \,;\, ([\iota, B, \iota] \setminus mid^{-1})$$

which shows that beside abides with below.

The other things that one may want to know about beside and below also say that it does not matter how you choose to bracket a picture, such as those in figure 1.17. First it is the case that $(R \,;\, \mathsf{snd}\, X) \leftrightarrow S = R \leftrightarrow (\mathsf{fst}\, X \,;\, S)$ and similarly by inverting both sides $R \updownarrow (S \,;\, \mathsf{fst}\, X) = (\mathsf{snd}\, X \,;\, R) \updownarrow S$. In particular, if $P \,;\, Q = Y \,;\, Z$ and $R = R \,;\, \mathsf{snd}\, Y$ and $\mathsf{fst}\, Z \,;\, S = S$ then $(R \,;\, \mathsf{snd}\, P) \leftrightarrow (\mathsf{fst}\, Q \,;\, S) = R \leftrightarrow S$.

Secondly it is the case that $[A, [B, C]] \,;\, (R \leftrightarrow S) \,;\, [[D, E], F] = ([A, B] \,;\, R \,;\, \mathsf{fst}\, D) \leftrightarrow (\mathsf{snd}\, C \,;\, S \,;\, [E, F])$ and dually $[[A, B], C] \,;\, (R \updownarrow S) \,;\, [D, [E, F]] = (\mathsf{fst}\, A \,;\, R \,;\, [D, E]) \updownarrow ([B, C] \,;\, S \,;\, \mathsf{snd}\, C)$. Most of the things that one needs to prove about these circuits are consequences of these last observations, setting various of the variables to the identity.

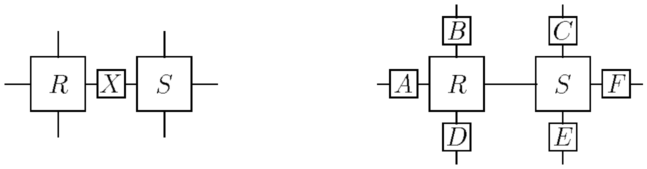



Figure 1.17: $(R \,;\, \mathsf{snd}\, X) \leftrightarrow S \;=\; R \leftrightarrow (\mathsf{fst}\, X \,;\, S)$ and $[A, [B, C]] \,;\, (R \leftrightarrow S) \,;\, [[D, E], F] = ([A, B] \,;\, R \,;\, \mathsf{fst}\, D) \leftrightarrow (\mathsf{snd}\, C \,;\, S \,;\, [E, F])$

### 1.4.2 Reflections

Throughout the discussion of 'beside' and 'below' we made use of the duality of the two operators. Taking the inverse of a $2 \rightarrow 2$ circuit corresponds to flipping the picture about the diagonal line between the domain and range – the line shown in figure 1.12. Since this turns vertical domain connections into horizontal range connections and vice versa, it turns below into beside and vice versa. That is the duality expressed by $(R \leftrightarrow S)^{-1} = (R^{-1}) \updownarrow (S^{-1})$.

Often, we find ourselves wanting to swap two of the wires of one of these four-sided tiles. Define $\langle a, b\rangle\, \mathsf{swp}\, R\, \langle c, d\rangle = \langle d, b\rangle\, R\, \langle c, a\rangle$, so that a picture of $\mathsf{swp}\, R$ is the same as one of $R$ except that the connections on the right and left are exchanged. The connections at the top and bottom are exactly the same. Amongst the useful things to know about $\mathsf{swp}$ are that

$$\begin{aligned}
\mathsf{swp}\,\mathsf{swp}\, R &= R \setminus 2 \\
(\mathsf{swp}\, R) \leftrightarrow (\mathsf{swp}\, S) &= \mathsf{snd}\, \mathit{swap} \,;\, \mathsf{swp}(S \leftrightarrow R) \,;\, \mathsf{fst}\, \mathit{swap} \\
(\mathsf{swp}\, R) \updownarrow (\mathsf{swp}\, S) &= \mathsf{swp}(R \updownarrow S)
\end{aligned}$$

Of course there is a dual operation: the circuit $(\mathsf{swp}\, R^{-1})^{-1}$ is the one that you get by reflecting $R$ about a horizontal line, with the top and bottom connections exchanged and the horizontal ones unchanged. This operator has similar properties, as you can check by taking inverses throughout the equations describing $\mathsf{swp}$.

### 1.4.3 Other orthogonally connected circuits

There are a number of other ways that circuits might be connected with the wires lying orthogonally to each other, and that might be interesting: for example it will be common to connect circuits with two inputs and one output in rows when doing things that are like adding up a row of numbers. There seem to be a vast number of possibilities corresponding to the combinations of connections that might be missing from the picture of beside and below. Three of these configurations are illustrated in figure 1.18.

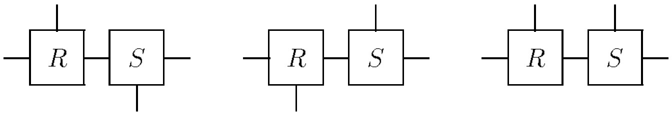


Figure 1.18: representation of $(R;S)\backslash 2$ and $\mathsf{snd}\,\pi_2{}^{-1};((\pi_1;R)\leftrightarrow(S;\pi_2{}^{-1}));\mathsf{fst}\,\pi_1$ and $((R\,;\pi_2{}^{-1})\leftrightarrow(S\,;\pi_2{}^{-1}))\,;\pi_2$

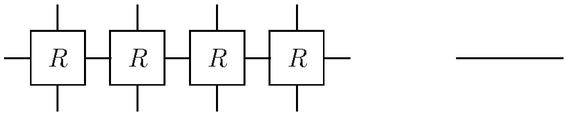


Figure 1.19: two instances of $\mathsf{row}\,R$

Fortunately, however, these are all just instances of beside, for example $(R\,;S)\setminus 2 = \mathsf{snd}\,\pi_1{}^{-1}\,;((R\,;\pi_2{}^{-1})\leftrightarrow(\pi_1\,;S))\,;\mathsf{fst}\,\pi_2$. This means that the existing results about beside carry over immediately to any new operator of this form that we might want. For example, suppose that $R\oplus S = \mathsf{snd}\,\pi_2{}^{-1}\,;((\pi_1\,;R)\leftrightarrow(S\,;\pi_2{}^{-1}))\,;\mathsf{fst}\,\pi_1$, which is the first configuration shown in the figure, then immediately from the properties of beside it follows that $[A,B]\,;(R\oplus S)\,;[C,D] = (A\,;R\,;\mathsf{fst}\,C)\oplus(\mathsf{snd}\,B\,;S\,;D)$. It is the existence of the inverses of the projections that make this unification possible, and this is one of the excuses for not confining ourselves to functions, nor indeed to total functions.

### 1.4.4 Rows

Having put two circuits next to each other with beside, the next step is of course to put another next to them and so on, as shown in figure 1.19. This is the same generalization that takes you from parallel composition to map, and the following requirements on $\mathsf{row}$ should look very like those for $\mathsf{map}$

$$\begin{array}{rcl}
\mathsf{snd}\,[\text{-}]\,;\mathsf{row}\,R & = & R\,;\mathsf{fst}\,[\text{-}] \\
\mathsf{row}\,R\,;\mathsf{fst}\,[\text{-}]^{-1} & = & \mathsf{snd}\,[\text{-}]^{-1}\,;R \\
\mathsf{snd}([m,n]\,;\mathit{app})\,;\mathsf{row}\,R & = & ((\mathsf{row}\,R\,;\mathsf{fst}\,m)\leftrightarrow(\mathsf{row}\,R\,;\mathsf{fst}\,n))\,;\mathsf{fst}\,\mathit{app} \\
\mathsf{row}\,R\,;\mathsf{fst}(\mathit{app}^{-1}\,;[m,n]) & = & \mathsf{snd}\,\mathit{app}^{-1}\,;((\mathsf{snd}\,m\,;\mathsf{row}\,R)\leftrightarrow(\mathsf{snd}\,n\,;\mathsf{row}\,R))
\end{array}$$

Notice that these four conditions cannot be abbreviated in the same way as those for map, by using inverse, because $(\mathsf{row}\,R)^{-1}\neq\mathsf{row}\,R^{-1}$.

The interface rule for row, by analogy with that for beside, would seem to be that if $D\,;A = Y\,;Z$ where $R:\mathsf{fst}\,Z\rightarrow\mathsf{snd}\,Y$, then

$$[A,\mathsf{map}\,B]\,;\mathsf{row}\,R\,;[\mathsf{map}\,C,D] = \mathsf{row}([A,B]\,;R\,;[C,D])$$

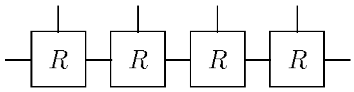



Figure 1.20: an instance of $\mathsf{rdl}\,R$

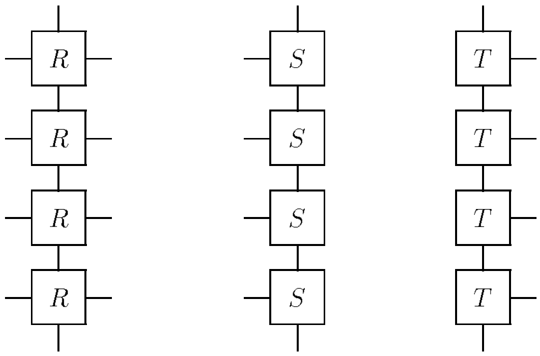



Figure 1.21: an instance of $\mathsf{col}\,R$, one of $\mathsf{rdr}\,S$, and one of $(\mathsf{rdl}(T^{-1}))^{-1}$

By induction from the rule about a component lying between two circuits beside each other, it follows that $\mathsf{fst}\,X\,;\mathsf{row}(R\,;\mathsf{snd}\,X) = \mathsf{row}(\mathsf{fst}\,X\,;R)\,;\mathsf{snd}\,X$. Similarly by induction from the rule relating $\mathsf{swp}$ and beside, we get a rule about reflecting a row, that $\mathsf{swp}\,\mathsf{row}\,R = \mathsf{snd}\,rev\,;\mathsf{row}\,\mathsf{swp}\,R\,;\mathsf{fst}\,rev$.

There is a row-like repeated form that deserves a special mention, namely $\mathsf{rdl}\,R$ read 'reduce (from the) left'. This is illustrated in figure 1.20 and defined by $\mathsf{rdl}\,R = \mathsf{row}(R;{\pi_2}^{-1});\pi_2$. Its properties are immediate from the corresponding properties of $\mathsf{row}$. It should be suggesting operations like summation to you: for example, if $\langle x, y\rangle\, add\,(x{+}y)$ then $\langle a, \langle p, q, r\rangle\rangle(\mathsf{rdl}\; add)(((a{+}p){+}q){+}r)$. The order of the bracketing – from the left – is important for non-associative operators, for example $\mathsf{rdl}\; apr$ or $\mathsf{rdl}(swap\,;apl)$, and explains the name.

### 1.4.5 Columns

Of course since beside has below as its dual there is going to be a corresponding dual for row which is column, defined by $\mathsf{col}\,R = (\mathsf{row}\,R^{-1})^{-1}$. An instance of this is illustrated in figure 1.21, and its properties are the duals of the properties of row.

$$
\begin{aligned}
\mathsf{fst}\,[\text{-}]\,;\mathsf{col}\,R &= R\,;\mathsf{snd}\,[\text{-}] \\
\mathsf{col}\,R\,;\mathsf{snd}\,[\text{-}]^{-1} &= \mathsf{fst}\,[\text{-}]^{-1}\,;R \\
\mathsf{fst}([m,n]\,;app)\,;\mathsf{col}\,R &= ((\mathsf{col}\,R\,;\mathsf{snd}\,m) \updownarrow (\mathsf{col}\,R\,;\mathsf{snd}\,n))\,;\mathsf{snd}\,app
\end{aligned}
$$

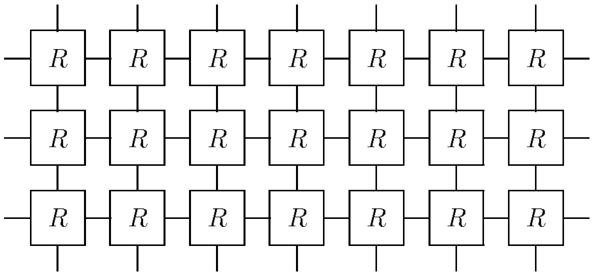



Figure 1.22: an instance of $\mathsf{row}\,\mathsf{col}\,R = \mathsf{col}\,\mathsf{row}\,R$

$$\mathsf{col}\,R\,;\mathsf{snd}(app^{-1}\,;[m,n]) \quad = \quad \mathsf{fst}\,app^{-1}\,;((\mathsf{fst}\,m\,;\mathsf{col}\,R)\updownarrow(\mathsf{fst}\,n\,;\mathsf{col}\,R))$$

and so on.

The column-shaped thing that corresponds to reduce from the left is reduce from the right, defined by $\mathsf{rdr}\,R = \mathsf{col}(R\,;{\pi_1}^{-1})\,;\pi_1$. It might seem that it would have been more logical to christen this something like 'reduce downwards', but again its name derives from the bracketing in the formula which describes the reduction of a function; for example, if $\langle x,y\rangle\ add\ (x+y)$ then $\langle\langle p,q,r\rangle,a\rangle\,(\mathsf{rdr}\ add)\,(p+(q+(r+a)))$.

Beware that reduce right is not the inverse-dual of reduce left! The easiest way to see the difference is to compare the examples of $\mathsf{rdr}\,R$ and $(\mathsf{rdl}(R^{-1}))^{-1}$ that are shown in figure 1.21. Of course $(\mathsf{rdr}(R^{-1}))^{-1}$ is yet a fourth different reduce-shaped circuit.

Since for any $R: 2 \rightarrow 2$ both $\mathsf{row}\,R: 2\rightarrow 2$ and $\mathsf{col}\,R: 2\rightarrow 2$, it is possible to make $\mathsf{row}\,\mathsf{col}\,R$ and $\mathsf{col}\,\mathsf{row}\,R$. Both of these will look like the example shown in figure 1.22, and in fact you can show by induction from the abides property of below and beside that $\mathsf{row}\,\mathsf{col}\,R = \mathsf{col}\,\mathsf{row}\,R$.

### 1.4.6 Horner's rule

Horner's rule is the name usually given to a method of evaluating polynomials without needing to raise powers or do unnecessary multiplications; it is encapsulated in the equality of $a_0 + xa_1 + \cdots + x^{n-2}a_{n-2} + x^{n-1}a_{n-1} + x^n a_n = a_0 + x(a_1 + \cdots + x(a_{n-2} + x(a_{n-1} + xa_n))\cdots)$. Naïve evaluation of the left-hand side would appear to require a quadratic number of multiplications, that of the right-hand side only a linear number. The equality is a consequence of a property that is usually expressed as the distribution of multiplication over addition. Specifically it follows from $x \times a + x \times b = x \times (a+b)$.

The rule applies to any structure with two operations that distribute in this way, and in particular it has a counterpart in circuits:

$$[R,R]\,;S = S\,;R \quad\Longrightarrow\quad ((\mathsf{tri}\,R)\setminus apr^{-1})\,;\mathsf{rdr}\,S = \mathsf{rdr}(\mathsf{tri}\,R\,;S)$$

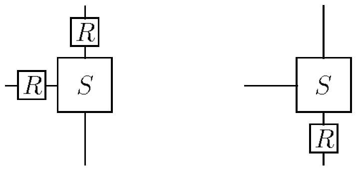



Figure 1.23: the hypothesis for Horner's rule, that $[R, R] \,;S = S \,; R$

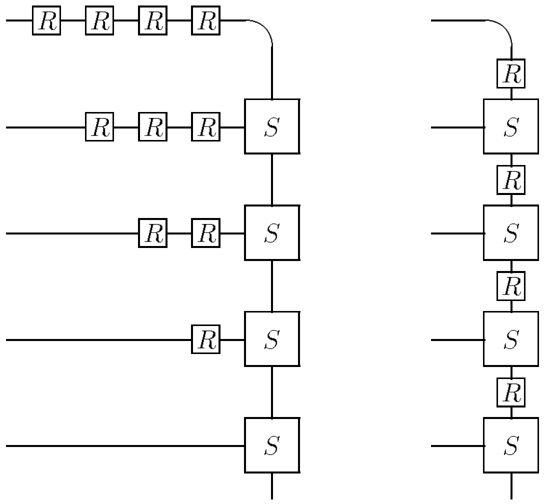



Figure 1.24: an instance of the consequence of Horner's rule, that $\mathsf{tri}\, R; apr^{-1};$ $\mathsf{rdr}\, S = apr^{-1} \,; \mathsf{rdr}(\mathsf{tri}\, R \,; S)$

The hypothesis is illustrated in figure 1.23 and an example of the consequence for a particular size appears in figure 1.24.

The most familiar instance of Horner's rule will convert a list of bits into the number that they represent in binary notation. Suppose $R$ is the circuit $(\times 2)$ that doubles a number, then recall that $\mathsf{tri}(\times 2)$ relates a list of zeroes and ones to a list of those bits weighted by the powers of two which they would represent in a binary number with its least significant bit first. Suppose that $S$ is the circuit $add$ that relates a pair of numbers to their sum, then $apr^{-1}$ ; $\mathsf{rdr}\ add$ relates a list of numbers to their sum; so $bin = \mathsf{tri}(\times 2) ; apr^{-1} ; \mathsf{rdr}\ add$ relates a list of bits to the number that they represent as a binary number, least significant bit first. (It also relates other lists of numbers to the same value, but that does not matter for the moment.)

The hypothesis of Horner's rule is satisfied by these two circuits, because the circuit on one side is $[(\times 2), (\times 2)] ; add$ which relates a pair of numbers to the sum of twice each of them, and that on the other is $\mathsf{fst}(\times 2)^0 ; add ; (\times 2) = add ; (\times 2)$ which relates a pair of numbers to twice their sum. These are the same circuit because $2x + 2y = 2(x + y)$, so $[(\times 2), (\times 2)] ; add = \mathsf{fst}(\times 2)^0 ; add ; (\times 2)$. Our formulation of Horner's rule therefore allows the conclusion that $bin = apr^{-1} ; \mathsf{rdr}(\mathsf{tri}(\times 2) ; add) = apr^{-1} ; \mathsf{rdr}\ step$ where $step = \mathsf{snd}(\times 2) ; add$, which is to say that a bit-vector can be converted into the number that it represents by a right-reduction of $step$ components, each of which doubles one input and adds it to the other.

There is a statement of Horner's rule for left reduction:

$$[R, R] ; S = S ; R \implies ((\mathsf{irt}\ R) \setminus apl^{-1}) ; \mathsf{rdl}\ S = \mathsf{rdl}(\mathsf{irt}\ R ; S)$$

Similarly there are statements for the inverses of both reductions. There are also a variety of similar results for row and column, and for grids. Their proofs by induction would be tedious and lengthy, but the whole point of this work is that such proofs once having been done, their consequences can be applied in the course of a design without the necessity of recourse to induction.

## 1.5 Transposition and zips

So far we have only seen wires crossed over in $rev$, but there is another cliché of circuit design which is even worse from the point of view of crossed wires. Suppose a circuit takes its input from a pair of buses, and calculates a function of the signals on pairs of wires drawn from corresponding positions in those buses. Somewhere in that circuit the designer has to arrange to interleave the buses, along the lines shown in figure 1.25, and that costs a number of wire crossings quadratic in the widths of the buses.

To describe such circuits in Ruby we need a new piece of plumbing, $zip$, defined by $\langle x, y \rangle\, zip\, z \iff \forall i.\ z_i = \langle x_i, y_i \rangle$. The law about circuit constructors

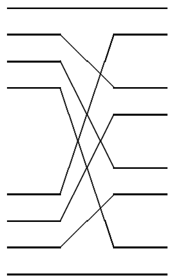


Figure 1.25: an instance of *zip*

that corresponds to this is that $[\mathsf{map}\,R, \mathsf{map}\,S]\,;zip = zip\,;\mathsf{map}[R,S]$. Another way of saying this is that $[\mathsf{map}\,R, \mathsf{map}\,S] \setminus zip = \mathsf{map}[R,S]$, but be careful because $(\mathsf{map}[R,S]) \setminus zip^{-1} \neq [\mathsf{map}\,R, \mathsf{map}\,S]$. Similarly $[\mathsf{tri}\,R, \mathsf{tri}\,S] \setminus zip = \mathsf{tri}[R,S]$, but not the other equation. This is because you can only interleave buses that are the same width, that is $zip^{-1}\,;zip = \mathsf{map}\,2$ but $zip\,;zip^{-1}$ is the type of a pair of lists that have the same length.

More generally any number of buses can be interleaved by transposing them with $trn$, defined by $x \; trn \; y \iff \forall i,j\,.\,x_{i,j} = y_{j,i}$, so $trn = trn^{-1}$, but be careful because $trn\,;trn^{-1} \neq \mathsf{map\,map}\,\iota$. Since $zip = 2\,;trn = trn\,;\mathsf{map}\,2$ there are laws like those about zipping but about transposition and map, such as $\mathsf{map\,map}\,R\,;trn = trn\,;\mathsf{map\,map}\,R$ and more suggestively $\mathsf{map\,tri}\,R\,;trn = trn\,;\mathsf{tri\,map}\,R$ and so on.

Of course because zip and transpose are 'expensive' it is a good idea to avoid them in the final form of a circuit, but having them in the language makes it much easier to do some calculations. One has to be able to say what it is that one would not want to build, as well as what to aim for.

### 1.5.1 Zipping rows together

One of the common optimizations in circuit design is to interleave a number of similar regular grid-shaped calculations, such as arise in dealing with binary representations of numbers, to shorten the wires that connect them, for example bringing bits of the same weight in different numbers together if they are to be added. This is the transformation that takes a number of arithmetic circuits and interleaves them to make the data-path of a processor.

In Ruby the validity of this transformation is captured by the observation that $([A,B] \setminus zip^{-1}) \leftrightarrow ([C,D] \setminus zip^{-1}) = \mathsf{snd}\,zip^{-1}\,;([A \leftrightarrow C, B \leftrightarrow D] \setminus zip^{-1})\,;\mathsf{fst}\,zip$ and of course the dual result for below. It generalizes by induction to

$$\begin{aligned}
\mathsf{row}([R,S] \setminus zip^{-1}) &= \mathsf{snd}\,zip^{-1}\,;([\mathsf{row}\,R, \mathsf{row}\,S] \setminus zip^{-1})\,;\mathsf{fst}\,zip \\
\mathsf{col}([R,S] \setminus zip^{-1}) &= \mathsf{fst}\,zip^{-1}\,;([\mathsf{col}\,R, \mathsf{col}\,S] \setminus zip^{-1})\,;\mathsf{snd}\,zip
\end{aligned}$$

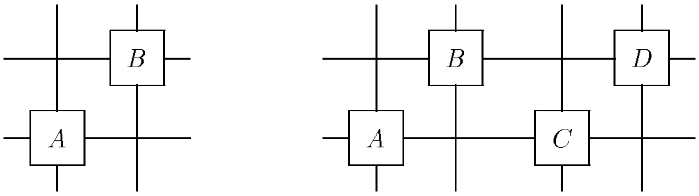



Figure 1.26: $[A, B] \setminus zip^{-1}$ and $([A, B] \setminus zip^{-1}) \leftrightarrow ([C, D] \setminus zip^{-1})$

$$\begin{aligned}
\mathsf{row}((\mathsf{map}\, R) \setminus zip^{-1}) &= \mathsf{snd}\, trn^{-1} \,;((\mathsf{map\,row}\, R) \setminus zip^{-1}) \,; \mathsf{fst}\, trn \\
\mathsf{col}((\mathsf{map}\, R) \setminus zip^{-1}) &= \mathsf{fst}\, trn^{-1} \,;((\mathsf{map\,col}\, R) \setminus zip^{-1}) \,; \mathsf{snd}\, trn
\end{aligned}$$

and by application of two of the above to such observations as

$$\mathsf{row\,col}((\mathsf{map}\, R) \setminus zip^{-1}) = ((\mathsf{map\,row\,col}\, R) \setminus zip^{-1}) \setminus [trn, trn]$$

## 1.6 Sequential circuits

So far we have intended to give the impression that all the circuits that we are talking about are combinational: that they behave as described in a steady state, in which they are presented with an unchanging input and deliver an unchanging output which depends only on that input. Of course no circuit is ever used in this way: at the very least there was a time before it was presented with any input; however the combinational description faithfully models the behaviour of a circuit that was presented with its input so long ago that it has settled down.

If an entire system is allowed to settle after each change in its input, the whole of the system is tied up for a settling time that depends on the longest propagation path, so it usual to place 'latches' – sometimes called 'registers' – at the outputs of the modules that make up a system. Each latch stores the output from its module in the previous steady state of that module, allowing that module to proceed with the next calculation while the modules that follow it are still settling into the previous steady state. In figure 1.27, suppose that the $F$ is presented with a succession $x_i$ of inputs. The output from the latch $\mathcal{D}$ at any time will be the value that was previously presented to it by $F$, so $G$ can be completing the calculation of $G(F(x_i))$ while $F$ is working on the calculation of $F(x_{i+1})$. When the circuit has settled the latch is 'clocked' – it is told to discard its current output and to propagate $F(x_{i+1})$ instead – by a signal not shown in the diagram. This technique of overlapping a succession of combinational calculations is called 'pipelining'. Since it has the effect of shortening the longest time to settle, it allows a greater number of calculations to be completed by a circuit in a given time. Notice, however,

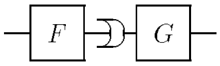



Figure 1.27: a pipeline $F ; \mathcal{D} ; G$

that there is no longer a single combinational description of what the whole circuit is doing.

It is usual to think of the latches in such a circuit as being the repository of the state of the circuit. The calculation being implemented by the circuit at any given time depends not only on the circuit and the inputs to it, but also on the values being held in latches. By means of these the calculation at any given time can be influenced by past calculations. In a pipeline like that in figure 1.27 the calculation, and so the output, depends only on a finite number of past inputs, in this case on only one; but in general there may be cycles in the circuit with the input to a latch depending on some function of its output. In that case the state held in the latch is an accumulation of information about the whole past history of calculations performed by the circuit, and the output can depend on an arbitrarily large number of past inputs. There is no way of giving a convenient and convincing account of such circuits in terms of a succession of independent combinational calculations.

The process of designing a sequential circuit appears to be significantly harder than that of designing a combinational one: even in the simple case of a pipeline there is an additional concern, that of making sure that the inputs that arrive together at a component are those for the same calculation; and more generally it is not possible to separate the concern of getting the time right from that of getting the value right. Fortunately, there is a large range of circuits for which it is possible to reason that a design which would work as a combinational circuit would work more or less the same way as a sequential circuit.

### 1.6.1 Time sequences

The sequential circuits with which we are concerned are those about which it is sound to reason in exactly the same way as those in previous sections. What that means is that the algebra is going to be the same as it was in previous sections, but the interpretation of the symbols will have to be different: the variables like $R$, $S$ and so on will represent sequential circuits; and the combining forms will be correspondingly different.

Think of a circuit as relating not values, but time-sequences of values. A time sequence is just a function from an index which represents the time at which a signal is observed to the value that the signal has at that time. We will use the integers to represent successive times at which a signal is changed:

a signal representing a Boolean, one of type *bool*, is a function $Z \rightarrow B$, and one representing a pair of Booleans, one of type $[bool, bool]$, is a function $Z \rightarrow (B \times B)$, that is a pair-valued function, and *not* a pair of functions. So for example to say that $not : bool \rightarrow bool$ is to say that in this model it represents a relation between a function of type $Z \rightarrow B$ and another of the same type, and to say that $xor : [bool, bool] \rightarrow bool$ is to say that it represents a relation between a function of type $Z \rightarrow (B \times B)$ and one of type $Z \rightarrow B$.

Which relation should *xor* be in this model? It should relate a pair of Booleans to their logical 'exclusive or', and it should do this at each time. Similarly, a sequential *not* should, at each instant, relate a Boolean to its negation.

Let $\mathcal{S}[\![R]\!]$ be the relation that $R$ represents in the sequential model, and $\mathcal{C}[\![R]\!]$ be the relation that it represented in the combinational model, then provided $R$ has no internal state, we want $x\, \mathcal{S}[\![R]\!]\, y \iff \forall i.\, x_i\, \mathcal{C}[\![R]\!]\, y_i$, and we say that any $R$ for which this is true is *stateless*.

Note that in the sequential model a circuit has a single time sequence in each of its domain and range. So, for example, a sequential *xor* relates a sequence of pairs of Booleans to a sequence of Booleans.

### 1.6.2 Composition, parallel composition and so on

Having decided what the interpretation of some circuits will be in the sequential model, what should the interpretation of composition be? At least for stateless circuits it is already determined, for if $R$ and $S$ are stateless we certainly want $R\,;S$ to be stateless, so the only possibility is to define $\mathcal{S}[\![R\,;S]\!]$ to be $\mathcal{S}[\![R]\!]\,;\mathcal{S}[\![S]\!]$ for general $R$ and $S$. Similarly, the inverse of a sequential circuit is interpreted as the relational converse of its interpretation: $\mathcal{S}[\![R^{-1}]\!] = \mathcal{S}[\![R]\!]^{-1}$, and so also for repeated composition.

In order to talk about the interpretation of parallel composition, we need to be able to relate a pair of sequences to the obvious sequence of pairs to which it corresponds: we already have a suitable relation, *zip* defined by $\langle x, y\rangle\, zip\, z \iff \forall i.\, \langle x_i, y_i\rangle = z_i$ and we define $\mathcal{S}[\![[R,S]]\!] = [\mathcal{S}[\![R]\!], \mathcal{S}[\![S]\!]] \setminus zip$ for general $R$ and $S$. In giving an interpretation for $\mathsf{map}$, we use *trn* to convert a time-sequence of $n$-sequences into an $n$-sequence of time-sequences. and define $\mathcal{S}[\![\mathsf{map}\, R]\!] = (\mathsf{map}\, \mathcal{S}[\![R]\!]) \setminus trn$ for general $R$. The interpretation of the other combining forms follows from these.

### 1.6.3 Delay and state

So far all the sequential circuits we can talk about have been stateless because all our combining forms preserve statelessness. In order to talk about circuits with state it will be enough for the present to introduce one new primitive: a *delay*, written $\mathcal{D}$. Its meaning in the sequential model is the relation

$$x\,\mathcal{D}\,y \iff \forall i.\, x_i = y_{i+1}$$

which says that delay relates two signals if the range signal is at all times the same as the domain signal had been exactly one time step earlier.

The inverse of a delay, an *anti-delay* written of course $\mathcal{D}^{-1}$, is also a useful concept. It represents the relation $x\,\mathcal{D}^{-1}\,y \iff \forall i.\, x_i = y_{i-1}$ which says that it relates two signals if the range signal is at all times the same as the domain signal would be exactly one time step later.

If these relations are to be implemented by circuits taking inputs from their domains and returning outputs in their ranges, then $\mathcal{D}$ is just a latch: the output after each clock is the same as the input immediately preceding it. The corresponding interpretation of $\mathcal{D}^{-1}$ as a circuit is altogether less implementable: it would have to predict *before* each clock what its input was going to be *after* that clock.

Conversely, if the relations are to be implemented by circuits taking inputs from their ranges and returning outputs in their domains, $\mathcal{D}^{-1}$ is a latch, and it is $\mathcal{D}$ that becomes unimplementable. What this means is that if an expression including $\mathcal{D}$ or $\mathcal{D}^{-1}$ were to be implemented by translating it into a circuit there would be these additional constraints on the choice of direction for each of the signal flows, that all $\mathcal{D}$ and $\mathcal{D}^{-1}$ in the expression be implemented the right way around.

So long as the time-sequences in the model are functions from all of the integers, positive and negative, so that there is no first time and no last time, you can show that $\mathcal{D}\,;\mathcal{D}^{-1} = \iota = \mathcal{D}^{-1}\,;\mathcal{D}$. There is a perfect symmetry between $\mathcal{D}$ and $\mathcal{D}^{-1}$ as presented here: you should not think of either of them as being more realistic, or more implementable than the other. Indeed, one of them is just the other one seen the other way around.

Even though $\mathcal{D}$ makes no sense in the combinational model, reasoning about circuits that contain $\mathcal{D}$ can be done in exactly the same way as about combinational circuits. Any calculation with a variable $R$ in it, a calculation that does not depend on the particular properties of $R$, can be instantiated by putting $\mathcal{D}$ for $R$. For example Horner's rule that if $[R, R]\,; S = S\,; R$ then $((\mathsf{tri}\,R) \setminus apr^{-1})\,; \mathsf{rdr}\,S = \mathsf{rdr}(\mathsf{tri}\,R\,; S)$ is a property of the combining forms like composition and reduce right, and not of $R$ or $S$. Consequently it is the case that if $[\mathcal{D}, \mathcal{D}]\,; S = S\,; \mathcal{D}$ then $((\mathsf{tri}\,\mathcal{D}) \setminus apr^{-1})\,; \mathsf{rdr}\,S = \mathsf{rdr}(\mathsf{tri}\,\mathcal{D}\,; S) = \mathsf{rdr}(\mathsf{snd}\,\mathcal{D}\,; S)$.

Of course there are things that are true about $\mathcal{D}$ which do depend on its being a particular circuit, for example that $\mathsf{tri}\,\mathcal{D}\,; \mathsf{tri}\,\mathcal{D}^{-1} = \mathsf{map}\,\iota$, but you should always keep in mind that it is also a perfectly normal component of the kind that we have been reasoning about all along. It is just this simplicity that makes the approach presented here so useful.

One of the remarkable properties of $\mathcal{D}$ is quite how polymorphic it is: $\mathcal{D}$ can be used to represent a unit delay applied to signals of any type, and the tightest type-specification that can be given is that $\mathcal{D} : \iota \rightarrow \iota$. Moreover $\mathcal{D} \backslash n = (\mathsf{map}\,\mathcal{D}) \backslash n$, so that for example $apl ; \mathcal{D} = apl ; \mathsf{map}\,\mathcal{D} = [\mathcal{D}, \mathsf{map}\,\mathcal{D}] ; apl$. This means that even though we may be talking about circuit components that will be realized as latches we do not *need* to be concerned with exactly what type of signals they handle, just with the delaying properties. The effect is a clean separation of two concerns: that of making sure that the correct values are handled, and that of ensuring that they arrive at the right times.

### 1.6.4 Timelessness

For any stateless circuit $R$ you can show that $R = R \backslash \mathcal{D}$. In these notes any circuit $R$ for which $R = R \backslash \mathcal{D}$ is said to be *timeless*. A timeless circuit is one that implements the same relation now as it did in the past, and similarly that it will in the future. Notice that timelessness is *not* the same as statelessness: $\mathcal{D}$ is most definitely not stateless, indeed it the essence of statefulness! Nevertheless since $\mathcal{D} = \mathcal{D}^{-1} ; \mathcal{D} ; \mathcal{D} = \mathcal{D} \backslash \mathcal{D}$ it is immediate from the definition that $\mathcal{D}$ is timeless.

Just as a stateless circuit is one that does not behave differently at any time according to the history of its inputs, so a timeless circuit is one that does not behave differently on account of the absolute time. A stateless circuit is necessarily timeless. An example of a timeful circuit would be one that generated a signal indicating time zero. A different sort of example would be $E$ for which $x \; E \; y$ if and only if $\forall i.\, x_{2i} = y_{2i}$. This circuit is not anchored at a single instant because $E \backslash \mathcal{D}^2 = E$, but nevertheless $E$ is not timeless since $E \backslash \mathcal{D} \neq E$.

One of the reasons that timelessness is interesting is that exactly like statelessness it is preserved by all of the circuit constructors that we have seen so far.

### 1.6.5 Slowing

So far we have ways of talking only about circuits in which all of the sequential components operate at the same rate. Sometimes it proves necessary to reason about circuits in which some components operate at different rates from others. In particular there are various common techniques which require some parts of a circuit to be clocked at twice the rate of other parts, or some such small multiple.

In order to talk about such circuits, we use a new primitive relation *pair* defined by $x \; pair \; y \iff \forall i.\, y_i = \langle x_{2i}, x_{2i+1} \rangle$. Define a 2-slow version of $R$ by $\mathsf{slow}\,R = [R, R] \backslash pair^{-1}$. The effect is that $\mathsf{slow}\,R$ separates the time-sequences in its domain and range each into two interleaved sequences, and

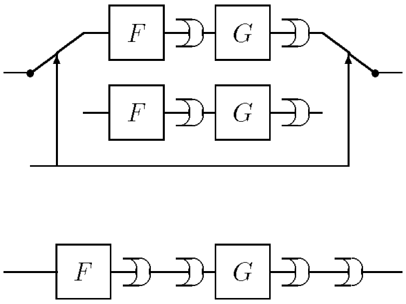



Figure 1.28: $\mathsf{slow}(F \,;\, \mathcal{D} \,;\, G \,;\, \mathcal{D})$ and $F \,;\, \mathcal{D}^2 \,;\, G \,;\, \mathcal{D}^2$

performs the calculation described by $R$ on each of the pairs of corresponding subsequences.

It is a matter of proof that if $R$ is an expression involving only the combining forms introduced so far and $\mathcal{D}$ and stateless components, then $\mathsf{slow}\, R$ is the same as the circuit obtained by replacing all occurrences of $\mathcal{D}$ in $R$ by $\mathcal{D}^2$. For example $\mathsf{slow}\, \mathcal{D} = \mathcal{D}^2$, and $\mathsf{slow}(F \,;\, \mathcal{D} \,;\, G \,;\, \mathcal{D}) = F \,;\, \mathcal{D}^2 \,;\, G \,;\, \mathcal{D}^2$. The proof is by structural induction on $R$.

This says that you can implement $\mathsf{slow}\, R$ either by following the recipe in the definition – making two copies of $R$ and supplying them with alternate inputs to derive alternate outputs – or by doubling up each latch in $R$ – effectively sharing the rest of the hardware in $R$ between two calculations, with the state of each calculation being held in the extra latch while the other calculation is being done.

### 1.6.6 Retiming

Retiming is a procedure for disposing latches through a circuit, with a view to improving properties like the length of the longest unbroken propagation path. The name and the formalisation are usually attributed to Leiserson [Leis81] although the technique is well-known and presentations of it are to be found throughout the literature.

The simplest form of retiming is the replacement of (timeless) $R$ by $R \setminus \mathcal{D}$ or by $R \setminus \mathcal{D}^{-1}$. We can get more complicated retiming laws by applying this transformation independently to different but possibly overlapping parts of a circuit. Horner's rule yields an example of such a composite retiming law. It says that if $S = [\mathcal{D}^{-1}, \mathcal{D}^{-1}] \,;\, S \,;\, \mathcal{D} = (2 \,;\, S) \setminus \mathcal{D}$, that is if $S$ is timeless, then $((\mathsf{tri}\, \mathcal{D}) \setminus apr^{-1}) \,;\, \mathsf{rdr}\, S = \mathsf{rdr}(\mathsf{snd}\, \mathcal{D} \,;\, S)$. A circuit like $\mathsf{rdr}(\mathsf{snd}\, \mathcal{D} \,;\, S)$ is said to be pipelined, because there is a latch between each of the stages of the reduction. You can read this instance of Horner's rule as follows: to implement a $\mathsf{rdr}\, S$ that is pipelined, implement $\mathsf{rdr}(\mathsf{snd}\, \mathcal{D} \,;\, S)$, and adjust the timing of the domain signals according to $(\mathsf{tri}\,\mathcal{D}) \setminus apr^{-1}$. If the circuit takes input from the domain, that adjustment consists of supplying each component of the input one time step earlier than the one to its left.

Such a specification is a *data skew*, and an expression like $(\mathsf{tri}\,\mathcal{D}) \setminus apr^{-1}$ can either represent a skewing circuit which is implemented and performs the required adjustment, or else represent a specification for the interface to the circuit which is implemented.

## 1.7 A systolic correlator

This section treats a design as an example of the style which we are advocating. Correlation is one of the most important functions in digital signal processing.

### 1.7.1 Specifying the correlator

The requirement is to calculate

$$c(t) = \sum_{i=0}^{N-1} d(t-i) \times r_i(t)$$

at each time $t$ given time sequences $d$ and $r_i$.

The first thing to do is to recast the specification into a more tractable notation. We separate the time and space dimensions by introducing a new sequence of values, $d'_i$, for which

$$c(t) = \sum_{i=0}^{N-1} d'_i(t) \times r_i(t) \quad \& \quad d'_i(t) = d(t-i)$$

the point being that the part with the arithmetic in it looks as though it can be implemented by a stateless circuit calculating

$$c = \sum_{i=0}^{N-1} d'_i \times r_i$$

and the other part looks like a shift-register.

Translating the stateless part of the specification, suppose we have components $mul$ and $acc$ for which $\langle x, y\rangle\; mul\; (x \times y)$ and $x\; acc\; (\sum_i x_i)$, then

$$\langle d', r\rangle\; (zip\; ;\, \mathsf{map}\; mul\; ;\, acc)\; c$$

There is a design decision made in dividing the inputs and outputs between domain and range. It is regrettably hard to modify this particular decision without starting again.

We can also divide the shift register into a stateless and a sequential part by introducing another sequence of values, $d''_i$, for which

$$d(t) = d''_i(t) \quad \& \quad d''_i(t-i) = d'_i(t)$$

for each $i$. The first part is stateless, and is a many-way *fork*. Suppose we call this fork $F$, so that $d\ F\ d''$. From the definition of $\mathcal{D}$, we can rewrite the second part as $d''_i\ \mathcal{D}^i\ d'_i$, or as $d''\ \mathsf{tri}\,\mathcal{D}\ d'$, so $\mathit{shift} = F\ ;\mathsf{tri}\,\mathcal{D}$. The whole correlator is implemented as a composition of these parts.

$$\langle d, r\rangle\ (\mathsf{fst}\,\mathit{shift}\ ;\mathit{zip}\ ;\mathsf{map}\,\mathit{mul}\ ;\mathit{acc})\ c$$

Notice that we can implement all of the components as functions from the domain to the range, so in a sense this is a complete design of a correlator. The rest of the development is going to be a matter of optimizing this initial implementation. We need to choose suitable lower level implementations of the many-way fork and the accumulator, so that we can eliminate $\mathsf{tri}\,\mathcal{D}$ and *zip*, both of which are inefficient in layout area. We also need to translate the implementation from one that operates on numbers to one that operates on binary representations of numbers.

### 1.7.2 Implementing the shift register

There are many different ways of building a many-way fork from a two-way fork. For example,

$$\begin{aligned} F &= {\pi_1}^{-1}\ ;\mathsf{row}(\pi_1\ ;\mathit{fork})\ ;\pi_1 \\ &= {\pi_2}^{-1}\ ;\mathsf{col}(\pi_2\ ;\mathit{fork})\ ;\pi_2 \end{aligned}$$

Here, we have allowed for the possibility that we may want the list in the range of $F$ to be of zero length. This in turn permits $r$ to be a list of length zero. We will choose to use the version built using $\mathsf{row}$ because we know that this circuit is composed with $\mathsf{tri}\,\mathcal{D}$ and we have a version of Horner's rule that matches.

$$\begin{aligned} \mathit{shift} &= {\pi_1}^{-1}\ ;\mathsf{row}(\pi_1\ ;\mathit{fork})\ ;\pi_1\ ;\mathsf{tri}\,\mathcal{D} \\ &= {\pi_1}^{-1}\ ;\mathsf{row}(\pi_1\ ;\mathit{fork})\ ;((\mathsf{tri}\,\mathcal{D}) \setminus \mathit{apr}^{-1})\ ;\pi_1 \\ &= {\pi_1}^{-1}\ ;\mathsf{row}(\pi_1\ ;\mathit{fork}\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1 \end{aligned}$$

The decision to implement the shift register using a $\mathsf{row}$ will influence design decisions such as the choice of how to build the accumulator.

### 1.7.3 Eliminating the *zip*

The next step is to use our results about zipping rows together to eliminate the *zip* from

$$\mathsf{fst}\, shift \,;\, zip$$

The components next to the zip are first cast into the form of two rows and these are then interleaved.

$$
\begin{array}{l}
\mathsf{fst}({\pi_1}^{-1} \,;\, \mathsf{row}(\pi_1 \,;\, fork \,;\, \mathsf{snd}\,\mathcal{D}) \,;\, \pi_1) \,;\, zip \\
\quad = \{\, \text{type of } zip \,\} \\
\qquad [{\pi_1}^{-1} \,;\, \mathsf{row}(\pi_1 \,;\, fork \,;\, \mathsf{snd}\,\mathcal{D}) \,;\, \pi_1, \mathsf{map}\,\iota] \,;\, zip \\
\quad = \{\, \text{implementing } \mathsf{map}\,\iota \text{ as a row} \,\} \\
\qquad [{\pi_1}^{-1} \,;\, \mathsf{row}(\pi_1 \,;\, fork \,;\, \mathsf{snd}\,\mathcal{D}) \,;\, \pi_1, {\pi_2}^{-1} \,;\, \mathsf{row}(\pi_2 \,;\, {\pi_1}^{-1}) \,;\, \pi_1] \,;\, zip \\
\quad = \{\, \text{rearranging terms} \,\} \\
\qquad [{\pi_1}^{-1}, {\pi_2}^{-1}] \,;\, [\mathsf{row}(\pi_1 \,;\, fork \,;\, \mathsf{snd}\,\mathcal{D}), \mathsf{row}(\pi_2 \,;\, {\pi_1}^{-1})] \,;\, [\pi_1, \pi_1] \,;\, zip \\
\quad = \{\ [{\pi_1}^{-1}, {\pi_2}^{-1}] = [{\pi_1}^{-1}, {\pi_2}^{-1}] \,;\, zip \,\} \\
\qquad [{\pi_1}^{-1}, {\pi_2}^{-1}] \,;\, zip \,; \\
\qquad\quad [\mathsf{row}(\pi_1 \,;\, fork \,;\, \mathsf{snd}\,\mathcal{D}), \mathsf{row}(\pi_2 \,;\, {\pi_1}^{-1})] \,; \\
\qquad\qquad [\pi_1, \pi_1] \,;\, zip \\
\quad = \{\, [\pi_1, \pi_1] \,;\, zip = zip^{-1} \,;\, \mathsf{fst}\, zip \,;\, \pi_1 \,\} \\
\qquad [{\pi_1}^{-1}, {\pi_2}^{-1}] \,;\, zip \,; \\
\qquad\quad [\mathsf{row}(\pi_1 \,;\, fork \,;\, \mathsf{snd}\,\mathcal{D}), \mathsf{row}(\pi_2 \,;\, {\pi_1}^{-1})] \,; \\
\qquad\qquad zip^{-1} \,;\, \mathsf{fst}\, zip \,;\, \pi_1 \\
\quad = \{\, \text{interleaving} \,\} \\
\qquad [{\pi_1}^{-1}, {\pi_2}^{-1}] \,;\, \mathsf{snd}\, zip \,;\, \mathsf{row}\, R \,;\, \pi_1 \\
\qquad \text{where } R = [\pi_1 \,;\, fork \,;\, \mathsf{snd}\,\mathcal{D}, \pi_2 \,;\, {\pi_1}^{-1}] \setminus zip^{-1} \\
\quad = \{\, {\pi_2}^{-1} \,;\, zip = \mathsf{map}\, {\pi_2}^{-1} \,\} \\
\qquad [{\pi_1}^{-1}, \mathsf{map}\, {\pi_2}^{-1}] \,;\, \mathsf{row}\, R \,;\, \pi_1
\end{array}
$$

Now for the interleaved components of which we have a row: the description we have for this cell is one that consists of two separate components and enough plumbing to make it appear that they are interleaved. A more efficient implementation can be obtained by a bit of simplification, using the knowledge that the left-most input to the row is provided through $\mathsf{fst}\, {\pi_1}^{-1}$ and the top input through $\mathsf{snd}\,\mathsf{map}\, {\pi_2}^{-1}$.

$$
\begin{array}{rcl}
[{\pi_1}^{-1}, {\pi_2}^{-1}] \,;\, R & = & [{\pi_1}^{-1}, {\pi_2}^{-1}] \,;\, [\pi_1 \,;\, fork \,;\, \mathsf{snd}\,\mathcal{D}, \pi_2 \,;\, {\pi_1}^{-1}] \setminus zip^{-1} \\
& = & [{\pi_1}^{-1}, {\pi_2}^{-1}] \,;\, [\pi_1 \,;\, fork \,;\, \mathsf{snd}\,\mathcal{D}, \pi_2 \,;\, {\pi_1}^{-1}] \,;\, zip^{-1}
\end{array}
$$

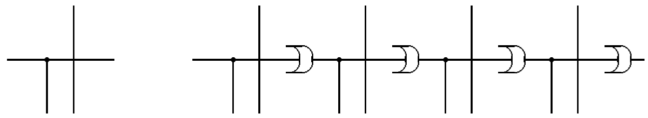


Figure 1.29: $dist = fork\ ;\mathsf{snd}\,\pi_1$ and $\mathsf{row}(dist\ ;\mathsf{snd}\,\mathcal{D})$

$$\begin{aligned} &= [fork\ ;\mathsf{snd}\,\mathcal{D}, {\pi_1}^{-1}]\ ; zip^{-1} \\ &= fork\ ;\mathsf{snd}(\pi_1\ ;\mathcal{D}\ ; {\pi_2}^{-1}) \\ &= dist\ ;\mathsf{snd}\,\mathcal{D}\ ;\mathsf{snd}\,{\pi_2}^{-1} \\ &\quad \text{where } dist = fork\ ;\mathsf{snd}\,\pi_1 \end{aligned}$$

so substituting for $R$ in a $\mathsf{row}\,R$ in its context

$$\begin{aligned} [{\pi_1}^{-1}, \mathsf{map}\,{\pi_2}^{-1}]\ ;\mathsf{row}\,R\ ;\pi_1 &= \mathsf{row}(dist\ ;\mathsf{snd}\,\mathcal{D})\ ;\mathsf{snd}\,{\pi_2}^{-1}\ ;\pi_1 \\ &= \mathsf{row}(dist\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1 \end{aligned}$$

and putting together what we have so far

$$CORR = \mathsf{row}(dist\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1\ ;\mathsf{map}\,mul\ ; acc$$

implements a relation for which $\langle r, d\rangle\ CORR\ c$.

Since we know that $acc$ can be implemented by rows or columns of components which add a pair of numbers, it looks as though we might be able to implement $CORR$ as a pair of rows, one below the other. The point of doing this is that $(\mathsf{row}\,S) \updownarrow (\mathsf{row}\,T) = \mathsf{row}(S \updownarrow T)$ and we can hope to make further simplifications in the component $S \updownarrow T$.

### 1.7.4 Implementing the accumulator

On the face of it, there is no choice here since we want to implement $acc$ as a row, it will have to be something like a left reduction of $add$, say $acc = apl^{-1}\ ;\mathsf{rdl}\,add$. This would be a reasonable implementation, but since it does not allow empty rows it would not be natural in the context of a potentially zero-width shift register. Define instead

$$\begin{aligned} acc &= {\pi_2}^{-1}\ ;\mathsf{fst}\,zero\ ;\mathsf{rdl}\,add \\ &\quad \text{where } x\ zero\ y \iff x = y = 0 \\ &= {\pi_2}^{-1}\ ;\mathsf{fst}\,zero\ ;\mathsf{row}(add\ ; {\pi_2}^{-1})\ ;\pi_2 \end{aligned}$$

and if we do indeed proceed in this direction

$$\begin{aligned} CORR = {} & \mathsf{row}(dist\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1\ ;\mathsf{map}\,mul\ ; \\ & {\pi_2}^{-1}\ ;\mathsf{fst}\,zero\ ;\mathsf{row}(add\ ; {\pi_2}^{-1})\ ;\pi_2 \end{aligned}$$

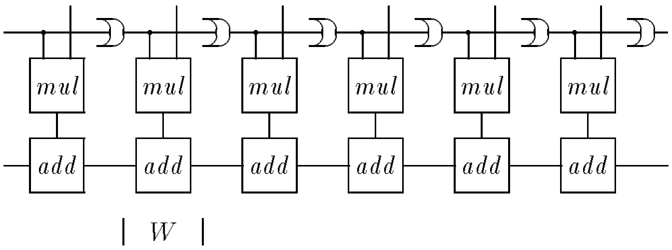



Figure 1.30: an instance of $\mathsf{rdl}(W\ ;\pi_2\ ;\mathsf{snd}\,\mathcal{D})$

$$
\begin{aligned}
= \quad & \mathsf{row}(dist\ ;[mul,\mathcal{D}])\ ;\pi_1\ ; \\
& \quad \pi_2{}^{-1}\ ;\mathsf{fst}\,zero\ ;\mathsf{row}(add\ ;\pi_2{}^{-1})\ ;\pi_2 \\
= \quad & \mathsf{fst}\,\pi_2{}^{-1}\ ; \\
& \quad ((\mathsf{fst}\,zero\ ;\mathsf{row}(add\ ;\pi_2{}^{-1}))\updownarrow \mathsf{row}(dist\ ;[mul,\mathcal{D}]))\ ; \\
& \qquad \mathsf{snd}\,\pi_1\ ;\pi_2 \\
= \quad & \mathsf{fst}(\pi_2{}^{-1}\ ;\mathsf{fst}\,zero)\ ; \\
& \quad \mathsf{row}((add\ ;\pi_2{}^{-1})\updownarrow(dist\ ;[mul,\mathcal{D}]))\ ; \\
& \qquad \pi_2\ ;\pi_1 \\
= \quad & \mathsf{fst}(\pi_2{}^{-1}\ ;\mathsf{fst}\,zero)\ ;\mathsf{row}(W\ ;\mathsf{snd}\,\mathsf{snd}\,\mathcal{D})\ ;\pi_2\ ;\pi_1 \\
& \text{where } W=(add\ ;\pi_2{}^{-1})\updownarrow(dist\ ;\mathsf{fst}\,mul) \\
= \quad & \mathsf{fst}(\pi_2{}^{-1}\ ;\mathsf{fst}\,zero)\ ;\mathsf{rdl}(W\ ;\pi_2\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1
\end{aligned}
$$

This completes the rearrangement of the correlator into a very regular circuit with few different sorts of components, and only local connections between those components. In fact there is only one sort of component other than latches: the cell $W$ performs the only 'word-level' operation that we need. By 'word' here we just mean a number: we are making a distinction with 'bit-level' operations which deal only with ones and noughts.

### 1.7.5 Making the circuit systolic

Unfortunately although the wires explicit in the structure of the row are all short – which is what we mean by saying that the connections are all local – there are some potentially long propagation paths in the circuit. The one which stands out in figure 1.30 is the lower path along the row, that through the accumulator. We would prefer to avoid these long paths, on the assumption that they will take a long time to settle into a known state. Our strategy is to trade latency for throughput by pipelining the long paths.

Yet another instance of Horner's rule can be used to pipeline this path.

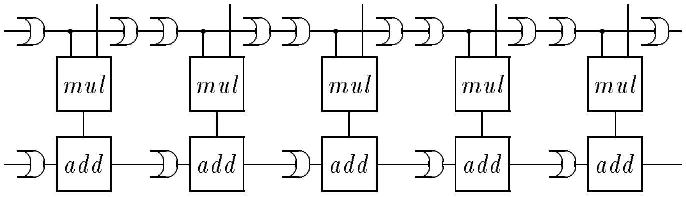



Figure 1.31: an instance of $\mathsf{rdl}(\mathsf{fst}\,\mathcal{D}\ ;W\ ;\pi_2\ ;\mathsf{snd}\,\mathcal{D})$

Knowing that $((\mathsf{irt}\,\mathcal{D}) \setminus apl^{-1})\ ;\mathsf{rdl}\,R = \mathsf{rdl}(\mathsf{fst}\,\mathcal{D}\ ;R)$ tempts us to adjust the timing of the inputs and implement not $CORR$ but

$$\begin{aligned}&((\mathsf{irt}\,\mathcal{D}) \setminus apl^{-1})\ ;CORR\\ &\quad= (\mathsf{irt}\,\mathcal{D}) \setminus apl^{-1}\ ;\mathsf{fst}({\pi_2}^{-1}\ ;\mathsf{fst}\,zero)\ ;\mathsf{rdl}(W\ ;\pi_2\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1\\ &\quad= \mathsf{fst}({\pi_2}^{-1}\ ;\mathsf{fst}\,zero)\ ;((\mathsf{irt}\,\mathcal{D}) \setminus apl^{-1})\ ;\mathsf{rdl}(W\ ;\pi_2\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1\\ &\quad= \mathsf{fst}({\pi_2}^{-1}\ ;\mathsf{fst}\,zero)\ ;\mathsf{rdl}(\mathsf{fst}\,\mathcal{D}\ ;W\ ;\pi_2\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1\end{aligned}$$

an example of which is shown in figure 1.31. This circuit has no combinational propagation paths longer than one cell. Such a design is called systolic, in this case word-level systolic, perhaps because the clocking of the latches pumps data around the circuit. As we use it here the word 'systolic' means just that there are no combinational paths (at a given level of detail).

The problem with this implementation of the correlator is that, contrary to appearances in our diagrams, latches tend to be much bigger and so more expensive than the other components in a circuit. The design in figure 1.31 has in each cell three latches two of which are big enough to store a value of $d$, and one of which is big enough to store a value of the accumulated sum.

Looking for an alternative strategy we might try skewing in the other direction

$$\begin{aligned}&((\mathsf{irt}\,\mathcal{D}^{-1}) \setminus apl^{-1})\ ;CORR\\ &\quad= \mathsf{fst}({\pi_2}^{-1}\ ;\mathsf{fst}\,zero)\ ;\mathsf{rdl}(\mathsf{fst}\,\mathcal{D}^{-1}\ ;W\ ;\pi_2\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1\end{aligned}$$

which introduces anti-delays into the accumulator. In order to be able to implement these by latches we would need to reverse the data-flow in the accumulator. This would not be too difficult to do. However, the real problem with this implementation is that 'unskewing' has cancelled the effect of the latches in the shift register, for

$$\begin{aligned}&\mathsf{rdl}(\mathsf{fst}\,\mathcal{D}^{-1}\ ;W\ ;\pi_2\ ;\mathsf{snd}\,\mathcal{D})\\ &\quad= \mathsf{fst}\,\mathcal{D}^{-1}\ ;\mathsf{rdl}(W\ ;\pi_2\ ;\mathsf{snd}\,\mathcal{D}\ ;\mathcal{D}^{-1})\ ;\mathcal{D}\\ &\quad= \mathsf{fst}\,\mathcal{D}^{-1}\ ;\mathsf{rdl}(W\ ;\pi_2\ ;\mathsf{fst}\,\mathcal{D}^{-1}\ ;\mathsf{snd}(\mathcal{D}\ ;\mathcal{D}^{-1}))\ ;\mathcal{D}\\ &\quad= \mathsf{fst}\,\mathcal{D}^{-1}\ ;\mathsf{rdl}(W\ ;\pi_2\ ;\mathsf{fst}\,\mathcal{D}^{-1})\ ;\mathcal{D}\end{aligned}$$

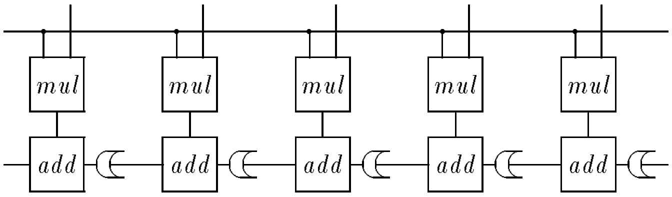



Figure 1.32: an instance of $\mathsf{rdl}(W\ ;\pi_2\ ;\mathsf{fst}\,\mathcal{D}^{-1})$

and there is now a long combinational propagation path along the top of the circuit, as illustrated in figure 1.32.

We have run up against a common problem in regular array design. We have chosen directions of data flow in such a way that attempting to make the circuit systolic results either in too many latches or in long combinational paths. The solution is to reverse the direction of one of the data paths and then to slow the circuit before skewing. Doubling up the delays in the shift register ensures that skewing only wipes out half of those delays, so that both paths end up being properly pipelined.

We choose to reverse the direction of the shift-register because that looks easier. Since $\mathsf{swp}\ \mathsf{row}\ R = \mathsf{snd}\ rev\ ;\mathsf{row}\ \mathsf{swp}\ R\ ;\mathsf{fst}\ rev$ and $\mathsf{swp}(dist\ ;\mathsf{snd}\,\mathcal{D}) = \mathsf{fst}\,\mathcal{D}^{-1}\ ;dist$ we can calculate that

$$\begin{array}{ll} \multicolumn{2}{l}{\langle d,r\rangle\ (\mathsf{row}(dist\ ;\mathsf{snd}\,\mathcal{D})\ ;\pi_1)\ s} \\ \iff & r\ ({\pi_2}^{-1}\ ;\mathsf{swp}\ \mathsf{row}(dist\ ;\mathsf{snd}\,\mathcal{D}))\ \langle s,d\rangle \\ \iff & r\ (rev\ ;{\pi_2}^{-1}\ ;\mathsf{row}(\mathsf{fst}\,\mathcal{D}^{-1}\ ;dist)\ ;\mathsf{fst}\ rev)\ \langle s,d\rangle \end{array}$$

We know from our first design that $s\ (\mathsf{map}\ mul\ ;acc)\ c$, so

$$r\ (rev\ ;{\pi_2}^{-1}\ ;\mathsf{row}(\mathsf{fst}\,\mathcal{D}^{-1}\ ;dist)\ ;\mathsf{fst}(rev\ ;\mathsf{map}\ mul\ ;acc))\ \langle c,d\rangle$$

and because $rev\ ;\mathsf{map}\ mul = \mathsf{map}\ mul\ ;rev$ and $rev\ ;acc = acc$ we can remove the internal $rev$. Clearly we do not want to implement the $rev$ on the left-hand end in silicon, so we play the trick of changing the way in which the inputs are presented to the circuit, and implementing $rev\ ;CORR'$ where $r\ CORR'\ \langle c,d\rangle$.

$$\begin{array}{rcl} rev\ ;CORR' & = & {\pi_2}^{-1}\ ;\mathsf{row}(\mathsf{fst}\,\mathcal{D}^{-1}\ ;dist)\ ;\mathsf{fst}(\mathsf{map}\ mul\ ;acc) \\ & = & {\pi_2}^{-1}\ ;\mathsf{row}(\mathsf{fst}\,\mathcal{D}^{-1}\ ;dist\ ;\mathsf{fst}\ mul)\ ; \\ & & \quad \mathsf{fst}({\pi_2}^{-1}\ ;\mathsf{fst}\ zero\ ;\mathsf{row}(add\ ;{\pi_2}^{-1})\ ;\pi_2) \end{array}$$

As before, we rearrange this into a pair of rows, one below the other.

$$\begin{array}{rcl} rev\ ;CORR' & = & {\pi_2}^{-1}\ ;\mathsf{fst}\,\mathsf{fst}\ zero\ ;\mathsf{row}(\mathsf{fst}\,\mathsf{snd}\,\mathcal{D}^{-1}\ ;W)\ ;\pi_2 \\ & & \text{where } W = (add\ ;{\pi_2}^{-1}) \updownarrow (dist\ ;\mathsf{fst}\ mul) \\ & = & {\pi_2}^{-1}\ ;\mathsf{fst}\,\mathsf{fst}\ zero\ ;\mathsf{rdl}(\mathsf{fst}\,\mathsf{snd}\,\mathcal{D}^{-1}\ ;W\ ;\pi_2) \end{array}$$

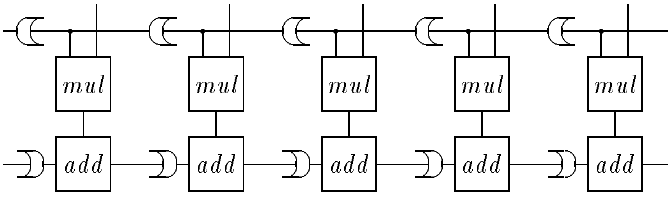



Figure 1.33: an instance of $\mathsf{rdl}(\mathsf{fst}[\mathcal{D}, \mathcal{D}^{-1}] \,;\, W \,;\, \pi_2)$

Next, we slow the circuit

$$\begin{aligned} rev \,;\, \mathsf{slow}\, CORR' &= \mathsf{slow}(rev \,;\, CORR') \\ &= {\pi_2}^{-1} \,;\, \mathsf{fst}\,\mathsf{fst}\, zero \,;\, \mathsf{rdl}(\mathsf{fst}\,\mathsf{snd}\,\mathcal{D}^{-2} \,;\, W \,;\, \pi_2) \end{aligned}$$

and retime using Horner's rule

$$\begin{aligned} &\mathsf{irt}\,\mathcal{D} \,;\, rev \,;\, \mathsf{slow}\, CORR' \\ &\quad= \mathsf{irt}\,\mathcal{D} \,;\, {\pi_2}^{-1} \,;\, \mathsf{fst}\,\mathsf{fst}\, zero \,;\, \mathsf{rdl}(\mathsf{fst}\,\mathsf{snd}\,\mathcal{D}^{-2} \,;\, W \,;\, \pi_2) \\ &\quad= {\pi_2}^{-1} \,;\, \mathsf{fst}\,\mathsf{fst}\, zero \,;\, ((\mathsf{irt}\,\mathcal{D}) \setminus apl^{-1}) \,;\, \mathsf{rdl}(\mathsf{fst}\,\mathsf{snd}\,\mathcal{D}^{-2} \,;\, W \,;\, \pi_2) \\ &\quad= {\pi_2}^{-1} \,;\, \mathsf{fst}\,\mathsf{fst}\, zero \,;\, \mathsf{rdl}(\mathsf{fst}(\mathcal{D} \,;\, \mathsf{snd}\,\mathcal{D}^{-2}) \,;\, W \,;\, \pi_2) \\ &\quad= {\pi_2}^{-1} \,;\, \mathsf{fst}\,\mathsf{fst}\, zero \,;\, \mathsf{rdl}(\mathsf{fst}[\mathcal{D}, \mathcal{D}^{-1}] \,;\, W \,;\, \pi_2) \end{aligned}$$

which is illustrated in figure 1.33. The left-hand side reminds us that we have only implemented a slow correlator and that the weights $r$ should be presented to the circuit in reverse order and time-skewed.

That completes the development of two word-level systolic correlators, one of which uses $3N$ latches, the other of which uses only $2N$ at the expense of a doubled clock speed.

### 1.7.6 Refining to a bit level implementation

The next stage of the development is to refine the operations to ones that combine and produce bit-vectors, and there we will stop. Earlier presentations of this circuit such as reference [Sheer88] have made this refinement step in one leap, appealing to 'well-known' implementations of the arithmetic functions. However since we can express representation and abstraction relations in the same notation as the circuit, we have the opportunity to bridge the gap between word- and bit-level. This is work still in progress at Glasgow and Oxford, so this section does no more than outline the shape that the refinement would have.

The strategy for refining from word- to bit-level is: to describe an abstraction relation that relates the bit-vectors operated on by the circuit to the numbers that they represent; to compose this relation with the word-level description of the correlator; and then to simplify the resulting expression. We will use a simple bit-parallel binary representation of numbers, most significant bit first. We know to choose this representation because we have done the development before using the least significant bit first representation and regretted the choice; the resulting circuit was hard to pipeline. Choosing the representation with the most-significant-bit first gives simpler data flow through the bit-level implementation of the cell $W$.

If $bit$ is the identity relation on bits, that is if it is the type of bits, and $bin = \mathsf{map}\, bit \,;\, rev \,;\, \mathsf{tri}(\times 2) \,;\, acc$, then since $bin^{-1} \,;\, bin = nat$ is the identity on natural numbers, $bin$ is an abstraction from bit vectors to the natural numbers. Moreover if $bin_n = n \,;\, bin$ then $bin_n \,;\, bin_n^{-1} = n \,;\, \mathsf{map}\, bit$ and $bin_n^{-1} \,;\, bin_n = nat_n$ where $nat_n$ is the type of those numbers in the range $0 \leq i < 2^n$ representable in $n$ bits, a sub-type of $nat$. So $bin_n$ abstracts $n$-bit bit-vectors to small enough numbers, and $bin_n^{-1}$ represents any small enough number as an $n$-bit bit-vector.

Suppose we are not going to want to give negative inputs to the correlator: then we can show that no part of the circuit need then deal with negative numbers, and that the output must be non-negative. The next thing to do is to decide on some appropriate widths for the bit-vectors. Since each of the component parts of $W$ is going to become a column of cells, and since we will want to interleave these columns, it will be simplest if we can arrange for everything to have the same width. This is a great simplification, but one that you need not make if you are determined to produce the smallest possible implementation of the circuit.

As in the circuit in reference [McCab82], we further simplify matters by assuming that the reference inputs $r_i$ are only a single bit wide. If the largest $d$-value can be represented by a $k$-wide bit vector – that is, if it is in $nat_k$ – then any value manipulated by the circuit can be represented by an $n$-wide bit-vector, provided $n \geq N + k$.

$$
\begin{aligned}
&N \,;\, \mathsf{map}\, nat_1 \,;\, \mathsf{irt}\,\mathcal{D} \,;\, rev \,;\, \mathsf{slow}\; CORR' \,;\, \mathsf{snd}\; nat_k \\
&\qquad = \quad N \,;\, \mathsf{map}\, nat_1 \,;\, \mathsf{irt}\,\mathcal{D} \,;\, rev \,;\, \mathsf{slow}\; CORR' \,;\, [nat_n, nat_k] \\
&\qquad = \quad N \,;\, \mathsf{map}\, nat_1 \,;\, \mathsf{irt}\,\mathcal{D} \,;\, rev \,;\, \mathsf{slow}\; CORR' \,;\, [nat_n, nat_n] \,;\, \mathsf{snd}\; nat_k
\end{aligned}
$$

Now our strategy is to replace each $nat_n$ by $bin_n^{-1} \,;\, bin_n$ and to push the representation relations down into the circuit, until the circuit can be described as a collection of components that we know how to implement. Doing this, we find that we need to implement

$$
\begin{aligned}
&[[bin_n, bin_n], bit] \,;\, W \,;\, \pi_2 \,;\, [bin_n, bin_n]^{-1} \\
&\qquad = \quad (([bin_n, bin_n] \,;\, add \,;\, bin_n^{-1} \,;\, {\pi_2}^{-1}) \updownarrow \\
&\qquad\qquad (dist \,;\, \mathsf{fst}([bin_n, bit] \,;\, mul \,;\, bin_n^{-1}))) \,;\, \pi_2
\end{aligned}
$$

We have shown elsewhere [Jon90] how expressions like $[bin_n, bin_n] ; add ; bin_n^{-1}$ can be implemented by arrays of simpler cells. Here, we state the necessary theorems without proof. Firstly

$$[bin_n, bin_n] ; add ; bin_n^{-1} \quad = \quad zip ; n ; {\pi_1}^{-1} ; \mathsf{snd}\ zero ; \mathsf{col}\ FA ; \pi_2$$

where the relation $FA$ is just a standard full-adder, a bit-level component that we know how to implement:

$$FA \quad = \quad [[bit, bit] ; add, bit] ; add ; add^{-1} ; [(\times 2)^{-1} ; bit^{-1}, bit^{-1}]$$

Similarly

$$\begin{aligned} [bin_n, bit] ; mul ; bin_n^{-1} \quad &= \quad \mathsf{col}\ M ; \pi_2 ; n \\ & \quad\quad \text{where } M = fork ; [\pi_2, [bit, bit] ; mul ; bit^{-1}] \end{aligned}$$

and the bit-multiplier $[bit, bit] ; mul ; bit^{-1}$ can be implemented by the same electronics as you would use to implement logical 'and'.

That implements the addition and multiplication parts of the cell as two separate columns, as illustrated in figure 1.34. It would be a good idea to eliminate some wire crossings, and to do this it will be necessary to interleave the columns. We start by merging the *dist* at the top of the diagram with the column of multiplier cells.

$$\begin{aligned} dist ; \mathsf{fst}(\mathsf{col}\ M ; \pi_2 ; n) \quad &= \quad \mathsf{col}\ M' ; \pi_2 ; n ; zip^{-1} \\ & \quad\quad \text{where } M' = dist ; \mathsf{fst}\ M ; lsh \end{aligned}$$

Now we are in a position to interleave the column of $M'$ cells with the column of $FA$ cells.

$$\begin{aligned} &[[bin_n, bin_n], bit] ; W ; \pi_2 ; [bin_n, bin_n]^{-1} \\ &\quad = \quad (zip ; n ; {\pi_1}^{-1} ; \mathsf{snd}\ zero ; \mathsf{col}\ FA ; \pi_2 ; {\pi_2}^{-1}) \updownarrow \\ &\quad\quad\quad\quad (\mathsf{col}\ M' ; \pi_2 ; n ; zip^{-1})) ; \pi_2 \\ &\quad = \quad [zip ; n, {\pi_1}^{-1} ; \mathsf{snd}\ zero] ; \mathsf{col}\ B ; \pi_2 ; (zip ; n)^{-1} \\ &\quad\quad\quad \text{where } B = ((swap \updownarrow M') ; \mathsf{snd}\ rsh) \leftrightarrow (FA \updownarrow swap) \end{aligned}$$

We have stated without justification a theorem about the interleaving of two columns. To approach this design properly, it would have been better to introduce a new higher order function corresponding to the way the cell $B$ is built from the cells $FA$ and $M'$ and to develop some of the algebra of that higher order function.

Figure 1.35 shows the column that we have just made. The relation $B$ can be implemented by a full-adder, an 'and' gate and some wiring; so we can stop here and plug everything back together.

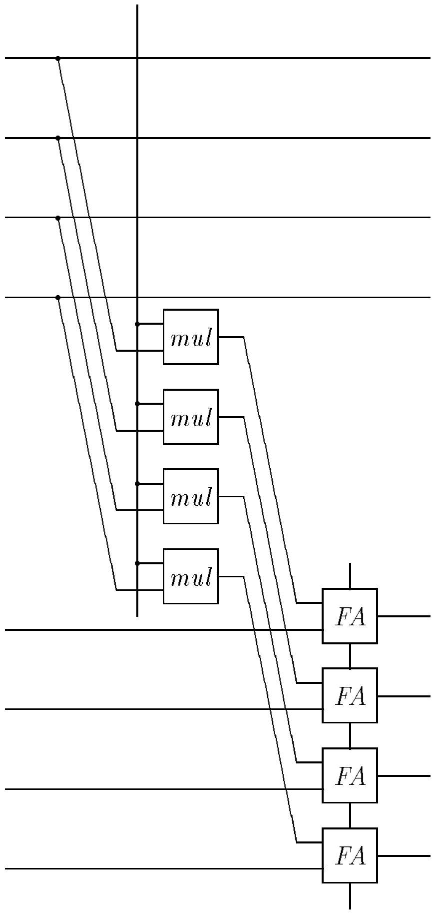



Figure 1.34: $[[bin_4, bin_4], bit] \,;W\,; \mathsf{snd}[bin_4, bin_4]^{-1}$ implemented by an interleaving of separate columns

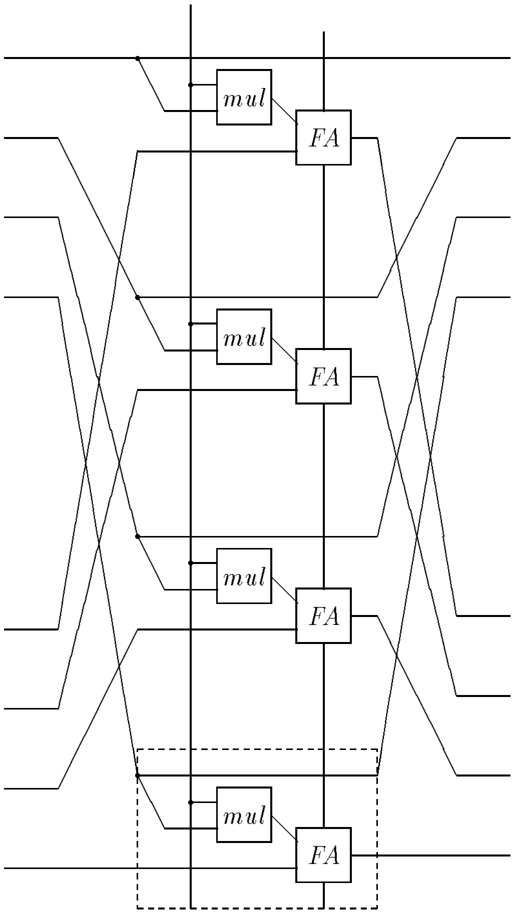



Figure 1.35: $[[bin_4, bin_4], bit]$ ; $W$ ; $\mathsf{snd}[bin_4, bin_4]^{-1}$ implemented by a single column of interleaved components

$$\begin{aligned}
&N \,;\, \mathsf{map}\, nat_1 \,;\, \mathsf{irt}\, \mathcal{D} \,;\, rev \,;\, \mathsf{slow}\, CORR' \,;\, \mathsf{snd}\, nat_k \\
&\quad = \; N \,;\, \mathsf{map}\, bit^{-1} \,;\, {\pi_2}^{-1} \,;\, \mathsf{fst}\,\mathsf{fst}\,\mathsf{map}\, zero \,; \\
&\qquad\quad \mathsf{rdl}(\mathsf{fst}[\mathcal{D}, \mathcal{D}^{-1}] \,;\, [[bin_n, bin_n], bit] \,;\, W \,;\, \pi_2 \,;\, [bin_n, bin_n]^{-1}) \,; \\
&\qquad\qquad [bin_n, bin_n] \,;\, \mathsf{snd}\, nat_k \\
&\quad = \; N \,;\, \mathsf{map}\, bit^{-1} \,;\, {\pi_2}^{-1} \,;\, \mathsf{fst}\,\mathsf{fst}\,\mathsf{map}\, zero \,; \\
&\qquad\quad \mathsf{rdl}([[\mathcal{D}, \mathcal{D}^{-1}] \,;\, zip \,;\, n, {\pi_1}^{-1} \,;\, \mathsf{snd}\, zero] \,;\, \mathsf{col}\, B \,;\, \pi_2 \,;\, (zip \,;\, n)^{-1}) \,; \\
&\qquad\qquad [bin_n, bin_n \,;\, nat_k] \\
&\quad = \; N \,;\, \mathsf{map}\, bit^{-1} \,;\, {\pi_2}^{-1} \,;\, [zip \,;\, \mathsf{map}\,\mathsf{fst}\, zero, \mathsf{map}({\pi_1}^{-1} \,;\, \mathsf{snd}\, zero)] \,; \\
&\qquad\quad \mathsf{rdl}(\mathsf{fst}\,\mathsf{map}[\mathcal{D}, \mathcal{D}^{-1}] \,;\, \mathsf{col}\, B \,;\, \pi_2) \,; \\
&\qquad\qquad zip^{-1} \,;\, [bin_n, bin_n \,;\, nat_k] \\
&\quad = \; N \,;\, \mathsf{map}\, bit^{-1} \,;\, {\pi_2}^{-1} \,;\, [\mathsf{map}\,\mathsf{fst}\, zero, \mathsf{map}({\pi_1}^{-1} \,;\, \mathsf{snd}\, zero)] \,; \\
&\qquad\quad \mathsf{row}\,\mathsf{col}(\mathsf{fst}[\mathcal{D}, \mathcal{D}^{-1}] \,;\, B) \,; \\
&\qquad\qquad \pi_2 \,;\, zip^{-1} \,;\, [bin_n, bin_n \,;\, nat_k]
\end{aligned}$$

The middle term on the right-hand side is just an $N$ by $n$ grid of identical cells, and this part would certainly be implemented as part of the circuit. The terms at either end of the expression are what remains of the representation relation. They describe how the input must be presented to the circuit, and how to read the output: $N \,; \mathsf{map}\, bit^{-1}$ means 'a list of $N$ one-bit numbers'; and $zip^{-1}$; $[bin_n, bin_n \,;\, nat_k]$ means 'the interleaving of an $n$-bit binary representation of one number and the $n$-bit representation of another number which could have been represented in $k$ bits'. The remaining terms describe the tying of certain lines to a level representing the bit zero, and the discarding of the signals on other lines.

### 1.7.7 Making the implementation systolic at the bit level

The circuit represented by $G = \mathsf{row}\,\mathsf{col}(\mathsf{fst}[\mathcal{D}, \mathcal{D}^{-1}] \,;\, B)$, illustrated in figure 1.36, has no long horizontal combinational paths because of the $[\mathcal{D}, \mathcal{D}^{-1}]$ through which each horizontal data-path passes at each cell. Moreover since the change of representation that we used did not involve any change in the direction of the data-flow, both the delays and anti-delays in the circuit remain implementable as latches.

There are however vertical unbroken combinational paths through each column of cells. In figure 1.35 these are the long vertical line that carries the reference bit, and a route through the carry-path that joins the $FA$ components. We must eliminate these long paths by skewing. We leave this task as an exercise for the reader. (Hint: the appropriate skewing law for columns is reminiscent of Horner's rule for right reductions:

$$(\mathsf{tri}\,\mathcal{D} \setminus apr^{-1}) \,;\, \mathsf{col}\, R \,;\, \mathsf{snd}\,\mathsf{tri}\,\mathcal{D}^{-1} = \mathsf{col}(\mathsf{snd}\,\mathcal{D} \,;\, R)$$

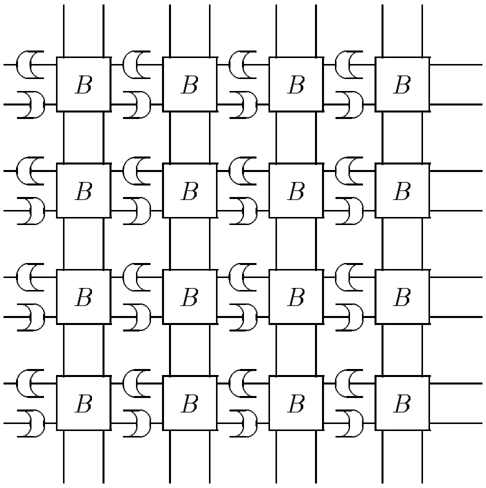



Figure 1.36: an instance of $\mathsf{row\,col}(\mathsf{fst}[\mathcal{D}, \mathcal{D}^{-1}] \,;\, B)$

and although we could skew each column in turn, it is simplest to exchange the row and column structure.)

## 1.8 Butterfly networks

Butterfly networks and algorithms are a design cliché in digital signal processing, just as arrays and grids are. Typical applications are sorting, fast Fourier transform, and the interconnection of processors and memories or of networks of processors. Both the 'cube connected cycle' and the 'hypercube' can be made by folding up a butterfly network. From the point of view of an algorithm designer, butterfly networks are interesting because they have many simple recursive decompositions. We need some extra Ruby notation in order to be able to describe butterflies.

### 1.8.1 The perfect shuffle

We have already made a lot of use of the wiring relation *zip* which was defined by $\langle x, y\rangle\ zip\ z \iff \forall i.\, z_i = \langle x_i, y_i\rangle$. Butterfly networks use a related piece of plumbing, the perfect shuffle or *riffle*. Consider a deck of fifty-two cards; it can be riffled by dividing in half, interleaving to give twenty-six pairs, and then 'unpairing' by forgetting about the pairing.

This permutation on even-length lists is actually used on silicon despite the crossovers that it introduces. We can describe it by composing three simpler wiring relations, $halve$, $zip$ and $pair^{-1}$ ('unpair'). The relation $halve$ relates a list of even length to a pair containing the first and second halves of the list so $2n \,;\, halve = app^{-1} \,;\, [n, n]$. The relation $pair$ defined by $x \; pair \; y \iff \forall i.\, y_i = \langle x_{2i}, x_{2i+1}\rangle$ divides a list of length $2n$ into an $n$-list of pairs. (We used it earlier when defining $\mathsf{slow}$.) Define

$$riffle \quad = \quad halve \,;\, zip \,;\, pair^{-1}$$

Some of the useful properties of $riffle$ are

$$\begin{aligned} 2n \,;\, riffle &= riffle \,;\, 2n \\ 2n \,;\, riffle \,;\, riffle^{-1} &= 2n \\ 2n \,;\, riffle^{-1} \,;\, riffle &= 2n \\ 2^k \,;\, riffle^k &= 2^k \end{aligned}$$

### 1.8.2 Two and interleave

Although we could now describe butterflies using the combining forms and wiring relations already introduced, two new combining forms simplify the description.

Let $R$ and $S$ both be of type $\mathsf{map}\,\iota \rightarrow \mathsf{map}\,\iota$. Define

$$R \mid S \quad = \quad [R, S] \setminus halve^{-1}$$

and

$$\mathsf{two}\, R \quad = \quad R \mid R$$

The relation $\mathsf{two}\, R$ relates by $R$ the first halves of the lists in its domain and range, and similarly the second halves. You can think of it as placing two copies of $R$ across a bus. Similarly $\mathsf{two}^n\, R$ places $2^n$ copies of $R$ across a bus.

The relation $\mathsf{ilv}\, R$ on the other hand relates by $R$ the even numbered elements of the lists in its domain and range, and similarly the odd numbered elements.

$$\mathsf{ilv}\, R \quad = \quad (\mathsf{two}\, R) \setminus riffle$$

Surprisingly enough, $\mathsf{two}$ and $\mathsf{ilv}$ can be exchanged

$$\mathsf{two}\,\mathsf{ilv}\, R \quad = \quad \mathsf{ilv}\,\mathsf{two}\, R$$

a fact that we will find very useful later. If we know that $R$ and $S$ relate only lists of equal length, then both $\mathsf{ilv}$ and $\mathsf{two}$ distribute over composition: $\mathsf{ilv}(R \,;\, S) = (\mathsf{ilv}\, R) \,;\, (\mathsf{ilv}\, S)$ and $\mathsf{two}(R \,;\, S) = (\mathsf{two}\, R) \,;\, (\mathsf{two}\, S)$.

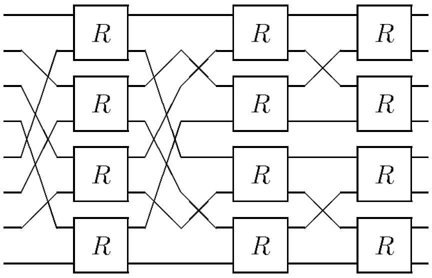



Figure 1.37: a butterfly of size 3

### 1.8.3 Fat composition

A butterfly network of size $k$ consists of a composition of $k$ ranks, each of which contains the same number of copies of the basic cell, but disposed differently across the bus. We need a way of describing this kind of composition of several different but similar relations. This version of composition, $⨟$ (read 'fat composition'), when applied to a list of relations composes them. This means that we need a notation for lists: write $\langle i : 0 \leq i < n : x_i \rangle$ for the list of length $n$, the $i$th element of which is $x_i$. So for example $⨟\langle i : 0 \leq i < 4 : R_i \rangle$ is the relation $R_0 \,;R_1 \,;R_2 \,;R_3$. You can think of $⨟$ as the quantifier corresponding to composition, just as $\sum$ is the quantifier corresponding to addition.

### 1.8.4 Describing the butterfly network

Let us assume that our basic cell $R$ is of type $2^{n+1} \rightarrow 2^{n+1}$ for some fixed $n$. Usually $n$ will be zero, for example when we build butterflies of comparators or of two way switches. However making $n$ a parameter will be a useful generalization when trying to understand the recursive structure of the network.

A butterfly network of size $k$ consists of $k$ columns or ranks, each containing $2^{k-1}$ copies of the basic cell. Each rank distributes these copies across the bus – which must be of width $2^{k+n}$ – in a different but regular way.

$$2^{n+k} \,; \bowtie R \quad = \quad ⨟\langle i : 0 \leq i < k : \mathsf{two}^i \, \mathsf{ilv}^{k-i-1} \, R \rangle$$

A butterfly of size zero is simply the identity on lists of length $2^n$. If we fix $n$ to be zero so that $R : 2 \rightarrow 2$ then a butterfly of size 3 has 3 ranks, each with 4 copies of $R$. It can be variously described as

$$\begin{aligned} 2^3 \,; \bowtie R \quad &= \quad \mathsf{ilv}^2 \, R \,; \mathsf{two}(\mathsf{ilv} \, R \,; \mathsf{two} \, R) \\ &= \quad \mathsf{ilv}^2 \, R \,; \mathsf{two} \, \mathsf{ilv} \, R \,; \mathsf{two}^2 \, R \end{aligned}$$

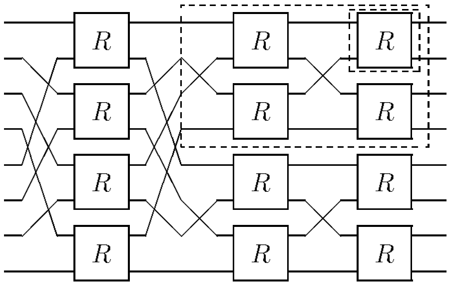



Figure 1.38: first recursive decomposition of the butterfly

$$\begin{aligned} &= \mathsf{ilv}^2\, R \,;\, \mathsf{ilv}\,\mathsf{two}\, R \,;\, \mathsf{two}^2\, R \\ &= \mathsf{ilv}(\mathsf{ilv}\, R \,;\, \mathsf{two}\, R) \,;\, \mathsf{two}^2\, R \end{aligned}$$

You can think of these descriptions as different ways of decomposing the same relation. Figure 1.37 shows one possible way of laying out the relation. There are of course many others.

Starting from our iterative description of the butterfly, we can use the properties of two, ilv and *riffle* to derive several alternative decompositions. Unwinding the definition of the butterfly from the left gives

$$\begin{aligned} 2^{n+k+1} \,;\bowtie R &= \mathbin{⨟}\langle i : 0 \le i < k+1 : \mathsf{two}^i\, \mathsf{ilv}^{(k+1)-i-1}\, R\rangle \\ &= (\mathsf{ilv}^k\, R) \,;\, \mathbin{⨟}\langle i : 1 \le i < k+1 : \mathsf{two}^i\, \mathsf{ilv}^{(k+1)-i-1}\, R\rangle \\ &= (\mathsf{ilv}^k\, R) \,;\, \mathbin{⨟}\langle j : 0 \le j < k : \mathsf{two}^{j+1}\, \mathsf{ilv}^{(k+1)-(j+1)-1}\, R\rangle \\ &= (\mathsf{ilv}^k\, R) \,;\, \mathbin{⨟}\langle j : 0 \le j < k : \mathsf{two}\,\mathsf{two}^j\, \mathsf{ilv}^{k-j-1}\, R\rangle \\ &= (\mathsf{ilv}^k\, R) \,;\, \mathsf{two}\, \mathbin{⨟}\langle j : 0 \le j < k : \mathsf{two}^j\, \mathsf{ilv}^{k-j-1}\, R\rangle \\ &= (\mathsf{ilv}^k\, R) \,;\, \mathsf{two}(2^{n+k} \,;\bowtie R) \end{aligned}$$

showing that a butterfly of size $k+1$ is made from $2^k$ copies of $R$ making up the first rank, some wiring, and two recursive instances of a butterfly of size $k$. This is perhaps a more familiar description of the butterfly. Figure 1.38 outlines an instance of a butterfly of size 2 and one of size 1 within a butterfly of size 3.

Similarly unwinding the fat composition from the right gives us

$$\begin{aligned} 2^{n+k+1} \,;\bowtie R &= \mathbin{⨟}\langle i : 0 \le i < k+1 : \mathsf{two}^i\, \mathsf{ilv}^{(k+1)-i-1}\, R\rangle \\ &= \mathbin{⨟}\langle i : 0 \le i < k : \mathsf{two}^i\, \mathsf{ilv}^{(k+1)-i-1}\, R\rangle \,;\, \mathsf{two}^k\, R \\ &= \mathbin{⨟}\langle i : 0 \le i < k : \mathsf{ilv}\,\mathsf{two}^i\, \mathsf{ilv}^{k-i-1}\, R\rangle \,;\, \mathsf{two}^k\, R \\ &= \mathsf{ilv}(\mathbin{⨟}\langle i : 0 \le i < k : \mathsf{two}^i\, \mathsf{ilv}^{k-i-1}\, R\rangle) \,;\, \mathsf{two}^k\, R \\ &= \mathsf{ilv}(2^{n+k} \,;\bowtie R) \,;\, \mathsf{two}^k\, R \end{aligned}$$

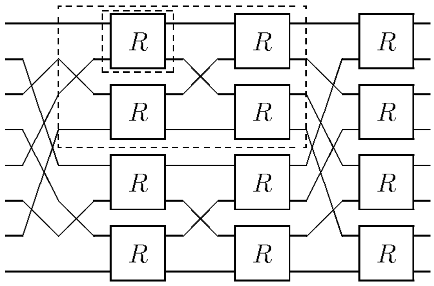



Figure 1.39: a second recursive decomposition

This time we can see that a butterfly consists of two smaller butterflies interleaved, composed with a rank consisting just of $\mathsf{two}^k\,R$. Figure 1.39 outlines this second recursive decomposition.

Because $\mathsf{two}$ and $\mathsf{ilv}$ can be exchanged, it must be the case that

$$\begin{aligned}
\mathsf{two}(2^{n+k}\,;\bowtie R) &= \mathsf{two}\,\fatsemi\langle j : 0 \le j < k : \mathsf{two}^j\,\mathsf{ilv}^{k-j-1}\,R\rangle \\
&= \fatsemi\langle j : 0 \le j < k : \mathsf{two}\,\mathsf{two}^j\,\mathsf{ilv}^{k-j-1}\,R\rangle \\
&= \fatsemi\langle j : 0 \le j < k : \mathsf{two}^j\,\mathsf{ilv}^{k-j-1}(\mathsf{two}\,R)\rangle \\
&= 2^{n+k+1}\,;\bowtie \mathsf{two}\,R
\end{aligned}$$

and

$$\mathsf{ilv}(2^{n+k}\,;\bowtie R) = 2^{n+k+1}\,;\bowtie \mathsf{ilv}\,R$$

which gives us another pair of decompositions of a butterfly each containing only a single instance of the next smaller butterfly, although with a larger component.

$$\begin{aligned}
2^{n+k+1}\,;\bowtie R &= (\mathsf{ilv}^k\,R)\,;(\bowtie \mathsf{two}\,R) \\
&= (\bowtie \mathsf{ilv}\,R)\,;(\mathsf{two}^k\,R)
\end{aligned}$$

The second of these decompositions is illustrated in figure 1.40: the larger outlined box is an instance of $\bowtie \mathsf{ilv}\,R$; and the smaller is just $\mathsf{ilv}^2\,R$ which is a degenerate instance of $\bowtie \mathsf{ilv}^2\,R$.

The butterfly has many more recursive decompositions because we need not divide it into a single rank and a recursive call: it can be the composition of two similar parts. The discovery of this decomposition is left as an exercise for the reader.

Finally, let us return to the case in which the basic cell is of type $2 \rightarrow 2$. In that case, we can relate $\mathsf{two}$ and $\mathsf{ilv}$ to $\mathsf{map}$ to get simpler descriptions of the circuit. Suppose that $R : 2 \rightarrow 2$, then

$$2^{k+1}\,;\mathsf{two}^k\,R = 2^{k+1}\,;((\mathsf{map}\,R) \setminus pair^{-1})$$

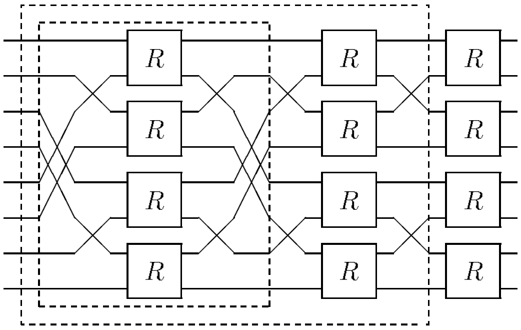


Figure 1.40: singly recursive view

In other words, each $R$ cell operates on pairs of adjacent elements from the lists in the domain and range. The form $((\mathsf{map}\,R) \setminus pair^{-1})$ arises so often that we abbreviate it to $\mathsf{pmap}\,R$. Similarly

$$2^{k+1}\,;\,\mathsf{ilv}^k\,R \quad = \quad 2^{k+1}\,;\,((\mathsf{pmap}\,R) \setminus \mathit{riffle}^{-1})$$

and we often abbreviate $((\mathsf{pmap}\,R) \setminus \mathit{riffle}^{-1})$ as $\mathsf{rpmap}\,R$. The recursive descriptions of a butterfly of $R : 2 \rightarrow 2$ can be rewritten as

$$\begin{aligned} 2^{k+1}\,;\,\bowtie R &= \mathsf{rpmap}\,R\,;\,\mathsf{two}(2^k\,;\,\bowtie R) \\ &= \mathsf{ilv}(2^k\,;\,\bowtie R)\,;\,\mathsf{pmap}\,R \end{aligned}$$

## 1.9 The Fourier transform

This outlines a development of a common digital signal-processing algorithm, having the aim of turning the specification into the Ruby description of a butterfly circuit. There is a fuller presentation of the calculation of the implementation from the specification in reference [Jon89].

### 1.9.1 The discrete Fourier transform

Twenty-five years ago Cooley and Tukey rediscovered an optimizing technique usually attributed to Gauss, who used it in hand calculation. They applied the technique to the discrete Fourier transform, reducing an apparently $O(n^2)$ problem to the almost instantly ubiquitous $O(n \log n)$ 'fast Fourier transform' [Cool65].

The fast Fourier transform is not of course a different transform, but a fast implementation of the discrete transform. Its greatest virtue lies in that it can be executed in $O(\log n)$ time on $O(n)$ processors in a uniform way – it lends itself to a low-latency high-throughput pipelined hardware implementation.

The discrete Fourier transform is defined in terms of the arithmetic on an integral domain. You can think of arithmetic on complex numbers, for a definite example, although there are applications where finite fields or vector spaces over integral domains are appropriate. The derivation depends only on the algebraic properties of the arithmetic, not on the underlying arithmetic itself, so everything said here about the algorithm will be true for finite fields and vector spaces as well.

The discrete Fourier transform of a vector $x$ of length $n$ is a vector $y$ of the same length for which

$$y_j \;=\; \sum_{k:0\le k<n} \omega^{j\times k} \times x_k$$

where $\omega$ is a principal $n$-th root of unity. (In the example of complex numbers, you can think of $\omega = e^{2\pi i/n}$.) The result, $y$, is sometimes called the 'frequency spectrum' of the sample $x$.

Even if the powers of $\omega$ are pre-calculated, it would appear that $O(n^2)$ multiplications are required to evaluate the whole of $y$ for any $x$. The fast algorithm avoids many of these by making use of the fact that $\omega^n = 1$. If $n$ is composite, the calculation can be divided into what amounts to a number of smaller Fourier transforms. Suppose $n = p \times q$, then by a change of variables

$$\begin{aligned} y_{pa+b} &= \sum_{c:0\le c<p}\ \sum_{d:0\le d<q} \omega^{(pa+b)(qc+d)} x_{qc+d} \\ &= \sum_{c:0\le c<p}\ \sum_{d:0\le d<q} (\omega^{pq})^{ac}(\omega^{p})^{ad}(\omega^{q})^{bc}\omega^{bd} x_{qc+d} \\ &= \sum_{d:0\le d<q} (\omega^{p})^{ad}\omega^{bd} \sum_{c:0\le c<p} (\omega^{q})^{bc} x_{qc+d} \end{aligned}$$

Since $\omega^q$ is a $p$-th root of unity, and $\omega^p$ is a $q$-th root of unity, it is not surprising that the above calculation leads to an implementation in which $p$-sized and $q$-sized transforms appear. It is harder, however, to see what that implementation might be.

### 1.9.2 Casting the algorithm in the notation

The first task in a calculation dealing with an algorithm is to cast the specification in the notation that will be used to handle the development. There are two things which we do in this stage.

One part appears to be largely a process of eliminating subscripts, since the usual convention is to specify separately each co-ordinate of an output vector.

The conventional understanding of a specification of the form $y_i = \ldots$ is that the subscript is universally quantified, so that this one equation formally represents a number of different equations, one for each value of $i$. To make clear that an algorithm operates uniformly at all co-ordinates of its output we write a single equation which defines the whole list of output values.

The other part of the translation is to manipulate the specification – which is usually an expression describing the output of a calculation for a given input – into the form of an application to that input of an expression representing the algorithm. The manipulation of the algorithm can then proceed without reference to the particular input.

The discrete Fourier transform was specified by

$$y_j \quad = \sum_{k:0\leq k<n} \omega^{j\times k} \times x_k$$

by which was meant that the output $y$ should be defined for each $j$ in the range $0 \leq j < n$, so meaning that

$$\begin{array}{ll}
 & y = \langle j : 0 \leq j < n : \sum \langle k : 0 \leq k < n : \omega^{j\times k} \times x_k \rangle\rangle \\
\Longleftrightarrow & \{\,\text{meaning of summation and } \mathsf{map}\text{, meaning of exponentiation}\,\} \\
 & \langle j : 0 \leq j < n : \langle k : 0 \leq k < n : \omega^{j\times k} \times x_k \rangle\rangle \; (\mathsf{map}\; acc)\; y \\
 & \text{where } acc = apl^{-1} \,;\mathsf{rdl}\; add = apr^{-1} \,;\mathsf{rdr}\; add \\
\Longleftrightarrow & \{\,\text{meaning of arithmetic exponentiation, associativity of } \times\,\} \\
 & \langle j : 0 \leq j < n : \langle k : 0 \leq k < n : ((\omega \times)^j)^k x_k \rangle\rangle \; (\mathsf{map}\; acc)\; y \\
\Longleftrightarrow & \{\,\text{meaning of } \mathsf{tri}\,\} \\
 & \langle j : 0 \leq j < n : \langle k : 0 \leq k < n : x_k \rangle\rangle \; (\mathsf{tri}\,\mathsf{tri}(\times\omega) \,;\mathsf{map}\; acc)\; y \\
\Longleftrightarrow & \{\,\text{meaning of } join\,\} \\
 & \langle k : 0 \leq k < n : x_k \rangle \; (J^{-1} \,; n \,; \mathsf{tri}\,\mathsf{tri}(\times\omega) \,;\mathsf{map}\; acc)\; y \\
 & \text{where } J = apl^{-1} \,;\mathsf{rdl}\; join = apr^{-1} \,;\mathsf{rdr}\; join \\
\Longleftrightarrow & x\; (J^{-1} \,; n \,; \mathsf{tri}\,\mathsf{tri}(\times\omega) \,;\mathsf{map}\; acc)\; y
\end{array}$$

Since $\omega$ depends on $n$, because $\omega^n = 1$, we will be honest and write $(\times\omega)$ using a new operator $\otimes$ for which $z \otimes n = z \times \omega$. This operation has the property, which will be useful later, that $(\otimes(p \times q))^q = (\otimes p)$. There are two instances of reduction of an associative relation in this description, and we can avoid commitment about how to implement these by introducing a new operator $\mathsf{red}$ for which $\mathsf{red}\; R = apl^{-1} \,;\mathsf{rdl}\; R = apr^{-1} \,;\mathsf{rdr}\; R$ for any associative $R$.

The term in brackets that relates $x$ to $y$ represents the discrete Fourier transform algorithm, but only if $x$ is a list of length $n$, so we will calculate

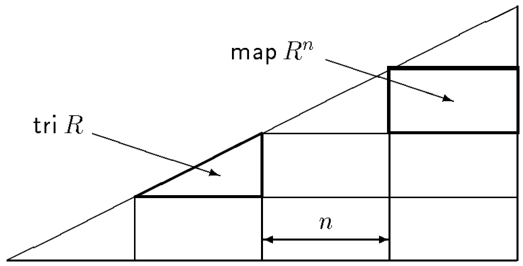



Figure 1.41: illustration of $\mathsf{tri}\, R$ divided into $\mathsf{map}\, n$ ; $\mathsf{tri}\,\mathsf{map}\, R^n$ ; $\mathsf{map}\,\mathsf{tri}\, R$

from the definition

$$n\,;\mathcal{F} \quad = \quad n\,;(\mathsf{red}\, join)^{-1}\,;n\,;\mathsf{tri}\,\mathsf{tri}(\otimes n)\,;\mathsf{map}\,\mathsf{red}\, add$$

### 1.9.3 Dividing large problems into smaller ones

Suppose $R$ is an algorithm or circuit for calculating some list-valued function of a list of values. If it is possible to express $R$ in the form $S\,;\mathsf{red}\, app$, then $S$ is an algorithm for constructing the same result in parts, and may be implementable by a number of independent parts. For example

$$(\mathsf{red}\, join)^{-1} \quad = \quad (\mathsf{red}\, join)^{-1}\,;\mathsf{map}(\mathsf{red}\, join)^{-1}\,;\mathsf{red}\, app$$

describes a divide-and-conquer strategy for fanning out a signal into some number of copies, and then independently fanning out each of those.

Similarly $\mathsf{red}\, app\,;R$ is an algorithm which constructs the same result as $R$ from a partition of the same input into a list of lists. If it is possible to 'simplify' $\mathsf{red}\, app\,;R$ into a form which has a parallel implementation, that gives a strategy for dividing the calculation of $R$. A particularly useful result in the present case is that

$$\mathsf{map}\, n\,;\mathsf{red}\, app\,;\mathsf{tri}\, R \quad = \quad \mathsf{tri}\,\mathsf{map}\, R^n\,;\mathsf{map}\,\mathsf{tri}\, R\,;\mathsf{map}\, n\,;\mathsf{red}\, app$$

which means that $\mathsf{tri}\, R$ can be implemented by a number of (smaller) independent instances of $\mathsf{tri}\, R$ and a triangular array of $\mathsf{map}\, R^n$ components. This equality depends on the restriction to a list-of-lists where every sublist has the same length.

In factorising the discrete Fourier transform, this rule is applied twice to an instance of an expression of the form $\mathsf{tri}\,\mathsf{tri}\, R$.

$$\begin{aligned}
&\mathsf{map}\,\mathsf{map}(\mathsf{map}\, q\,;\mathsf{red}\, app)\,;(\mathsf{map}\, p\,;\mathsf{red}\, app)\,;\mathsf{tri}\,\mathsf{tri}\, R \\
&\quad = \quad \mathsf{map}\,\mathsf{map}(\mathsf{map}\, q\,;\mathsf{red}\, app)\,;\mathsf{tri}\,\mathsf{map}\, R^p\,;\mathsf{map}\,\mathsf{tri}\, R\,;\mathsf{map}\, p\,;\mathsf{red}\, app
\end{aligned}$$

$$\begin{aligned}
&= \mathsf{tri\,map}(\mathsf{tri\,map}\,R^{p\times q} \,;\, \mathsf{map\,tri}\,R^p) \,; \\
&\quad \mathsf{map\,map}(\mathsf{map}\,q \,;\, \mathsf{red}\,app) \,;\, \mathsf{map\,tri}\,R \,;\, \mathsf{map}\,p \,;\, \mathsf{red}\,app \\
&= \mathsf{tri\,map}(\mathsf{tri\,map}\,R^{p\times q} \,;\, \mathsf{map\,tri}\,R^p) \,;\, \mathsf{map\,tri}(\mathsf{tri\,map}\,R^q \,;\, \mathsf{map\,tri}\,R) \,; \\
&\quad \mathsf{map\,map}(\mathsf{map}\,q \,;\, \mathsf{red}\,app) \,;\, \mathsf{map}\,p \,;\, \mathsf{red}\,app \\
&= \mathsf{tri\,map\,tri\,map}\,R^{p\times q} \,; \\
&\quad \mathsf{tri\,map\,map\,tri}\,R^p \,;\, \mathsf{map\,tri\,tri\,map}\,R^q \,;\, \mathsf{map\,tri\,map\,tri}\,R \,; \\
&\quad \mathsf{map\,map}(\mathsf{map}\,q \,;\, \mathsf{red}\,app) \,;\, \mathsf{map}\,p \,;\, \mathsf{red}\,app
\end{aligned}$$

This factorisation corresponds to the two changes of variables in the earlier calculation with summations. The order of the terms in $R$ does not matter because terms in $\mathsf{map}$ and $\mathsf{tri}$ with the same relation necessarily commute with each other.

Since the $R$ in question is $(\otimes n)$, this is the point to observe that some powers of $R$ are going to be cancellable, specifically that

$$\begin{aligned}
\mathsf{tri\,map\,tri\,map}(\otimes n)^n &= \mathsf{map\,map\,map\,map}(\otimes 1) \\
&= \mathsf{map\,map\,map\,map}(\times\omega)^0
\end{aligned}$$

where $(\times\omega)^0$ is the identity on the type underlying the arithmetic. That term can then be cancelled by absorbing it into any of the other three similar terms; this corresponds to the cancelling of $\omega^n$ in the calculations with summations.

### 1.9.4 Dividing the discrete Fourier transform

Suppose that $n = p \times q$. The factorisation of the $n$-point transform proceeds, as suggested above, by simplifying a specific instance of $\mathsf{map}\,q \,;\, \mathsf{red}\,app \,;\, n \,;\, \mathcal{F}$. The particular instance is chosen – with hindsight, of course – so that a term can be cancelled later.

$$\begin{aligned}
&p \,;\, \mathsf{map}\,q \,;\, \mathsf{red}\,app \,;\, \mathcal{F} \\
&\quad = \quad p \,;\, \mathsf{map}\,q \,;\, \mathsf{red}\,app \,;\, n \,;\, \mathcal{F} \\
&\quad = \{\text{ definition of } \mathcal{F} \} \\
&\qquad p \,;\, \mathsf{map}\,q \,;\, \mathsf{red}\,app \,;\, (\mathsf{red}\,join)^{-1} \,;\, n \,;\, \mathsf{tri\,tri}(\otimes n) \,;\, \mathsf{map\,red}\,add \\
&\quad = \{\text{ factorising } \mathsf{red}\,join \} \\
&\qquad p \,;\, \mathsf{map}\,q \,;\, \mathsf{red}\,app \,;\, (\mathsf{red}\,join)^{-1} \,;\, (\mathsf{map\,red}\,join)^{-1} \,; \\
&\qquad q \,;\, \mathsf{map}\,p \,;\, \mathsf{red}\,app \,;\, \mathsf{tri\,tri}(\otimes n) \,;\, \mathsf{map\,red}\,add \\
&\quad = \{\text{ promoting } \mathsf{red}\,app \text{ over joins} \} \\
&\qquad p \,;\, \mathsf{map}\,q \,;\, (\mathsf{red}\,join)^{-1} \,;\, (\mathsf{map\,red}\,join)^{-1} \,;\, q \,;\, \mathsf{map}\,p \,; \\
&\qquad \mathsf{map\,map\,red}\,app \,;\, \mathsf{red}\,app \,;\, \mathsf{tri\,tri}(\otimes n) \,;\, \mathsf{map\,red}\,add
\end{aligned}$$

$$
\begin{aligned}
&= \{\, \text{factorising } \mathsf{tri\,tri} \,\} \\
&\qquad p \,;\, \mathsf{map}\, q \,;\, (\mathsf{red}\, \mathit{join})^{-1} \,;\, (\mathsf{map\,red}\, \mathit{join})^{-1} \,; \\
&\qquad \mathsf{tri\,map\,map\,tri}(\otimes n)^{p} \,;\, \mathsf{map\,tri\,tri\,map}(\otimes n)^{q} \,;\, \mathsf{map\,tri\,map\,tri}(\otimes n) \,; \\
&\qquad q \,;\, \mathsf{map}\, p \,;\, \mathsf{map\,map\,red}\, \mathit{app} \,;\, \mathsf{red}\, \mathit{app} \,;\, \mathsf{map\,red}\, \mathit{add} \\
&= \{\, \text{promoting } \mathsf{map\,red}\, \mathit{add} \text{ over } \mathsf{red}\, \mathit{app} \,\} \\
&\qquad p \,;\, \mathsf{map}\, q \,;\, (\mathsf{red}\, \mathit{join})^{-1} \,;\, (\mathsf{map\,red}\, \mathit{join})^{-1} \,; \\
&\qquad \mathsf{tri\,map\,map\,tri}(\otimes q) \,;\, \mathsf{map\,tri\,tri\,map}(\otimes p) \,;\, \mathsf{map\,tri\,map\,tri}(\otimes n) \,; \\
&\qquad q \,;\, \mathsf{map}\, p \,;\, \mathsf{map\,map}(\mathsf{map\,red}\, \mathit{app} \,;\, \mathsf{red}\, \mathit{add}) \,;\, \mathsf{red}\, \mathit{app} \\
&= \{\, \text{associativity of } \mathit{add} \,\} \\
&\qquad p \,;\, \mathsf{map}\, q \,;\, (\mathsf{red}\, \mathit{join})^{-1} \,;\, (\mathsf{map\,red}\, \mathit{join})^{-1} \,; \\
&\qquad \mathsf{map\,tri\,tri\,map}(\otimes p) \,;\, \mathsf{map\,tri\,map\,tri}(\otimes n) \,;\, \mathsf{tri\,map\,map\,tri}(\otimes q) \,; \\
&\qquad \mathsf{map\,map}(\mathsf{map\,red}\, \mathit{add} \,;\, \mathsf{red}\, \mathit{add}) \,;\, q \,;\, \mathsf{map}\, p \,;\, \mathsf{red}\, \mathit{app}
\end{aligned}
$$

This is a watershed in the calculation: the algorithm has now been teased apart into sufficiently many sufficiently small sub-calculations that we can begin to see how it might be re-arranged into smaller instances of the same algorithm.

The strategy from this point is to use a number of facts about the arithmetic to eliminate some map operators from the expression. Firstly the arithmetic is deterministic, and $R \,;\, \mathit{join}^{-1} = \mathit{join}^{-1} \,;\, [R, R]$ for deterministic relations $R$, so $(\mathsf{red}\, \mathit{join})^{-1} ; \mathsf{map}\, R = R ; (\mathsf{red}\, \mathit{join})^{-1}$. Secondly since multiplication distributes over addition, so does $\otimes$ and $\mathsf{map}(\otimes k) \,;\, \mathsf{red}\, \mathit{add} = \mathsf{red}\, \mathit{add} \,;\, (\otimes k)$. To do this the order of some of the operators must be changed by composing both sides with a transposition.

$$
\begin{aligned}
&\mathit{trn} \,;\, p \,;\, \mathsf{map}\, q \,;\, \mathsf{red}\, \mathit{app} \,;\, \mathcal{F} \\
&\quad = \{\, \text{previous calculation} \,\} \\
&\qquad q \,;\, \mathsf{map}\, p \,;\, (\mathsf{red}\, \mathit{join})^{-1} \,;\, (\mathsf{map\,red}\, \mathit{join})^{-1} \,;\, \mathsf{map\,map}\, \mathit{trn} \,; \\
&\qquad \mathsf{map\,tri\,tri\,map}(\otimes p) \,;\, \mathsf{map\,tri\,map\,tri}(\otimes n) \,;\, \mathsf{tri\,map\,map\,tri}(\otimes q) \,; \\
&\qquad \mathsf{map\,map}(\mathsf{map\,red}\, \mathit{add} \,;\, \mathsf{red}\, \mathit{add}) \,;\, q \,;\, \mathsf{map}\, p \,;\, \mathsf{red}\, \mathit{app} \\
&\quad = \{\, \text{transposing pointwise operations} \,\} \\
&\qquad q \,;\, \mathsf{map}\, p \,;\, (\mathsf{red}\, \mathit{join})^{-1} \,;\, (\mathsf{map\,red}\, \mathit{join})^{-1} \,; \\
&\qquad \mathsf{map\,tri\,map\,tri}(\otimes p) \,;\, \mathsf{map\,tri\,tri\,map}(\otimes n) \,;\, \mathsf{tri\,map\,tri\,map}(\otimes q) \,; \\
&\qquad \mathsf{map\,map}(\mathit{trn} \,;\, \mathsf{map\,red}\, \mathit{add} \,;\, \mathsf{red}\, \mathit{add}) \,;\, q \,;\, \mathsf{map}\, p \,;\, \mathsf{red}\, \mathit{app} \\
&\quad = \{\, \text{commutativity of } \mathit{add} \,\} \\
&\qquad q \,;\, \mathsf{map}\, p \,;\, (\mathsf{red}\, \mathit{join})^{-1} \,;\, (\mathsf{map\,red}\, \mathit{join})^{-1} \,; \\
&\qquad \mathsf{map\,tri\,map\,tri}(\otimes p) \,;\, \mathsf{map\,tri\,tri\,map}(\otimes n) \,;\, \mathsf{tri\,map\,tri\,map}(\otimes q) \,; \\
&\qquad \mathsf{map\,map}(\mathsf{map\,red}\, \mathit{add} \,;\, \mathsf{red}\, \mathit{add}) \,;\, q \,;\, \mathsf{map}\, p \,;\, \mathsf{red}\, \mathit{app}
\end{aligned}
$$

$$
\begin{aligned}
&= \{\text{ distributivity of } \otimes \text{ over } add\ \} \\
&\quad q \,;\, \mathsf{map}\, p \,;\, (\mathsf{red}\, join)^{-1} \,;\, (\mathsf{map\,red}\, join)^{-1} \,;\, \mathsf{map\,tri\,map\,tri}(\otimes p) \,; \\
&\quad \mathsf{map\,map\,map\,red}\ add \,;\, \mathsf{map\,tri\,tri}(\otimes n) \,;\, \mathsf{tri\,map\,tri}(\otimes q) \,; \\
&\quad \mathsf{map\,map\,red}\ add \,;\, q \,;\, \mathsf{map}\, p \,;\, \mathsf{red}\ app \\
&= \{\text{ deterministic arithmetic }\} \\
&\quad q \,;\, \mathsf{map}\, p \,;\, (\mathsf{red}\, join)^{-1} \,;\, \mathsf{tri\,map\,tri}(\otimes p) \,; \\
&\quad \mathsf{map\,map\,red}\ add \,;\, \mathsf{tri\,tri}(\otimes n) \,;\, (\mathsf{red}\, join)^{-1} \,;\, \mathsf{tri\,map\,tri}(\otimes q) \,; \\
&\quad \mathsf{map\,map\,red}\ add \,;\, q \,;\, \mathsf{map}\, p \,;\, \mathsf{red}\ app \\
&= \{\text{ promoting } q \text{ over joins }\} \\
&\quad \mathsf{map}\, p \,;\, (\mathsf{red}\, join)^{-1} \,;\, p \,;\, \mathsf{tri\,map\,tri}(\otimes p) \,;\, \mathsf{map\,map\,red}\ add \,; \\
&\quad \mathsf{tri\,tri}(\otimes n) \,; \\
&\quad \mathsf{map}\, q \,;\, (\mathsf{red}\, join)^{-1} \,;\, q \,;\, \mathsf{tri\,map\,tri}(\otimes q) \,;\, \mathsf{map\,map\,red}\ add \,; \\
&\quad \mathsf{red}\ app
\end{aligned}
$$

There are two occurrences of similar expressions in the right-hand side, differing only in the parameter $p$ or $q$. Each of these can be shown in the same way to satisfy

$$
\begin{aligned}
&\mathsf{map}\, p \,;\, (\mathsf{red}\, join)^{-1} \,;\, p \,;\, \mathsf{tri\,map\,tri}(\otimes p) \,;\, \mathsf{map\,map\,red}\ add \\
&\quad = \quad \mathsf{map}\, p \,;\, \mathsf{map}((\mathsf{red}\, join)^{-1} \,;\, p) \,;\, trn \,;\, \mathsf{tri\,map\,tri}(\otimes p) \,;\, \mathsf{map\,map\,red}\ add \\
&\quad = \quad \mathsf{map}(p \,;\, (\mathsf{red}\, join)^{-1} \,;\, p \,;\, \mathsf{tri\,tri}(\otimes p) \,;\, \mathsf{map\,red}\ add) \,;\, trn \\
&\quad = \quad \mathsf{map}(p \,;\, \mathcal{F}) \,;\, trn
\end{aligned}
$$

so showing that

$$
\begin{aligned}
&q \,;\, \mathsf{map}\, p \,;\, trn \,;\, \mathsf{red}\ app \,;\, \mathcal{F} \\
&\qquad = \quad \mathsf{map}(p \,;\, \mathcal{F}) \,;\, trn \,;\, \mathsf{tri\,tri}(\otimes n) \,;\, \mathsf{map}(q \,;\, \mathcal{F}) \,;\, trn \,;\, \mathsf{red}\ app
\end{aligned}
$$

Now $trn$ is its own inverse, and $p \,;\, \mathsf{map}\, q \,;\, \mathsf{red}\ app$ is also a bijection on the domain of the right-hand side, so both can be carried over to the other side of the equation.

$$
\begin{aligned}
n \,;\, \mathcal{F} \quad = \quad & (\mathsf{red}\ app)^{-1} \,;\, trn \,;\, \mathsf{map}(p \,;\, \mathcal{F}) \,;\, trn \,;\, \mathsf{tri\,tri}(\otimes n) \,; \\
& \mathsf{map}(q \,;\, \mathcal{F}) \,;\, trn \,;\, \mathsf{red}\ app
\end{aligned}
$$

The remaining asymmetry in the expression is annoying, but merely superficial for of course $trn \,;\, \mathsf{tri\,tri}(\otimes n) = \mathsf{tri\,tri}(\otimes n) \,;\, trn$.

So long as both $p$ and $q$ are strictly less than $n$ we can use this decomposition as a definition of $n \,;\, \mathcal{F}$ for any composite $n$. The decomposition can be read – taking terms from left to right – as a divide-and-conquer algorithm for

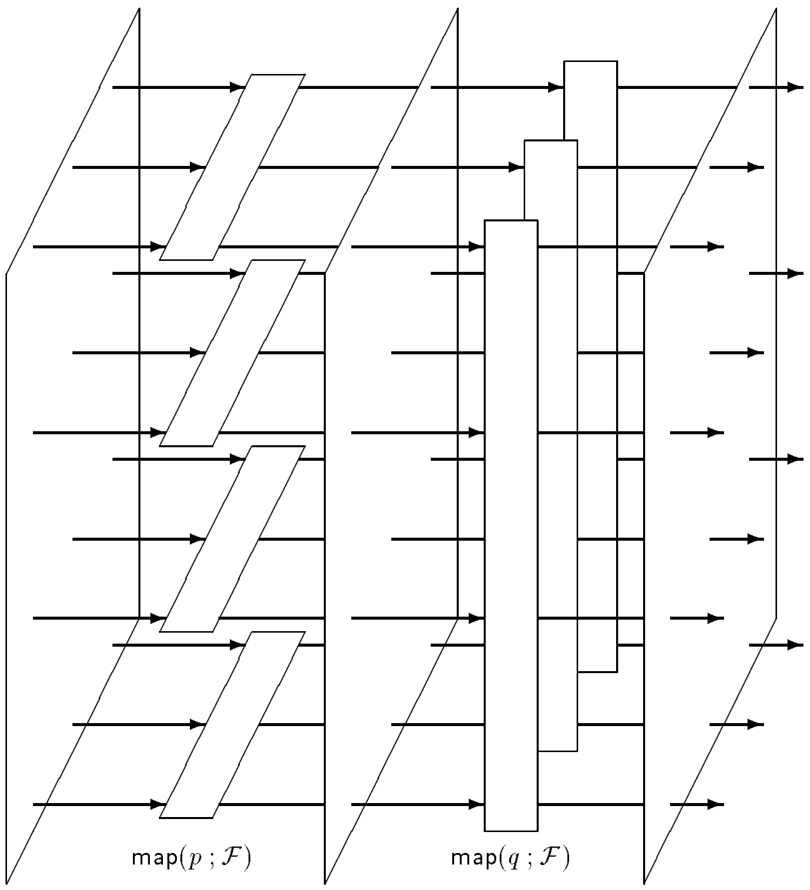



Figure 1.42: recursive decomposition of $(p \times q)\,;\mathcal{F}$

implementing transforms of composite width: divide the input into $p$ chunks of length $q$; interleave them; apply an array (of $q$) independent $p$-point transforms; interleave the results; modify by scaling the $\langle i, j\rangle$-th signal by $(\otimes n)^{i\times j}$; apply an array (of $p$) independent $q$-point transforms; interleave the results; and finally concatenate the $q$ resulting lists, each of which is of length $p$, into a single $n$-list. This is the algorithm known as the 'fast Fourier transform'. The scaling factors in $\mathsf{tri\,tri}(\otimes n)$ can of course be pre-calculated for any given $p$ and $q$ and are known as 'twiddle factors'.

### 1.9.5 Outline of an implementation

The usual recursive 'butterfly' implementation of the fast Fourier transform applies only to transforms on vectors of length $2^n$ for some $n$. This is because it is very easy to do two-point transforms: because minus one is the principal square root of unity, the two-point transform $\Phi = 2\,;\mathcal{F}$ relates $\langle x_0, x_1\rangle$ to $\langle x_0 + x_1, x_0 - x_1\rangle$ and requires no multiplications.

For higher powers of two, the factorisation used is

$$2n\,;\mathcal{F} \quad = \quad \mathit{pair}\,;\mathit{trn}\,;\mathsf{map}(n\,;\mathcal{F})\,;\mathit{trn}\,;\mathsf{tri\,tri}(\otimes 2n)\,;\mathsf{map}\,\Phi\,;\mathit{trn}\,;\mathit{halve}^{-1}$$

The only explicit multiplications in this factorisation are in the $\mathsf{tri\,tri}(\otimes 2n)$ which can be implemented by an array of $2n$ multiplications only $n-1$ of which are non-trivial. The factorisation is used recursively on the $n\,;\mathcal{F}$ term until only two-point transforms remain.

The usual way of implementing this algorithm – that is, the usual way of laying out the circuit – is to divide $\mathcal{F}$ into two homogeneous parts: let $\mathcal{F} = \mathcal{S}\,;\mathcal{F}'$ where

$$2n\,;\mathcal{F}' \quad = \quad \mathit{halve}\,;\mathsf{map}(n\,;\mathcal{F}')\,;\mathsf{tri\,tri}(\otimes 2n)\,;((\mathsf{map}\,\Phi)\setminus \mathit{trn})\,;\mathit{halve}^{-1}$$

and

$$2n\,;\mathcal{S} \quad = \quad \mathit{pair}\,;\mathit{trn}\,;\mathsf{map}(n\,;\mathcal{S})\,;\mathit{halve}^{-1}$$

Looking back to the discussion of butterfly circuits, you will see that the recursion

$$\begin{aligned} 2n\,;\mathcal{B} \quad &= \quad (\mathsf{map}(n\,;\mathcal{B})\,;(\mathsf{map}\,\Phi)\setminus \mathit{trn})\setminus \mathit{halve}^{-1} \\ &= \quad \mathsf{two}(n\,;\mathcal{B})\,;\mathsf{rpmap}\,\Phi \end{aligned}$$

has a solution $\mathcal{B} = (\bowtie \Phi^{-1})^{-1}$. This recursion is almost the same as that for $\mathcal{F}'$, and the solution need only be adjusted to accommodate the twiddle factors. Alternately, the twiddle factors might be calculated in a pre-pass using another co-located butterfly of the same shape as $\mathcal{B}$.

Similarly, the solution to the recursion for $\mathcal{S}$ is a permutation, which is a tree of transpositions. It is that very thorough shuffle which appears inscrutably in many implementations of the fast transform, and which reverses the bits of the index of the position of a value in a vector.

The derivation of the fast algorithm did not depend on a particular arithmetic: any integral domain would do. So you can use this derivation to lead into the design of a circuit in which the twiddle-factor multipliers operate on complex numbers, or on small integers with modulo arithmetic. More importantly, the chosen arithmetic can be the pointwise operations on time-sequences, and the derivation of the butterfly circuit can just as easily be used to lead to a sequential circuit.

## 1.10 References

[Bird88] R. S. Bird, *Lectures on constructive functional programming*, Programming Research Group technical monograph PRG–69, September 1988.

[Cool65] J. W. Cooley and J. W. Tukey, *An algorithm for the machine computation of complex Fourier series*, Mathematics of Computation, **19**, 1965. pp. 297–301.

[Dav89] K. Davis and J. Hughes (eds.), *Functional Programming, Glasgow 1989*, Springer Workshops in Computing, 1990.

[Jon89] G. Jones, *Deriving the fast Fourier algorithm by calculation*, in [Dav89]. (Programming Research Group technical report PRG–TR–4–89)

[Jon90] G. Jones and M. Sheeran, *Relations and refinement in circuit design*, in [Morg90].

[Leis81] C. E. Leiserson, *Area efficient VLSI computation*, Ph.D. thesis, Carnegie-Mellon University, Pittsburg, Pennsylvania, October 1981.

[McCab82] M. M. McCabe, A. P. H. McCabe, B. Abrambepola, I. N. Robinson and A. G. Cory, *New algorithms and architectures for VLSI*, GEC Journal of Science & Technology, Vol. 48, No. 2, 1982. pp. 68–75.

[Miln88] G. J. Milne (ed.), *The fusion of hardware design and verification*, North-Holland, 1988.

[Morg90] C. C. Morgan (ed.), *Proc BCS FACS workshop on Refinement*, Hursley, January 1990. (To appear from Springer 1990)

[Sheer88] M. Sheeran, *Retiming and slowdown in Ruby*, in [Miln88]. pp. 289–308.