Published in *Integration, the VLSI Journal*, Vol. 14, No. 3, February 1993, pp. 333–360.

Systematic serialisation of array-based architectures

W.W.C. Luk
Programming Research Group
Oxford University Computing Laboratory
11 Keble Road, Oxford, England OX1 3QD

Abstract

This paper describes the use of Ruby, a language of functions and relations, to develop serialised implementations of array-based architectures. Our Ruby expressions contain parameters which can be varied to produce a wide range of designs with different space-time trade-offs. Such expressions can be obtained by applying correctness-preserving transformations to an initial simple description. This approach provides a unified treatment of serialisation schemes similar to LPGS (Locally Parallel Globally Sequential) and LSGP (Locally Sequential Globally Parallel) partitioning methods, and will be illustrated by the development of a variety of circuits for convolution.

Keywords: Ruby, parametrised design, serialisation, correctness-preserving transformations, systolic arrays.

1 Introduction

An attraction of array-based architectures, such as systolic networks, is the opportunity for customising them to cater for a specific application. One way of achieving customisation is to start from a clear but perhaps inefficient algorithmic description, and apply successive correctness-preserving transformations until the result meets some given constraints such as size or performance requirements.

Serialisation, which is related to algorithm partitioning [6], is an important example of such transformations; it enables the behaviour of a large array of processors to be emulated by fewer processors supported by auxiliary components such as buffers and multiplexers. One may serialise a design so that, for instance, the resulting device has a given size – usually at the expense of computation time.

Recently various methods (see for example [2], [5], [14], [15], [16], [17], [18]) have been proposed for developing serialised architectures. Most of these methods involve studying optimisation techniques applicable to semantic models of different forms of recurrence equations. This paper, on the other hand, adopts an algebraic framework based on the Ruby language [4], which supports a style of development by pattern matching and equational reasoning; similar frameworks have been advocated for software development [1].

As an example of our approach, consider transforming a one-dimensional systolic array of n processors, R^n, into its serialised version, $\mathcal{S}_n\, R$. We are interested in equations of the form

$$R^n \quad = \quad A \,;\, \mathcal{S}_n\, R \,;\, B, \qquad (1)$$

which can be used as rewrite rules to transform expressions that match one side of the equation. The game is therefore to find appropriate definitions for the function $\mathcal{S}_n$ which returns a serialised version of a given component, and for the relations A and B which capture the interface constraints so that the expressions on the two sides of the above equation possess the same behavioural interpretation. We give an example in the form of Equation 1 in Section 3.3. A design derived using the proposed method can be found in Figure 11, which contains a serialised version of the convolver shown in Figure 10 (the block labelled $\underline{cmx}_M$ in Figure 11 corresponds to a multiplexer which selects either the external input signals or the feedback signals. The definition of $\underline{cmx}$ is given in Section 3.3).

Our work can be considered to be an extension of the study reported in Reference [5], where Ruby is used in deriving a bit-serial adder. Besides suggesting additional transformations, such as those embodied in Equation 1, we hope to achieve three more objectives. The first is to discover abstractions which capture serialised circuits and the associated interface constraints so that the resulting representations are sufficiently detailed while remaining tractable. It turns out that all such abstractions can be defined using existing primitives in Ruby; in particular, we have been able to express our serialisation facilities (such as $\underline{ev}$ and $\underline{cmx}$ defined in Section 3.3) in terms of a single primitive known as $\underline{bundle}$ [5]. Having a minimum number of basic primitives allows us to keep our mathematical framework simple.

Our second objective is to develop descriptions containing parameters which can be varied to produce a wide range of designs with different space-time trade-offs. It is sometimes believed that rigorous derivations in the style of this paper, while undoubtedly increasing confidence in the correctness of the resulting designs, will always remain a time-consuming business; the benefits of re-using such intellectual efforts are clear. A specific strength of our approach is the capability of deriving a parametrised expression which corresponds to an entire family of architectures. The quantitative aspects of such generic representations can be summarised in a *feature table*, an example of which can be found in Table 1 in Section 4.5. Notice that there is no need to understand our language in order to use the result of our derivation: given the performance characteristics of the available computational elements such as adders and multipliers, a designer can simply substitute the appropriate data in the feature table to work out the smallest circuit of a particular speed, or the fastest circuit of a particular size. Indeed, to encourage re-use of previous design experience, feature tables can be captured in a computer-based tool to facilitate exploring designs.

The third and final objective of our work is to formulate procedures to guide the development of such parametrised design descriptions. In addition to generating designs which are correct by construction, such procedures offer a means of documenting design decisions by recording the steps in the development of appropriate implementations. Possible evolution routes can be summarised in a *design tree* which provides a simple method of structuring the design space, indicating how different designs are related. The root of a design tree corresponds to a clear but perhaps abstract or inefficient design, and the leaves correspond to complex designs with acceptable efficiency and in sufficient detail for manufacturing; the branches are labelled with the transformations used in deriving one or more designs from another. An example of such a tree can be found in Figure 18 in Section 4.5. We expect our language, together with the associated transformations, feature tables and design trees, to become a useful complement to other methodologies for synthesising serialised representations of array-based architectures.

The rest of the paper is structured as follows. Section 2 reviews the basic elements of our design framework. Section 3 then introduces our approach for representing serialised designs and the associated strategies for their development; in particular a key equation for serialisation, Equation 24, is derived in Section 3.3. It is followed by Section 4 which illustrates the techniques using a variety of convolver circuits. Some concluding remarks are presented in Section 5.

2 Design representation

The formalism that we use is based on Ruby, a language for describing and reasoning about circuit designs. The background and the details of this approach have been described elsewhere (see [4], [5], [10], [11]), and only the definitions and concepts relevant to our discussion will be introduced here.

A design will be represented by a binary relation of the form $x\ R\ y$ where x, y represent the interface signals and belong respectively to the domain and range of R. For instance, a squaring operation can be described by

$$x\ sqr\ y \quad \Leftrightarrow \quad x^2 = y$$

or, more succinctly, by

$$x\ sqr\ x^2.$$

Transformed or composite circuits are described by functions which take one or more relations representing circuits as arguments and return a relation as result. As an example, the converse of R is defined by

$$x\ R^{-1}\ y \quad \Leftrightarrow \quad y\ R\ x.$$

It can be considered as a reflected version of R.

2.1 Binary compositions

Two components Q and R can be connected together if they share a compatible interface s which is hidden in the composite circuit $Q;R$ (Figure 1a),

$$x\,(Q;R)\,y \quad \Leftrightarrow \quad \exists s.\,(x\ Q\ s)\ \&\ (s\ R\ y). \tag{2}$$

This is, of course, just the common definition of relational composition. It is simple to show that relational composition is associative, and that $(Q;R)^{-1} = R^{-1};Q^{-1}$.

As shown later, many theorems in this framework can be expressed in the form $R = P^{-1};Q;P$. The pattern $P^{-1};Q;P$ – in words 'Q conjugated by P' – will be abbreviated to $Q\backslash P$.

If there are no connections between Q and R, the composite is represented by parallel composition $[Q, R]$ (Figure 1b), where

$$\langle x_0, x_1\rangle\,[Q,R]\,\langle y_0, y_1\rangle \quad \Leftrightarrow \quad (x_0\ Q\ y_0)\ \&\ (x_1\ R\ y_1). \tag{3}$$

One can check quickly that $[P,Q];[R,S] = [P;R,\ Q;S]$, and that $[P,Q]^{-1} = [P^{-1}, Q^{-1}]$. A collection of such theorems constitutes a calculus for reasoning about designs, which can usually be used without the need to refer to the meaning of the symbols such as P and Q.

There are several operations involving pairs of signals that we will require. First of all, given that ι is the identity relation, we have the abbreviations

$$\begin{aligned}\mathsf{fst}\,R &= [R,\iota],\\ \mathsf{snd}\,R &= [\iota,R].\end{aligned}$$

Next, the relation *fork* can be used to duplicate a signal, since $x\ \mathit{fork}\ \langle x,x\rangle$. Extracting an element from a pair is achieved by the projection relations π_1 and π_2, defined by $\langle x,y\rangle\ \pi_1\ x$

and $\langle x, y\rangle\ \pi_2\ y$. Finally, we need to be able to swap the elements of a pair: $\langle x, y\rangle\ swap\ \langle y, x\rangle$. Examples of theorems involving these operations include

$$\begin{array}{rclcl} \mathsf{fst}\,Q\ ;\ \mathsf{snd}\,R & = & \mathsf{snd}\,R\ ;\ \mathsf{fst}\,Q & = & [Q,R], \\ fork\ ;\ \pi_1 & = & fork\ ;\ \pi_2 & = & \iota, \\ [Q,R]\backslash swap & = & swap\ ;\ [Q,R]\ ;\ swap & = & [R,Q]. \end{array}$$

It should also be clear that $fork;[\pi_1,\pi_2] = [\iota,\iota]$, and that ${\pi_1}^{-1};\pi_1 = \iota \neq \pi_1;{\pi_1}^{-1}$ and similarly for π_2.

Tuples are ordered collections of elements, and they are appended by the operator '$\hat{}$' (binary append), such that $\langle a,b,c\rangle\hat{}\langle d,e\rangle = \langle a,b,c,d,e\rangle$. The relations *apl* (append left) and *apr* (append right) are given by $\langle x, xs\rangle\ apl\ \langle x\rangle\hat{}xs$ and $\langle xs, x\rangle\ apr\ xs\hat{}\langle x\rangle$. It is easy to show that $[Q,[R,S]]\ \backslash\ apl = [[Q,R],S]\ \backslash\ apr = [Q,R,S]$.

A rectangular component with connections on every side is modelled by a relation that relates 2-tuples, with the two components in the domain corresponding to signals for the west and north side and those in the range corresponding to signals for the south and east side. Such components can be assembled together by the beside ($\leftrightarrow$) and below ($\updownarrow$) operators (Figure 1c and Figure 1d):

$$\begin{array}{rcl} \langle a,\langle b,c\rangle\rangle\,(Q\leftrightarrow R)\,\langle\langle p,q\rangle,r\rangle & \Leftrightarrow & \exists s.\ \langle a,b\rangle\ Q\ \langle p,s\rangle\ \&\ \langle s,c\rangle\ R\ \langle q,r\rangle, \\ \langle\langle a,b\rangle,c\rangle\,(Q\updownarrow R)\,\langle p,\langle q,r\rangle\rangle & \Leftrightarrow & \exists s.\ \langle a,s\rangle\ Q\ \langle p,q\rangle\ \&\ \langle b,c\rangle\ R\ \langle s,r\rangle. \end{array}$$

Since $(Q\updownarrow R)^{-1} = Q^{-1}\leftrightarrow R^{-1}$, theorems that have been proved for beside can readily be adapted for below.

It is also useful to have a conjugate operator for pairs:

$$Q\backslash\!\backslash[R,S] \quad = \quad [S^{-1},R^{-1}];Q;[R,S].$$

Given that the conjugate operators have a lower precedence than all other operators except relational composition, one can show that $Q\backslash\!\backslash R = R^{-1}\backslash swap\ ;\ Q\ ;\ R$, and that $\mathsf{snd}Q^{-1};R;\mathsf{fst}Q = R\backslash\!\backslash(\mathsf{fst}Q)$.

2.2 Repeated compositions

Let us now look at the ways that we describe one- and two-dimensional arrays of components. Repeated relational composition of a given relation R forms a *chain* (Figure 2a) which contains cascaded copies of R; it is defined inductively by

$$R^1 \quad = \quad R, \tag{4}$$
$$R^{n+1} \quad = \quad R^n\ ;\ R. \tag{5}$$

Given that $Q;R = R;Q$, one can use induction on n to prove that $Q;R^n = R^n;Q$, and, from that, a distributive theorem for chains:

$$(Q\ ;\ R)^n \quad = \quad Q^n\ ;\ R^n. \tag{6}$$

Simple examples of such Q and R are components which calculate the power of numbers, like $sqr;cube = cube;sqr$. This theorem will also be useful for pipelining circuits as we shall see later.

Repeated parallel composition, $\mathsf{map}\,R$ (Figure 2b), relates two equal-length tuples given that the corresponding elements of the tuples are related by R:

$$\text{if } \#x = \#y = N \quad \text{then} \quad x\ (\mathsf{map}\,R)\ y \quad\Leftrightarrow\quad \forall i : 0 \le i < N.\ x_i\ R\ y_i.$$

For clarity, sometimes we shall make explicit the number of R's in a map and write it as $\mathsf{map}_N\, R$. This expression can be considered to be an abbreviation of $\mathsf{map}\, R \setminus N$ where N is the identity relation on N-tuples.

We shall also need the relation $\triangle R$ (Figure 2c) which relates two equal-length tuples such that their i-th elements relate to each other according to R^i:

$$\text{if } \#x = \#y = N \quad \text{then} \quad x\,(\triangle\, R)\, y \quad \Leftrightarrow \quad \forall i : 0 \leq i < N.\, x_i\, R^i\, y_i.$$

The $\triangle$ function will be useful in formulating distributive theorems for a *row* of components (Figure 2d). A row of components is built from repeated composition of beside, and can be described by

$$\begin{array}{l} \text{if } \#x = \#y = N \quad \text{then} \\ \langle a, x\rangle\,(\mathsf{row}\, R)\,\langle y, b\rangle \quad \Leftrightarrow \quad \exists s.\,(s_0 = a) \;\&\; (s_N = b) \;\&\; \forall i : 0 \leq i < N.\, \langle s_i, x_i\rangle\, R\, \langle y_i, s_{i+1}\rangle. \end{array}$$

The properties of columns of components (Figure 2e) can be obtained by exploiting the fact that $\mathsf{col}\, R \;=\; (\mathsf{row}\, R^{-1})^{-1}$.

There is a distributive law for a row of components which is similar to the law given by Equation 6 for a chain of components: on the assumption that $\mathsf{snd}\, A; R; [B, B] = \mathsf{fst}\, B; R$, one can show that

$$\mathsf{row}\,(R\;;\;\mathsf{snd} B) \quad = \quad \mathsf{snd}\,\triangle A\;;\;\mathsf{row}\, R\;;\;\triangle B \setminus apr^{-1}. \tag{7}$$

On some occasions we shall need to interleave an array of components from two equal-length tuples. This can be achieved by *zip*, defined by

$$\text{if } \#x = \#y = \#z = N \quad \text{then} \quad \langle x, y\rangle\; zip\; z \quad \Leftrightarrow \quad \forall i : 0 \leq i < N.\, \langle x_i, y_i\rangle = z_i,$$

so that, for instance, $\langle\langle 1,2,3\rangle, \langle 4,5,6\rangle\rangle\; zip\; \langle\langle 1,4\rangle, \langle 2,5\rangle, \langle 3,6\rangle\rangle$. Some simple examples of zipping arrays together include (Figure 3):

$$\begin{array}{rcl} [\mathsf{map}\, Q,\, \mathsf{map}\, R] & = & \mathsf{map}\,[Q, R] \setminus zip^{-1}, \\ [\mathsf{row}\, Q,\, \mathsf{row}\, R] \setminus zip^{-1} & = & \mathsf{row}\,([Q, R]\setminus zip^{-1}) \setminus\!\!\setminus \mathsf{fst}\, zip^{-1}. \end{array}$$

These transformations, sometimes described as transposition, are studied in greater detail in Reference [9].

3 A framework for serialisation

3.1 Streams and delays

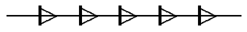
So far we have been using relations to model a static situation – the steady state behaviour of a circuit at a particular instant of time. To deal with sequential circuits, an expression is interpreted as a relation that relates a *stream* in its domain to a stream in its range. For our purpose, a stream can be considered to be a doubly-infinite tuple containing data at successive clock 'ticks'. Notice that the clock is an abstract means for specifying data synchronisation, and it may be realised either by a global synchronous clock or by some hand-shaking mechanism.

We shall use x_t to denote the t-th element from some reference point – such as the time when the circuit is initialised – in the stream x; given that x_t is a tuple, $x_{t,i}$ is its i-th element. An adder can then be described in the stream model as

$$x\; add\; y \quad \Leftrightarrow \quad \forall t.\, x_{t,0} + x_{t,1} \;=\; y_t.$$

For simplicity combinational circuits will be described in the static form. Such expressions should be interpreted as relations on streams in composite expressions involving sequential elements. In most cases, such as in the absence of conditionals, the same algebraic theorems can be applied to expressions representing either combinational or sequential systems; so, where appropriate, designs can be developed in the simpler static framework and then promoted to the sequential language. Readers interested in the formal aspects of the language – such as the description of an appropriate type discipline – may wish to consult References [3] and [4].

There are only two primitives that do not possess a static interpretation. The first primitive is *delay*, $\mathcal{D}$, which is defined by $x\,\mathcal{D}\,y \Leftrightarrow \forall t.\, x_{t-1} = y_t$. An *anti-delay* $\mathcal{D}^{-1}$ is such that $\mathcal{D};\mathcal{D}^{-1} = \mathcal{D}^{-1};\mathcal{D} = \iota$. A latch is modelled by a delay with data flowing from domain to range, or by an anti-delay with data flowing from range to domain. We shall use the symbols —▷— and ▽ to represent delays for horizontal and vertical dataflows respectively, so for instance



is a picture of $\mathcal{D}^5$. Similarly —◁— and △ represent anti-delays for horizontal and vertical dataflows.

For a circuit C which contains no primitives that possess a measure of absolute time, it is the case that $\mathcal{D};C = C;\mathcal{D}$. This property, called timelessness [3], is precisely the condition required for Equations 6 and 7 to be valid; and the motivation for applying these equations as rewrite rules is to distribute latches among the array of components in order to reduce the longest combinational path. This process is usually called *retiming* [7], and examples of deriving pipelined circuits based on an algebraic treatment of retiming can be found elsewhere [4], [11].

3.2 Circuits with feedback

Latches are also used in serial circuits to prevent unbuffered loops in feedback paths. A design R containing an internal feedback path s can be modelled by the loop construct σ [10]:

$$x\;(\sigma R)\;y \quad\Leftrightarrow\quad \exists s.\,\langle x,s\rangle\;R\;\langle s,y\rangle. \tag{8}$$

The picture in Figure 4a gives a possible geometrical interpretation of the above equation: for instance, we could have drawn the feedback wire to the left of R.

Given that $rsh = \iota{\leftrightarrow}\iota$ (right shift) and $lsh = \iota{\updownarrow}\iota$ (left shift) such that $rsh^{-1} = lsh$ [4], we can define

$$\sigma R \quad=\quad {\pi_1}^{-1}\;;\;(R \leftrightarrow swap) \mathbin{\backslash\!\backslash} (\mathsf{fst}\,fork^{-1})\;;\;\pi_2$$

Another feedback configuration (Figure 4b) may sometimes prove to be convenient for rectangular components with six connections:

$$\langle x,u\rangle\;(\nu R)\;\langle y,v\rangle \quad\Leftrightarrow\quad \exists s.\,\langle\langle x,s\rangle,u\rangle\;R\;\langle y,\langle v,s\rangle\rangle, \tag{9}$$

and this can be deduced from the definition

$$\nu R \quad=\quad \sigma\,(\mathsf{snd}\,swap \setminus rsh\;;\;R\;;\;rsh\;;\;swap).$$

A simple law involving both σ and ν is $\nu\,(Q{\updownarrow}R) = \mathsf{snd}\,\sigma\,(R\backslash swap);Q$.

These functions obey a number of laws; for instance one or more pieces of wires, represented by ι or by $[\iota,\iota]$, can be bent in various ways to form a loop,

$$\sigma\,[\iota,\iota] \quad=\quad \iota, \tag{10}$$
$$\nu\,lsh \quad=\quad [\iota,\iota]. \tag{11}$$

Components not on the feedback path can be expressed outside the loop constructs,

$$\sigma\,(\mathsf{fst}\,Q;R;\mathsf{snd}\,S) \;=\; Q\;;\;\sigma R\;;\;S, \tag{12}$$

$$\nu\,([\mathsf{fst}\,P,\,Q];R;[S,\,\mathsf{fst}\,T]) \;=\; [P,Q]\;;\;\nu R\;;\;[S,\,T], \tag{13}$$

while components on the feedback path can be moved from one end of the path to the other end,

$$\sigma\,(\mathsf{snd}\,Q\;;\;R) \;=\; \sigma(R\;;\;\mathsf{fst}\,Q), \tag{14}$$

$$\nu\,(\mathsf{fst}\,(\mathsf{snd}\,Q)\;;\;R) \;=\; \nu(R\;;\;\mathsf{snd}\,(\mathsf{snd}\,Q)). \tag{15}$$

Two feedback loops can be composed in series and in parallel,

$$\begin{aligned} \sigma Q\;;\;\sigma R &= \sigma\,(Q \leftrightarrow R), \\ [\sigma Q,\,\sigma R] &= \sigma\,([Q,R]\setminus zip^{-1}), \end{aligned}$$

or placed beside and below each other,

$$\begin{aligned} \nu Q \leftrightarrow \nu R &= \nu\,(\nu\,([Q,R]\setminus zip^{-1} \mathbin{\backslash\!\backslash} \mathsf{snd}\,zip^{-1})\;;\;\mathsf{snd}\,swap), \\ \nu Q \updownarrow \nu R &= \nu\,(Q \updownarrow R \mathbin{\backslash\!\backslash} \mathsf{snd}\,zip^{-1}). \end{aligned}$$

From Equations 2 and 8, it is clear that

$$Q\;;\;R \;=\; \sigma\,[Q,R], \tag{16}$$

which suggests that $Q \leftrightarrow R \;=\; \nu\,([Q,R]\setminus zip^{-1}\;;\;\mathsf{snd}\,swap)$. These equations lead to the following ways of expressing a chain and a row using the loop functions,

$$R^n \;=\; \sigma\,(apl\;;\;\mathsf{map}_n\,R\;;\;apr^{-1}), \tag{17}$$

$$\mathsf{row}_n\,R \;=\; \nu\,(\mathsf{fst}\,apl\;;\;\mathsf{map}_n\,R\setminus zip^{-1}\;;\;\mathsf{snd}\,(apr^{-1};swap)). \tag{18}$$

The expressions on the right-hand side of the above equations are shown in Figures 5 and 6. In the forthcoming paragraphs, we shall show how the kernel expressions in these equations, $\mathsf{map}_n\,R$ and $\mathsf{map}_n\,R\setminus zip^{-1}$, can be replaced by expressions involving a single copy of R after serialisation.

3.3 Facilities for serialisation

There are four constructs in our framework to deal with serialisation: the relations $\underline{bundle}$, $\underline{ev}$, and $\underline{cmx}$, and the function slow. The basic construct, $\underline{bundle}$ [5], can be used to define the other three constructs.

One can regard $\underline{bundle}$ as a component for converting between serial and parallel data,

$$x\;\underline{bundle}_n\;y \;\Leftrightarrow\; \forall t,i : 0 \le i < n.\, x_{nt+i} = y_{t,i}.$$

Here each y_t is an n-tuple consisting of successive elements of x, and hence x can be considered to be proceeding n times as quickly as y. On the other hand, since a stream of n-tuples may be decomposed into n streams, one may also consider x as a time-multiplexed version of the n streams in y. Hence $\underline{bundle}$ can be used to reduce the number of physical channels for a circuit by time-domain multiplexing.

It is clear that $\underline{bundle}_n;\underline{bundle}_n^{-1} \;=\; \iota$ and $\underline{bundle}_n^{-1};\underline{bundle}_n \;=\; \mathsf{map}_n\,\iota$. Note that $\underline{bundle}$ is not timeless, since $\mathcal{D}^n;\underline{bundle}_n \;=\; \underline{bundle}_n;\mathcal{D}$. It is the other primitive, besides $\mathcal{D}$, that does not possess a static interpretation.

An interesting observation is that $[Q, R] \setminus \underline{bundle}_2^{-1}$ can be interpreted to be a dynamically reconfigurable circuit that behaves like Q and like R on alternate clock cycles. A similar representation [4] has been used to describe *slowdown* [7], a method for characterising systems that may accommodate data for several independent computations. An n-slow version of a circuit, say R, can be obtained by replacing every delay in R by n delays in series, and similarly for anti-delays. It can be defined by

$$\mathsf{slow}_n\, R \;=\; (\mathsf{map}\, R) \setminus \underline{bundle}_n^{-1}.$$

One can show that if R represents a combinational circuit, then $\mathsf{slow}_n\, R = R$, and that $\mathsf{slow}_n\, \mathcal{D} = \mathcal{D}^n$. Furthermore, slow_n distributes through operators such as row, so $\mathsf{slow}_n\, (\mathsf{row}\, R) = \mathsf{row}\, (\mathsf{slow}_n\, R)$.

Another useful component is one that relates a signal and its sampled version. It is denoted by $\underline{ev}_n$ and pronounced 'every n-th', and it involves a range stream consisting of every n-th element of its domain stream:

$$\langle \cdots, x_0, x_1, x_2, x_3, x_4, \cdots \rangle \; \underline{ev}_3 \; \langle \cdots, x_0, x_3, x_6, x_9, x_{12}, \cdots \rangle.$$

We can define it using $\underline{bundle}$,

$$\underline{ev}_n \;=\; \underline{bundle}_n \;;\; apl^{-1} \;;\; \pi_1,$$

so that

$$x \; \underline{ev}_n \; y \;\Leftrightarrow\; \forall t.\, x_{nt} = y_t.$$

The relation $\underline{ev}_n$ shares with $\mathcal{D}$ the property that it can be applied to streams of any structure. For example, $\underline{ev}_n;[Q, R] = [\underline{ev}_n, \underline{ev}_n];[Q, R]$. However, while $\underline{ev}_n^{-1};\underline{ev}_n = \iota$, it is the case that $x\, (\underline{ev}_n;\underline{ev}_n^{-1})\, y \Leftrightarrow \forall t.\, y_{nt} = x_{nt}$. That is, only every n-th element of x and of y are required to be equal.

Assuming that R satisfies $\mathsf{map}\, R; apl^{-1}; \pi_1 = apl^{-1}; \pi_1; R$, one can show that

$$\underline{ev}_n \;;\; R \;=\; \mathsf{slow}_n\, R \;;\; \underline{ev}_n.$$

That is, the effect of sampling the domain signal of R is identical to that of sampling the range signal of $\mathsf{slow}_n\, R$. Note also that $\mathcal{D}^n;\underline{ev}_n = \underline{ev}_n;\mathcal{D}$, and that

$$\underline{bundle}_2 \;=\; fork \;;\; [\underline{ev}_2,\, \underline{ev}_2 \setminus \mathcal{D}^{-1}]. \tag{19}$$

Finally, we need a *cycling multiplexer* $\underline{cmx}_n$ which repeatedly identifies its range signal with the first of its two domain signals for one cycle and then with the other domain signal for $n-1$ cycles; that is, it satisfies

$$x \; \underline{cmx}_n \; y \;\Leftrightarrow\; \forall t.\, (x_{nt,0} = y_{nt}) \;\&\; (\forall i : 1 \le i < n.\, x_{nt+i,1} = y_{nt+i}).$$

This component can be defined by

$$\underline{cmx}_n \;=\; \mathsf{map}\, (\underline{bundle}_n \;;\; apl^{-1}) \;;\; [\pi_1,\, \pi_2] \;;\; apl \;;\; \underline{bundle}_n^{-1}.$$

For $n = 2$, one can use $\mathcal{D}^{-1};\underline{bundle}_2;\pi_1 = \underline{bundle}_2;\pi_2$ and $\pi_1{}^{-1} \;;\; \mathcal{D} \setminus \underline{bundle}_2 \;;\; \pi_2 = \iota$ to show that

$$\begin{aligned} \underline{cmx}_2 \;&=\; [\underline{ev}_2,\, \mathcal{D}^{-1};\underline{ev}_2] \;;\; \underline{bundle}^{-1} & (20) \\ &=\; \mathsf{snd}\, (\mathcal{D}^{-1};\underline{ev}_2;\underline{ev}_2^{-1};\mathcal{D}) \;;\; \underline{cmx}_2. & (21) \end{aligned}$$

For $n > 1$, we can prove that $\mathcal{D}^n ; \underline{cmx}_n \;=\; \underline{cmx}_n ; \mathcal{D}^n$ and that

$$\begin{aligned} \mathit{fork} \;;\; \underline{cmx}_n \;&=\; \iota, &(22)\\ \mathsf{fst}\,(\underline{ev}_n ; \underline{ev}_n^{-1}) \;;\; \underline{cmx}_n \;&=\; \underline{cmx}_n. &(23)\end{aligned}$$

Given the above definitions, we can now proceed to develop serialisation theorems for our operators. The intuitive idea is to circulate data through a processor n times to emulate the effect of n cascaded processors, using $\underline{cmx}_n$ to control when to accept feedback data and $\underline{ev}_n^{-1}$ and $\underline{ev}_n$ to inject and to reject dummy data when the processor is in feedback mode. Let us first derive a serialisation theorem for binary composition (in the following derivation, curly brackets are used to enclose hints explaining why the expressions above and below them are equal):

$$\begin{aligned} & Q \;;\; R \\ = \quad & \{\text{Equation 16 and } \underline{bundle}_2^{-1} ; \underline{bundle}_2 \;=\; [\iota, \iota]\} \\ & \sigma\,(\underline{bundle}_2^{-1} \;;\; \underline{bundle}_2 \;;\; [Q, R]) \\ = \quad & \{\text{Equations 20 and 12}\} \\ & \underline{ev}_2^{-1} \;;\; \sigma\,(\mathsf{snd}\,(\underline{ev}_2^{-1} ; \mathcal{D}) \;;\; \underline{cmx}_2 \;;\; \underline{bundle}_2 \;;\; [Q, R]) \\ = \quad & \{\underline{bundle}_2^{-1} ; \underline{bundle}_2 \;=\; [\iota, \iota], \text{ Equation 19, } P^{-1} ; Q ; P \;=\; Q \backslash P \text{ and } \mathcal{D} ; \mathcal{D}^{-1} \;=\; \iota\} \\ & \underline{ev}_2^{-1} \;;\; \sigma\,(\mathsf{snd}\,(\underline{ev}_2^{-1} ; \mathcal{D}) \;;\; \underline{cmx}_2 \;;\; [Q, R] \,\backslash\, \underline{bundle}_2^{-1} \;;\; \mathit{fork} \;;\; [\mathcal{D} ; \mathcal{D}^{-1} ; \underline{ev}_2,\, \mathcal{D} ; \underline{ev}_2 ; \mathcal{D}^{-1}]) \\ = \quad & \{\mathit{fork} ; [\mathcal{D}, \mathcal{D}] \;=\; \mathcal{D} ; \mathit{fork} \text{ and Equation 14}\} \\ & \underline{ev}_2^{-1} \;;\; \sigma\,(\mathsf{snd}\,(\mathcal{D}^{-1} ; \underline{ev}_2 ; \underline{ev}_2^{-1} ; \mathcal{D}) \;;\; \underline{cmx}_2 \;;\; [Q, R] \,\backslash\, \underline{bundle}_2^{-1} \;;\; \mathcal{D} \;;\; \mathit{fork} \;;\; \mathsf{snd}\,(\underline{ev}_2 ; \mathcal{D}^{-1})) \\ = \quad & \{\text{Equations 12 and 21}\} \\ & \underline{ev}_2^{-1} \;;\; \sigma\,(\underline{cmx}_2 \;;\; [Q, R] \,\backslash\, \underline{bundle}_2^{-1} \;;\; \mathcal{D} \;;\; \mathit{fork}) \;;\; \underline{ev}_2 \;;\; \mathcal{D}^{-1}. \qquad (24) \end{aligned}$$

Note that from the implementation point of view, in the right-hand side expression of Equation 24, the presence of $\mathcal{D}$ in the loop body constrains $[Q, R] \backslash \underline{bundle}_2^{-1}$ to be a function: inputs in its domain and outputs in its range.

Equation 24 suggests a similar theorem for serialising a chain of components (Figure 7):

$$\begin{aligned} R^n \;&=\; \underline{ev}_n^{-1} \;;\; \sigma\,(\underline{cmx}_n \;;\; \mathsf{map}\, R \,\backslash\, \underline{bundle}_n^{-1} \;;\; \mathcal{D} \;;\; \mathit{fork}) \;;\; \underline{ev}_n \;;\; \mathcal{D}^{-1} \\ &=\; \sigma\,(\underline{cmx}_n \;;\; \mathsf{slow}_n\, R \;;\; \mathcal{D} \;;\; \mathit{fork}) \,\backslash\, \underline{ev}_n \;;\; \mathcal{D}^{-1}, \qquad (25) \end{aligned}$$

which corresponds to Equation 1 with $\mathcal{S}_n\, R \;=\; \sigma\,(\underline{cmx}_n \;;\; \mathsf{slow}_n\, R \;;\; \mathcal{D} \;;\; \mathit{fork})$, $A \;=\; \underline{ev}_n^{-1}$, and $B \;=\; \underline{ev}_n \;;\; \mathcal{D}^{-1}$.

A similar theorem for a row of components is

$$\mathsf{row}_n\, R \;=\; \nu\,(\mathsf{fst}\, \underline{cmx}_n \;;\; \mathsf{slow}_n\, R \;;\; \mathsf{snd}\,(\mathcal{D} ; \mathit{fork})) \,\backslash\!\backslash\, [\underline{bundle}_n, \underline{ev}_n] \;;\; \mathsf{snd}\, \mathcal{D}^{-1} \qquad (26)$$

(Figure 8). One can develop a simpler version of this equation as follows. Define $\mu R \;=\; \sigma\,(R \backslash \mathit{swap})$. Given that $R \;=\; R \backslash\!\backslash \mathsf{snd}\,[\iota, \iota]$, it can be shown that $\mu\,(\nu R) \;=\; \mu R$. This result, together with Equation 22, can then be used to show that

$$\mu\,(\mathsf{fst}\, \mathcal{D} \;;\; \mathsf{slow}_n\, R) \,\backslash\, \underline{bundle}_n \;=\; \mu\,(\mathsf{fst}\, \mathcal{D} \;;\; \mathsf{row}_n\, R).$$

This equation has been used in deriving a bit-serial adder [5].

The corresponding theorems for a column of components can be obtained by appealing to the fact that $\mathsf{row}\, R \;=\; (\mathsf{col}\, R^{-1})^{-1}$. A rectangular array of processors R can be captured by

$\mathsf{row}\,(\mathsf{col}\,R)$. It can be serialised both vertically and horizontally by serialising first in one direction, then in the other.

It is also possible to extend the serialisation theorems to cover a network of heterogeneous components [9]; for instance Equation 24 is easily generalised to deal with a chain of dissimilar processors.

3.4 Strategies for serialisation

There are two common strategies for serialisation [6]. The first strategy is known as cut-and-pile [16] or LPGS (Locally Parallel Globally Sequential) strategy. In this scheme, a design is divided into blocks of processors which will then be serialised to become a single block of processors.

The first step for systematic serialisation corresponding to the LPGS scheme is to divide an array of components into smaller subarrays. This transformation, also called clustering, involves expressing a chain as a chain of chains,

$$R^{mn} \quad = \quad (R^m)^n .$$

For rows, we need the relation $group_n$ which relates the elements of an $(m \times n)$-tuple and those of an n-tuple of m-tuples:

$$\text{if } \#x = mn \quad \text{then} \quad \langle x_i \,|\, 0 \le i < mn \rangle \quad group_n \quad \langle \langle x_{mi+j} \,|\, 0 \le j < m \rangle \,|\, 0 \le i < n \rangle .$$

So $\langle 1,2,3,4,5,6 \rangle \; group_3 \; \langle \langle 1,2 \rangle, \langle 3,4 \rangle, \langle 5,6 \rangle \rangle$. One can then show that

$$\mathsf{row}_{mn}\, R \quad = \quad \mathsf{row}_n\,(\mathsf{row}_m\, R) \setminus\!\!\setminus \mathsf{fst}\, group_n^{-1} . \tag{27}$$

Substituting these results in Equations 25 and 26, we obtain a serialised circuit which is itself a chain or a row of components,

$$\begin{aligned} R^{mn} &= \sigma\,(\underline{cmx}_n \;;\; \mathsf{slow}_n\, R^m \;;\; \mathcal{D} \;;\; fork) \setminus \underline{ev}_n \;;\; \mathcal{D}^{-1}, & (28)\\ \mathsf{row}_{mn}\, R &= \nu\,(\mathsf{fst}\,\underline{cmx}_n \;;\; \mathsf{row}_m\, \mathsf{slow}_n\, R \;;\; \mathsf{snd}\,(\mathcal{D};fork)) \\ &\quad \setminus\!\!\setminus [\underline{bundle}_n; group_n^{-1},\ \underline{ev}_n] \;;\; \mathsf{snd}\,\mathcal{D}^{-1}. & (29) \end{aligned}$$

Notice that this transformation may result in a long feedback path whose length is proportional to the number of components in the array. This path can be eliminated by reflecting half of the components in a vertical axis and transposing the resulting configuration. Sometimes such operations improve the wiring within an array of components at the expense of complicating the array interface; they are covered in greater detail elsewhere [9]. Figure 9 contains an example of a circuit before and after reflections and transposition. For a concrete example, the reader will need to jump ahead to Figure 19, which contains a transposed version of the circuit shown in Figure 13.

The coalescing [16] or LSGP (Locally Sequential Globally Parallel) serialisation scheme, on the other hand, involves dividing a design into blocks, each of which will then be serialised to reduce the number of processors used. This method may preserve locality – without further transposition – at the expense of introducing additional latches and multiplexers.

Our serialisation theorems (Equations 25 and 26, for instance) tell us that after serialising the individual blocks, there will be anti-delays and $\underline{ev}$'s and $\underline{ev}^{-1}$'s left between the serialised blocks which may need to be removed. We can achieve this elimination using distributive theorems, such as Equation 6.

As an example, we now show in a few steps how the internal $\underline{ev}$'s and $\underline{ev}^{-1}$'s can be removed from a chain. First of all, a simple induction will show that given $Q;S = \iota \neq S;Q$,

$$(Q \; ; \; R \; ; \; S)^n \quad = \quad Q \; ; \; (S \; ; \; Q \; ; \; R)^n \; ; \; S. \tag{30}$$

Second, it is also simple to show that, for any A,

$$\begin{array}{ll} & \underline{ev}_n \; ; \; \underline{ev}_n^{-1} \; ; \; \sigma\,(\underline{cmx}_n \; ; \; A) \\ = & \quad \{Q \; ; \; \sigma R \; = \; \sigma\,(\mathsf{fst}\; Q \; ; \; R)\} \\ & \sigma\,(\mathsf{fst}\,(\underline{ev}_n \; ; \; \underline{ev}_n^{-1}) \; ; \; \underline{cmx}_n \; ; \; A) \\ = & \quad \{\text{Equation 23}\} \\ & \sigma\,(\underline{cmx}_n \; ; \; A). \end{array} \tag{31}$$

Armed with the two preceding equations, we can deduce that, given that $B \; = \; \sigma\,(\underline{cmx}_n;A)$,

$$\begin{array}{ll} & (B \setminus \underline{ev}_n)^m \\ = & \quad \{\text{Equation 30, since } \underline{ev}_n^{-1};\underline{ev}_n \; = \; \iota \neq \underline{ev}_n;\underline{ev}_n^{-1}\} \\ & (\underline{ev}_n \; ; \; \underline{ev}_n^{-1} \; ; \; B)^m \; \setminus \; \underline{ev}_n \\ = & \quad \{\text{Equation 31}\} \\ & B^m \setminus \underline{ev}_n. \end{array} \tag{32}$$

This final result can be used in the LSGP scheme for a chain of components:

$$\begin{array}{ll} & (R^n)^m \\ = & \quad \{\text{serialise } R^n \text{ using Equation 25}\} \\ & (\sigma\,(\underline{cmx}_n \; ; \; \mathsf{slow}_n\, R \; ; \; \mathcal{D} \; ; \; \mathit{fork}) \setminus \underline{ev}_n \; ; \; \mathcal{D}^{-1})^m \\ = & \quad \{\text{retime using Equation 6 since } \sigma\,(\underline{cmx}_n \; ; \; \mathsf{slow}_n\, R \; ; \; \mathcal{D} \; ; \; \mathit{fork}) \setminus \underline{ev}_n \text{ is timeless}\} \\ & (\sigma\,(\underline{cmx}_n \; ; \; \mathsf{slow}_n\, R \; ; \; \mathcal{D} \; ; \; \mathit{fork}) \setminus \underline{ev}_n)^m \; ; \; \mathcal{D}^{-m} \\ = & \quad \{\text{eliminate internal } \underline{ev}\text{'s and } \underline{ev}^{-1}\text{'s using Equation 32}\} \\ & (\sigma\,(\underline{cmx}_n \; ; \; \mathsf{slow}_n\, R \; ; \; \mathcal{D} \; ; \; \mathit{fork}))^m \setminus \underline{ev}_n \; ; \; \mathcal{D}^{-m}. \end{array}$$

Similarly, given that $R' \; = \; \mathsf{fst}\,\underline{cmx}_n \; ; \; \mathsf{slow}_n\, R \; ; \; \mathsf{snd}\,(\mathcal{D};\mathit{fork})$, it can be shown that

$$\mathsf{row}_m\,(\mathsf{row}_n\, R) \quad = \quad \mathsf{row}_m\,(\nu\, R') \setminus\!\!\setminus [\triangle\mathcal{D}^{-n};\underline{bundle}_n, \; \underline{ev}_n] \; ; \; \mathsf{snd}\,\mathcal{D}^{-m}.$$

Notice that since the LPGS scheme can be considered as a systematic application of serialisation techniques, we can apply it to each block obtained after the initial clustering step of the LSGP method.

3.5 Procedures for systematic serialisation

In this section we shall summarise the steps for obtaining a serialised design based on the LPGS scheme and the LSGP scheme. Let us deal with the LPGS scheme first:

1. Express the computation using operators such as row.

2. Determine the degree of serialisation based on the number of input/output pins and the input/output bandwidth. The degree of serialisation may be controlled by grouping the processors into clusters before serialising.

3. Decide whether pipelining is required from the desired performance of the system and from the delay characteristics of the components used. If so, one may have to go back to the original architecture before serialisation and apply retiming to that architecture, and then serialise the result again. The clustering technique can also be used to control the degree of pipelining.

4. Investigate whether it is possible and worthwhile to eliminate long feedback wires by transposing the circuit.

Of course, the above procedure is intended only for guidance and it needs to be applied judiciously. The main point to note is that it is easier to retime a circuit before serialising it, since serialisation introduces additional data dependencies which may render retiming difficult.

Next, we shall summarise the steps for obtaining a serialised design based on the LSGP scheme.

1. After expressing the computation in Ruby, divide the circuit into clusters of processors depending on the availability of input/output pins and on the input/output bandwidth.

2. Apply the LPGS scheme to each cluster.

3. Eliminate, by conditional distributive theorems, samplers and non-implementable delays between the array components.

A detailed example will be considered in the next section.

4 Convolver designs

In this section, we shall develop a number of convolver designs to demonstrate the techniques presented in the preceding sections. A circuit for one-dimensional convolution with time-varying coefficients can be specified as follows: given the data stream x, and the coefficient stream w such that for all t, $\#w_t = N$, compute the result stream y given by

$$y_t = \sum_{0 \leq i < N} w_{t,i} \times x_{t-i}. \tag{33}$$

It is clear from the specification that multipliers and adders will be required. These components will be denoted respectively by *mult* and *add*: $\langle a, b\rangle \, \mathit{mult} \, a \times b$, and $\langle a, b\rangle \, \mathit{add} \, a + b$. For simplicity our discussions will be confined to designs with a global synchronous clock.

We shall start with the well-known design in Figure 10. This circuit, $Cv1$, can be captured in Ruby simply as a row of $Cv1cell$s. The projection relation $\pi_2;\pi_1$ is used to select the first of the second range signal – corresponding to the result of the calculation – for output.

$$\begin{aligned} Cv1 &= \mathsf{row}_N \, Cv1cell \; ; \; \pi_2 \; ; \; \pi_1, \\ Cv1cell &= Cvcell \; ; \; \mathsf{snd}\,\mathsf{snd}\,\mathcal{D}, \\ Cvcell &= (\mathsf{snd}\,\mathit{mult} \; ; \; \mathit{add} \; ; \; {\pi_2}^{-1}) \updownarrow (\mathit{fork} \; ; \; \mathsf{snd}\,\pi_1). \end{aligned}$$

These expressions can themselves be obtained from ones which are closer to the behavioural specification in Equation 33; the method is well-documented in the literature (see [4] and [11]) and we shall not spend any time on this step.

Various implementations can be derived for $Cv1$. We shall first look at a simple LPGS scheme; then we shall address the problem of introducing pipelining in various ways. Next, designs are obtained by applying the LSGP method. Finally we summarise the process and compare the trade-offs of the designs.

4.1 LPGS design

First of all, our aim is to obtain a systematically-serialised version of $Cv1$ using the LPGS method. Given that M is a factor of N such that $MK = N$, we shall derive $Cv2$ – which contains K copies of $Cvcell$ – from $Cv1$ which has MK copies of $Cvcell$:

$$\begin{aligned} & Cv1 \\ = \quad & \{\text{Equation 29 and } \mathsf{fst}\,(\underline{bundle}_n ; group_n^{-1}) \;;\; \pi_2 \;=\; \pi_2\} \\ & [\underline{ev}_M^{-1},\ group_M ; \underline{bundle}_M^{-1}] \;;\; Cv2 \;;\; \underline{ev}_M \;;\; \mathcal{D}^{-1} \end{aligned}$$

where

$$\begin{aligned} Cv2 &= \nu\,(\mathsf{fst}\,\underline{cmx}_M \;;\; Cv2cell \;;\; \mathsf{snd}\,(\mathcal{D} \;;\; fork)) \;;\; \pi_2 \;;\; \pi_1, \\ Cv2cell &= \mathsf{row}_K\,(\mathsf{slow}_M\; Cv1cell) \\ &= \mathsf{row}_K\,(Cvcell \;;\; \mathsf{snd}\,\mathsf{snd}\,\mathcal{D}^M). \end{aligned}$$

An instance of $Cv2$ is shown in Figure 11. Note that in the diagrams we tried to avoid bending wires rather than to draw the components to scale, with the unfortunate effect that a multiplexer appears to be larger than a multiplier-adder.

4.2 LPGS design with pipelining

To introduce pipelining in the design, the array will first be retimed using Equation 7 which can be written as

$$\mathsf{row}_n\, R \;=\; \mathsf{row}_n\,(R \;;\; \mathsf{snd}\,\mathcal{D}) \setminus\!\setminus \mathsf{fst}\,\triangle\mathcal{D}^{-1} \;;\; \mathsf{snd}\,\mathcal{D}^{-n} \qquad (34)$$

on the assumption that R is timeless.

$$\begin{aligned} & Cv1 \\ = \quad & \{\text{Equation 27}\} \\ & \mathsf{snd}\,group_M \;;\; \mathsf{row}_M\,(\mathsf{row}_K\; Cv1cell) \;;\; \pi_2 \;;\; \pi_1 \\ = \quad & \{\text{Equation 34}\} \\ & \mathsf{snd}\,(group_M \;;\; \triangle\mathcal{D}) \;;\; Cv3 \;;\; \mathcal{D}^{-M} \end{aligned}$$

where

$$\begin{aligned} Cv3 &= \mathsf{row}_M\; Cv3cell \;;\; \pi_2 \;;\; \pi_1, \\ Cv3cell &= \mathsf{row}_K\; Cv1cell \;;\; \mathsf{snd}\,\mathcal{D}. \end{aligned}$$

An example of $Cv3$ is shown in Figure 12.

Let $PQ = M$. We shall now cluster the array and serialise the resulting design to get $Cv4$ (Figure 13),

$$\begin{aligned} & Cv3 \\ = \quad & \{\text{Equation 29}\} \\ & [\underline{ev}_P^{-1},\ group_P ; \underline{bundle}_P^{-1}] \;;\; Cv4 \;;\; \underline{ev}_P \;;\; \mathcal{D}^{-1} \end{aligned}$$

where

$$\begin{aligned} Cv4 &= \nu\,(\mathsf{fst}\,\underline{cmx}_P \;;\; Cv4cell \;;\; \mathsf{snd}\,(\mathcal{D} \;;\; fork)) \;;\; \pi_2 \;;\; \pi_1, \\ Cv4cell &= \mathsf{row}_Q\,(\mathsf{slow}_P\; Cv3cell) \\ &= \mathsf{row}_Q\,(\mathsf{row}_K\,(Cvcell \;;\; \mathsf{snd}\,\mathsf{snd}\,\mathcal{D}^P) \;;\; \mathsf{snd}\,\mathcal{D}^P). \end{aligned}$$

4.3 Pipelining the multiply-adders

One can pipeline the multiply-adders instead, by reversing the coefficients using the relation *rev*, defined by $x \; rev \; \langle x_{\#x-i-1} \mid 0 \leq i < \#x \rangle$. Given that $wiring = fork; \mathsf{snd}\,\pi_1$ and $madd = \mathsf{snd}\, mult; add; {\pi_2}^{-1}$, the relevant properties are

$$\mathsf{row}_n\,(wiring \;;\; \mathsf{snd}\,\mathcal{D}) \setminus\!\!\setminus \mathsf{fst}\,rev \;=\; \mathsf{fst}\,\mathcal{D}^n \;;\; \mathsf{row}_n\,(\mathsf{fst}\,\mathcal{D}^{-1} \;;\; wiring) \;;\; \mathsf{snd}\,\mathcal{D}^n, \tag{35}$$

$$\mathsf{row}\,madd \;=\; \mathsf{snd}\,rev \;;\; \mathsf{row}\,madd \tag{36}$$

since addition is commutative and associative.

We shall also need a reflected version of Equation 34 that says that given timeless R,

$$\mathsf{row}_n\,R \;=\; \mathsf{fst}\,\mathcal{D}^{-n} \;;\; \mathsf{row}_n\,(\mathsf{fst}\,\mathcal{D} \;;\; R) \setminus\!\!\setminus \mathsf{fst}\,(\triangle\mathcal{D}\backslash rev), \tag{37}$$

and two results concerning alternative bracketing of composite expressions,

$$[[P,Q],R] \,;\, (\mathsf{snd}\,X \;;\; A) \updownarrow B \;;\; [T,[U,V]] \;=\; (\mathsf{fst}\,P;A;[T,U]) \updownarrow ([Q,R];B;[X,V]), \tag{38}$$

$$\mathsf{row}\,(Q \updownarrow R) \;=\; \mathsf{row}\,Q \updownarrow \mathsf{row}\,R. \tag{39}$$

We can then transform $Cv1$,

$$
\begin{aligned}
& Cv1 \\
= \;& \{\text{definition of } Cv1\} \\
& \mathsf{row}_N\,(madd \updownarrow wiring \;;\; \mathsf{snd}\,\mathsf{snd}\,\mathcal{D}) \;;\; \pi_2 \;;\; \pi_1 \\
= \;& \{\text{Equations 38 and 39}\} \\
& \mathsf{row}_N\,madd \updownarrow \mathsf{row}_N\,(wiring;\mathsf{snd}\,\mathcal{D}) \;;\; \pi_2 \;;\; \pi_1 \\
= \;& \{\text{Equations 36 and 38}\} \\
& \mathsf{row}_N\,madd \updownarrow (\mathsf{row}_N\,(wiring;\mathsf{snd}\,\mathcal{D}) \;;\; \mathsf{fst}\,rev) \;;\; \pi_2 \;;\; \pi_1 \\
= \;& \{rev = rev^{-1} \text{ and Equation 35}\} \\
& \mathsf{row}_N\,madd \updownarrow ([\mathcal{D}^N, rev] \;;\; \mathsf{row}_N\,(\mathsf{fst}\,\mathcal{D}^{-1}; wiring) \;;\; \mathsf{snd}\,\mathcal{D}^N) \;;\; \pi_2 \;;\; \pi_1 \\
= \;& \{\text{Equations 38, 39 and definition of } Cvcell\} \\
& [\mathsf{snd}\,\mathcal{D}^N, rev] \;;\; \mathsf{row}_N\,(\mathsf{fst}\,\mathsf{snd}\,\mathcal{D}^{-1} \;;\; Cvcell) \;;\; \mathsf{snd}\,\mathsf{snd}\,\mathcal{D}^N \;;\; \pi_2 \;;\; \pi_1 \\
= \;& \{\text{Equation 37 and } \mathsf{snd}\,\mathsf{snd}\,\mathcal{D}^n;\pi_2;\pi_1 = \pi_2;\pi_1\} \\
& [\mathsf{fst}\,\mathcal{D}^{-N}, \triangle\mathcal{D}^{-1}; rev] \;;\; \mathsf{row}_N\,(\mathsf{fst}\,\mathsf{fst}\,\mathcal{D} \;;\; Cvcell) \;;\; \pi_2 \;;\; \pi_1 \\
= \;& \{\text{define } Cv5 = \mathsf{row}_N\,Cv5cell \;;\; \pi_2 \;;\; \pi_1\} \\
& [\mathsf{fst}\,\mathcal{D}^{-N}, \triangle\mathcal{D}^{-1}; rev] \;;\; Cv5
\end{aligned}
$$

where $Cv5cell = \mathsf{fst}\,\mathsf{fst}\,\mathcal{D} \;;\; Cvcell$ (Figure 14).

How should we choose between $Cv1$ and $Cv5$? Given that T_m and T_a are respectively the combinational delay of cells *mult* and *add*, and T_p is the propagation delay of x across the cell *wiring* and that $T_m + T_a > T_p$, $Cv5$ should run faster than $Cv1$ because its clock speed is restricted by $(N-1)\,T_p + T_m + T_a$ rather than by $T_m + NT_a$. However, $Cv5$ has twice the latency of $Cv1$, and it requires an extra triangular array of latches for skewing. Another possibility is that, although not shown in the diagram or in the calculation, additional latches may be required in $Cv5$ since the row of multiply-adders need to handle larger numbers than the broadcast circuit $\mathsf{row}\,wiring$. More wires may be needed when the circuit is refined to bit-level which in turn require more latches for them to be pipelined.

Next, let us proceed to serialise $Cv5$:

$$\begin{array}{cl} & Cv5 \\ = & \{\text{ Equation } 27\} \\ & \mathsf{snd}\, group_M \ ; \ \mathsf{row}_M\,(\mathsf{row}_K\ Cv5cell) \ ; \ \pi_2 \ ; \ \pi_1 \\ = & \{\text{Equation } 29\} \\ & [\underline{ev}_M^{-1},\ group_M ; \underline{bundle}_M^{-1}] \ ; \ Cv6 \ ; \ \underline{ev}_M \ ; \ \mathcal{D}^{-1} \end{array}$$

where

$$\begin{array}{rcl} Cv6 & = & \nu\,(\mathsf{fst}\,\underline{cmx}_M \ ; \ Cv6cell \ ; \ \mathsf{snd}\,(\mathcal{D} \ ; \ fork)) \ ; \ \pi_2 \ ; \ \pi_1, \\ Cv6cell & = & \mathsf{row}_K\,(\mathsf{fst}\,\mathsf{fst}\,\mathcal{D}^M \ ; \ Cvcell). \end{array}$$

An instance of $Cv6$ is shown in Figure 15.

4.4 LSGP design

In this section we explore two ways of serialising $Cv1$ by the LSGP method.

Let us first group the elements of $Cv1$ into a row of M clusters, each containing K components. These clusters will then be individually serialised. Finally the non-implementable delays will be eliminated by pipelining.

To eliminate the internal $\underline{ev}$'s and $\underline{ev}^{-1}$'s, we shall need the result

$$\mathsf{row}_m\,(R' \setminus\!\!\setminus \mathsf{snd}\,\underline{ev}_n) \quad = \quad (\mathsf{row}_m\ R') \setminus\!\!\setminus \mathsf{snd}\,\underline{ev}_n \tag{40}$$

where $R' = \nu\,(\mathsf{fst}\,\underline{cmx}_n \ ; \ R)$. A similar result for chains, given by Equation 32, has been derived in Section 3.4. $Cv1$ can now be transformed,

$$\begin{array}{cl} & Cv1 \\ = & \{\text{ Equation } 27\} \\ & \mathsf{snd}\, group_M \ ; \ \mathsf{row}_M\,(\mathsf{row}_K\ Cv1cell) \ ; \ \pi_2 \ ; \ \pi_1 \\ = & \{\text{Equation 26 and let } Cv7cell = \nu\,(\mathsf{fst}\,\underline{cmx}_K \ ; \ \mathsf{slow}_K\ Cv1cell \ ; \ \mathsf{snd}\,(\mathcal{D} \ ; \ fork))\} \\ & \mathsf{snd}\, group_M \ ; \ \mathsf{row}_M\,([\underline{ev}_K^{-1},\ \underline{bundle}_K^{-1}] \ ; \ Cv7cell \ ; \ \mathsf{snd}\,(\mathcal{D}^{-K} \ ; \ \underline{ev}_K)) \ ; \ \pi_2 \ ; \ \pi_1 \\ = & \{\text{Equation } 40\} \\ & [\underline{ev}_K^{-1},\ group_M ; \mathsf{map}\,\underline{bundle}_K^{-1}] \ ; \ \mathsf{row}_M\,(Cv7cell \ ; \ \mathsf{snd}\,\mathcal{D}^{-K}) \ ; \ \mathsf{snd}\,\underline{ev}_K \ ; \ \pi_2 \ ; \ \pi_1 \\ = & \{\text{Equation } 34\} \\ & [\underline{ev}_K^{-1},\ group_M ; \mathsf{map}\,\underline{bundle}_K^{-1} ; \triangle\mathcal{D}^K] \ ; \ Cv7 \ ; \ \underline{ev}_K \ ; \ \mathcal{D}^{-M} \end{array}$$

where

$$\begin{array}{rcl} Cv7 & = & \mathsf{row}_M\ Cv7cell \ ; \ \pi_2 \ ; \ \pi_1, \\ Cv7cell & = & \nu\,(\mathsf{fst}\,\underline{cmx}_K \ ; \ \mathsf{slow}_K\ Cv1cell \ ; \ \mathsf{snd}\,(\mathcal{D} \ ; \ fork)) \\ & = & \nu\,(\mathsf{fst}\,\underline{cmx}_K \ ; \ Cvcell \ ; \ \mathsf{snd}\,\mathsf{snd}\,\mathcal{D}^K \ ; \ \mathsf{snd}\,(\mathcal{D} \ ; \ fork)). \end{array}$$

An example of $Cv7$ is shown in Figure 16.

A more flexible approach is to have an additional level of clustering in $Cv1$, so that each serialised component may contain several copies of $Cv1cell$. Let $M = PQ$ and

$$Cv1' \quad = \quad \mathsf{row}_M\ Cv1'cell \ ; \ \pi_2 \ ; \ \pi_1,$$

where $Cv1'cell = \mathsf{row}_K\ Cv1cell$, so that

$$Cv1 \ = \ \mathsf{snd}\, group_M \ ; \ Cv1'.$$

We now cluster $Cv1'$ and apply serialisation to each of the resulting clusters,

$$\begin{aligned}
& Cv1' \\
= \ & \{\text{Equation 27}\} \\
& \mathsf{snd}\, group_Q \ ; \ \mathsf{row}_Q\, (\mathsf{row}_P\ Cv1'cell) \ ; \ \pi_2 \ ; \ \pi_1 \\
= \ & \{\text{Equation 26} \\
& \quad \text{and let } Cv8cell = \nu\, (\mathsf{fst}\, \underline{cmx}_P \ ; \ \mathsf{slow}_P\ Cv1'cell \ ; \ \mathsf{snd}\, (\mathcal{D} \ ; \ fork))\} \\
& \mathsf{snd}\, group_Q \ ; \ \mathsf{row}_Q\, ([\underline{ev}_P^{-1}, \ \underline{bundle}_P^{-1}] \ ; \ Cv8cell \ ; \ \mathsf{snd}\, (\mathcal{D}^{-P} \ ; \ \underline{ev}_P)) \ ; \ \pi_2 \ ; \ \pi_1 \\
= \ & \{\text{Equation 40 and Equation 34}\} \\
& [\underline{ev}_P^{-1}, \ group_Q ; \mathsf{map}\, \underline{bundle}_P ; \triangle\mathcal{D}^P] \ ; \ Cv8 \ ; \ \underline{ev}_P \ ; \ \mathcal{D}^{-Q}
\end{aligned}$$

where

$$\begin{aligned}
Cv8 & = \mathsf{row}_Q\ Cv8cell \ ; \ \pi_2 \ ; \ \pi_1, \\
Cv8cell & = \nu\, (\mathsf{fst}\, \underline{cmx}_P \ ; \ \mathsf{row}_K\, (\mathsf{slow}_P\ Cv1cell) \ ; \ \mathsf{snd}\, (\mathcal{D} \ ; \ fork)) \\
& = \nu\, (\mathsf{fst}\, \underline{cmx}_P \ ; \ \mathsf{row}_K\, (Cvcell \ ; \ \mathsf{snd}\,\mathsf{snd}\, \mathcal{D}^P) \ ; \ \mathsf{snd}\, (\mathcal{D} \ ; \ fork)).
\end{aligned}$$

Figure 17 shows an example of $Cv8$. Notice that circuits with identical structure to $Cv7$ can be obtained by substituting K by 1 and P by K in $Cv8$.

We can, of course, continue our design exploration, but let us now turn to examining the trade-offs of the designs that we have already obtained.

4.5 Design trade-offs

Figure 18 contains a design tree summarising the relationships between the various convolvers, whose features are shown in Table 1. Each entry in the table consists of an expression involving N, the number of coefficients in the convolution, K, a parameter controlling the degree of serialisation and pipelining, and P, a parameter controlling the degree of serialisation. One way to derive the formulae in Table 1 is to adopt the technique of non-standard interpretation [8].

For designs which have not been serialised such as $Cv1$, $Cv3$ and $Cv5$, a smaller K will give a faster circuit at the expense of a larger latency and a larger number of latches, which in turn may lead to a larger circuit consuming more power. For LPGS designs such as $Cv2$, $Cv4$ and $Cv6$, a smaller K (and a larger P in case of $Cv4$) will produce a more compact circuit capable of running at a higher clock frequency, but it will take more cycles between successive valid outputs as indicated by the larger slowdown factor. For LSGP designs such as $Cv7$ and $Cv8$, a large K and a large P correspond to designs with a small number of cells but a large slowdown factor.

To choose between the alternatives, a designer with the task of developing a convolver for a given size and performance can first check, from the available size of multiply-adders, whether the convolver can be constructed without serialisation. If so, the next step is to determine whether the performance requirements can be met by designs such as $Cv1$, $Cv3$ and $Cv5$ with the appropriate degree of pipelining. Otherwise serialisation will be necessary, and in that case the LPGS scheme may be tried first since it corresponds to the LSGP scheme with a single cluster. If localised connections are essential, then one may have to apply the LSGP scheme with multiple clusters which usually results in more latches and multiplexers.

Earlier (and also in Section 3.4) we have briefly mentioned that transposition [9] may be able to localise long wires in a LPGS design, hence reducing the T_f term in expressions for minimum cycle time in Table 1. The result of transposing the design in Figure 13 can be found in Figure 19. Notice that the coefficients have to be re-arranged so that they match the sample of x delayed by the designated number of clock cycles; this re-arrangement is captured by the relation *revzip* shown in the figure.

As for LSGP designs, it is usually desirable to control the cycling multiplexers by a single control signal derived from a counter. This control signal will be pipelined when we retime the circuit to remove non-implementable anti-delays – step 3 of the proposed LSGP design procedure outlined in Section 3.5.

5 Summary

Techniques for systematically serialising array-based architectures have been presented in this paper. The main innovations include the adoption of expressions which can be instantiated to describe a variety of circuits with different degrees of serialisation; the use of abstractions, like $\underline{ev}$ and $\underline{cmx}$, to establish theorems for serialising designs; and the formulation of procedures to guide the development of serial architectures.

An advantage of our approach is the explicit representation of interface constraints; this facilitates the construction of composite circuits from simpler components. Moreover, the concise and uniform representation of circuits and their interface, coupled with an equational style of transformation, leads to a simple development framework. Although it may not be possible to automate our techniques as fully as other methods [18], the range of designs with which we can deal – such as an architecture serialised by an arbitrary mixture of LSGP and LPGS transformations – appears to be wider. Furthermore, the techniques presented in this paper have already been used in developing implementations based on Field-Programmable Gate Arrays [12].

Our plan for future work includes the following. First, it may be possible to provide a more formal, and perhaps more general, definition of serialisation which can then be used to justify Equations 24, 25 and 26. Second, the scope of our method should be extended to cover other common architectural idioms such as triangular- or tree-shaped networks. Third, it may prove fruitful to study the connection between our method and other formalisms for serialising designs (see for example [2], [6], [14], [15], [16], [17], [18]). Finally, it is our intention to incorporate techniques for implementing serialisation in our prototype design system (see [12], [13]), and to make it compatible with other circuit design tools and cell libraries.

Acknowledgement

Thanks to Mark Aagaard, Tony Hoare, Andrew Kay, Richard Stamper and the referee for their comments, and to L.S. Wong for help in preparing some of the diagrams for this paper. The support of Rank Xerox (UK) Limited is gratefully acknowledged.

References

[1] Bird, R. S., Lectures on constructive functional programming, in: M. Broy, ed., Constructive Methods in Computing Science, vol. F55, Nato ASI Series (Springer-Verlag, 1989) pp. 151–216.

[2] Bu, J., E. F. Deprettere and P. Dewilde, A design methodology for fixed-size systolic arrays, in: S. Y. Kung et. al., eds., Proc. International Conference on Application-Specific Array Processors (IEEE Computer Society Press, 1990) pp. 591–602.

[3] Jones, G. and M. Sheeran, Timeless truths about sequential circuits, in: S. K. Tewksbury, B. W. Dickinson and S. C. Schwartz, eds., Concurrent Computations: Algorithms, Architectures and Technology (Plenum Press, 1988) pp. 245–259.

[4] Jones, G. and M. Sheeran, Circuit design in Ruby, in: J. Staunstrup, ed., Formal Methods for VLSI Design (North-Holland, 1990) pp. 13–70.

[5] Jones, G. and M. Sheeran, Deriving bit-serial circuits in Ruby, in: A. Halaas and P. B. Denyer, eds., Proc. VLSI 91 (1991) pp. 3a.3.1–3a.3.10.

[6] Kung, S. Y., *VLSI Array Processors*, Prentice Hall, 1988.

[7] Leiserson, C. E. and J. B. Saxe, Optimizing synchronous circuitry by retiming, in: R. Bryant, ed., Third Caltech Conference on Very Large Scale Integration (Computer Science Press, 1983) pp. 87–116.

[8] Luk, W., Analysing parametrised designs by non-standard interpretation, in: Kung, S.Y. et. al., eds., Proc. International Conference on Application-Specific Array Processors (IEEE Computer Society Press, 1990) pp. 133–144.

[9] Luk, W., Optimising designs by transposition, in: G. Jones and M. Sheeran, eds., Designing Correct Circuits (Springer-Verlag, 1991) pp. 332–354.

[10] Luk, W. and G. Brown, A systolic LRU processor and its top-down development, Science of Computer Programming 15 (2–3) (1990) 217–233.

[11] Luk, W. and G. Jones, From specification to parametrised architectures, in: G. J. Milne, ed., The Fusion of Hardware Design and Verification (North-Holland, 1988) pp. 267–288.

[12] Luk, W. and I. Page, Parameterising designs for FPGAs, in: W. Moore and W. Luk (eds), *FPGAs* (Abingdon EE&CS Books, 1991) pp. 284–295.

[13] Luk, W., G. Jones and M. Sheeran, Computer-based tools for regular array design, in: J. V. McCanny, J. McWhirter and E. E. Swartzlander Jr., eds., Systolic Array Processors (Prentice Hall, 1989) pp. 589–598.

[14] Moldovan, D. I. and J. A. B. Fortes, Partitioning and mapping algorithms into fixed size systolic arrays, IEEE Trans. Comput. C-35 (1) (1986) 1–12.

[15] Moreno, J. and T. Lang, A graph-based approach to map matrix algorithms onto application-specific multi-processor arrays, Proc. International Conference of the Chilean Computer Society, 1990.

[16] Navarro, J. J., J. M. Llaberia and M. Valero, Partitioning: an essential step in mapping algorithms into systolic array processors, IEEE Computer, 20 (7) (1987) 77–89.

[17] Parhi, K. K., C.-Y. Wang and A. P. Brown, Synthesis of control circuits in folded DSP architectures, IEEE J. Solid-State Circuits 27 (1) (1992) 29–43.

[18] Van Dongen, V., Mapping uniform recurrences onto small size arrays, in: E. H. L. Aarts et. al., eds., PARLE'91, LNCS 505 (Springer-Verlag 1991) pp. 191–208.

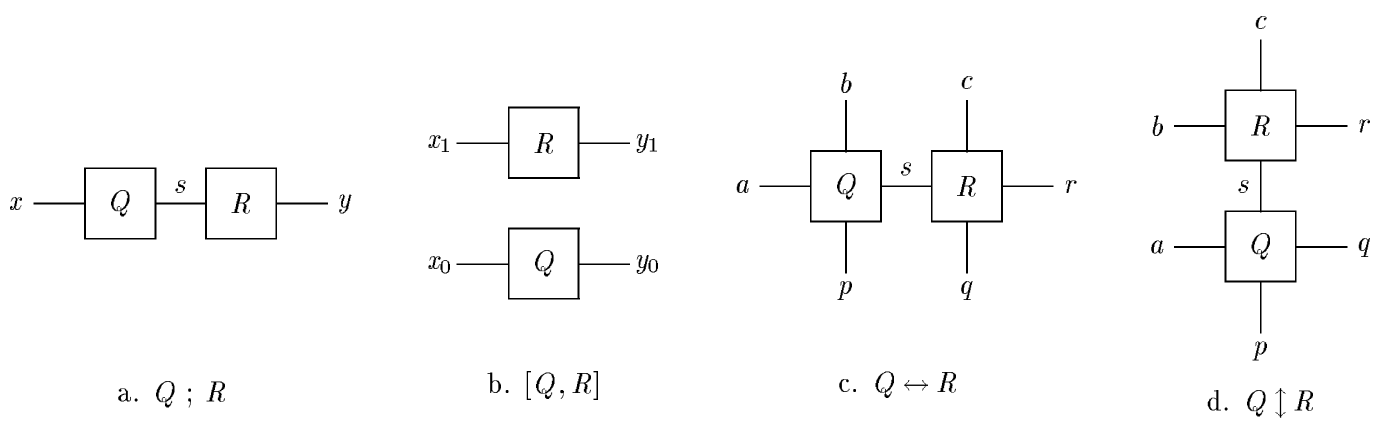


Figure 1 Binary compositions.

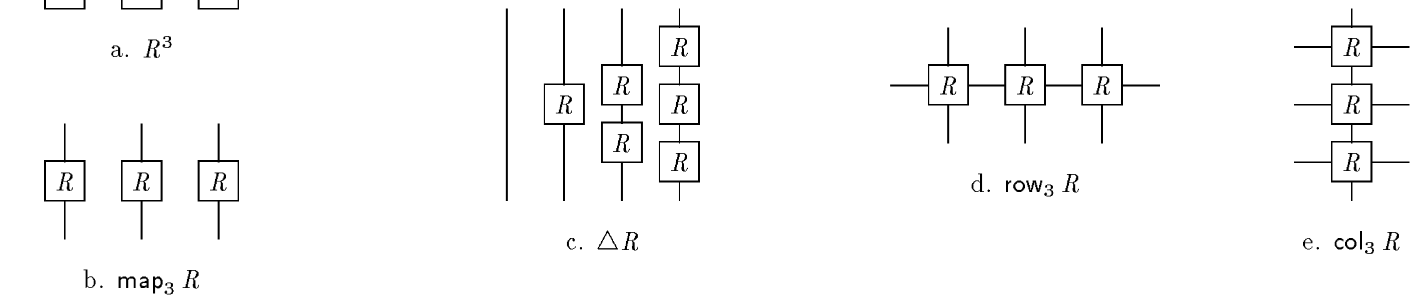
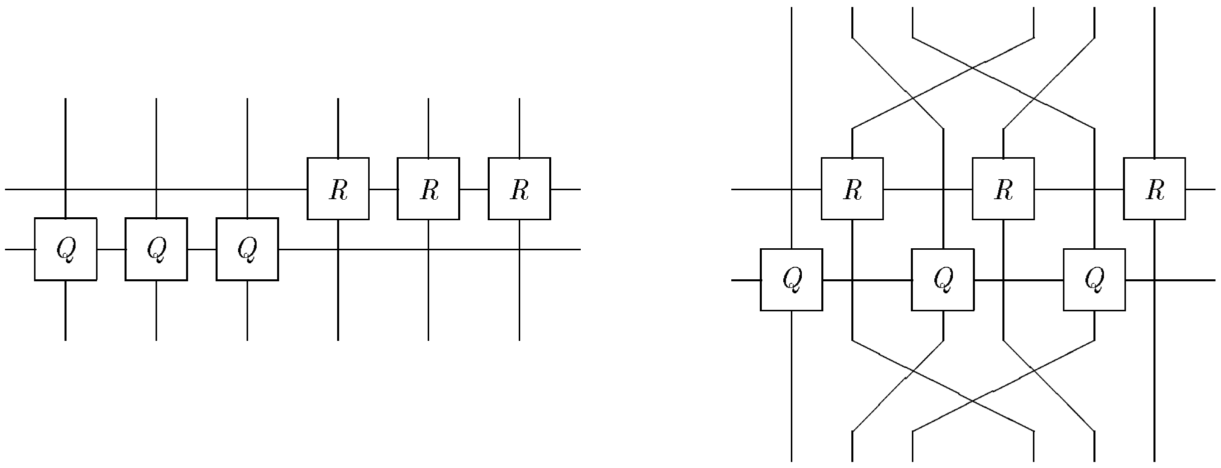
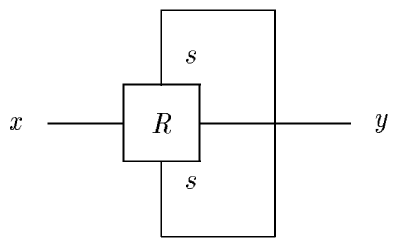


Figure 2 Repeated compositions.

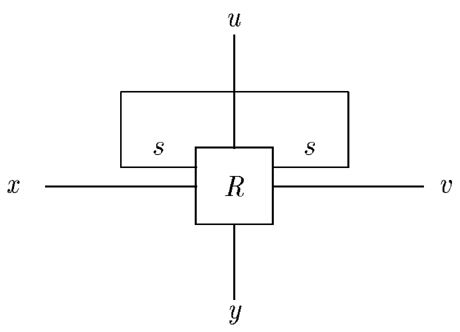


$[\mathsf{row}\,Q,\ \mathsf{row}\,R] \setminus zip^{-1}$ $\mathsf{row}\,([Q, R]\backslash zip^{-1}) \mathbin{\backslash\!\backslash} \mathsf{fst}\; zip^{-1}$

Figure 3 An instance of a theorem involving *zip*.



a. $x\ (\sigma\, R)\ y \ \Leftrightarrow\ \exists s.\, \langle x, s\rangle\ R\ \langle s, y\rangle$



b. $\langle x, u\rangle\ (\nu\, R)\ \langle y, v\rangle \ \Leftrightarrow\ \exists s.\, \langle\langle x, s\rangle, u\rangle\ R\ \langle y, \langle v, s\rangle\rangle$

Figure 4 Functions that describe feedback configurations.

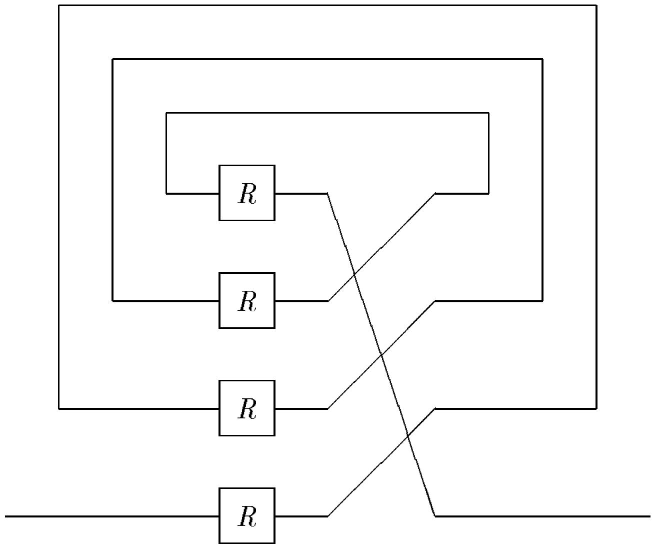


Figure 5 An instance of $\sigma\,(apl\ ;\ \mathsf{map}_n\ R\ ;\ apr^{-1})$, $n = 4$.

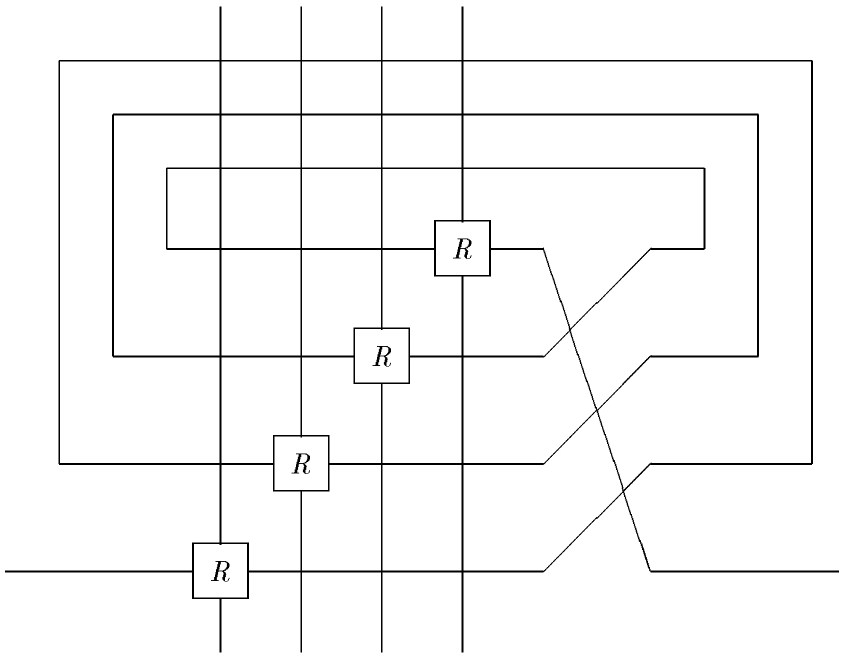


Figure 6 An instance of $\nu\,(\mathsf{fst}\,apl\ ;\ \mathsf{map}_n\,R \setminus zip^{-1}\ ;\ \mathsf{snd}\,(apr^{-1};swap))$, $n = 4$.

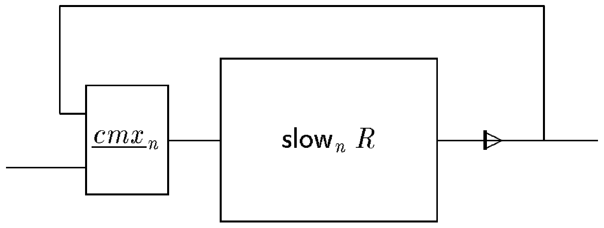


Figure 7 An instance of $\sigma\,(\underline{cmx}_n\ ;\ \mathsf{slow}_n\ R\ ;\ \mathcal{D}\ ;\ \mathit{fork})$.

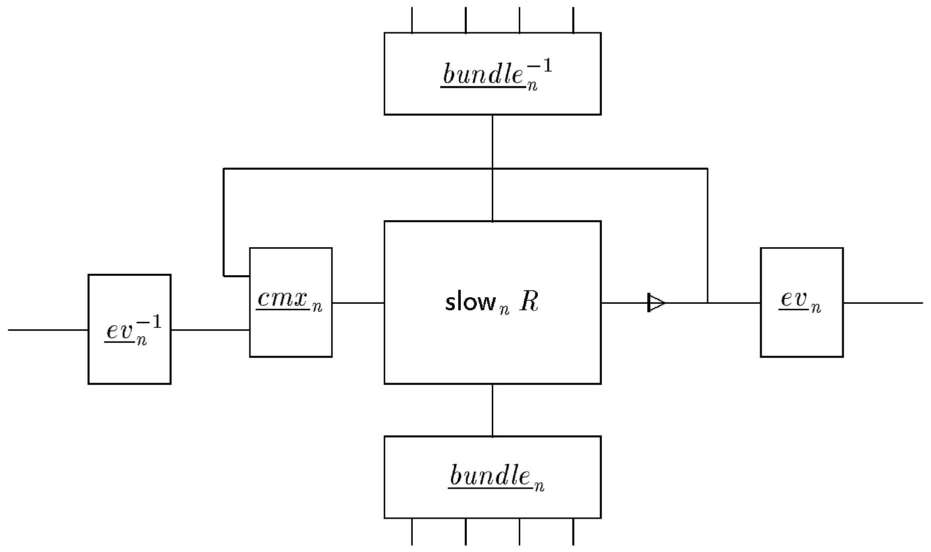


Figure 8 An instance of $\nu\,(\mathsf{fst}\,\underline{cmx}_n\ ;\ \mathsf{slow}_n\ R\ ;\ \mathsf{snd}\,(\mathcal{D};\mathit{fork}))\ \backslash\!\backslash\ [\underline{bundle}_n, \underline{ev}_n]$, $n = 4$.

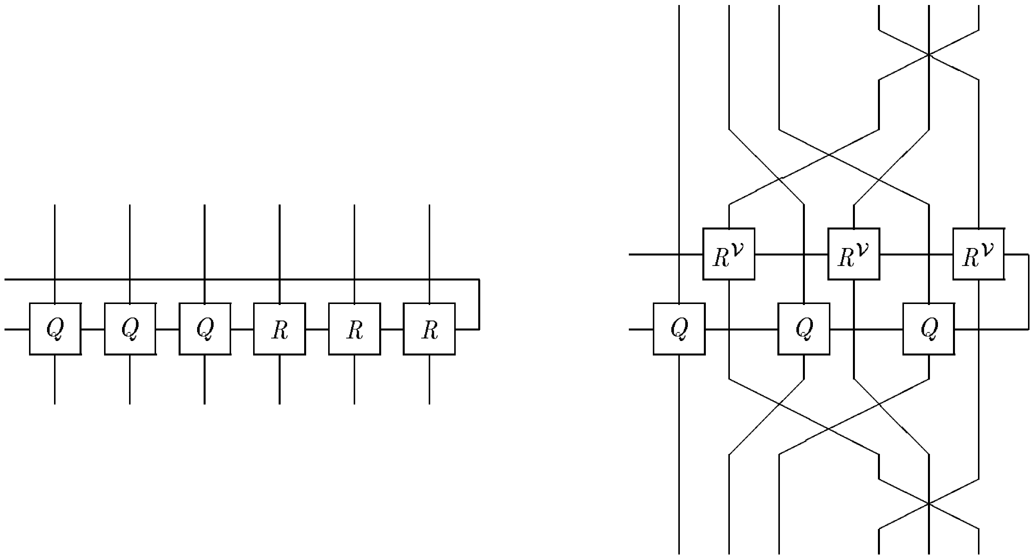


Figure 9 Transforming a row of components with a feedback wire. $R^{\mathcal{V}}$ corresponds to reflecting R in a vertical axis.

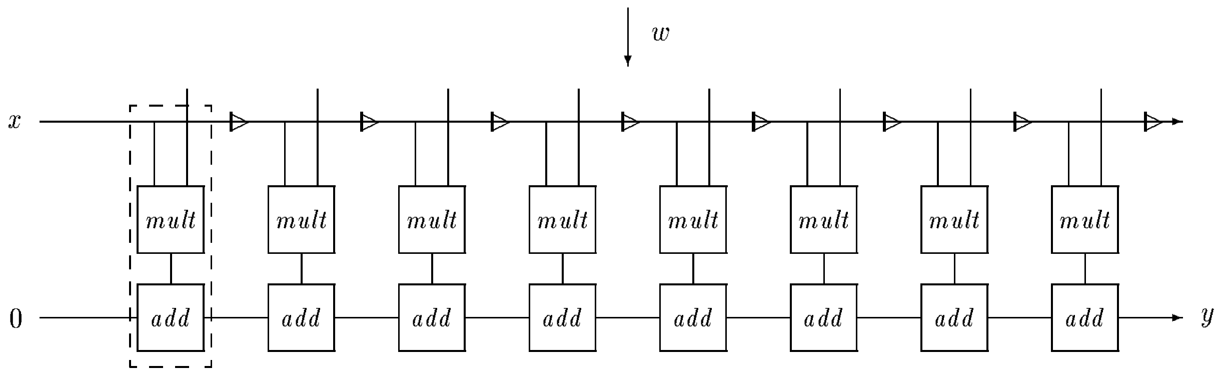
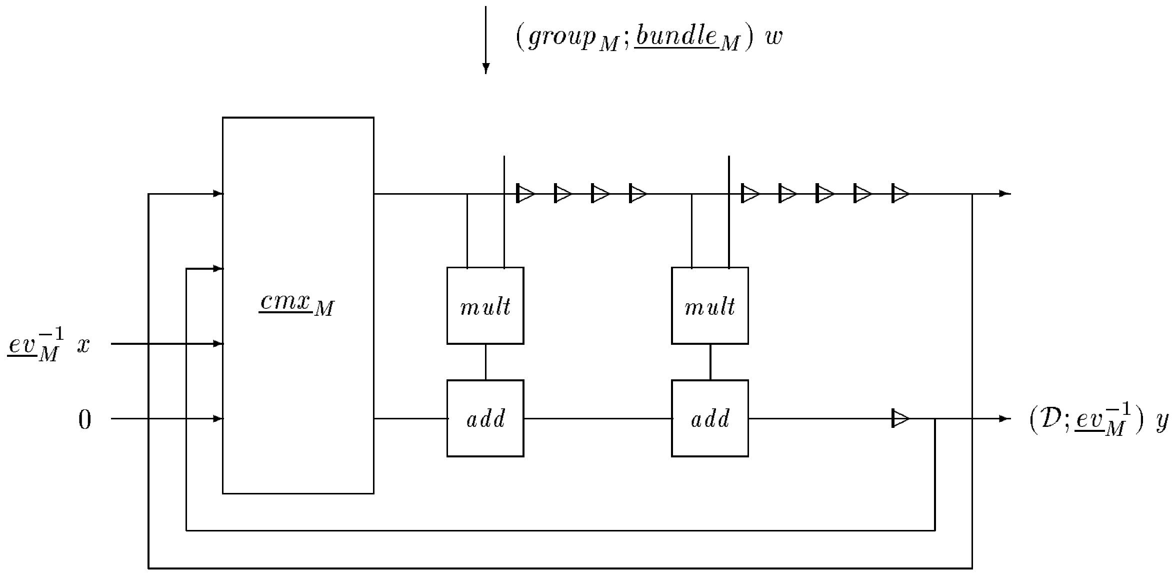


Figure 10 Design $Cv1$ $(N = 8)$. An instance of $Cvcell$ is enclosed in the dashed box.



$(group_M; \underline{bundle}_M)\ w$
$\underline{cmx}_M$
$mult$
$mult$
add
add
$\underline{ev}_M^{-1}\ x$
0
$(\mathcal{D}; \underline{ev}_M^{-1})\ y$

Figure 11 Design $Cv2$ $(N = 8,\ M = 4,\ K = 2)$.

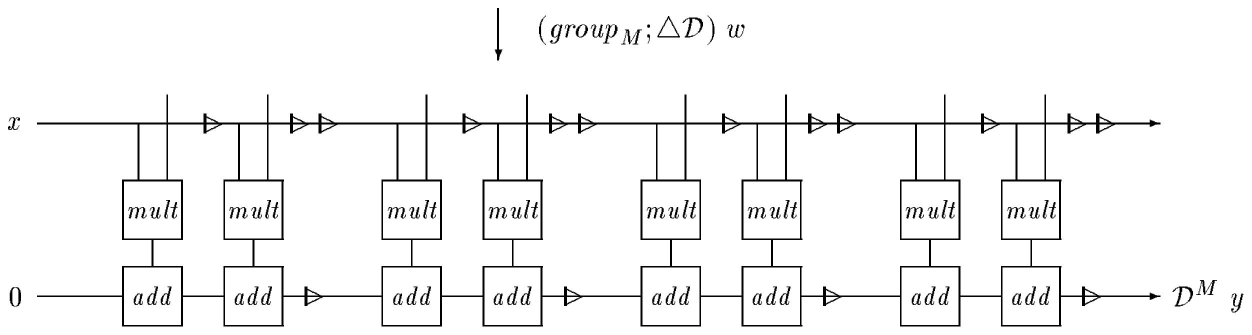
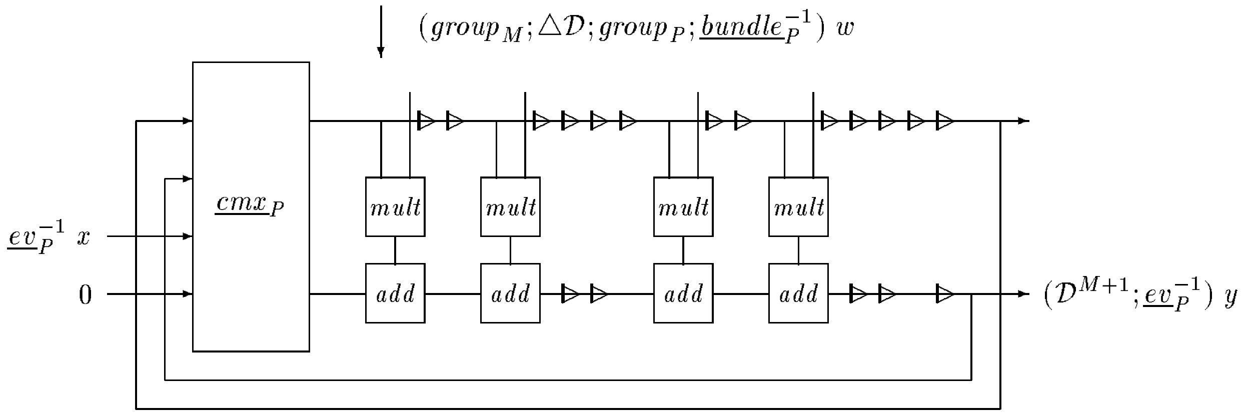


Figure 12 Design $Cv3$ $(N = 8,\ M = 4,\ K = 2)$.



$(group_M;\triangle\mathcal{D};group_P;\underline{bundle}_P^{-1})\ w$
$\underline{cmx}_P$
$\underline{ev}_P^{-1}\ x$
0
mult
mult
mult
mult
add
add
add
add
$(\mathcal{D}^{M+1};\underline{ev}_P^{-1})\ y$

Figure 13 Design $Cv4$ $(N = 8,\ M = 4,\ K = P = Q = 2)$.

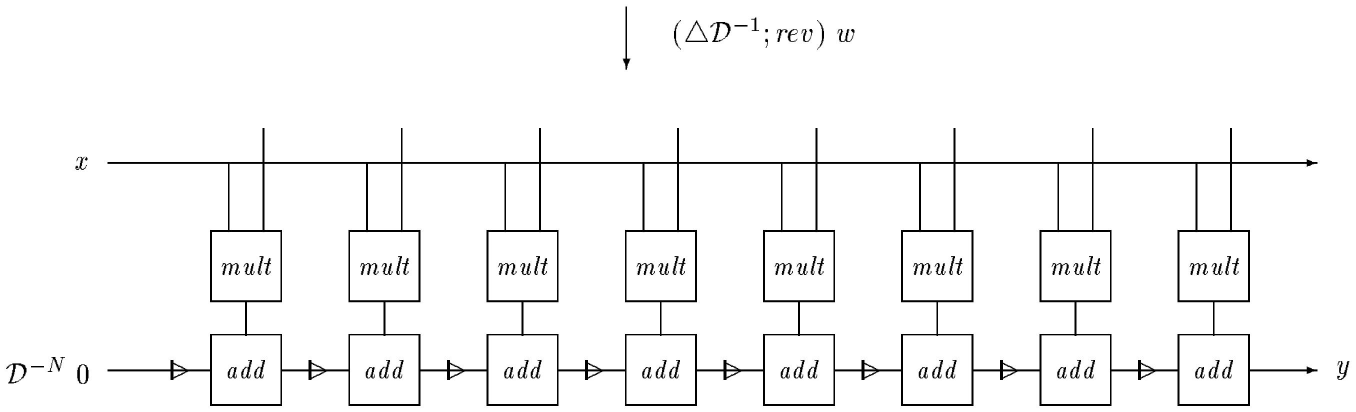


Figure 14 Design $Cv5$ $(N = 8)$.

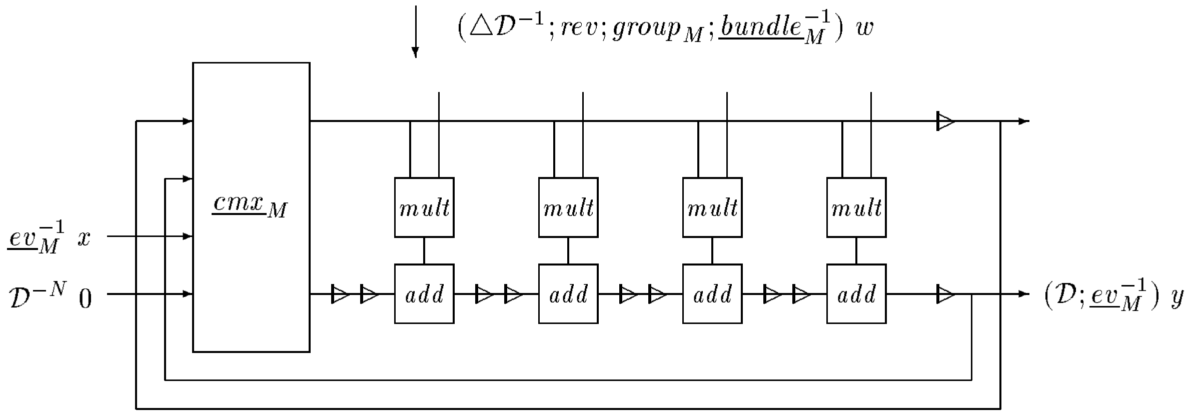

$(\triangle\mathcal{D}^{-1}; rev; group_M; \underline{bundle}_M^{-1})\ w$
$\underline{cmx}_M$
$\underline{ev}_M^{-1}\ x$
$\mathcal{D}^{-N}\ 0$
$mult$
$mult$
$mult$
$mult$
add
add
add
add
$(\mathcal{D}; \underline{ev}_M^{-1})\ y$

Figure 15 Design $Cv6$ $(N = 8,\ M = 2,\ K = 4)$.

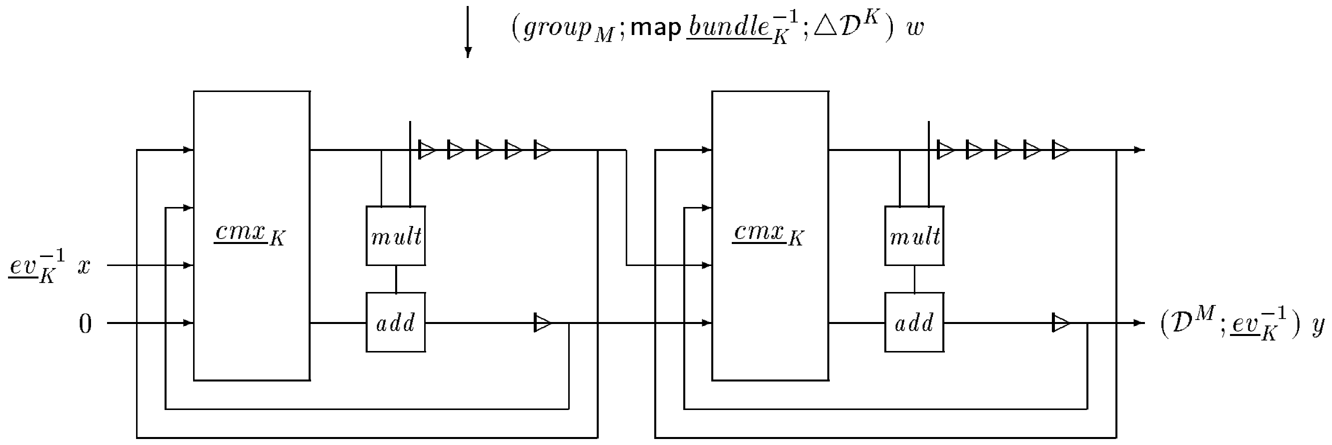

$(group_M ; \mathsf{map}\ \underline{bundle}_K^{-1} ; \triangle\mathcal{D}^K)\ w$
$\underline{cmx}_K$
$mult$
add
$\underline{ev}_K^{-1}\ x$
0
$\underline{cmx}_K$
$mult$
add
$(\mathcal{D}^M ; \underline{ev}_K^{-1})\ y$

Figure 16 Design $Cv7$ $(N = 8,\ M = 2,\ K = 4)$.

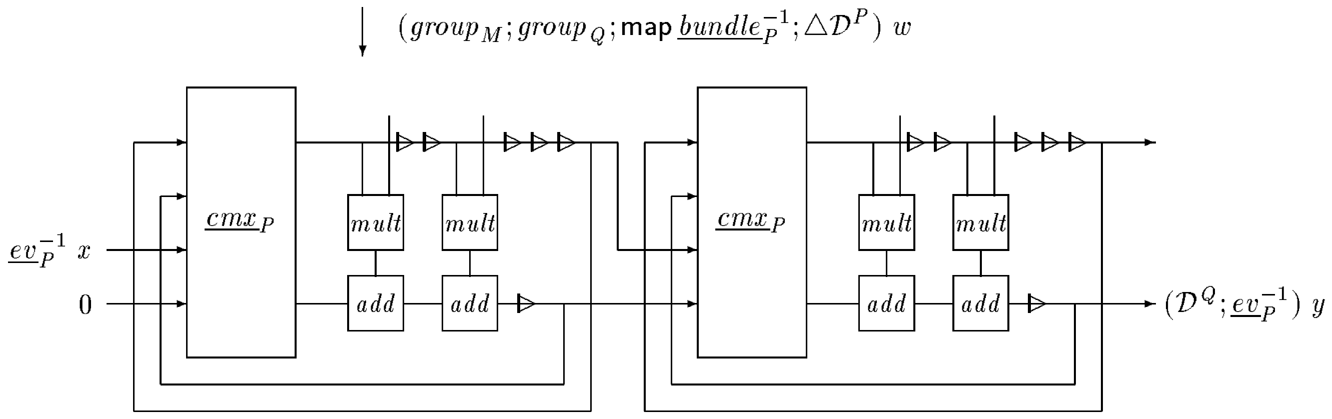

$(group_M; group_Q; \mathsf{map}\ \underline{bundle}_P^{-1}; \triangle\mathcal{D}^P)\ w$
$\underline{cmx}_P$
$mult$
$mult$
add
add
$\underline{cmx}_P$
$mult$
$mult$
add
add
$\underline{ev}_P^{-1}\ x$
0
$(\mathcal{D}^Q; \underline{ev}_P^{-1})\ y$

Figure 17 Design $Cv8$ $(N = 8,\ K = P = Q = 2)$.

$$\begin{tikzcd}
 & Cv1 \arrow[rr, "\text{reverse coefficients}"] \arrow[dl, "\text{LPGS}"'] \arrow[d, "\text{pipelining}"] \arrow[dr, "\text{LSGP}"] & & Cv5 \arrow[d, "\text{LPGS}"] \\
Cv2 & Cv3 \arrow[d, "\text{LPGS}"] & Cv7,\ Cv8 & Cv6 \\
 & Cv4 & &
\end{tikzcd}$$

Figure 18 A design tree summarising how the convolver designs are related to one another.

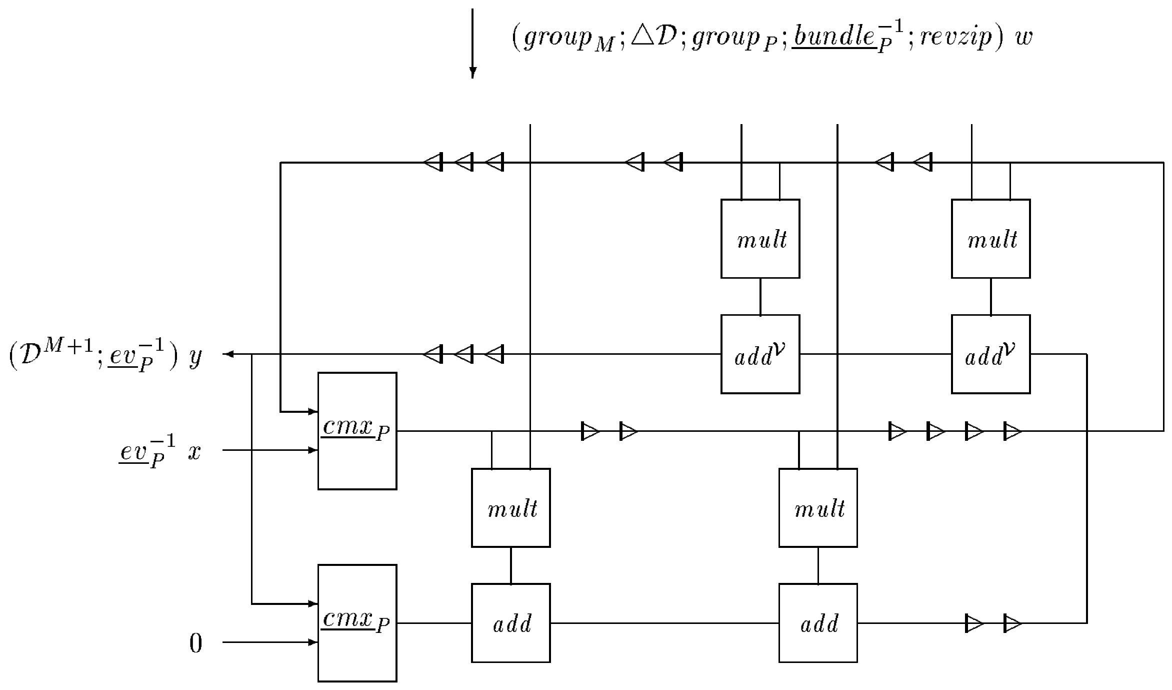

$(group_M; \triangle\mathcal{D}; group_P; \underline{bundle}_P^{-1}; revzip)\ w$
$(\mathcal{D}^{M+1}; \underline{ev}_P^{-1})\ y$
$\underline{ev}_P^{-1}\ x$
0
$\underline{cmx}_P$
$\underline{cmx}_P$
mult
mult
mult
mult
$add^{\mathcal{V}}$
$add^{\mathcal{V}}$
add
add

Figure 19 Transposed version of Design $Cv4$ ($N = 8$, $M = 4$, $K = P = Q = 2$, and $\langle x + y, x\rangle\ add^{\mathcal{V}}\ y$).

Table 1 A feature table comparing convolver designs.

Design	Minimum cycle time	Latency (cycles)	Slow-down factor	Number of *Cvcell* in array	Number of latches in array	Number of $\underline{cmx}$'s in array
$Cv1$	$T_m + NT_a$	$N-1$	1	N	N	0
$Cv2$	$T_m + KT_a$ $+T_f(K)$	$\frac{N^2}{K}$	$\frac{N}{K}$	K	$N+2$	1
$Cv3$	$T_m + KT_a$	$\frac{N(K+1)-K}{K}$	1	N	$\frac{N(K+2)}{K}$	0
$Cv4$	$T_m + KT_a$ $+T_f(N/P)$	$\frac{NP(K+1)}{K}$	P	$\frac{N}{P}$	$\frac{N(K+2)+2K}{K}$	1
$Cv5$	$(N-1)T_p + T_m$ $+T_a$	$2(N-1)$	1	N	N	0
$Cv6$	$(K-1)T_p + T_m$ $+T_a + T_f(K)$	$\frac{N(2N-1)}{K}$	$\frac{N}{K}$	K	$N+2$	1
$Cv7$	$T_m + T_a$ $+T_f(1)$	$K(N-1)+N$	K	$\frac{N}{K}$	$\frac{N(K+2)}{K}$	$\frac{N}{K}$
$Cv8$	$T_m + KT_a$ $+T_f(K)$	$\frac{KP(N-1)+N}{K}$	P	$\frac{N}{P}$	$\frac{N(KP+2)}{KP}$	$\frac{N}{KP}$

N:	the number of stages of convolution (the width of the coefficient stream),
T_m, T_a:	the combinational delay of cell *mult*, *add*,
T_p:	the propagation delay of x across the wiring cell $fork;\mathsf{snd}\,\pi_1$,
$T_f(n)$:	the propagation delay associated with the feedback path (depending on n) and the cycling multiplexer.