Journal of VLSI Signal Processing, 5, 7–20 (1993)

Pipelining and Transposing Heterogeneous Array Designs

WAYNE LUK

Programming Research Group, Oxford University Computing Laboratory, 11 Keble Road, Oxford, England OX1 3OD

Received December 12, 1991; Revised May 21, 1992.

Abstract. This paper describes a scheme for representing heterogeneous array circuits, in particular those which have been optimized by pipelining or by transposition. Equations for correctness-preserving transformations of these parametric representations are presented. The method is illustrated on developing novel pipelined designs for parallel division. It is estimated that, for a field-programmable gate array implementation, the speed of an integer divider can be doubled at the expense of a 50 percent increase in area.

1. Introduction

The regularity and modularity of array-based circuits offer two important advantages: they facilitate the efficient implementation of these circuits in VLSI technology, and they simplify their description and transformation so that optimized designs can be built rapidly and correctly. We have presented an algebraic framework and the related computer-based tools (see [1], [2], [3], [4]) for structuring and refining parametrized representations of array-based circuits. Our approach has been used to develop a wide range of word-level and bit-level devices, including designs for digital signal processing [5], arithmetic circuits [2], multi-level storage managers [6] and butterfly networks [1].

The purpose of this paper is to illustrate how this framework captures various ways of pipelining and transposing designs, together with the resulting trade-offs. Although systematic methods for pipelining [2] and for transposition [7] have been introduced in the past, the study of optimization schemes involving both of these transformations is novel. In particular, we compare several methods for improving the speed of circuits that contain counter-flowing data. An example of this kind of circuit, in this case a component used in an integer divider, is shown in figure 1. It is not clear whether there is an efficient scheme for distributing latches to pipeline the broadcast circuitry as well as the array of fulladders (labelled *fadd* in the figure), because signals flow in opposite directions in these two circuits. One of our solutions, which will be detailed in a later section, involves reversing the broadcast direction and eliminating long connections by transposition. The trade-offs of these transformations will be analyzed for a field-programmable gate array implementation.

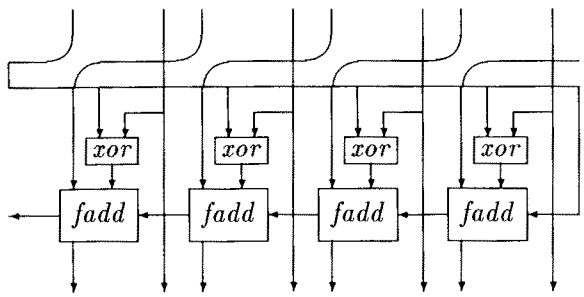


Fig. 1. A circuit with counter-flowing data.

Some readers may question the necessity of using a language specific to array-based circuits. In our opinion the results obtained justify the use of such a language and the associated techniques for circuit development: no other design methodology that we are aware of deals uniformly with both pipelining and transposition. Moreover, an algebraic approach provides a concise and systematic means of parametrizing designs and a simple transformation framework based on pattern matching and equational reasoning.

Another advantage of this method is its flexibility of allowing the user to select where to begin developing a design. Since our transformations are correctness-preserving, the only requirement on the initial design description is that it behaves as desired. In the interest of clarity, this description does not need to be implementable. On the other hand, it is often useful to transform a given structure which is asserted to behave correctly to see if it leads to an attractive implementation;

*Due to difficulties in typsetting, different fonts are used for Ruby expressions in main text and in figures.

that assertion can always be checked in a separate step. Where appropriate, simulation is another way of increasing confidence in the correctness of designs. The behavior of all circuit descriptions in Section 6, including the initial and final designs, can be simulated using an interpreter of our language.

The transformations discussed in this paper are intended to guide the development of parametrized design descriptions. They provide a means of organizing and codifying design experience, allowing such expertise to be deployed rapidly in new applications. They can also be used to document design decisions by recording what alternatives have been explored, how these alternatives are related to one another, and why a particular design has been chosen. A *design tree* can often be used to summarize the relationship between designs; see figure 13 for an example.

A further advantage of this method is the possibility of deriving formulae for estimating the quality of parametrized design representations [4]. Some examples of such formulae can be found in table 1. These formulae are technology-independent, since they express the features of a composite design in terms of the features of basic building blocks supplied by the user. They can be used without the knowledge of how they are derived.

For readers who wish to obtain an overview of our approach, figure 4 provides a number of examples describing various patterns of connecting processors in our notation. Figure 5 contains a theorem for pipelining designs—a triangle in the diagram represents an abstract view of a latch. Figure 8 gives a flavor of how a row of components can be transposed. A parallel divider design is shown in figure 9; using our transformations the repeated unit *DV0cell* can be replaced by the cells in figures 10, 11 or 12. Figure 13 summarizes the relationship between the divider cells that we derive, and table 1 quantifies their salient features. Finally, figure 15 presents possible implementations of two divider cells in field-programmable gate array technology.

Table 1. Comparing divider designs.

Design	Minimum Cycle Time	Latency (cycles)	Number of Latches in Array
DV0cell	$(N-1)T_B + T_X + NT_F$	0	0
DV1cell	$(N-1)T_B + T_X + KT_F$	$\frac{N}{K}$	$\frac{N(3N-K+2)}{2K}$
DV2cell	$NT_B + T_X + NT_F$	0	0
DV3cell	$NT_B + T_X + KT_F$	$\frac{N}{K}$	$\frac{N(N+2)}{K}$
DV4cell	$T_B + T_X + KT_F$	$\frac{N}{K}$	$\frac{N(N+2)}{K}$

N: the number of adders in a cell,
K: clustering the adders in a cell in groups of K adders,
T_X, T_F: the combinational delay of the cell *xor*, *fadd*,
T_B: the broadcast delay through an X_i cell.

2. Design Representation

The formalism that we use is based on Sheeran's relational framework and the author's heterogeneous combinators. The background and the details of this approach have been described elsewhere (see [1], [8]), and only the definitions and concepts relevant to our discussion will be introduced here.

A design will be described 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 example, an inverter can be specified as

$$x\ inv\ y \Leftrightarrow (x = 0 \wedge y = 1) \vee (x = 1 \wedge y = 0), \quad (1)$$

or, more succinctly, as $0\ inv\ 1 \wedge 1\ inv\ 0$.

The converse R^{-1} of a relation R is defined by $x\ (R^{-1})\ y \Leftrightarrow y\ R\ x$, and the identity relation is given by $x\ \iota\ y \Leftrightarrow x = y$. If R is a function, then $R\ x$ is the value of R for the argument x.

Objects in our notation are either atoms (such as numbers or relations) or tuples of objects: for instance the object $\langle 0, \langle 1, 2\rangle\rangle$ is a 2-tuple containing the number 0 and the tuple $\langle 1, 2\rangle$. A tuple is an ordered collection of elements, with the empty tuple denoted by $\langle\ \rangle$. Given that x is a tuple, $\#x$ represents the number of elements in it, and x_i (where $0 \leq i < \#x$) is its i-th element. Tuples are concatenated by the functions *apl* (append left), *apr* (append right), or '^' (binary append), so that

$$\begin{aligned} apl\,\langle a, \langle b, c, d, e\rangle\rangle &= apr\,\langle\langle a, b, c, d\rangle, e\rangle \\ &= \langle a, b, c\rangle\hat{}\langle d, e\rangle \\ &= \langle a, b, c, d, e\rangle. \end{aligned}$$

A rectangular circuit with connections on every side is modeled by a relation that relates 2-tuples, with the 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. In general, composite signals are represented as tuples with the position of a particular signal corresponding to its relative position, and with its structure—the grouping of signals—reflecting the logical organization of adjacent signals.

Given a circuit R with connections on all four sides, we can use the generic reverse function *recrev* which recursively reverses a tuple and all its component tuples,

$$recrev\ x = x \text{ if } x = \langle\ \rangle \text{ or if } x \text{ is an atom,}$$

$$recrev(\langle x\rangle \hat{}\ xs) = (recrev\ xs)\hat{}\langle recrev\ x\rangle \text{ otherwise,}$$

to define $R^{\triangledown}$ and $R^{\mathcal{H}}$ which denote the reflection of R in a vertical and in a horizontal axis:

$$\langle x, y\rangle R^{\triangledown}\langle x', y'\rangle \Leftrightarrow \langle y', recrev\ y\rangle R\langle recrev\ x', x\rangle,$$

$$\langle x, y\rangle R^{\mathcal{H}}\langle x', y'\rangle \Leftrightarrow \langle recrev\ x, x'\rangle R\langle y, recrev\ y'\rangle.$$

Clearly $(R^{\triangledown})^{\mathcal{H}} = (R^{\mathcal{H}})^{\triangledown}$.

3. Combinators

Combinators are higher-order functions that capture common patterns of computation as parametrized expressions. Both behavior and structure can be described by these patterns: they express the behavior of a composite device in terms of the behavior of its components, and they also describe the connection of components to form the composite device.

3.1. Binary Composition

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 2a),

$$x\,(Q;R)\,y \Leftrightarrow \exists s \cdot (x\ Q\ s) \wedge (s\ R\ y). \qquad (2)$$

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 2b), where

$$\langle x_0, x_1\rangle[Q, R]\langle y_0, y_1\rangle \Leftrightarrow (x_0\ Q\ y_0) \wedge (x_1\ R\ y_1). \qquad (3)$$

It is simple to show that $[P, Q]; [R, S] = [P; R, Q; S]$, and that $[P, Q]^{-1} = [P^{-1}, Q^{-1}]$. Our framework contains a collection of such theorems, which can be used for reasoning about designs.

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

$$\mathsf{fst}\ R = [R, \iota],$$

$$\mathsf{snd}\ R = [\iota, R].$$

Next, the relation *fork* can be used to duplicate a signal, since $x\ fork\ \langle x, x\rangle$. 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

$$\mathsf{fst}\ Q\ ;\ \mathsf{snd}\ R = \mathsf{snd}\ R\ ;\ \mathsf{fst}\ Q = [Q, R],$$

$$fork\ ;\ \pi_1 = fork\ ;\ \pi_2 = \iota,$$

$$[Q, R]\backslash swap = swap\ ;\ [Q, R]\ ;\ swap = [R, Q].$$

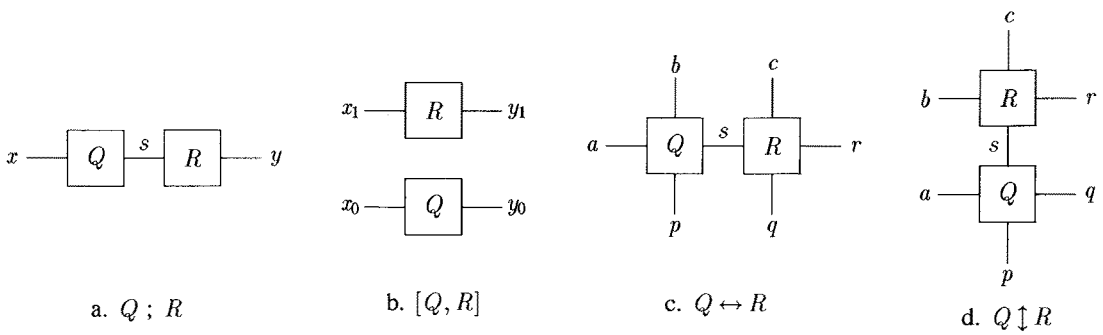
Two rectangular components with connections on every side can be assembled together by the beside ($\leftrightarrow$) and below ($\updownarrow$) operators (figure 2c and figure 2d):

$$\langle a, \langle b, c\rangle\rangle(Q \leftrightarrow R)\langle\langle p, q\rangle, r\rangle$$

$$\Leftrightarrow \exists s \cdot \langle a, b\rangle Q\langle p, s\rangle \wedge \langle s, c\rangle R\langle q, r\rangle,$$

$$\langle\langle a, b\rangle, c\rangle(Q \updownarrow R)\langle p, \langle q, r\rangle\rangle$$

$$\Leftrightarrow \exists s \cdot \langle a, s\rangle Q\langle p, q\rangle \wedge \langle b, c\rangle R\langle s, r\rangle.$$



a. $Q\ ;\ R$ b. $[Q, R]$ c. $Q \leftrightarrow R$ d. $Q \updownarrow R$

Fig. 2. Some binary combinators.

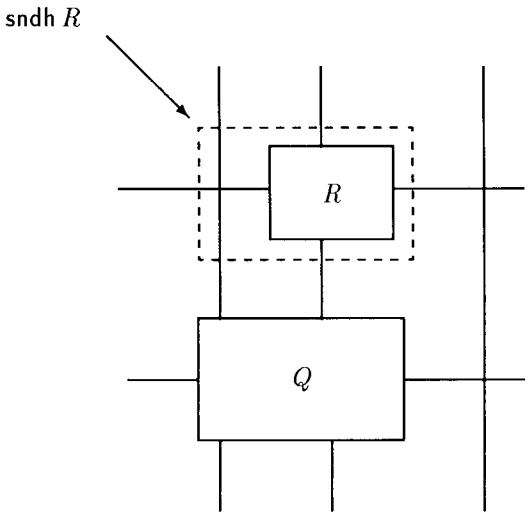


Fig. 3. The picture of fsth (Q ↕ (sndh R)).

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 useful to have the following abbreviations which are similar to the fst, snd and \ operators,

$$\text{fsth } R = R \leftrightarrow \mathit{swap},$$

$$\text{sndh } R = \mathit{swap} \leftrightarrow R,$$

$$\text{fstv } R = R \updownarrow \mathit{swap},$$

$$\text{sndv } R = \mathit{swap} \updownarrow R,$$

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

A simple example involving some of these abbreviations is shown in figure 3.

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

Finally, note that unappending the last element of a tuple followed by appending its first element corresponds to a shift left, and similarly for shift right,

$$\mathit{shl} = \mathit{apr}^{-1}; \mathit{apl},$$

$$\mathit{shr} = \mathit{apl}^{-1}; \mathit{apr}.$$

Clearly $\mathit{shl}^{-1} = \mathit{shr}$, since $(Q ; R)^{-1} = R^{-1}; Q^{-1}$.

3.2. Heterogeneous Combinators

One can check that in general the binary operators (such as parallel composition) are not commutative, and that only relational composition is associative. For instance, since $\langle\langle x, y\rangle, z\rangle$ and $\langle x, \langle y, z\rangle\rangle$ are distinct tuples, $[[P, Q], R] \neq [P, [Q, R]]$. We now introduce a class of prefix combinators, called heterogeneous combinators, each of which takes a tuple of components and returns a binary relation corresponding to the composite circuit; the components that are wired together can be different from one another. So given that $\#x = \#y = \#R = N$, a *heterogeneous chain* (figure 4a) and a *heterogeneous map* (figure 4b) are described respectively by

$$a\ (;\ R)\ b \Leftrightarrow \exists s \cdot (s_0 = a) \wedge (s_N = b) \wedge \forall i\colon 0 \leq i < N \cdot s_i R_i s_{i+1},$$

$$x\ (\|\ R)\ y \Leftrightarrow \forall i\colon 0 \leq i < N \cdot x_i R_i y_i.$$

Similarly, a heterogeneous row (figure 4c) is described by

$$\langle a, x\rangle(\leftrightarrow R)\langle y, b\rangle \Leftrightarrow \exists s \cdot (s_0 = a) \wedge (s_N = b) \wedge \forall i\colon 0 \leq i < N \cdot \langle s_i, x_i\rangle R_i \langle y_i, s_{i+1}\rangle.$$

A heterogeneous column (figure 4d) can be described in a similar fashion. The correctness of these descriptions can be proved using the recursive definitions of these combinators in terms of the binary operators shown earlier, but we shall not go into such details.

Given $\Psi \in \{;, \|, \leftrightarrow, \updownarrow\}$ and $0 \leq N \leq \#R$, we shall adopt the abbreviation

$$\underset{i<N}{\Psi} R_i = \Psi\langle R_i \mid 0 \leq i < N\rangle,$$

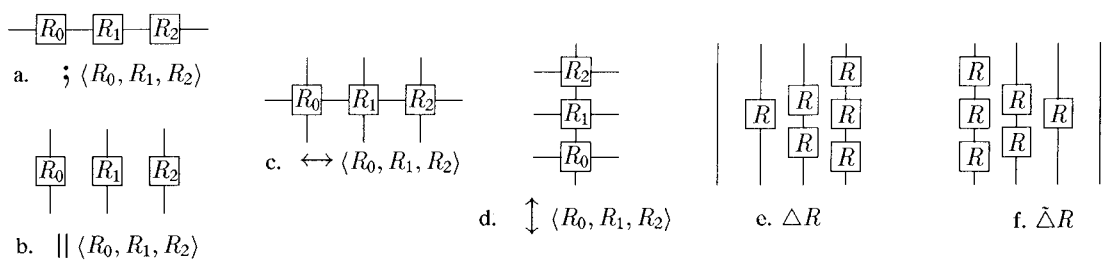


Fig. 4. Some heterogeneous and homogeneous combinators.

so that one can write

$$\left(\updownarrow_{i<N} R_i\right)^{-1} = \leftrightarrow_{i<N} (R_i^{-1}).$$

3.3. Homogeneous Combinators

For many applicatons the generality of heterogeneous combinators is not required. If R_i does not depend on i, then the following homogeneous combinators [1] can be used to describe arrays of identical components.

$$R^N = ;_{i<N} R \qquad \text{(chain)},$$

$$\mathsf{map}_N\, R = \|_{i<N} R \qquad \text{(map)},$$

$$\triangle_N\, R = \|_{i<N} R^i \qquad \text{(triangle)},$$

$$\tilde{\triangle}_N\, R = \|_{i<N} R^{N-i-1} \qquad \text{(reverse triangle)},$$

$$\mathsf{row}_N\, R = \leftrightarrow_{i<N} R \qquad \text{(row)},$$

$$\mathsf{col}_N\, R = \updownarrow_{i<N} R \qquad \text{(column)}.$$

Instances of the $\triangle$ and $\tilde{\triangle}$ operators are shown in figure 4e and figure 4f. The subscripts associated with these combinators correspond to the size of the arrays, often omitted when they can be deduced from context.

4. Sequential Circuits and Pipelining

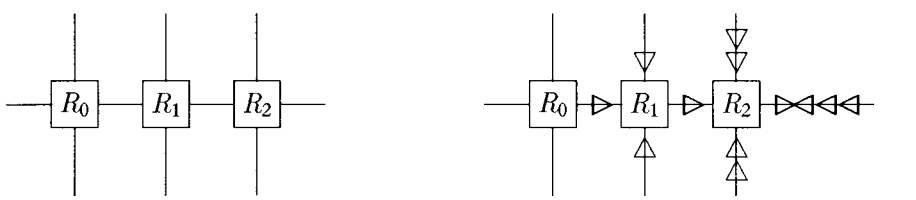
So far we have been using relations to model a static situation—the steady state behavior of a circuit at a particular instant of time. To deal with sequential circuits, an expression can be considered as relating a stream in its domain to a stream in its range. For our purpose, a stream can be considered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

For instance, an inverter, *Inv*, can be specified as $x\ Inv\ y \Leftrightarrow \forall t \cdot x_t\ inv\ y_t$, where *inv* is the corresponding static description given earlier in equation 1. For simplicity, combinational circuits will be described in the static form; they 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 [9].

A delay $\mathfrak{D}$ is defined by $x\ \mathfrak{D}\ y \Leftrightarrow \forall t \cdot y_t = x_{t-1}$. An *anti-delay* $\mathfrak{D}^{-1}$ is such that $\mathfrak{D};\ \mathfrak{D}^{-1} = \mathfrak{D}^{-1};\ \mathfrak{D} = \iota$. A latch is modeled 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 $\mathfrak{D}^5$. Similarly —◁— and △ represent anti-delays for horizontal and vertical dataflows.

4.1. Retiming

Retiming [10] is a method for pipelining a circuit by introducing and relocating latches. It can be applied to circuits containing no primitives which possess a measure of absolute time. For such circuits, simultaneously delaying every domain signal and anti-delaying every range signal will not alter the behavior: $R = \mathfrak{D};\ R;\ \mathfrak{D}^{-1} = R \backslash \mathfrak{D}^{-1}$. Circuits with this property are known as *timeless* [9]. The following theorems state that one can distribute delays within a chain or a row of timeless components so long as additional delays and anti-delays are placed at the edges of the circuits (figure 5):



$$\leftrightarrow_{i<N} R_i \quad = \quad \leftrightarrow_{i<N} (R_i; \mathsf{snd}\, \mathcal{D}) \setminus\!\setminus (\mathsf{fst}\, \triangle \mathcal{D}^{-1}); \mathsf{snd}\, \mathcal{D}^{-N}$$

Fig. 5. A theorem for retiming a row of heterogeneous components ($N = 3$).

$$\mathop{;}_{i<N} R_i = \mathop{;}_{i<N} (R_i \,;\, \mathfrak{D});\ \mathfrak{D}^{-N}, \tag{4}$$

$$\mathop{\leftrightarrow}_{i<N} R_i = \mathsf{snd}\ \triangle\mathfrak{D};\ \mathop{\leftrightarrow}_{i<N} (R_i \,;\, \mathsf{snd}\ \mathfrak{D});\ [\triangle\mathfrak{D}^{-1},\ \mathfrak{D}^{-N}]$$

$$= \mathop{\leftrightarrow}_{i<N} (R_i \,;\, \mathsf{snd}\ \mathfrak{D}) \backslash\backslash (\mathsf{fst}\ \triangle\mathfrak{D}^{-1});\ \mathsf{snd}\ \mathfrak{D}^{-N}. \tag{5}$$

Note that the boundary condition $\mathfrak{D}^{-N}$ indicates that the pipelined circuits have an additional latency of N cycles, and the expression $\triangle\mathfrak{D}^{-1}$ provides information about data-skews associated with the transformation.

4.2. Controlling Pipelining

Although pipelining can increase the throughput of a system, it may also increase its latency and the amount of area and power dissipation from latches both within the array and at its periphery for data-skewing. To synthesize expressions that denote designs with variable degrees of pipelining, we first use *clustering theorems* to express the target array as an array of clusters of components; latches are then distributed between clusters using retiming theorems. In general a cluster with a larger number of components will result in a slower design with smaller area and latency.

It is obvious how to cluster a chain; just express it as a chain of chains:

$$\mathop{;}_{i<NK} R_i = \mathop{;}_{i<N} \left(\mathop{;}_{j<K} R_{iK+j} \right). \tag{6}$$

To cluster rows, we need the relation $group_n$ to format an $(m \times n)$-tuple to form an n-tuple of m-tuples,

$$x\ group_n\ y \Leftrightarrow \forall i, j\colon 0 \le i < n,\ 0 \le j < m \cdot (\#y = n) \wedge (\#x = m \times n) \wedge (y_{i,j} = x_{mi+j}),$$

so that, for example, $\langle 1, 2, 3, 4, 5, 6\rangle\ group_3\ \langle\langle 1, 2\rangle, \langle 3, 4\rangle, \langle 5, 6\rangle\rangle$. We can use *group* to format the signals to match the tuple structure required by a row of rows,

$$\mathop{\leftrightarrow}_{i<NK} R_i = \mathop{\leftrightarrow}_{i<N} \left(\mathop{\leftrightarrow}_{j<K} R_{iK+j} \right) \backslash\backslash \mathsf{fst}\ group_N^{-1}. \tag{7}$$

One can then apply the retiming theorems described earlier, treating the chain or row of K R_{iK+j} components as a single unit:

$$\mathop{;}_{i<NK} R_i = \mathop{;}_{i<N} \left(\mathop{;}_{j<K} R_{iK+j} \,;\, \mathfrak{D} \right);\ \mathfrak{D}^{-N}, \tag{8}$$

$$\mathop{\leftrightarrow}_{i<N} \left(\mathop{\leftrightarrow}_{j<K} R_{iK+j} \right) = \mathop{\leftrightarrow}_{i<N} \left(\mathop{\leftrightarrow}_{j<K} R_{iK+j};\ \mathsf{snd}\ \mathfrak{D} \right) \backslash\backslash (\mathsf{fst}\ \triangle\mathfrak{D}^{-1});\ \mathsf{snd}\ \mathfrak{D}^{-N}. \tag{9}$$

There are also retiming theorems to deal with two-dimensional arrays; such details can be found in [1], [2], [5].

5. Transposition

Transposition theorems relate circuits interleaved in different ways. A general discussion of using transposition to optimize designs can be found elsewhere [7]; for this paper we shall just give a few examples illustrating the idea. The objective of transposition is to distribute the components of a circuit so as to shorten interconnections or to vary its aspect ratio.

There are two common patterns of wiring for describing transposed circuits: *tran* and *bend*. The transpose relation interleaves the elements of a tuple of tuples,

$$x\ tran\ y \Leftrightarrow \forall i, j\colon 0 \le i < \#x,\ 0 \le j < \#y \cdot (\#x_i = \#y) \wedge (y_{j,i} = x_{i,j}),$$

where $x_{i,j} = (x_i)_j$. For example, $\langle\langle 1, 2, 3\rangle, \langle 4, 5, 6\rangle\rangle$ *tran* $\langle\langle 1, 4\rangle, \langle 2, 5\rangle, \langle 3, 6\rangle\rangle$. Notice that *tran* is only defined for tuples in which all sub-tuples have the same length.

A *bend* is a piece of wire that connects its domain signals and leaves its range signal unspecified,

$$\langle x, y\rangle\ bend\ z \Leftrightarrow x = y.$$

Note that $bend = fork^{-1};\ \pi_1^{-1};\ \pi_2$.

5.1. Transposing a Chain

The theorems that will be introduced relate circuits with bends. A bend can be added, for instance, to make the two ends of a chain of components to point in the same direction (figure 6):

$$x \left(\mathop{;}_{i<N} R_i \right) y \Leftrightarrow \langle x, y\rangle \left(\mathsf{fst} \mathop{;}_{i<N} R_i;\ bend \right) z.$$

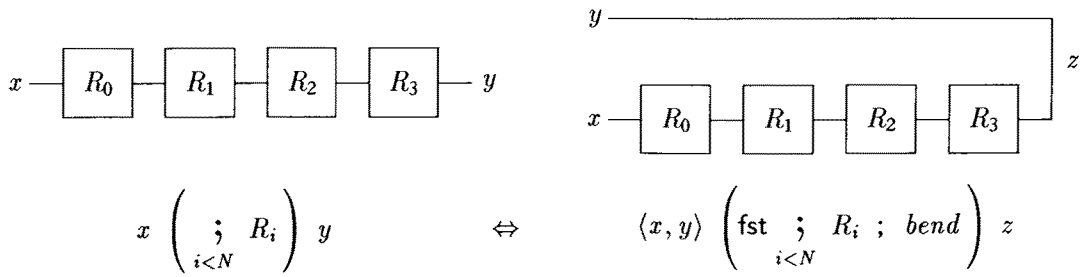


Fig. 6. Adding a vertical bend to a chain ($N = 4$).

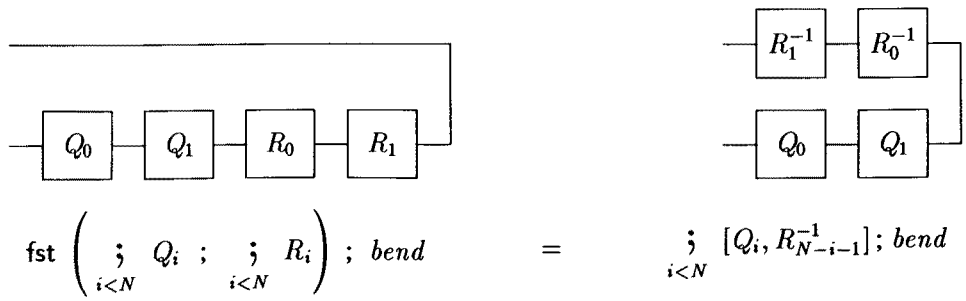


Fig. 7. Transposing a chain with a vertical bend ($N = 2$).

We can now move some of the components from the lower branch to the upper branch (figure 7), since

$$\begin{aligned}&\mathsf{fst}\left(\mathop{;}_{i<N} Q_i \;;\; \mathop{;}_{i<N} R_i\right);\ \mathit{bend}\\ &\quad=\left[\mathop{;}_{i<N} Q_i,\ \mathop{;}_{i<N} R_{N-i-1}^{-1}\right];\ \mathit{bend}\\ &\quad=\mathop{;}_{i<N}[Q_i, R_{N-i-1}^{-1}];\ \mathit{bend}. \quad (10)\end{aligned}$$

This transformation has two effects. First, it eliminates the long wire in the left-hand circuit which may cause undesirable propagation delay. Second, it also changes the aspect ratio of a circuit—for instance given that all components are of width w and height h, equation 10 equates a circuit of size $2Nw \times h$ with a circuit of size $Nw \times 2h$.

5.2. *Transposing a Row*

A bend can also be added to a component with connections on every side:

$$\langle x, y\rangle R\langle u, v\rangle \Leftrightarrow \langle\langle x, v\rangle, y\rangle(\mathsf{fstv}\ R;\ \mathsf{snd}\ \mathit{bend};\ \pi_1)u.$$

One can move the component to the upper branch provided that it is reflected vertically,

$$\begin{aligned}&\mathsf{fstv}\ R\ ;\ \mathsf{snd}\ \mathit{bend}\\ &\qquad=(\mathsf{sndv}\ R^{\nabla})\backslash\backslash(\mathsf{fst}\ \mathit{recrev});\ \mathsf{snd}\ \mathit{bend}.\end{aligned}$$

From this, an equation for transposing a circuit with two components can be derived:

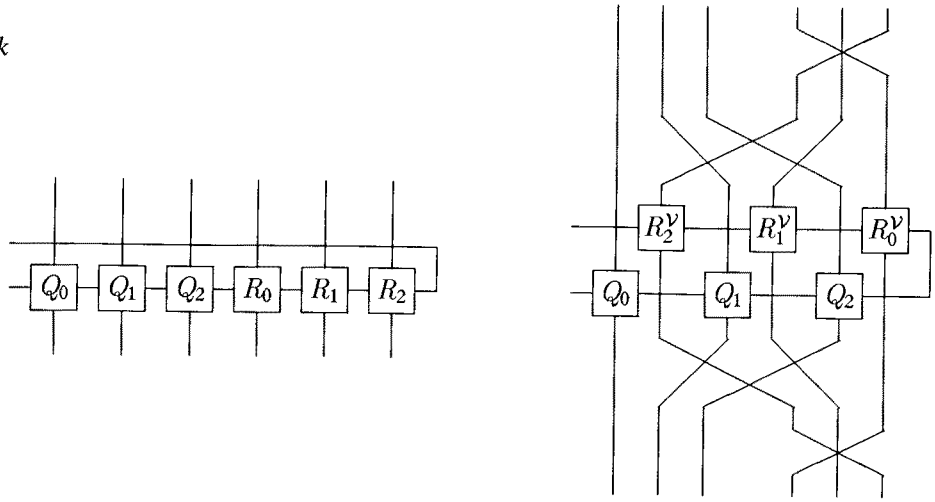
$$\begin{aligned}&\mathsf{fstv}\ (Q \leftrightarrow R);\ \mathsf{snd}\ \mathit{bend}\\ &\quad=([Q, R^{\nabla}]\backslash\mathit{tran})\backslash\backslash\mathsf{fst}(\mathsf{snd}\ \mathit{recrev});\ \mathsf{snd}\ \mathit{bend}.\end{aligned}$$

If both components are themselves rows of N components, then we can move half of the components from the lower branch to the upper branch and transpose the resulting circuit (figure 8). So given that $\mathit{srev} = \mathit{tran}$; snd *recrev*, we have

$$\begin{aligned}&\mathsf{fstv}\left(\mathop{\leftrightarrow}_{i<N} Q_i \leftrightarrow \mathop{\leftrightarrow}_{i<N} R_i\right);\ \mathsf{snd}\ \mathit{bend}\\ &=\left(\left[\mathop{\leftrightarrow}_{i<N} Q_i,\ \mathop{\leftrightarrow}_{i<N} R_{N-i-1}^{\nabla}\right]\backslash\mathit{tran}\right)\\ &\qquad\backslash\backslash\mathsf{fst}(\mathsf{snd}\ \mathit{recrev});\ \mathsf{snd}\ \mathit{bend}\\ &=\mathop{\leftrightarrow}_{i<N}([Q_i, R_{N-i-1}^{\nabla}]\backslash\mathit{tran})\backslash\backslash\mathsf{fst}\ \mathit{srev}\ ;\ \mathsf{snd}\ \mathit{bend}.\end{aligned}$$

Note that the wiring cell *tran* can itself be expressed as a row of *trancell*,

$$\begin{aligned}\mathit{tran} = \mathit{apl}^{-1};\ &\mathsf{fst}\ \mathsf{map}(\mathit{apr}^{-1}\backslash\pi_1);\\ &\mathsf{row}(\mathsf{col}\ \mathit{trancell});\ \pi_2 \quad (12)\end{aligned}$$



$$\mathsf{fstv}\left(\underset{i<N}{\leftrightarrow} Q_i \leftrightarrow \underset{i<N}{\leftrightarrow} R_i\right); \mathsf{snd}\, bend = \underset{i<N}{\leftrightarrow} ([Q_i, R^V_{N-i-1}] \setminus tran) \setminus\!\setminus \mathsf{fst}\, srev \;;\; \mathsf{snd}\, bend$$

Fig. 8. Transposing a row with a vertical bend ($N = 3$).

where *trancell* = ((fst *swap*)*shl*)\(snd *apr*). Theorems like equation 5 can then be applied to pipeline the array of *trancell*. Hence it is possible to transpose an array of pipelined components without introducing long wires, provided that the boundary circuits such as *tran* are themselves pipelined as well.

Further details on transposing linear and rectangular arrays with multiple bends can be found in [7].

6. Parallel Division

In this section an example will be used to illustrate the application of the techniques expounded in the preceding sections. Based on the nonrestoring division algorithm [11] one can construct an iterative array (figure 9) for unsigned parallel division with divident D, divisor d, quotient q and remainder r. Given that $!c$ represents a circuit that generates a constant signal c,

$$x\ (!c)\ y \Leftrightarrow x = y = c,$$

this divider design can be captured in our notation as

$$DV0 = \mathsf{fst}(\pi_2^{-1}; \mathsf{fst}\ !1;\ apl);$$

$$(\mathsf{row}\ DV0cell \leftrightarrow DV0cell')\setminus\!\setminus(\mathsf{fst}\ apr);\ \mathsf{snd}\ AFcell,$$

$$DV0cell = \mathsf{snd}\ \pi_1^{-1};\ shl \leftrightarrow BXF;\ shrbend.$$

We shall explain *shl* first, and deal with *shrbend* later on. *shl* shifts its input downwards towards the most significant bit and can be implemented by a column of wiring cells $\iota 2$:

$$shl = \mathsf{col}\ \iota 2 = \mathsf{col}\ [\iota, \iota].$$

The bottom output of *shl* controls the function of the programmable adder/subtracter *BXF* and must be initialized to 1 to set *BXF* up as a subtracter. This initialization is performed by the interface circuit π_2^{-1}; fst !1. *BXF* itself consists of a column of XF_i cells, each containing a fulladder *fadd* and a circuit X_i that uses an exclusive-or gate *xor* to control the inversion of the divisor.

The digits d_i of the divisor are hardwired as static coefficients, with d_0 as the most significant bit ($d_0 = 0$ if d represents an unsigned number); this prevents the dynamic alteration of the divisor but enables fewer latches to be used in pipelining—we shall come back to this later. The output of the top X_i cell is fed into the carry input of its neighoring *fadd* cell by the wiring cell $bend^{-1}$.

$$BXF = \mathsf{snd}\ bend^{-1};\ \underset{i<N}{\updownarrow} XF_i,$$

$$XF_i = X_i \leftrightarrow fadd,$$

$$X_i = \mathsf{fst}(\pi_1^{-1};\ \mathsf{snd}\ !d_i);\ \mathsf{sndv}(fork;\ [\pi_2,\ xor]).$$

Notice that the *fadd* cell described above has all inputs in its domain and outputs in its range, and is different from the one in figure 1 whose carry output is in the domain of the corresponding relation.

At the bottom of *DV0cell* another bending cell, *shrbend*, is used to connect the bottom output of *shl* to the bottom input if its neighboring X_0 cell, and to duplicate and to pass to the right the bottom carry output of the adder column:

$$shrbend = \mathsf{fst}(shr;\ \mathsf{fst}\ bend);\ dupshl,$$

$$dupshl = \mathsf{fst}(\pi_2;\ fork);\ (\mathsf{snd}\ \pi_2;\ shl)\setminus(\mathsf{snd}\ apl).$$

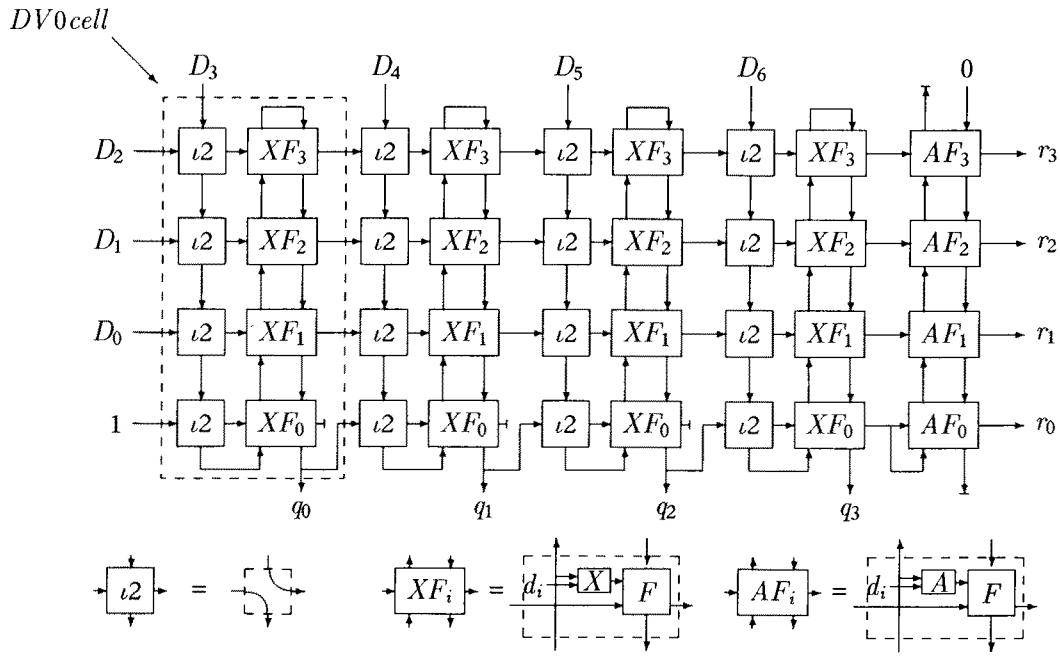


Fig. 9. Design *DV*0 ($L = 7$, $N = 4$, $X = xor$, $A = and$, $F = fadd$).

The rightmost *DV*0*cell*, called *DV*0*cell'*, has a simplified *shrbend* as the bottom carry output of the adders is not duplicated:

$$DV0cell' = \mathsf{snd}\ \pi_1^{-1};\ shl \leftrightarrow BXF;\ shrbend',$$

$$shrbend' = \mathsf{fst}(shr;\ \mathsf{fst}\ bend;\ \pi_2).$$

An *AFcell* is placed to the right of *DV*0*cell'*. This is a remainder-correction circuit with a structure similar to that of a *DV*0*cell*. It can be omitted if only the quotient is required.

$$AFcell = fork;\ \mathsf{fst}\ (apl^{-1};\ \pi_1);\ \pi_1^{-1};\ \mathsf{snd}(!0;\ \pi_2^{-1});$$

$$Bcell \updownarrow \left(\updownarrow_{i<N} AF_i \right);\ \pi_2^2,$$

$$Bcell = shr;\ \mathsf{fst}\ bend,$$

$$AF_i = A_i \leftrightarrow fadd,$$

$$A_i = \mathsf{fst}(\pi^{-1};\ \mathsf{snd}\ !d_i);\ \mathsf{sndv}(fork;\ [\pi_2,\ and]).$$

Given that *nat* maps a tuple of bits (most significant bit first) into its numerical value, the correctness of *DV*0 with an L-bit D and N-bit d can be established by showing that

$$\langle\langle D_i | 0 \le i < N-1\rangle, \langle D_i | N-1 \le i < L\rangle\rangle$$

$$DV0\ \langle q, r\rangle \Leftrightarrow nat\ D = (nat\ q) \times (nat\ d) + nat\ r.$$

One should also work out ways to detect invalid inputs and outputs such as having a zero divisor or overflow in addition or subtraction. We omit such details since deriving designs from a behavioral description (see [1], [2], [5]) is not the main theme of this paper.

It is simple to use equation 5 to pipeline *DV*0 by placing latches between *DV*0*cells*. Now suppose that even this transformation does not provide a fast enough circuit. Two methods of pipelining the column of adders will be presented below.

6.1. Decomposing Circuits with Counter-Flowing Data

The main problem with pipelining *DV*0*cell* is the presence of counter-flowing data in an XF_i cell—remember that a delay with a rightward or a downward signal flow can be implemented by a latch, but a delay with a leftward or an upward signal flow cannot be implemented. The first solution that we offer is to decompose the column of XF_i cells into a column of X_i cells beside a column of *fadd* cells,

$$\updownarrow_{i<N} XF_i = \updownarrow_{i<N} (X_i \leftrightarrow fadd)$$

$$= \left(\updownarrow_{i<N} X_i \right) \leftrightarrow \mathsf{col}_N\ fadd,$$

and pipeline the column of fulladders using a reflected version of equation 5,

$$\mathsf{col}_N\, fadd = (\mathsf{col}_N(fadd;\mathsf{fst}\ \mathfrak{D}))\backslash\backslash(\mathsf{snd}\ \triangle\mathfrak{D}^{-1});\ \mathsf{fst}\ \mathfrak{D}^{-N}$$
$$= \mathsf{fst}\ \tilde{\triangle}\mathfrak{D};\ \mathsf{col}_N(fadd;\ \mathsf{fst}\ \mathfrak{D});\ [\mathfrak{D}^{-N}, \tilde{\triangle}\mathfrak{D}^{-1}].$$

The $\tilde{\triangle}\mathfrak{D}^{-1}$ that is introduced between adjacent *DV0cells* can be cancelled by inserting N delays between them, since $\tilde{\triangle}_N\mathfrak{D}^{-1};\ \mathfrak{D}^N = \mathfrak{D};\ \triangle\mathfrak{D}$. This gives

$$DV0cell;\ \mathfrak{D}^N = \mathsf{snd}\ \pi_1^{-1};\ shl \leftrightarrow BXF';\ shrbend,$$
$$BSF' = \mathsf{snd}\ bend^{-1};\ \updownarrow_{i<N} X_i \leftrightarrow faddD,$$
$$faddD = \mathsf{fst}\ \tilde{\triangle}\mathfrak{D};\ \mathsf{col}_N(fadd;\mathsf{fst}\ \mathfrak{D});\ \mathsf{snd}\ (\mathfrak{D};\ \triangle\mathfrak{D}).$$

A similar procedure can be used to pipeline *AFcell*.

Given that T_X and T_F represent respectively the propagation delay of *xor* and *fadd*, and T_B is the delay due to broadcasting the control signal upwards through an X_i cell, the critical path delay of a *DV0cell* with pipelined fulladders is given by $(N-1)T_B + T_X + T_F$. This cell has a latency of N cycles and has $N(N-1) + N + N(N+1)/2 = N(3N+1)/2$ latches (an N-bit register which may be required to hold the divisor is not included in this figure). By clustering the N fulladders into M groups of K fulladders ($N = MK$) and then applying the column version of equation 5 as before, we obtain

$$DV1cell = \mathsf{snd}\ \pi_1^{-1};\ shl \leftrightarrow BXF1;\ shrbend,$$
$$BXF1 = \mathsf{snd}\ bend^{-1};\ \updownarrow_{i<N} X_i \leftrightarrow faddD1,$$
$$faddD1 = \mathsf{fst}(group_M;\ \tilde{\triangle}\mathfrak{D});\ \mathsf{col}_M(\mathsf{col}_K fadd;\ \mathsf{fst}\ \mathfrak{D});\ \mathsf{snd}(\mathfrak{D};\triangle\mathfrak{D};group_M^{-1}).$$

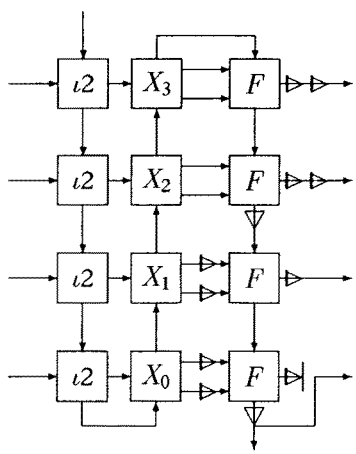
One can reduce the value of M in *faddD1* to produce circuits that work at a lower speed and with fewer latches; figure 10 shows a *DV1cell* pipelined by every two fulladders. On the other hand a faster circuit can be obtained—at the expense of introducing even more latches—by pipelining the column of X_i cells as well. In the following we shall discuss an alternative way of pipelining *DV0cell* which uses fewer latches.

6.2. *Reversing the Broadcast Direction*

The basic idea of this method is to reverse the direction of broadcasting the control signal in X_i by introducing a wire that connects the bottom output of *shl* to the top input of the column of XF_i cells (figure 11),

$$DV2cell = topbend;\ (\mathsf{fsth}\ shl\ ;\ \mathsf{fst}\ bend) \leftrightarrow \left(\updownarrow_{i<N} XF_i\right);\ \mathsf{fst}\ \pi_2;\ dupshl,$$
$$topbend = \mathsf{snd}(\pi_1^{-1};\ \mathsf{snd}\ bend^{-1};\ shr;\ \mathsf{snd}\ fork).$$

It can be shown that a *DV2cell* behaves the same as a *DV0cell*: *DV2cell* = *DV0cell*. However, now that counter-flowing data have been eliminated, the column of XF_i cells can be pipelined by inserting latches between M groups each containing K cells ($N = MK$) to form a *DV3cell*:



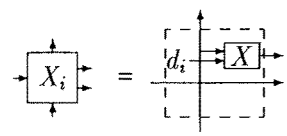


Fig. 10. Design *DV1cell* ($N = 4$, $M = K = 2$, $X = xor$, $F = fadd$).

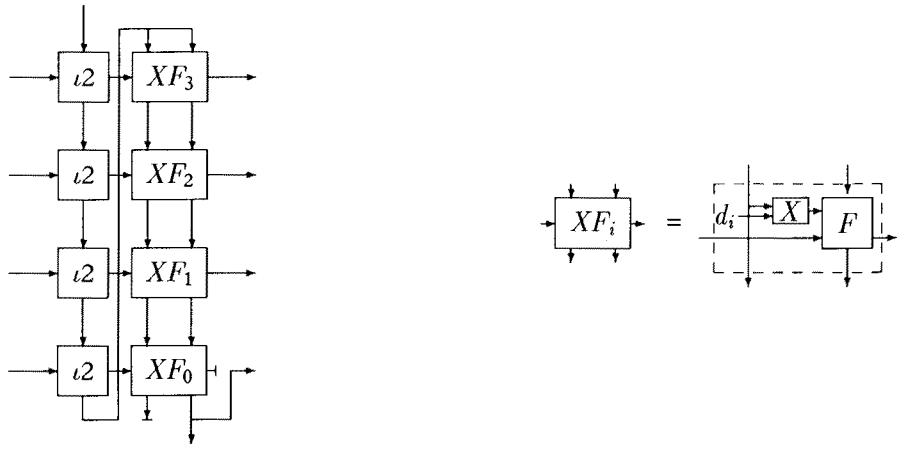


Fig. 11. Design *DV2cell* ($N = 4$, $X = xor$, $F = fadd$).

$$\updownarrow_{i<N} XF_i = A;\ \updownarrow_{i<M}\left(\updownarrow_{j<K} XF_{iK+j};\ \mathsf{fst}\ \mathfrak{D}\right);\ B \qquad (13)$$

where

$$A = \mathsf{fst}\ (group_M\ ;\ \tilde{\triangle}\mathfrak{D}),$$

$$B = [\mathfrak{D}^{-M},\ \tilde{\triangle}\mathfrak{D}^{-1};\ group_M^{-1}].$$

As before, the $\tilde{\triangle}\mathfrak{D}^{-1}$ between adjacent *DV3cells* can be eliminated by composing it with $\mathfrak{D}^M$. One can check that a fully-pipelined *DV*3cell ($K = 1$) has a critical path delay of $NT_B + T_X + T_F$ and a latency of N cycles, but it only has $N(N-1)/2 + 2N + N(N+1)/2 = N(N+2)$ latches. Fewer skewing latches are required since an XF_i cell has only one horizontal input while *fadd* has two horizontal inputs. Hence for $N > 3$, it will be more economical to pipeline a *DV2cell* rather than a *DV0cell*.

A similar divider circuit with unidirectional data flow has been proposed independently by Parhi [12].

6.3. *Introducing Transposition*

A pipelined *DV3cell* can be further optimized by transposition to eliminate the broadcast wire and the associated delay of NT_B. The resulting design, *DV4cell*, can be obtained in three steps. First of all, since *shl* = $\mathsf{col}_\iota 2$, we can split it into two halves using the following theorem:

$$\mathsf{col}_{2n}R = (\mathsf{col}_n R \updownarrow \mathsf{col}_n R)\backslash\backslash\mathsf{snd}\ group_2^{-1}.$$

The resulting array is then transposed using the column version of equation 11,

$$\mathsf{fsth}(\mathsf{col}\ R \updownarrow \mathsf{col}\ R);\ \mathsf{fst}\ bend$$

$$= \mathsf{col}(\mathsf{sndh}\ R^{\mathfrak{JC}} \updownarrow \mathsf{fsth}\ R)\backslash\backslash\mathsf{snd}\ frev\ ;\ \mathsf{fst}\ bend$$

where $frev = tran\ ;\ \mathsf{fst}\ recrev$.

Next, a similar procedure can be used to transpose the arrays of pipelined XF_i cells and pipelined AF_i cells, provided that a *bend* and a broadcast wire are first added so that they become respectively

$$\mathsf{fsth}\left(\updownarrow_{i<N}(XF_i;\ \mathsf{fst}\ \mathfrak{D})\right);\ \mathsf{fst}\ bend$$

and

$$\mathsf{fsth}\left(\updownarrow_{i<N}(AF_i;\ \mathsf{fst}\ \mathfrak{D})\right);\ \mathsf{fst}\ bend.$$

These arrays can then be split in half and transposed as before.

Finally, we also need to be able to transpose the triangular-shaped arrays of latches introduced by pipelining the column of XF_i cells (see equation 13). One way to achieve this is to use

$$\tilde{\triangle}_{2n}Q\backslash group_2 = \|_{i<n}\,[Q^{n+i},\ Q^{n-i-1}]\backslash frev$$

to deduce that

$$\mathsf{fst}\ \tilde{\triangle}Q;\ \mathsf{fsth}\updownarrow_{i<2n} R_i;\ \mathsf{fst}\ bend$$

$$= \left(\updownarrow_{i<n} R_i'\right)\backslash\backslash\mathsf{snd}(frev;\ group_2^{-1});\ \mathsf{fst}\ bend$$

where $R_i' = \mathsf{fst}[Q^{n+i},\ Q^{n-i-1}];\ \mathsf{sndh}\ R_{n-i-1}^{\mathfrak{JC}} \updownarrow \mathsf{fsth}\ R_{n+i}$. Substituting into this equation with $Q = \mathfrak{D}$ and $R = XF_i;\ \mathsf{fst}\ \mathfrak{D}$ then completes the major development steps for *DV4cell*.

There is a further opportunity for speed improvement: as discussed in the preceding section, equation 12 enables the peripheral transposition to be expressed as a row of wiring cells which can then be pipelined if desired.

The amount of pipelining in this circuit can also be controlled by grouping cells into clusters before pipelining and transposition. If each cluster contains K cells, then the array has $N(N + 2)/K$ latches, N/K cycles latency and $T_B + T_X + KT_F$ critical path delay. A *DV4cell* pipelined by every two adders is shown in figure 12.

Figure 13 summarizes the relationship between the divider designs that we discussed, and table 1 compares their salient features.

6.4. Performance Evaluation

We are developing a suite of computer-based tools to support the design framework described in this paper. An overview of these tools is shown in figure 14, and further details can be found in [3], [4] and [13]. The design system has been used to implement the divider designs on a CHS2x4 board which contains several field-programmable gate arrays known as Configurable Array Logic (CAL) chips. Table 2 compares the features of *DV0cell* (with horizontal outputs latched), *DV1cell* and *DV4cell*, where N is the number of adders in each design. Since it is difficult to measure the maximum operation speed using our current system, the critical path delays are estimated using the manufacturer's data [14]: 2 nanoseconds for CAL cell routing delay, and 8 nanoseconds for CAL cell computational delay. In the table both *DV1cell* and *DV4cell* are pipelined by $N/2$ adders; the corresponding CAL implementations for $N = 4$ are shown in figure 15.

One can deduce from table 2 that for large N, both *DV1cell* and *DV4cell* need 50 percent more CAL cells than *DV0cell*, and both of them have a latency twice as much as that of *DV0cell*. On the other hand, *DV1cell* is 1.8 times faster than *DV0cell*, while *DV4cell* is 2.2 times faster than *DV0cell*. For instance, a divider with an 8-bit divisor ($N = 9$) can compute 2.6 million divisions per second when implemented as a *DV0cell*; the corresponding figures for a *DV1cell* and a *DV4cell* are respectively 4.6 and 5.7 million divisions per second. Since the latched fulladder used in a *DV4cell* has a critical path delay of only 44 nanoseconds, it can be used with up to three routing cells to form the basic unit of a fully-pipelined divider circuit operating at 20 million divisions per second.

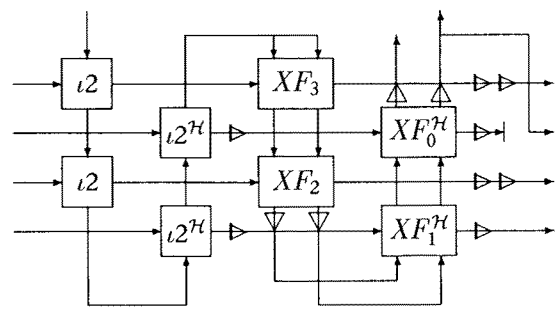


Fig. 12. Design *DV4cell* with partial pipelining.

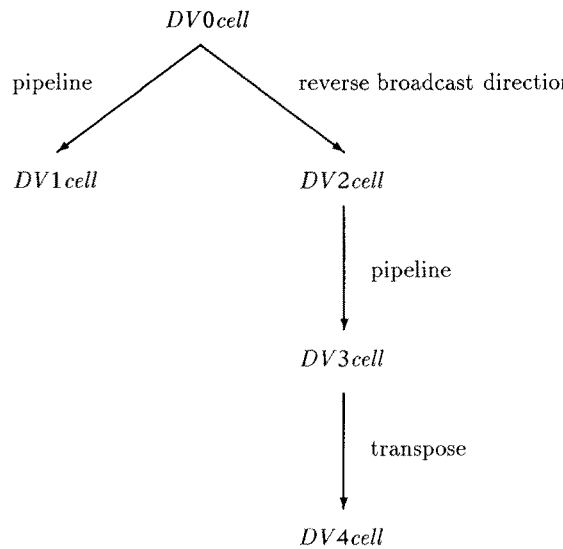


Fig. 13. Relating divider designs.

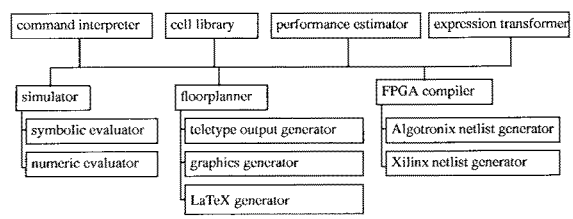


Fig. 14. Overview of design tools.

Table 2. Comparing divider designs in Configurable Array Logic.

Design	Number of CAL cells	Latency	Critical Path Delay
DV0cell	$8N + 8$	1 cycle	$40N + 18$ ns
DV1cell	$12N + 18$	2 cycles	$22N + 20$ ns
DV4cell	$12N + 24$	2 cycles	$18N + 12$ ns

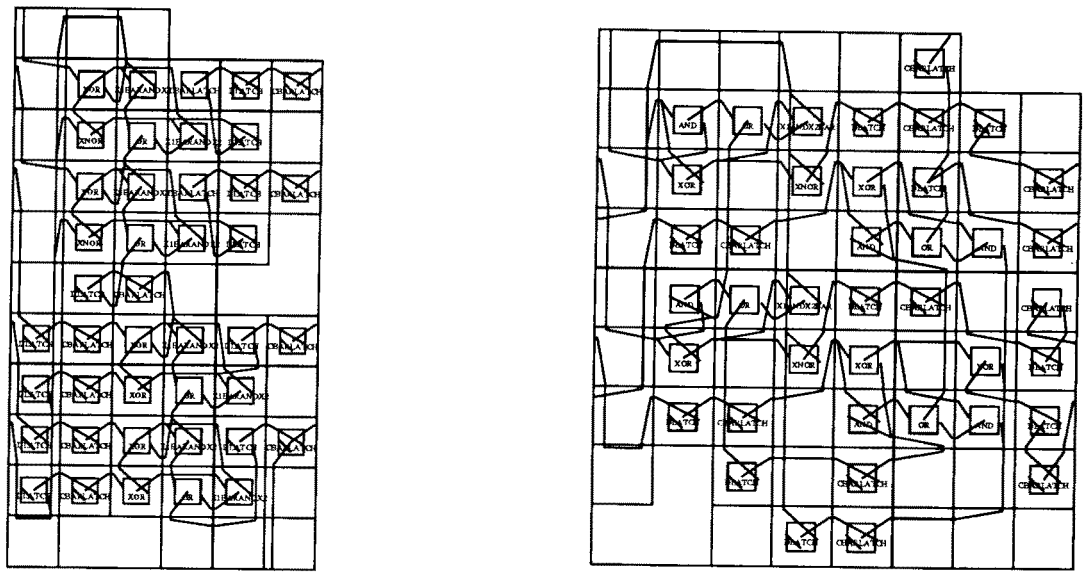


Fig. 15. CAL implementation of *DV*1*cell* and *DV*4*cell*.

6.5. *Other Possibilities*

Three remarks are in order. First, the above development can also be used for a division circuit with the divisor supplied externally. Only minor changes need to be made to the descriptions for capturing the propagation of the divisor from one *DV*0*cell* to another *DV*0*cell* and also to *AFcell*; so for instance, *shl* and *BXF* will become

$$shl2 = \mathsf{col}(\mathsf{fstv}\ \iota 2),$$

$$BXF2 = \mathsf{snd}\ bend^{-1};$$

$$\mathsf{col}(X2 \leftrightarrow (\mathsf{fst}\ shr;\ \mathsf{fstv}\ fadd)),$$

$$X2 = \mathsf{sndv}(fork;\ [\pi_2,\ fork;\ [xor,\ \pi_1]]).$$

However, a fully-pipelined column of adder cells need N^2 extra skewing latches for the lines carrying the divisor. This outweighs the use of an N-bit register to hold the divisor, although it is more flexible as different divisors can be used in successive clock cycles. Moreover, for a field-programmable gate array implementation it may be preferable to reconfigure the array to accommodate different values of divisors, if the divisor is updated relatively infrequently.

The second remark concerns the efficiency of our implementation. In addition to the nonrestoring division array described in this section, two other common parallel dividers are the carry-lookahead array (which is also based on the nonrestoring principle) and the restoring division array [11]. The carry-lookahead array removes the need for pipelining the adders by using carry-save and carry-lookahead adders. However, the circuit is more complex and less regular than the simple rectangular array of *DV*0, and there is still the problem of broadcast delay. As for the restoring array, each of its cells consists of a column of fulladders whose final carry output controls an adjacent column of multiplexers. The column of adders can be pipelined in the same way as *DV*1*cell*, but changing the direction of broadcasting the control signal for the multiplexers does not lead to as much reduction in the number of latches as in *DV*2*cell* above.

Our final remark is about the proposed optimization scheme. The methods of pipelining and transposition appear to be applicable in general to circuits involving broadcasting such as other arithmetic arrays [11] and semi-systolic arrays [6].

7. Conclusion

We have presented a framework for optimizing array-based circuits. The key objectives of this framework are to provide a uniform description of arrays of heterogeneous and homogeneous components, to devise procedures for transforming these descriptions, and to explore the trade-offs involved in these transformations. The techniques discussed formalize and generalize design experience; in particular, they allow incorporating the

appropriate degree of pipelining and transposition to cater for specific applications. The approach is supported by computer-based tools which enable designs to be developed and checked rapidly.

Current work is centered on two fronts. We are extending our transformation framework to cover design serialization [15] in order to widen the range of solutions that satisfy given constraints, such as requirements on speed, size and shape of the circuit. It is also our intention to interface our methods and tools to other development techniques, circuit design aids and cell libraries.

Acknowledgments

I thank Andrew Kay, Keshab Parhi, Richard Stamper and the referees for their comments. The support of Rank Xerox (UK) Limited, Scottish Enterprise and Algotronix Limited is gratefully acknowledged.

References

1. G. Jones and M. Sheeran, "Circuit design in Ruby," in (J. Staunstrup, ed.), *Formal Methods for VLSI Design*, North-Holland, 1990, pp. 13–70.
2. W. Luk and G. Jones, "From specification to parametrized architectures," in (G.J. Milne, ed.), *The Fusion of Hardware Design and Verification*, North-Holland, 1988, pp. 267–288.
3. W. Luk, 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*, Englewood Cliffs, NJ: Prentice Hall, 1989, pp. 589–598.
4. W. Luk, "Analyzing parametrized designs by nonstandard interpretation," in (S.Y. Kung et al., eds.), *Proc. International Conference on Application-Specific Array Processors*, IEEE Computer Society Press, 1990, pp. 133–144.
5. W. Luk and G. Jones, "The derivation of regular synchronous circuits," in (K. Bromley, S.Y. Kung, and E. Swartzlander, eds.), *Proc. International Conference on Systolic Arrays*, IEEE Computer Society Press, 1988, pp. 305–314.
6. W. Luk and G. Brown, "A systolic LRU processor and its top-down development," *Science of Computer Programming*, vol. 15, 1990, pp. 217–233.
7. W. Luk, "Optimizing designs by transposition," in (G. Jones and M. Sheeran, eds.), *Designing Correct Circuits*, Berlin/New York: Springer-Verlag, 1991, pp. 332–354.
8. W. Luk, "Specifying and developing regular heterogeneous designs," in (L. Claesen, ed.), *Formal VLSI Specification and Synthesis*, North-Holland, 1990, pp. 391–409.
9. G. Jones 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.
10. C.E. Leiserson and J.B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, 1991, pp. 5–35.
11. K. Hwang, *Computer Arithmetic*, New York: Wiley, 1979.
12. K.K. Parhi, "A systematic approach for design of digit-serial signal processing architectures," *IEEE Trans. on Circuits and Systems*, vol. 38, 1991, pp. 358–375.
13. W. Luk and I. Page, "Parametrizing designs for FPGAs," in (W. Moore and W. Luk, eds.), *FPGAs*, Abingdon, England. Abingdon EE&CS Books, 1991, pp. 284–295.
14. Algotronix Limited, *CAL 1024 Datasheet* 1990.
15. W. Luk, "Systematic serialization of array-based architectures," to appear in *Integration*, Special Issue on Algorithms and VLSI Architectures.

Wayne Luk received his M.A., M.Sc. and D.Phil. degrees in engineering and computing science from Oxford University. He is Rank Xerox Fellow at Oriel College, Oxford and is also a member of the Faculty of Mathematical Sciences of Oxford University. His research interests include algorithms and architectures for processor arrays, and techniques and tools for building reliable high-performance systems.