ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**Mary Sheeran, Chalmers Technical University, Sweden, and Satnam Singh,
University of Glasgow, UK (Eds)**

Designing Correct Circuits

Proceedings of the 3rd Workshop on Designing Correct Circuits
(DCC96), Båstad, Sweden, 2-4 September 1996

ISBN: 3-540-76102-0

Paper:

Serialising Heterogeneous and Non- Factorisable Processor Arrays

Wayne Luk, Freddy Mang and Chuen Ser Ng

Published in collaboration with the
British Computer Society

Serialising Heterogeneous and Non-Factorisable Processor Arrays

Wayne Luk, Freddy Mang and Chuen Ser Ng

Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, England

Abstract

This paper presents two classes of transformations for serialising an array of processors. The first serialises a heterogeneous array of size mk to one of size m . The second serialises an array from n to m components where m does not need to be a factor of n . Our transformations are expressed in the Ruby language, which provides a concise notation for capturing the serialised array and the boundary conditions; in particular many design constraints can be represented in the form of conjugate $(Q^{-1} ; R ; Q)$ or transposed conjugate $([Q, P]^{-1} ; R ; [P, Q])$ expressions. Design trade-offs in performance and resource requirements of various serialisation schemes are discussed. We illustrate our approach using several arithmetic and recursive filter designs, and their implementation using programmable hardware is outlined.

1 Introduction

A powerful hardware design technique is to emulate a large array of processors by a network of fewer processors with auxiliary components such as buffers and multiplexers. This technique, often known as serialisation [5], offers a development path for producing an implementation which will best match the performance and resource requirements of a given application. Since serial designs tend to have a lower computational capacity but a higher degree of pipelining than the corresponding parallel versions, they are particularly suitable for implementation using field-programmable gate arrays [1] which have a limited size but a relatively large number of registers and multiplexers for pipelining and for implementing control circuitry.

The serialisation transformations discussed in this paper are illustrated in Figure 1 and Figure 2. Figure 1(a) shows an array of four processors, $R_0 \dots R_3$. Its serial version, shown in Figure 1(b), consists of two processors, R_{02} and R_{13} , and there are two control components S and S' selecting data from either the feedback loop or the external connections. One of the transformations discussed in Section 5 can be used to map R_0 and R_2 onto R_{02} , and R_1 and R_3 onto R_{13} ; it also produces the control requirements for S and S' and the boundary conditions so that the behaviour of the serial design can be formally related to that of the original. Figure 2 illustrates another transformation, presented in Section 9, which serialises an array of four halfadders into three; it can be used in deriving the control requirements for the control components $d_0 \dots d_2$ and Q as well as the boundary conditions not shown in the diagram.

Figure 1: (a) An array of four processors. (b) A serial version of (a) which involves only two processors. S and S' are switches selecting data from either the feedback loop or the external connections.

Our transformations are expressed in the Ruby notation ([3], [6]). An algebraic formulation of serialisation in Ruby provides a concise and reusable framework for capturing design descriptions and interface constraints, minimising the

Figure 2: (a) An array of four halfadders. (b) A serial version of (a) which involves only three halfadders. $d_0 \dots d_2$ and Q are control components and the triangular-shaped element denotes a register.

need for the intricate calculations often required when dealing with graph-based models. In particular, an algebraic framework can be used for deriving expressions with parameters which can be varied to produce a wide range of implementations with different trade-offs in performance and resource usage. Previous work on using Ruby for serialisation includes derivation of a bit-serial adder [4], transforming designs by interlacing [8], and a unified treatment of LPGS (Locally Parallel Globally Sequential) and LSGP (Locally Sequential Globally Parallel) partitioning methods [5].

This paper extends the above work in two directions. First, we describe how transformations in [5] can be generalised to cater for heterogeneous processors arrays – arrays of components which can be different from one another – as shown in Figure 1. Second, we present various schemes for serialising a non-factorisable array, such that an array of n processors is transformed into one with only m processors, where $m < n$ and m does not need to be a factor of n ; an example is shown in Figure 2. The trade-offs of these schemes will be discussed.

Serial designs present a challenge to designers since they are less regular and require more complex control than the corresponding parallel versions. A notable feature of our method is the extensive use of conjugate and transposed conjugate expressions (see Section 2), which highlights the symmetry of the serial designs that we derive. The use of conjugation results in compact and general representations for serial designs, and abstract entities (such as *cmx* in Section 4) can be refined to different implementations when the direction of dataflow is taken into account.

In the following we first review the Ruby notation. Various serialisation schemes will then be introduced, and their features will be compared. Our approach will be illustrated by several examples involving recursive filtering and arithmetic, and we outline the implementation of a serial architecture using programmable hardware.

2 Ruby

Ruby is a language of functions and relations. It provides a framework for describing designs and deriving them using correctness-preserving transformations. Further details of Ruby can be found elsewhere (such as [3], [5] and [9]), and the following summarises the main features of Ruby that we use in subsequent sections.

A design in Ruby is described by a binary relation of the form $x R y$ where x and y represent the interface signals, and belong respectively to the domain and range of R . For example, an increment operation can be described by the *inc* relation,

$$x \text{ inc } y \Leftrightarrow x + 1 = y$$

or, more concisely, as $x \text{ inc } (x + 1)$.

Combinators are functions that take one or more relations as arguments and return a relation as result. As an example, the converse of R is defined by

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

Two components Q and R are connected in series to form the composite circuit $Q ; R$ if they share a common hidden interface s (Figure 3a),

$$x (Q ; R) y \Leftrightarrow \exists s : (x Q s) \wedge (s R y).$$

Other ways of combining two components, such as parallel composition (Figure 3b), beside (Figure 3c) and below (Figure 3d), can be defined in a similar way. A component with connections on all four sides is modelled by a relation that relates two-tuples: the two components in the domain correspond to signals for the west and the north sides, and those in the range correspond to signals for the south and the east sides.

Figure 3: Binary compositions: (a) $Q ; R$, (b) $[Q, R]$, (c) $Q \leftrightarrow R$, (d) $Q \downarrow R$.

We shall show later that many of our transformed expressions 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 $Q \setminus P$ when appropriate. Similarly, the transposed conjugate operator is useful for components with two elements in both their domain and range:

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

There are several operations involving pairs of signals. First, given that ι is the identity relation, we have the abbreviations $\text{fst } R = [R, \iota]$ and $\text{snd } R = [\iota, R]$. Next, *swap* is used to reverse the components of a signal, $\langle x, y \rangle \text{ swap } \langle y, x \rangle$, and *fork* is used to duplicate a signal, $x \text{ fork } \langle x, x \rangle$. *mfork_n* is a fork with n -branches instead of two. The pair projection relations π_1 and π_2 , defined by $\langle x, y \rangle \pi_1 x$ and $\langle x, y \rangle \pi_2 y$, relate a signal pair to its first or second component. The tuple projection relation $\epsilon_{k,n}$ relates an n -tuple to its k th element: given $0 \leq k < n$, we have $\langle x_0, x_1, \dots, x_{n-1} \rangle (\epsilon_{k,n}) x_k$. We shall also need left and right shifts which rearrange the brackets:

$$\begin{aligned} \langle \langle x, y \rangle, z \rangle &\text{ lsh } \langle x, \langle y, z \rangle \rangle, \\ \langle x, \langle y, z \rangle \rangle &\text{ rsh } \langle \langle x, y \rangle, z \rangle. \end{aligned}$$

Some of the transformations will require interleaving an array of components from two equal-length tuples. Denoting the number of elements in x as $\#x$, the relation *zip* is given by

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

For instance, $\langle \langle 1, 2, 3 \rangle, \langle 4, 5, 6 \rangle \rangle \text{ zip}_3 \langle \langle 1, 4 \rangle, \langle 2, 5 \rangle, \langle 3, 6 \rangle \rangle$.

3 Array of components

To describe an array of components, we use a higher-order function that takes a mapping from indices to components and returns a binary relation corresponding to the composite design; similar techniques have been used elsewhere

([7],[9]). Given that there are n components $R_0 \dots R_{n-1}$, their repeated series composition is given by the *index chain* combinator (Figure 4a):

$$a \text{ (ichain}_n R) b \Leftrightarrow \exists s : (s_0 = a) \wedge (s_n = b) \wedge (\forall i : 0 \leq i < n : s_i R_i s_{i+1}).$$

A useful shorthand is to denote $\text{ichain}_n R$ by $\text{ichain}_n^i R_i$. For instance, if $R_i = Q^i$, then we can write $\text{ichain}_n R$ as $\text{ichain}_n^i Q^i$.

Repeated parallel composition can be described by the *index map* function: given two tuples x and y such that $\#x = \#y = n$, we have (Figure 4b):

$$x \text{ (imap}_n R) y \Leftrightarrow \forall i : 0 \leq i < n : x_i R_i y_i.$$

Similarly, an *index row* (Figure 4c) can be described by

$$\langle a, x \rangle \text{ (irow}_n R) \langle y, b \rangle \Leftrightarrow \exists s : (s_0 = a) \wedge (s_n = b) \wedge \forall i : 0 \leq i < n : \langle s_i, x_i \rangle R_i \langle y_i, s_{i+1} \rangle.$$

The correctness of these descriptions can be shown using their inductive definitions in terms of the corresponding binary operators such as series and parallel composition, for instance,

$$\begin{aligned} \text{ichain}_0 R &= \iota, \\ \text{ichain}_{n+1} R &= \text{ichain}_n R ; R_n. \end{aligned}$$

To describe arrays of identical components, we can use

$$\begin{aligned} Q^n &= \text{ichain}_n^i Q, \\ \text{map}_n Q &= \text{imap}_n^i Q, \\ \text{row}_n Q &= \text{irow}_n^i Q. \end{aligned}$$

Figure 4: Pictorial descriptions of index combinators: (a) $\text{ichain}_3 R$, (b) $\text{imap}_3 R$, (c) $\text{irow}_3 R$.

4 Sequential designs and serialisation

To deal with sequential designs, a Ruby expression can be interpreted as a relation that relates a *stream* in its domain to a stream in its range. An adder can be described informally in the stream model as

$$\langle \dots, \langle x_0, y_0 \rangle, \langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots \rangle \text{ add } \langle \dots, x_0 + y_0, x_1 + y_1, x_2 + y_2, \dots \rangle$$

Let $x(t)$ as the t -th element from some reference point – such as the time when the design is initialised – in the stream x . If $x(t)$ is a tuple, then $x(t)_i$ is its i -th element. Then the adder can be described by

$$x \text{ add } y \Leftrightarrow \forall t : x(t)_0 + x(t)_1 = y(t).$$

There are two primitives that can only be interpreted in the stream model. The first 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. A delay is represented by a triangular-shaped element in the figures (see Figure 2b).

The second primitive is $\mathcal{B}_n$ (known as bundle in [4] and [5]), which relates serial and parallel data:

$$x \mathcal{B}_n y \Leftrightarrow \forall t, i : 0 \leq i < n : x(nt+i) = y(t)_i.$$

For example,

$$\langle \dots, x_0, x_1, x_2, x_3, \dots \rangle \mathcal{B}_3 \langle \dots, \langle x_0, x_1, x_2 \rangle, \langle x_3, x_4, x_5 \rangle, \dots \rangle.$$

Several more constructs will prove convenient for transforming a parallel design into its serial counterpart. All of these constructs can be expressed in terms of the Ruby definitions presented earlier [5]. Their names are underlined to indicate that they do not have a combinational interpretation.

The first is a sampler, $\underline{ev}_n$ (read as every n -th), whose range can be regarded as a sampled version of the domain every n cycles:

$$x \underline{ev}_n y \Leftrightarrow \forall t : x(nt) = y(t).$$

For example,

$$\langle \dots, x_0, x_1, x_2, \dots \rangle \underline{ev}_3 \langle \dots, x_0, x_3, x_6, \dots \rangle$$

The second component is a cycling multiplexer, $\underline{cmx}_n$. It relates the range signal to the first domain component at every n -th cycle, and to the second domain component at other cycles:

$$x \underline{cmx}_n y \Leftrightarrow \forall t : (x(nt)_0 = y(nt)) \wedge (\forall i : 1 \leq i < n : x(nt+i)_1 = y(nt+i)).$$

For instance,

$$\langle \dots, \langle x_0, y_0 \rangle, \langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots \rangle \underline{cmx}_3 \langle \dots, x_0, y_1, y_2, x_3, y_4, y_5, \dots \rangle$$

Clearly a $\underline{cmx}$ can be implemented using multiplexers with the selection signal coming from a chain of suitably initialised registers in a loop.

Next, it can be shown that a design Q conjugated by $\mathcal{B}_n^{-1}$ is the same as one obtained by replacing each $\mathcal{D}$ in Q by n copies of $\mathcal{D}$'s in series, and similarly for each $\mathcal{D}^{-1}$ in Q . This is commonly known as slowdown, defined by

$$\text{slow}_n Q = (\text{map}_n Q) \setminus \mathcal{B}_n^{-1}.$$

A slowdown design can cope with n -interleaved computations, and is often produced as a result of pipelining [3]. The heterogeneous version of $\text{slow}_n Q$ can be obtained by substituting $\text{imap}_n^i Q_i$ for $\text{map}_n Q$:

$$\text{islow}_n^i Q_i = (\text{imap}_n^i Q_i) \setminus \mathcal{B}_n^{-1}.$$

This design behaves like $Q_0, \dots, Q_{n-1}$ at successive cycles and then repeats. One can regard it as a dynamically reconfigurable circuit ([2], [5]).

An important feature of any serialisation scheme is the presence of feedback loops. A design R containing an internal feedback path s can be modelled by the loop construct φ (Figure 5a):

$$x (\varphi R) y \Leftrightarrow \exists s : \langle x, s \rangle R \langle y, s \rangle.$$

Another feedback configuration (Figure 5b) may sometimes be convenient:

$$x (\mu R) y \Leftrightarrow \exists s : \langle s, x \rangle R \langle y, s \rangle.$$

Figure 5: Functions that describe feedback configurations. (a) $x (\varphi R) y$, (b) $x (\mu R) y$.

5 Serialising factorisable arrays

An array of $n = mk$ processors can be serialised to produce an array of m slowdown processors with feedback (Figure 6). A delay is introduced for storing intermediate results, and cycling multiplexers are used to select the external connections every k -th cycle for a new emulation:

$$\begin{aligned} R^{mk} &= \underline{ev}_k^{-1} ; \varphi(\underline{cmx}_k ; \text{slow}_k R^m ; \mathcal{D} ; \underline{cmx}_k^{-1}) ; \underline{ev}_k ; \mathcal{D}^{-1} \\ &= (\varphi((\text{slow}_k R^m ; \mathcal{D}) \setminus \underline{cmx}_k^{-1})) \setminus \underline{ev}_k ; \mathcal{D}^{-1}. \end{aligned} \quad (1)$$

An instance of the right-hand side of Equation 1 is shown in Figure 6. This equation is a generalised version of a similar one introduced in [5], which cannot deal with designs with counter-flowing data streams; further explanations and examples can be found in [5].

Figure 6: A serial circuit for emulating R^{mk} which can accommodate counter-flowing data streams. The delay does not allow data to flow from right to left, but it can be cancelled by an anti-delay; see Example 1.

Notice that $fork$ is an appropriate implementation of $\underline{cmx}_k^{-1}$ when data flow from domain to range, since $fork ; \underline{cmx}_k = \iota$. This observation gives us the flexibility of dealing with designs with counter-flowing data streams: one can implement $\underline{cmx}_k^{-1}$ by $fork$ when data flow from domain to range, and implement $\underline{cmx}_k$ by $fork^{-1}$ when data flow from range to domain.

Figure 7: The component $\underline{cmx}_k$ (a) can be implemented by a $fork^{-1}$ (b) if data flow from right to left.

To accommodate designs with heterogeneous components, Equation 1 can be generalised by substituting $\text{ichain}_{mk}^i R_i$ for R^{mk} , and $\text{ichain}_m^i (\text{islow}_k^j R_{j+m+i})$ for $\text{slow}_k R^m$,

$$\text{ichain}_{mk}^i R_i = \varphi(\underline{cmx}_k ; \text{ichain}_m^i (\text{islow}_k^j R_{j+m+i}) ; \mathcal{D} ; \underline{cmx}_k^{-1}) \setminus \underline{ev}_k ; \mathcal{D}^{-1}. \quad (2)$$

$$\text{islow}_k^j R_j = \mathcal{B}_k ; \text{imap}_k^j R_j ; \mathcal{B}_k^{-1}. \quad (3)$$

Consider the behaviour of $\text{islow}_k^j R_j$: each processor behaves differently at each cycle (see Figure 8). One can check that if all R_i 's are identical, this equation becomes Equation 1.

Figure 8: (a) The figure shows $\text{imap}_k^j (\text{ichain}_m^i R_{j+m+i}) \setminus \mathcal{B}_k^{-1}$, with $k = 4$, $m = 3$. The dotted box indicates that the second processor behaves as R_1, R_4, R_7 and R_{10} in successive cycles. The shaded squares show how the three processors behave at a particular cycle. (b) If all R s are identical, the structure in (a) reduces to a chain of R , provided that R is combinational.

Let us consider an example to illustrate how these transformations can be applied.

Example 1. A linear recursive filter with time-invariant coefficients can be described as follows: given the data stream x , and the coefficients $a_1, a_2, \dots, a_n$ and $b_1, b_2, \dots, b_n$, the filter is to compute y_t , given by:

$$\forall t. y_t = \sum_{1 \leq i \leq n} a_i \times y_{t-i} + \sum_{1 \leq i \leq n} b_i \times x_{t-i}$$

In Ruby, a fully parallel recursive filter can be described as (Figure 9):

$$RF = \text{ichain}_n^i (R_i ; \text{snd}(\text{fst } \mathcal{D}^{-1})) ; F$$

Figure 9: An instance of a recursive filter RF , with $n = 4$.

where R_i (Figure 10a) and F satisfy (the detailed definitions are not given here since we are only interested in the behaviour of the cells):

$$\begin{aligned} \langle y, \langle u, x \rangle \rangle R_i \langle y', \langle u, x' \rangle \rangle & \quad \text{where } y' = b_{n-i} \times u + y \\ \text{and } x' & = a_{n-i} \times u + x, \\ \langle y, \langle x, x \rangle \rangle F y & \end{aligned}$$

Figure 10: An instance of a cell for recursive filter RF , (a) R_i (b) R'_i with $n = 4$ and $m = k = 2$.

Given $n = mk$, let us serialise the recursive filter design using Equation 2,

$$\begin{aligned}
 RF &= \text{ichain}_n^i(R_i ; \text{snd}(\text{fst } \mathcal{D}^{-1})) ; F \\
 &= \text{ichain}_{mk}^i(R_i ; \text{snd}(\text{fst } \mathcal{D}^{-1})) ; F \\
 &= \varphi(\underline{cmx}_k ; \text{ichain}_m^i(\text{islow}_k^j R_{jm+i} ; \text{snd}(\text{fst } \mathcal{D}^{-k})) ; \mathcal{D} ; \underline{cmx}_k^{-1} \setminus \underline{ev}_k ; \mathcal{D}^{-1} ; F \\
 &= \varphi(\underline{cmx}_k ; \text{ichain}_m^i(R'_i ; \text{snd}(\text{fst } \mathcal{D}^{-k})) ; \mathcal{D} ; \underline{cmx}_k^{-1} \setminus \underline{ev}_k ; \mathcal{D}^{-1} ; F
 \end{aligned}$$

where $R'_i = \text{islow}_k^j R_{jm+i}$. R'_i should satisfy (Figure 10b):

$$\begin{aligned}
 \langle y_t, \langle u_t, x_t \rangle \rangle R'_i \langle y'_t, \langle u_t, x'_t \rangle \rangle \quad \text{where} \quad y'_t &= b_\tau \times u_t + y_t, \\
 x'_t &= a_\tau \times u_t + x_t \\
 \text{and} \quad \tau &= ((n - i - 1) - tm) \bmod n.
 \end{aligned}$$

Next, the direction of data flow is identified, and $\underline{cmx}_k^{-1}$ is replaced by *fork* (or appropriately, $\underline{cmx}_k$ by fork^{-1}). Given that $\underline{cmx}_k$ is polymorphic,

$$[\text{map}_2 \iota, \text{map}_2(\text{map}_2 \iota)] ; \underline{cmx}_k = \text{zip}_2 ; \text{snd } \text{zip}_2 ; [\underline{cmx}_k, \text{map}_2 \underline{cmx}_k].$$

Hence

$$\begin{aligned}
 RF &= \varphi(\text{zip}_2 ; \text{snd } \text{zip}_2 ; [\underline{cmx}_k, \text{map}_2 \underline{cmx}_k] ; \text{ichain}_m^i(R'_i ; \text{snd}(\text{fst } \mathcal{D}^{-k})) ; \mathcal{D} ; \\
 &\quad [\underline{cmx}_k^{-1}, \text{map}_2 \underline{cmx}_k^{-1}] ; \text{snd } \text{zip}_2^{-1} ; \text{zip}_2^{-1} \setminus \underline{ev}_k ; \mathcal{D}^{-1} ; F. \\
 &= \varphi(\text{zip}_2 ; \text{snd } \text{zip}_2 ; [\underline{cmx}_k, [\text{fork}^{-1}, \underline{cmx}_k]] ; \text{ichain}_m^i(R'_i ; \text{snd}(\text{fst } \mathcal{D}^{-k})) ; \mathcal{D} ; \\
 &\quad [\text{fork}, [\underline{cmx}_k^{-1}, \text{fork}]] ; \text{snd } \text{zip}_2^{-1} ; \text{zip}_2^{-1} \setminus \underline{ev}_k ; \mathcal{D}^{-1} ; F. \\
 &= (\varphi(\text{zip}_2 ; \text{snd } \text{zip}_2 ; [\underline{cmx}_k, [\text{fork}^{-1}, \underline{cmx}_k]] ; \text{ichain}_m^i(R'_i ; \text{snd}(\text{fst } \mathcal{D}^{-k})) ; \mathcal{D} ; \\
 &\quad [\text{fork}, [\underline{cmx}_k^{-1}, \text{fork}]] ; \text{snd } \text{zip}_2^{-1} ; \text{zip}_2^{-1}) ; F) \setminus \underline{ev}_k ; \mathcal{D}^{-1}.
 \end{aligned}$$

The core of the serialised design is shown in Figure 11.

6 Simple methods for serialising non-factorisable arrays

The method proposed in Section 5 can only be applied to factorisable arrays. This section explores two methods of serialising non-factorisable arrays, which are simple but possibly inefficient. Another method will be introduced in the next section. In all three cases a design of n components is serialised to use only m components.

Figure 11: An instance of a serial recursive filter with only two processors. The two unlabelled components on the left are $\underline{cmx}_k$, and the one on the right is $\underline{cmx}_k^{-1}$. Note that the delay and the anti-delay in dotted lines cancel each other, resulting in an implementable design. Interface constraints are not shown in the diagram.

Given that $n = ab + c$, such that $0 \leq c < b$ and $b + c = m$, the number of processors in the transformed array, we can apply Equation 1 to serialise the first ab processors to b processors; another c processors are required to complete the emulation (Figure 12). For instance, given that R is combinational, we have:

$$\begin{aligned} R^{ab+c} &= \varphi(\underline{cmx}_a ; R^b ; \mathcal{D} ; \underline{cmx}_a^{-1}) \setminus \underline{ev}_a ; \mathcal{D}^{-1} ; R^c \\ &= (\varphi(\underline{cmx}_a ; R^b ; \mathcal{D} ; \text{fork}) ; R^c) \setminus \underline{ev}_a ; \mathcal{D}^{-1}. \end{aligned} \quad (4)$$

However, it is not difficult to see that this method can be inefficient in processor usage, since effectively c processors are used only once every a cycles.

Figure 12: A simple but possibly inefficient serialisation scheme. There are m processors in this system: $b + c = m$.

Let $n = mq + r$, such that $0 \leq r < m$. A more sophisticated scheme can be obtained by accommodating, for every q cycles each involving m processors, an extra cycle that iterates through r of the m processors (Figure 13). This requires a component which selects one input for q cycles and then the other input for one cycle. The component that we introduce is known as a *delayed* cycling multiplexer, a cycling multiplexer whose range corresponds to the second component of its domain for the first d cycles:

$$\underline{dcm}_{n,d} = \underline{cmx}_n \setminus \mathcal{D}^{-(n-d)}.$$

For instance,

$$\langle \dots, \langle x_0, y_0 \rangle, \langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots \rangle \underline{dcm}_{3,2} \langle \dots, y_0, y_1, x_2, y_3, y_4, x_5, y_6, y_7, x_8, \dots \rangle$$

Like $\underline{cmx}$, $\underline{dcm}$ can be implemented by fork^{-1} when data flow from range to domain, and by a hardware multiplexer when data flow from domain to range.

The delayed cycling multiplexer that we require for the serial array is $\underline{dcm}_{q+1,q}$. One can then partition the m processors into r processors. We shall also need the conditional involvement of another $m - r$ processors, with the selection controlled by the component $\underline{dcm}_{q+1,q}$:

$$R^n = \varphi(\underline{cmx}_{q+1} ; R^r ; (\text{snd } R^{m-r}) \setminus \underline{dcm}_{q+1,q} ; \mathcal{D} ; \underline{cmx}_{q+1}^{-1}) \setminus \underline{ev}_{q+1} ; \mathcal{D}^{-1}. \quad (5)$$

Figure 13: Another scheme for serialising non-factorisable arrays that is more efficient. The architecture contains both feedforward and feedback connections.

Although this solution involves fewer processors than the previous scheme and requires only two cycling multiplexers, there is still a source of inefficiency because for every $q + 1$ cycles, $m - r$ processors are not used in the last cycle.

For both serialisation schemes that we consider above, a new emulation cycle always begins with a processor at the boundary, which is connected via a $\underline{cmx}$ component to the external interface. In the following section we show that relaxing this condition can lead to more efficient implementations.

7 Serialising non-factorisable arrays efficiently

This section presents an efficient scheme for serialising arrays of combinational elements – its extension to deal with sequential components requires the introduction of further concepts and will not be attempted here. The basic idea is that a new emulation of a large array by a small one can be initiated immediately at the next available processor, which may be situated in the middle of the serialised array. Hence multiplexers have to be inserted along the chain of processors to allow external inputs and outputs at an internal processor. There may be additional wiring for the serialised design which can result in routing congestions. However this may not be a serious drawback for multi-layer printed circuit boards or field-programmable gate arrays with a hierarchical routing network [1].

Let us elaborate this method for the case of serialising a chain from length n to m . The two key questions are: (a) which processors will need external connections, and how should they be modified to accommodate external connections? (b) What are the interface constraints that relate the original design and the serialised version?

To answer question (a), let $p = \gcd(m, n)$, the greatest common divisor of m and n . Intuitively one can imagine dividing up the m -element chain into chains of length p each with external connections. Label the processors C_i in the serialised array from $i = 0$ to $i = m - 1$ (Figure 14). The domain of processor i will need to connect to the external interface if i is divisible by p . Since the initiation of another emulation must be preceded by the termination of the previous emulation, the range of processor i will need to connect to the external interface if $i + 1$ is divisible by p . To simplify the description, we shall include components (called S_i and T_i , shown in Figure 15) which connect every element in the chain to the external interface. We then specify the conditions under which some of these components can be reduced to floating wires, and can be eliminated when the processors they connected to do not require external connection.

The answer to question (b) can be structured into two parts. The first part involves the component P^{-1} in the domain and P in the range (Figure 14) to convert between serial and parallel data, and to introduce additional data values which will be ignored when the serialised array is operating in the feedback mode. The second part involves the component WD in the domain and WR in the range to manage floating wires so that they are consistent with those from the S_i and T_i components, and to route the external connections to the appropriate processor within the array.

Below we present a formula for transforming a chain of n processors into m processors. Given that $m' = m/p$ and $n' = n/p$, one can show that (Figure 14):

$$\begin{aligned} R^n &= (WD ; \mu (\text{irow}_m^i C_i ; \text{snd } \mathcal{D}) ; WR) \setminus P. \\ C_i &= S_i ; R ; T_i. \end{aligned} \quad (6)$$

The core of the transformed array is a row of m processors, each of which consists of an R component that is inserted between two control blocks S_i and T_i , where $0 \leq i < m$. P and P^{-1} represent the interface constraint at the range and the domain, where WD (Wiring at Domain) and WR (Wiring at Range) capture the wiring interface constraint.

Figure 14: This diagram shows $(WD ; \mu (\text{irow}_m^i C_i ; \text{snd } \mathcal{D}) ; WR) \setminus P$, a serial version for emulating R^n , where $m = 3$. Details of C_i are described in Figure 15; implementation examples of P and P^{-1} are described in Figure 16 and Figure 20.

Before describing the components that control the connection to the external interface, we need to introduce $\underline{fcm}_{n,d}$, a delayed cycling multiplexer (defined in the previous section) with a period of n cycles which has an additional d delays at its first domain component:

$$\begin{aligned} \underline{fcm}_{n,d} &= \text{fst } \mathcal{D}^d ; \underline{dcm}_{n,d} \\ &= [\mathcal{D}^n, \mathcal{D}^{n-d}] ; \underline{cmx}_n ; \mathcal{D}^{-(n-d)}. \end{aligned}$$

For instance,

$$\langle \dots, \langle x_0, y_0 \rangle, \langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle, \dots \rangle \underline{fcm}_{3,2} \langle \dots, y_0, y_1, x_0, y_3, y_4, x_3, y_6, y_7, x_6, \dots \rangle$$

The $\underline{fcm}$ component controls the selection of whether to connect to the preceding processor (the second domain component) or to the external interface. Given $\text{lcm}(m, n)$ represents the least common multiple of m and n , a processor requiring external connection does so every $\text{lcm}(m, n)/m = n'$ cycles. For the i -th processor in the serialised chain, the control block S_i provides an external connection for the domain of R at cycle u_i if $(u_i m + i)$ is a multiple of m :

$$S_i = \begin{cases} \text{swap} ; \underline{fcm}_{n', u_i} & \text{if } p \text{ divides } i, \\ \pi_1 & \text{otherwise,} \end{cases} \quad (7)$$

Figure 15: (a) Each C_i consists of S_i , R and T_i . Two possible implementations for S_i are shown in (b) and (c), and the possible implementations for T_i are shown in (d) and (e). (b) fcm is used if there is external connection to this processor. (c) If there is no external connection, a floating wire is introduced. (d) Anti-delays are introduced when output is required at this processor. (e) If output is not required, a floating wire is introduced. See Examples 2 (Figure 16) and 3 (Figure 20).

where $(u_i m + i) \bmod n = 0$.

Similarly, the control block T_i for the range of R is given by

$$T_i = \begin{cases} \text{fork} ; \text{fst } \mathcal{D}^{-v_i} & \text{if } p \text{ divides } i + 1, \\ \pi_2^{-1} & \text{otherwise,} \end{cases} \quad (8)$$

where $(v_i m + i + 1) \bmod n = 0$.

Each processor R is connected by an S_i block to the external interface P through the intermediate wiring component WD , and by a T_i block to P^{-1} through WR . When an emulation cycle is completed, an output must be followed by an input to initiate the next emulation cycle; this is reflected in Equation 7 which describe external connections for input when p divides i , and in Equation 8 for output when p divides $i + 1$. Otherwise, floating wires are introduced to ensure interface consistency.

$$P^{-1} = \mathcal{B}_{m'} ; \text{map}_{m'} \underline{ev}_{n'}^{-1}. \quad (9)$$

$$P = \text{map}_{m'} \underline{ev}_{n'} ; \mathcal{B}_{m'}^{-1}. \quad (10)$$

The boundary conditions P^{-1} and P (Equations 9 and 10) include the components $\mathcal{B}_{m'}$ and $\mathcal{B}_{m'}^{-1}$, which convert data from serial to parallel and vice versa, and $\underline{ev}_{n'}^{-1}$ and $\underline{ev}_{n'}$, which respectively injects and rejects added data when the processors are in the feedback mode. The components WD and WR are given by:

$$\begin{aligned} WD &= WW^v ; WDF, \\ WR &= WRF ; WW. \end{aligned} \quad (11)$$

The component WW , and its reflection in a vertical axis WW^v , are to compute the correct order of external connections to the array,

$$\langle dom_0, dom_1, \dots, dom_{m'-1} \rangle WW \langle ran_0, ran_1, \dots, ran_{m'-1} \rangle, \quad (12)$$

where $ran_t = dom_s$ and $t = (v_s m' + s - (n' - 1)) / n'$. When $t = s$ as in Example 3, then WW^v and WW are reduced to identity.

Figure 16: Emulation of a chain of five incrementers by three incrementers. The components labelled S_i in the diagram correspond to $swap ; \underline{fcm}_{5,u_i}$.

The wiring relations WDF and WRF are given by:

$$WDF = (mfork_{m'} ; [\epsilon_{0,m}, \epsilon_{p,m}, \epsilon_{2p,m}, \epsilon_{3p,m}, \dots, \epsilon_{xp,m}])^{-1}, \quad (13)$$

$$WRF = mfork_{m'} ; [\epsilon_{p-1,m}, \epsilon_{2p-1,m}, \epsilon_{3p-1,m}, \dots, \epsilon_{m-1,m}], \quad (14)$$

where x is the largest integer satisfying $xp \leq m$. Recall also that $\epsilon_{j,m}$ selects the j -th element ($0 \leq j < m$) from a sequence of size m .

Since input is only required at the i -th processor when p divides i , the purpose of WDF is to connect the inputs to the required processors and add floating wires to those which do not require input. Similarly the wiring pattern WRF removes the floating wires leaving only the relevant connections to the outputs (see Examples 2 and 3).

When n and m are co-prime such that $p = \gcd(m, n) = 1$ as in Example 2, then all values of i and $i + 1$ are divisible by p . In other words, external connections are required for all S_i and T_i components and floating wires are not required. This is in contrast with the situation in Example 3 where $p \neq 1$, so inputs are only required at even S_i , outputs are only required at odd T_i , and floating wires are introduced for the remaining control components.

Example 2. To serialise a chain of five incrementers into three, apply Equation 6 with $n = 5$, $m = 3$ and $R = inc$. Since $p = \gcd(5, 3) = 1$, $m' = m = 3$ and $n' = n = 5$, we deduce from Equations 7 and 8 that there are connections to all the processors within the array (Figure 16) with $u_0 = 0, u_1 = 3, u_2 = 1$, and $v_0 = 3, v_1 = 1, v_2 = 4$. Figure 17 shows the domain parts in greater detail: the data values of the wires $w_0, w_1, \dots, w_9$ at successive clock cycles are given in Table 1.

Cycle number	w_0	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9
0	x_0	x_0	x_1	x_2	x_0	x_1	x_2	$f(x_0)$	$f^2(x_0)$	$f^3(x_0)$
1	x_1	x_3	x_4	x_5	$\perp$	$\perp$	$\perp$	$f^4(x_0)$	$f^5(x_0)$	$f(x_1)$
2	x_2	x_6	x_7	x_8	$\perp$	$\perp$	$\perp$	$f^2(x_1)$	$f^3(x_1)$	$f^4(x_1)$
3	x_3	x_9	x_{10}	x_{11}	$\perp$	$\perp$	$\perp$	$f^5(x_1)$	$f(x_2)$	$f^2(x_2)$
4	x_4	x_{12}	x_{13}	x_{14}	$\perp$	$\perp$	$\perp$	$f^3(x_2)$	$f^4(x_2)$	$f^5(x_2)$
5	x_5	x_{15}	x_{16}	x_{17}	x_3	x_4	x_5	$f(x_3)$	$f^2(x_3)$	$f^3(x_3)$
6	x_6	x_{18}	x_{19}	x_{20}	$\perp$	$\perp$	$\perp$	$f^4(x_3)$	$f^5(x_3)$	$f(x_4)$
7	x_7	x_{21}	x_{22}	x_{23}	$\perp$	$\perp$	$\perp$	$f^2(x_4)$	$f^3(x_4)$	$f^4(x_4)$
8	x_8	x_{24}	x_{25}	x_{26}	$\perp$	$\perp$	$\perp$	$f^5(x_4)$	$f(x_5)$	$f^2(x_5)$
9	x_9	x_{27}	x_{28}	x_{29}	$\perp$	$\perp$	$\perp$	$f^3(x_5)$	$f^4(x_5)$	$f^5(x_5)$
10	x_{10}	x_{30}	x_{31}	x_{32}	x_6	x_7	x_8	$f(x_6)$	$f^2(x_6)$	$f^3(x_6)$

Table 1: This table shows the values at successive clock cycles on the wires $w_0, \dots, w_9$ in Figure 17. ‘Don’t care’ values $\perp$ are due to the $\underline{ev}_5^{-1}$ components. We define f by $x \text{ inc }^j y \Leftrightarrow \forall t : y_t = f^j(x_t)$.

8 Comparing serialisation methods

Table 2 summarises the features of the three serialisation schemes (Equations 4, 5 and 6) described in the last two sections. One would expect that the fastest serial circuit should be able to achieve n/m cycles per emulation, where m is the size of the serial array and n is the size of the array being emulated. This optimal performance is achieved by the third scheme: it can be shown that the first scheme requires $c(n - m)/(m(m - c))$ more cycles per emulation than the third scheme, and the second scheme requires $(m - r)/m$ more cycles per emulation than the third scheme.

However, the third scheme needs additional multiplexers and connection paths when compared with the other two schemes. In particular, it requires more intensive wiring since inputs may have to be distributed to some processors in the serialised array; for instance when m and n are coprime, the inputs will need to be distributed to all processors in the array. This may not be a serious concern for multi-layer printed circuit boards or field-programmable gate arrays with a hierarchical routing network [1].

The first scheme has the advantage that only a simple feedback arrangement is required. The second scheme requires both feedforward and feedback connections. The second scheme may be a good compromise between the first and the third scheme: although it is not as fast as the third scheme, it is faster than the first scheme and it uses less resources than the third scheme.

Design	Number of processors	Number of multiplexers	Critical path	Cycles per emulation
R^n	n	0	$n \times T_R$	1
Eqn. 4	m	1	$\max((m - c) \times T_R + T_c, c \times T_R)$	$(n - c)/(m - c)$
Eqn. 5	m	2	$m \times T_R + 2 \times T_c$	$q + 1$
Eqn. 6	m	m'	$m \times T_R + m' \times T_c$	n/m

m' is equal to $m/\gcd(m, n)$.

c : the number of processors outside the loop in Figure 12.

T_R : combinational delay due to the processor R .

T_c : combinational delay due to a cycling multiplexer.

Cycles per emulation: the average number of cycles the serial design takes to emulate the corresponding parallel design.

Table 2: Emulation of n processors by m processors using three serialisation schemes, where $n = mq + r$ and that $0 \leq r < m$.

9 Serialising a non-factorisable row of components

Equation 6 can be generalised to give a formula for serialising a row of components:

$$\begin{aligned} \text{row}_n R &= (\text{fst } WD ; (\mu (\text{irow}_m^i C_i ; \text{snd } \mathcal{D})) \setminus \text{zip}_m^{-1} ; \text{snd } WR) \parallel [\mathcal{BB}_{n,m}^{-1}, P]. \\ C_i &= S_i ; R ; T_i. \end{aligned} \quad (15)$$

All the expressions are as defined in the previous section apart from S_i and T_i ,

$$S_i = \begin{cases} \text{rsh} ; \text{fst } (\text{swap} ; \underline{fcm}_{n', u_i}) & \text{if } p \text{ divides } i, \\ \text{rsh} ; \text{fst } \pi_1 & \text{otherwise,} \end{cases} \quad (16)$$

$$T_i = \begin{cases} \text{snd } (\text{fork} ; \text{fst } \mathcal{D}^{-v_i}) ; \text{rsh} & \text{if } p \text{ divides } i + 1, \\ \text{snd } \pi_2^{-1} ; \text{rsh} & \text{otherwise,} \end{cases} \quad (17)$$

where $(u_i m + i) \bmod n = 0$ and $(v_i m + i + 1) \bmod n = 0$.

Note that we need a component more general than $\mathcal{B}$ to capture the interface constraints: one that converts between a stream of n -tuples and a stream of m -tuples,

$$x \mathcal{BB}_{n,m} y \Leftrightarrow \forall t, i : 0 \leq i < n : x(t)_i = y((nt + i) \text{ div } m)_{(nt+i) \bmod m}.$$

It can be defined by $\mathcal{BB}_{n,m} = \mathcal{B}_n^{-1} ; \mathcal{B}_m$. For example,

$$\langle \dots, \langle x_0, y_0, z_0 \rangle, \langle x_1, y_1, z_1 \rangle, \dots \rangle \mathcal{BB}_{3,2} \langle \dots, \langle x_0, y_0 \rangle, \langle z_0, x_1 \rangle, \langle y_1, z_1 \rangle, \dots \rangle.$$

Figure 17: Effect of boundary condition (P^{-1} ; WD) in Example 2: the data values on the wires labelled $w_0, \dots, w_9$ are shown in Table 1. Each $fcm_{5,d}$ delays its first input by d cycles.

Figure 18: A serial design which emulates a row of n processors by m processors, where $n = 5$ and $m = 3$. Components labelled C_i are described in Figure 19. BB represents a component involving parallel-serial conversion.

Figure 19: The component C_i for Equation 15.

Example 3. To implement a digit incrementer of length six using only four halfadders, apply Equation 15 with $n = 6$, $m = 4$ and $R = \text{hadd}$ (Figure 20). Then $p = \gcd(m, n) = 2$, $n' = 3$ and $m' = 2$. Note that since $p = 2$, cycling multiplexers and external connections for inputs exist only at positions with even indices: $u_0 = 0$ and $u_2 = 1$. Anti-delays and external connections for outputs exist only at positions with odd indices: $v_1 = 1$ and $v_3 = 2$ (see Equations 16 and 17). Figure 21 shows how the core of the array hadds in Figure 20 can be realised in hardware; in particular the component fcm can be implemented by a multiplexer controlled by a counter. Figure 22 illustrates the implementation of this circuit in Xilinx 6200 programmable hardware technology: each cell in this technology can be programmed to be a two-input logic gate, or a multiplexer, or a D latch [1]. While there is a hierarchical network of connections in the 6200 architecture, only nearest-neighbour connections are used in our implementation.

Figure 20: Serialising a row of six halfadders to four halfadders.

 Figure 21: A hardware implementation of the core of the serial array *hadds* in Figure 20, without the anti-delays. The initial values of the latches are labelled on top of their triangular symbol. The input and output wire names are the same as those labelled in Figure 20.

Figure 22: The implementation of the serial halfadder array in Xilinx 6200 programmable hardware technology. The component D_T corresponds to a D latch initialised to one, while the component D_F corresponds to a D latch initialised to zero.

10 Summary

The purpose of our work is to study flexible and efficient techniques for serialising array-based designs. In this paper we introduce various strategies for serialising heterogeneous processor arrays and non-factorisable arrays, and we discuss their trade-offs and their implementation. The refinement, verification and unification of these strategies are being explored, and we have been extending the treatment of non-factorisable arrays to include sequential components. Prototype software has been developed to automate some of the calculations involved in the proposed serialisation schemes.

Further work includes widening the range of solutions that satisfy given constraints, automating the application and verification of serialisation transformations, covering designs of a greater variety, and evaluating our approach using realistic applications and implementations.

Acknowledgements

Thanks to Koala Choi, Teddy Wu and Kenny Hui for their help, to Mike Dean, Geraint Jones, Ole Rasmussen and the referees for their comments and suggestions, to Xilinx Development Corporation and others who have been supporting us throughout this project.

References

- [1] Churcher S, Kean T, Wilkie B. The XC6200 FastMap Processor Interface. In: Moore W, Luk W (eds) Field Programmable Logic and Applications. LNCS 975, Springer, 1995, pp 36-43
- [2] Hogg J, Singh S, Sheeran M. New HDL research challenges posed by dynamically reprogrammable hardware. In: Proc. APCHDL'96, 1996
- [3] Jones G, Sheeran M. Circuit design in Ruby. In: Staunstrup J (ed) Formal Methods for VLSI Design. North-Holland, 1990, pp 13-70

- [4] Jones G, Sheeran M. Deriving bit-serial circuits in Ruby. In: Halaas A, Denyer PB (eds) Proc. VLSI 91. 1991, pp 3a.3.1-3a.3.10
- [5] Luk W. Systematic serialisation of array-based architectures. In: Integration. Vol 14, No 3, 1993, pp 333-360
- [6] Luk W. A declarative approach to incremental custom computing. In: Athanas P, Pocek KL (eds) Proc. FCCM95. IEEE Computer Society Press, 1995, pp 164-172
- [7] Luk W. Optimizing designs by transposition. In: Jones G, Sheeran M (eds) Designing Correct Circuits. Springer-Verlag, 1991, pp 332-354
- [8] Sandum O. Multiple clocks and Ruby. Technical Report ID-TR:1994-153, Department of Computer Science, Technical University of Denmark. August 1994
- [9] Sharp R, Rasmussen O. Using a language of functions and relations for VLSI specification. In: Peyton Jones S (ed) FPCA'95. ACM, 1995, pp 45-54