THE DERIVATION OF REGULAR SYNCHRONOUS CIRCUITS

WAYNE LUK and GERAINT JONES

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

Abstract. An approach to derive parametrised representation of regular synchronous circuits from their specification is presented. A number of word level and bit level rank evaluator designs are developed to illustrate the techniques described.

INTRODUCTION

Design derivation involves recording the steps in developing a product from a specification, and the justification of design decisions in proceeding from one step to the next. This paper outlines a framework for deriving regular synchronous circuits so that they can be described succinctly and derived in a rigorous manner. This will also provide a basis for documenting design commitments and for evaluating design tradeoffs.

A specification describes the desired behaviour of a system. For the sake of clarity an expressive notation should be used to capture the specification precisely and comprehensibly. Our specifications are general predicates on the observable behaviour of the product. Familiar mathematical operators, such as the summation operator $\sum$, can be used in the specification.

Architectures are expressed in a restricted notation which allows efficient structures to be developed by stepwise refinement. The strength of regular architectures is that a relatively small number of predefined structures appears to be sufficient to capture the main design idioms. These structures correspond to common patterns of computation that can be laid out efficiently on a two-dimensional surface; they can be parametrised to cater for requirements of different applications. Our notation also possesses elegant algebraic properties which enable designs to be transformed by simple substitution of equal expressions. In particular, there are theorems that can be used to pipeline a circuit to any desirable degree.

The derivation of designs in our framework consists of two steps: rewriting the specification in terms of predefined structures to obtain a draft architecture, and optimising that architecture by successive correctness-preserving transformations using algebraic theorems. These steps can be repeated for developing, at a lower level of abstraction, architectures that still satisfy the original specification.

SPECIFYING AND DESCRIBING PARAMETRISED ARCHITECTURES

The design process starts with an unambiguous specification of what the system is to accomplish. Predicate logic provides a formalism to specify requirements clearly and precisely. A simple adder, for instance, can be defined by $ADD(x,y,z) \stackrel{\text{def}}{=} z = x + y$. For many systems, such as those for signal processing, this step is normally quite straightforward as the operations to be performed are generally well understood.

Next comes the problem of describing an architecture. Expressing both specification and architecture in the same notation simplifies the task of associating them. An architecture f is described by a binary relation [4,5], and is defined in the form $x\, f\, y \stackrel{\text{def}}{=} P(x,y)$ where x, y represent the interface signals and belong to $dom(f)$ and $rng(f)$ (domain and range of f) respectively, and P is a predicate describing the intended behaviour. The design process can then be seen as finding an appropriate expression for f that relates the interface signals x, y according to the specification P [2].

CH2603-9/88/0000/0305$01.00 © 1988 IEEE

The converse f^{-1} of a circuit f is defined by $x\,(f^{-1})\,y \stackrel{\text{def}}{=} y\,f\,x$ and corresponds to reflecting f in a line orthogonal to the direction of signal flow. If f happens to be a function, then $f\,x$ represents the value of f for the argument x. If there is a need to distinguish infix operators from relations the former will be underlined, so $y = a\,\underline{f}\,b \stackrel{\text{def}}{=} y = f\,\langle a, b\rangle \equiv \langle a, b\rangle\,f\,y$.

For a circuit with ports on every side of its bounding box, the following convention is used to partition the interface signals into domain and range: signals for the west or north side are allocated to the domain, and those for the south or east side to the range. If there are two or more signals in the domain (or range), the signals are represented as *tuples* with the position of a particular signal corresponding to its relative position, and the tuple structure – the grouping of signals – reflecting the logical organisation of adjacent signals. As usual, subscripting a tuple with i extracts its i^{th} element (i starts from zero), so x_0 is the first element of tuple x. $x_{i,j}$ is defined to be $(x_i)_j$. The number of elements in x is written as $\#x$.

The empty tuple is denoted by $\langle\rangle$. Tuples are formed by constructors *append left* (app_L) and *append right* (app_R), which append an element to the left and to the right of a tuple respectively, so that

$$a\ \underline{app_L}\ \langle b, c, d\rangle = \langle a, b, c\rangle\ \underline{app_R}\ d = \langle a, b, c, d\rangle.$$

The following operations are used to manipulate tuples (I is defined to be $dom(x)$, and in case of *zip*, $\forall\, i \in I.\ dom(x_i) = J$):

$$\begin{array}{rcll}
x\ el_n\ z & \stackrel{\text{def}}{=} & z = x_{n-1} & \text{(select } n^{th}\ x, 1 \le n \le \#x\text{)},\\
x\ zip\ z & \stackrel{\text{def}}{=} & z = \langle\langle x_{i,j} | i \in I\rangle | j \in J\rangle & \text{(interleaving elements of } x\text{)},\\
\langle y, x\rangle\ dist_L\ z & \stackrel{\text{def}}{=} & z = \langle\langle y, x_i\rangle | i \in I\rangle & \text{(distribute } y \text{ to left of each } x_i\text{)},\\
\langle x, y\rangle\ dist_R\ z & \stackrel{\text{def}}{=} & z = \langle\langle x_i, y\rangle | i \in I\rangle & \text{(distribute } y \text{ to right of each } x_i\text{)},\\
x\ rev\ z & \stackrel{\text{def}}{=} & z = \langle x_{\#I-i-1} | i \in I\rangle & \text{(reverse } x\text{)},\\
x\ id\ z & \stackrel{\text{def}}{=} & z = x & \text{(identity relation)}.
\end{array}$$

These can be used to describe organisation of data, or can be implemented as wiring cells.

Circuits are combined by *circuit combinators* which are higher order functions mapping components to the circuit in which they are wired together. The definitions of some circuit combinators are given in Figure 1; because of lack of space the base case of inductive definitions is not given. These circuit combinators correspond to simple and regular patterns of interconnections, some of which are shown in Figure 2. They are parametrised by the type of components that they combine. Some of them, such as pipe, map, triangle and reduce, are parametrised by the array size as well: each combinator describes a family of circuits with the same structure.

Three wiring cells will be named: $tj2 \stackrel{\text{def}}{=} [id, id]$, which corresponds to a T-junction of two branches, $id2 \stackrel{\text{def}}{=} id \parallel id$, which corresponds to extending two buses of wires, and $rev2 \stackrel{\text{def}}{=} [el_2, el_1]$, which corresponds to a cross-over circuit. They can be used to describe common wiring patterns; $tj_H^{(N)} \stackrel{\text{def}}{=} (/_V^{(N)} tj2^{-1})^{-1}$ and $tj_V^{(N)} \stackrel{\text{def}}{=} (/_H^{(N)} tj2^{-1})^{-1}$, for instance, represent circuits that broadcast a signal to $N+1$ destinations.

For systems with sequential elements, we use streams – infinite tuples with the innermost subscript representing time – to describe and reason about them. A stream operator relates a single stream in its domain to a single stream in its range. Square brackets will be used to indicate the interleaving of a tuple of streams to form a stream of tuples, so that $[[x, y], z]_t = \langle\langle x_t, y_t\rangle, z_t\rangle$. A stream operator will be denoted by capitalising the first letter of the corresponding static operator; thus $[x, y]\ Add\ z \stackrel{\text{def}}{=} \forall\, t.\ z_t = x_t\ \underline{add}\ y_t = x_t + y_t$. The same symbols are used for both stream and static circuit combinators – no confusion should arise since stream combinators will be confined to stream components and static combinators will be confined to static components.

A delay $\mathcal{D}$ is defined by $x\mathcal{D}y \stackrel{\text{def}}{=} \forall\, t.\ y_t = x_{t-1}$. x_t's with $t < 0$ can be regarded as undefined values or values defined by initialisation. An *anti-delay* $\mathcal{D}^{-1}$ is such that $\mathcal{D};\ \mathcal{D}^{-1} = \mathcal{D}^{-1};\ \mathcal{D} = Id$. 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. From its definition $\mathcal{D}$ can be used on all types of signals, so

$$
\begin{array}{rcll}
x\,(f;\,g)\,z & \stackrel{\text{def}}{=} & \exists y.\,(x\,f\,y) \wedge (y\,g\,z) & \text{(sequential composition)} \\
\langle x, y\rangle\,(f \parallel g)\,\langle x', y'\rangle & \stackrel{\text{def}}{=} & (x\,f\,x') \wedge (y\,g\,y') & \text{(parallel composition)} \\
\mathbf{fst}\,f & \stackrel{\text{def}}{=} & f \parallel id & \text{(apply to first)} \\
\mathbf{snd}\,f & \stackrel{\text{def}}{=} & id \parallel f & \text{(apply to second)} \\
x\,[f, g]\,\langle p, q\rangle & \stackrel{\text{def}}{=} & (x\,f\,p) \wedge (x\,g\,q) & \text{(construction)} \\
f^{n+1} & \stackrel{\text{def}}{=} & f;\,f^{n} & \text{(pipe)} \\
app_L;\,\propto^{(n+1)} f & \stackrel{\text{def}}{=} & f \parallel \propto^{(n)} f;\,app_L & \text{(map)} \\
app_R;\,\Delta_L^{(n+1)} f & \stackrel{\text{def}}{=} & \mathbf{fst}(\propto f;\,\Delta_L^{(n)} f);\,app_R & \text{(left triangle)} \\
app_L;\,\Delta_R^{(n+1)} f & \stackrel{\text{def}}{=} & \mathbf{snd}(\propto f;\,\Delta_R^{(n)} f);\,app_L & \text{(right triangle)} \\
\langle a, \langle b, c\rangle\rangle(f \rightarrow g)\langle\langle p, q\rangle, r\rangle & \stackrel{\text{def}}{=} & \exists s.\,\langle a, b\rangle f\langle p, s\rangle \wedge \langle s, c\rangle g\langle q, r\rangle & (f \text{ beside } g) \\
\langle\langle a, b\rangle, c\rangle(f \downarrow g)\langle p, \langle q, r\rangle\rangle & \stackrel{\text{def}}{=} & \exists s.\,\langle b, c\rangle f\langle s, r\rangle \wedge \langle a, s\rangle g\langle p, q\rangle & (f \text{ above } g) \\
f \underline{\downarrow} g & \stackrel{\text{def}}{=} & el_1^{-1};\,((el_1;\,f) \downarrow (g;\,el_2^{-1}));\,el_2 & \text{(flat above)} \\
\mathbf{snd}\,app_R;\,\backslash_H^{(n+1)} f & \stackrel{\text{def}}{=} & (\backslash_H^{(n)} f \rightarrow f);\,\mathbf{fst}\,app_R & \text{(horizontal array)} \\
\mathbf{fst}\,app_L;\,\backslash_V^{(n+1)} f & \stackrel{\text{def}}{=} & (\backslash_V^{(n)} f \downarrow f);\,\mathbf{snd}\,app_L & \text{(vertical array)} \\
/_H f & \stackrel{\text{def}}{=} & \backslash_H(f;\,el_2^{-1});\,el_2 & \text{(horizontal reduce)} \\
/_V f & \stackrel{\text{def}}{=} & \backslash_V(f;\,el_1^{-1});\,el_1 & \text{(vertical reduce)} \\
\backslash^{(m,n)} f & \stackrel{\text{def}}{=} & \backslash_V^{(m)}(\backslash_H^{(n)} f) & \text{(rectangular array)} \\
app_R \parallel app_R;\,\mathbf{trail}^{(n+1)}(P, Q) & \stackrel{\text{def}}{=} & (\backslash_H P \rightarrow Q) \downarrow (\mathbf{trail}^{(n)}(P, Q) \rightarrow \backslash_V P);\; app_R \parallel app_R & (Q\text{'s on trailing diagonal of a square array of } P\text{'s})
\end{array}
$$

Note: prefix circuit combinators have a higher binding power than infix ones, and sequential composition has the least binding power, so $\propto f;\ g \rightarrow h$ means $(\propto f);\ (g \rightarrow h)$. Bracketed superscripts are size parameters, often omitted when they can be deduced from environment.

Figure 1. Definitions of circuit combinators

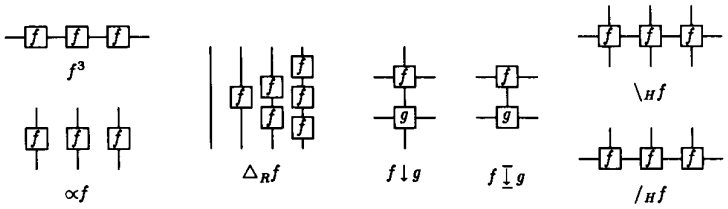


Figure 2. Pictures of some circuit combinators

that for example $\mathcal{D};\propto F = \propto\mathcal{D};\propto F = \propto(\mathcal{D};\,F)$.

DERIVING DESIGNS

A specification needs to be recast in circuit combinators before the algebraic theorems outlined below can be applied. This can usually be done by the method of *divide-and-conquer*: decompose the specification into components which can be implemented by known architectures, and combine the architectures by circuit combinators. For instance, if specification $\mathcal{P}$ can be decomposed into $\mathcal{P}1$ and $\mathcal{P}2$ which can be implemented by architectures $P1$ and $P2$ respectively, that is, if

$$
\begin{aligned}
\mathcal{P}(x, y) &\equiv \exists z.\,\mathcal{P}1(x, z) \wedge \mathcal{P}2(y, z) \\
&\equiv \exists z.\,(x\,P1\,z) \wedge (z\,P2\,y),
\end{aligned}
$$

then $\mathcal{P}(x, y) \equiv x\,(P1;\,P2)\,y$. A designer will know how to implement certain specifications directly: for example given that $\forall t.\ dom(x_t) = K$, then

$$
\forall t.\ y_t = \sum_{0 \le k < \#K} x_{t,k} \quad \equiv \quad x\,([!0, Id];\,/_H Add)\,y \quad \text{where } \langle p, q\rangle\,add\,r \stackrel{\text{def}}{=} p + q = r.
$$

$(x\,(!c)\,y$ is defined to be $\forall t.\, y_t = c$.) This theorem can be proved by induction on the size of x. Of course, this is one of many possible ways to implement summation.

The algorithm is fixed once the specification has been expressed in circuit combinators that define the shape of the circuit. The expression can then be refined by algebraic theorems which state the behavioural equivalence of distinct expressions. Circuits are transformed by substitution of equals using these theorems. The theorems discussed in this paper can be divided into several classes.

The two sides of a *bracketing theorem* represent different views of the same circuit layout; it is like bracketing the same circuit layout in different ways. The most familiar example is the associative law of sequential composition: $f;\,(g;\,h) = (f;\,g);\,h$. Other examples include distributive laws like $\propto(f;\,g) = \propto f;\propto g$, and $\backslash_H(\mathsf{snd}g;\,f) = \mathsf{snd}\propto g;\,\backslash_H f$ and $(/_V^{(n)}f^{-1})^{-1} \mathbin{\underline{\top}} (/_H^{(n)}g) = (f \mathbin{\underline{\top}} g)^n$.

Flipping theorems express the reflectional symmetries in regular structures; they relate circuits to their mirror images. Some examples are $(f;\,g)^{-1} = g^{-1};\,f^{-1}$ and $(\backslash_V f)^{-1} = \backslash_H(f^{-1})$.

Zipping theorems relates circuits with connections interleaved in different ways and can be used to interleave bundles of wires to minimise their length and the number of cross-overs. Here are a couple of examples to convey their flavour: $(zip;\,f;\,zip)^n = zip;\,f^n;\,zip$ and $\backslash_H(zip;\propto f;\,zip) = \mathsf{snd}zip;\,(zip;\propto(\backslash_H f);\,zip);\,\mathsf{fst}zip$.

Retiming is a well-known technique to achieve pipelining by relocating latches to reduce combinational rippling. For systems containing no primitives which possess a measure of absolute time, adding n delays to every domain signal and n anti-delays to every range signal (or vice versa) will not alter its behaviour: $\mathcal{D}^n;\,F;\,\mathcal{D}^{-n} = \mathcal{D}^{-n};\,F;\,\mathcal{D}^n = F$. *Retiming theorems* for our circuit combinators can then be derived, such as

$$F^n = (F;\,\mathcal{D})^n;\,\mathcal{D}^{-n}, \tag{1}$$

$$\backslash_H^{(n)}F = \mathsf{snd}\,\triangle_R\mathcal{D};\,\backslash_H^{(n)}(F;\,\mathsf{snd}\,\mathcal{D});\,\triangle_R\mathcal{D}^{-1} \parallel \mathcal{D}^{-n}, \tag{2}$$

$$\backslash^{(m,n)}F = \triangle_L\mathcal{D} \parallel \triangle_R\mathcal{D};\,\backslash^{(m,n)}(F;\,\mathcal{D});\,(\mathcal{D}^{-m};\,\triangle_R\mathcal{D}^{-1}) \parallel (\mathcal{D}^{-n};\,\triangle_L\mathcal{D}^{-1}), \tag{3}$$

$$F^{k\times n} = (F^k;\,\mathcal{D})^n;\,\mathcal{D}^{-n}, \tag{4}$$

$$\backslash_H^{(n)}(\backslash_H^{(k)}F) = \mathsf{snd}\,\triangle_R\mathcal{D};\,\backslash_H^{(n)}(\backslash_H^{(k)}F;\,\mathsf{snd}\,\mathcal{D});\,\triangle_R\mathcal{D}^{-1} \parallel \mathcal{D}^{-n}, \tag{5}$$

$$\backslash^{(m,n)}(\backslash^{(k,k)}F) = \triangle_L\mathcal{D} \parallel \triangle_R\mathcal{D};\,\backslash^{(m,n)}(\mathsf{trail}^{(k)}(F,\,F;\,\mathcal{D}));\,(\mathcal{D}^{-m};\,\triangle_R\mathcal{D}^{-1}) \parallel (\mathcal{D}^{-n};\,\triangle_L\mathcal{D}^{-1}). \tag{6}$$

These theorems are still true if all delays are changed to anti-delays and anti-delays to delays. The significance of theorems (4–6) is that a single expression, such as $(F^k;\,\mathcal{D})^n$, represents an array with elements which are themselves arrays so that the degree of pipelining can be controlled by adjusting the size of the component arrays. See [2] for a more detailed discussion of this topic.

Other theorems correspond to well-known numerical identities, such as for negation and addition of integers: $\propto negate;\,add = add;\,negate$.

DATA REFINEMENT

One often designs a word level circuit first and then refines it to bit level. Data refinement is a useful technique that allows bit level architectures to be developed from the corresponding word level architectures. The essential idea is to use an *abstraction function* to map concrete data (such as bits) to abstract data (such as integers), and to ensure that for a given concrete signal x the abstract representation of the range signal of the concrete operation f_c is the same as the range signal of the corresponding abstract operation f_a on abstract data: $abs\,(f_c\,x) = f_a\,(abs\,x)$.

An abstraction function mapping a tuple of bits to a number can be defined as

$$\begin{aligned} abs\,\langle\rangle &\stackrel{\text{def}}{=} 0, \\ abs\,(xs\,\underline{app_R}\,x) &\stackrel{\text{def}}{=} 2 \times abs\,xs + x \end{aligned}$$

where $x \in \{0,1\}$ and $\forall i \in dom(xs).\, xs_i \in \{0,1\}$. The first element of the tuple is the most significant bit.

To illustrate the design procedure, consider developing a bit level architecture *adds* to increment a number, represented by a tuple of bits, by one or zero. That is, given that xs is a tuple of bits and that y is a single bit, *adds* has to satisfy

$$abs\,(adds\,\langle xs, y\rangle) = abs\,xs\ +\ y. \tag{7}$$

To find an expression for *adds*, consider the induction case of right hand side of (7):

$$\begin{aligned} abs\,(xs\ \underline{app_R}\ x)\ +\ y &= 2 \times abs\,xs\ +\ x\ +\ y \\ &= 2 \times (abs\,xs\ +\ c)\ +\ s \quad \text{where } 2 \times c + s = x + y \\ &= 2 \times abs\,(adds\,\langle xs, c\rangle)\ +\ s \quad \text{(by induction hypothesis)} \\ &= abs\,((adds\,\langle xs, c\rangle)\ \underline{app_R}\ s) \\ &= abs\,(p\ \underline{app_L}\ ps\ \underline{app_R}\ s) \quad \text{where } p\ \underline{app_L}\ ps = adds\langle xs, c\rangle, \end{aligned}$$

so if $adds\,\langle xs\ \underline{app_R}\ x, y\rangle = p\ \underline{app_L}\ ps\ \underline{app_R}\ s$ then (7) will be satisfied. Moreover, if

$$\begin{aligned} \langle x, y\rangle\ hadd\,\langle c, s\rangle &\stackrel{\text{def}}{=} x + y = 2 \times c + s, \quad \text{then since} \\ (\backslash_V f)\langle as\ \underline{app_R}\ a, b\rangle &= \langle b', as'\ \underline{app_R}\ a'\rangle \quad \text{where } \langle b', as'\rangle = (\backslash_V f)\langle as, z\rangle\ \wedge\ \langle z, a'\rangle = f\langle a, b\rangle, \end{aligned}$$

the definition $adds \stackrel{\text{def}}{=} \backslash_V hadd$; app_L will satisfy the induction case of (7). This definition also satisfies the base case of (7).

As another example, consider implementing a function sm by a bit level architecture *smbs*. Given two numbers x and y, $x\ \underline{sm}\ y \stackrel{\text{def}}{=} 1$ if $x < y$, otherwise $x\ \underline{sm}\ y \stackrel{\text{def}}{=} 0$. This time let us represent the most significant bit as the *last* element of the tuple. The abstraction function then becomes $abs'\,x \stackrel{\text{def}}{=} abs\,(rev\,x)$. *smbs* should satisfy

$$smbs\,\langle xs, ys\rangle = (abs'\,xs)\,\underline{sm}\,(abs'\,ys). \tag{8}$$

smbs can be derived by noting that

$$abs'\,(xs\ \underline{app_R}\ x)\ <\ abs'\,(ys\ \underline{app_R}\ y)\ \equiv\ (x < y)\ \vee\ ((x = y)\ \wedge\ (abs'\,xs\ <\ abs'\,ys)),$$

which gives

$$abs'\,(xs\ \underline{app_R}\ x)\ \underline{sm}\ abs'\,(ys\ \underline{app_R}\ y)\ =\ (x\ \underline{smbit}\ y)\ \underline{or}\ ((x\ \underline{eq}\ y)\ \underline{and}\ (abs'\,xs\ \underline{sm}\ abs'\,ys))$$

where $x\ \underline{smbit}\ y = y\ \underline{and}\ (not\ x)$, $x\ \underline{eq}\ y \stackrel{\text{def}}{=} 1$ if $x = y$ otherwise $x\ \underline{eq}\ y \stackrel{\text{def}}{=} 0$, and $\underline{and}$, $\underline{or}$ and *not* are bit level and-gates, or-gates and inverters respectively. Furthermore it can be shown that $smbs \stackrel{\text{def}}{=} [!0, zip];\ /_H smb$ where $p\ \underline{smb}\,\langle q, r\rangle = (q\ \underline{smbit}\ r)\ \underline{or}\ ((q\ \underline{eq}\ r)\ \underline{and}\ p)$ satisfies (8), which follows from

$$\left.\begin{aligned} g\,\langle\langle\rangle, \langle\rangle\rangle &= c \\ g\,\langle xs\ \underline{app_R}\ x,\ ys\ \underline{app_R}\ y\rangle &= f\,(g\,\langle xs, ys\rangle, \langle x, y\rangle) \end{aligned}\right\} \ \equiv\ g = [!c, zip];\ /_H f.$$

RANK EVALUATION: SPECIFICATION AND WORD LEVEL DESIGNS

A circuit for rank evaluation can be specified as follows [3]: given a window of length $N + 1$ and input x, compute output y such that

$$\forall t.\ y_t = \sum_{1 \le n \le N} x_{t-n}\ \underline{sm}\ x_t.$$

A preliminary architecture can be derived by decomposing the specification into several stages each of which can be directly implemented by the architectures on the left hand side of the following equations:

$$\begin{aligned} x\ (Tj2;\ \mathsf{snd}(Tj_H^{(N)};\ El_1))\ ya &\equiv \forall t.\ ya_t = \langle x_t, xs_t\rangle \ \text{ where } \forall n : 0 \le n < N.\ xs_{t,n} = x_t, \\ ya\ (Dist_L)\ yb &\equiv \forall t, n : 0 \le n < N.\ yb_{t,n} = \langle x_t, x_t\rangle, \\ yb\ (\triangle_R \mathsf{fst}\mathcal{D};\ \propto \mathsf{fst}\mathcal{D})\ yc &\equiv \forall t, n : 0 \le n < N.\ yc_{t,n} = \langle x_{t-n-1}, x_t\rangle, \\ yc\ (\propto Sm)\ yd &\equiv \forall t, n : 0 \le n < N.\ yd_{t,n} = x_{t-n-1}\ \underline{sm}\ x_t, \\ yd\ ([!0, Id];\ /_H Add)\ y &\equiv \forall t.\ y_t = \sum_{0 \le n < N} x_{t-n-1}\ \underline{sm}\ x_t = \sum_{1 \le n \le N} x_{t-n}\ \underline{sm}\ x_t. \end{aligned}$$

The intermediate streams ya, yb, yc and yd can be eliminated by sequential composition to give

$$x\ Re0\ y \equiv \forall t.\, y_t = \sum_{1 \le n \le N} x_{t-n}\ \underline{sm}\ x_t, \text{ where}$$

$$Re0 \stackrel{\text{def}}{=} Tj2;\ \mathsf{snd}(Tj_H^{(N)};\ El_1);\ Dist_L;\ \triangle_R \mathsf{fst}\mathcal{D};\ \propto \mathsf{fst}\mathcal{D};\ \propto Sm;\ [!0, Id];\ /_H Add.$$

By implementing $Dist_L$ as an array of wiring cells for broadcasting: $Dist_L = \backslash_H[Id2, El_1];\ El_1$, by noting that

$$\mathsf{snd}(\propto F;\ \triangle_R G);\ \backslash_H[Id2, El_1];\ El_1 = \backslash_H[Id2, El_1];\ El_1;\ \propto \mathsf{snd}F;\ \triangle_R \mathsf{snd}G,$$

and by using theorem (2) and other bracketing theorems, $Re0$ can be transformed to $Re1$:

$$Re1 \stackrel{\text{def}}{=} [!0, Tj2];\ Re1cell^N;\ El_1 \text{ where } Re1cell \stackrel{\text{def}}{=} (\mathsf{fst}\mathcal{D};\ [Sm, Id2])\ \underline{\downarrow}\ Add.$$

$Re1$ can be pipelined by every K cells by retiming theorem (4) so that $Re2;\ \mathcal{D}^{-M} = Re1$, where

$$Re2 \stackrel{\text{def}}{=} [!0, Tj2];\ (Re1cell^K;\ \mathcal{D})^M;\ El_1 \quad (\text{given } K > 0 \text{ and } K \times M = N).$$

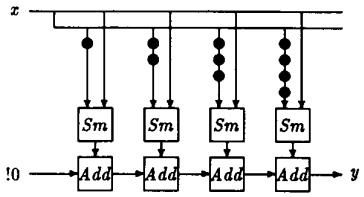


Figure 3. $Re0$

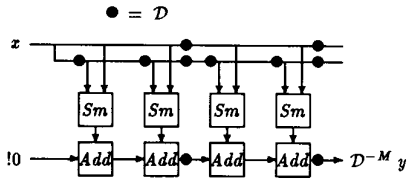


Figure 4. $Re2$ $(K = 2, M = 2)$

A faster circuit can be obtained by pipelining the Sm cell and the adder,

$$Re3 \stackrel{\text{def}}{=} [!0, Tj2];\ Re3cell^N;\ El_1 \text{ where } Re3cell \stackrel{\text{def}}{=} (\mathsf{fst}\mathcal{D};\ [Sm;\ \mathcal{D}, Id2])\ \underline{\downarrow}\ (Add;\ \mathcal{D}),$$

which satisfies $\mathsf{fst}\mathcal{D};\ Re3;\ \mathcal{D}^{-(N+1)} = Re1$.

All circuits derived so far have unidirectional flow data. Circuits with counter-flowing data can be derived by using the following theorem to move the T-junction from the top left-hand corner of $Re1$ to the top right-hand corner:

$$Tj2;\ (/_V(\mathsf{fst}\mathcal{D};\ [F, Id2])^{-1})^{-1};\ El_1 = El_2^{-1};\ (/_V[F, \mathsf{fst}\mathcal{D}^{-1}]^{-1})^{-1};\ \mathsf{snd}\,Tj2^{-1};\ El_1;\ Rev,$$

and since $\mathsf{snd}Rev;\ /_H Add = /_H Add$, $Re1$ can be transformed to $Re4$:

$$Re4 \stackrel{\text{def}}{=} [!0, El_2^{-1}];\ Re4cell^N;\ \mathsf{snd}\,Tj2^{-1};\ El_1 \text{ where } Re4cell \stackrel{\text{def}}{=} [Sm, \mathsf{fst}\mathcal{D}^{-1}]\ \underline{\downarrow}\ Add.$$

$Re4$ can be pipelined in several ways. We can make it 2-*slow* by doubling every delay and anti-delay in the circuit [2] and then apply retiming theorem (1) to give

$$\begin{aligned} Re5 &\stackrel{\text{def}}{=} [!0, El_2^{-1}];\ Re5cell^N;\ \mathsf{snd}\,Tj2^{-1};\ El_1, \\ Re5cell &\stackrel{\text{def}}{=} [Sm, \mathcal{D}^{-1} \parallel \mathcal{D}]\ \underline{\downarrow}\ (Add;\ \mathcal{D}). \end{aligned}$$

However, only one of a pair of adjacent processors is active at any time, hence doubling the cycle time for a particular computation. This can be avoided by pipelining $Re4$ by every K cells ($K > 1$ and $K \times M = N$), so that $Re6;\ \mathcal{D}^{-M} = Re4$, where

$$\begin{aligned} Re6 \stackrel{\text{def}}{=} [!0, El_2^{-1}];\ Re6cell^M;\ \mathsf{snd}\,Tj2^{-1};\ El_1 \quad \text{and} \quad Re6cell &\stackrel{\text{def}}{=} Re4cell^{K-1};\ Re4cell', \\ Re4cell' &\stackrel{\text{def}}{=} [Sm, \mathsf{snd}\mathcal{D}]\ \underline{\downarrow}\ (Add;\ \mathcal{D}). \end{aligned}$$

An alternative way of partially pipelining $Re4$ would have $Re7;\ \mathcal{D}^{-(K-1)M} = Re4$, where

$$\begin{aligned} Re7 &\stackrel{\text{def}}{=} [!0, El_2^{-1}];\ Re7cell^M;\ \mathsf{snd}\,Tj2^{-1};\ El_1, \\ Re7cell &\stackrel{\text{def}}{=} Re4cell'^{K-1};\ Re4cell. \end{aligned}$$

Some simple measures of design tradeoffs for the architectures developed are given in Table 1.

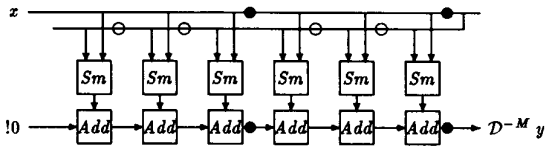


Figure 5. Design *Re*6 ($K = 3$, $M = 2$)

● = $\mathcal{D}$
○ = $\mathcal{D}^{-1}$

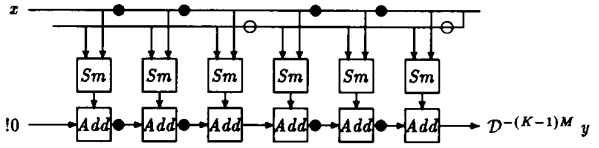


Figure 6. Design *Re*7 ($K = 3$, $M = 2$)

Table 1. Comparison of word level rank evaluator designs

Data flow	*Design*	*Min. clock period (for a particular computation)*	*Latency*	*Slow-down by 2*	*Number of latches in array*
Unidir. flow	*Re*1	$T_{Sm} + NT_{Add}$	N	No	N
	*Re*2 ($K > 0$)	$T_{Sm} + KT_{Add}$	$\frac{N(K+1)}{K}$	No	$\frac{N(K+3)}{K}$
	*Re*3	$max(T_{Sm}, T_{Add})$	$2N + 1$	No	$5N$
Contra-flow	*Re*4	$T_{Sm} + NT_{Add}$	N	No	N
	*Re*5	$2(T_{Sm} + T_{Add})$	$3N$	Yes	3N
	*Re*6 ($K > 1$)	$T_{Sm} + KT_{Add}$	$\frac{N(K+1)}{K}$	No	$\frac{N(K+1)}{K}$
	*Re*7 ($K > 1$)	$max((K-1)T_P + T_{Sm} + T_{Add}, T_{Sm} + 2T_{Add})$	$\frac{N(2K-1)}{K}$	No	$\frac{N(2K-1)}{K}$

N: number of word level rank evaluator cells
K: clustering cells in groups of K cells
T_P: propagation delay across a cell
T_{Sm}, T_{Add}: the combinational delay of cell Sm, Add

RANK EVALUATION: DATA REFINEMENT AND BIT LEVEL DESIGNS

Rel cell can be separated into a combinational part *Rel cella* and a register:

$$\begin{array}{rcl} \mathit{Rel cell} & = & \mathsf{snd}(\mathsf{fst}\mathcal{D});\ \mathit{Rel cella} \quad \text{where} \quad \mathit{rel cella} \stackrel{\text{def}}{=} \mathit{sma} \mathbin{\underline{\uparrow}} \mathit{add} \\ & & \qquad\qquad\qquad\quad \text{and} \qquad \mathit{sma} \stackrel{\text{def}}{=} [\mathit{sm}, \mathit{id}2]. \end{array}$$

rel cellc, the concrete (bit level) implementation of *rel cella*, should satisfy

$$\mathit{abs};\ \mathit{rel cella} = \mathit{rel cellc};\ \mathit{abs}. \tag{9}$$

abs is defined in the section on data refinement. For convenience, we shall use *abs* to denote the abstraction of all bit components in a tuple, for example $\mathit{abs}\,\langle\langle\langle 0,1\rangle,\langle 1,1\rangle\rangle,\langle 1,0\rangle\rangle = \langle\langle 1,3\rangle,2\rangle$.

If *smc* and *addc* are the concrete implementations of *sma* and *add* respectively, then (9) will hold if $\mathit{rel cellc} = \mathit{smc} \mathbin{\underline{\uparrow}} \mathit{addc}$, and $\mathit{smc};\ \mathit{abs} = \mathit{abs};\ \mathit{sma}$ and $\mathit{addc};\ \mathit{abs} = \mathit{abs};\ \mathit{add}$. Since

$$\begin{array}{rcl} \propto\mathit{rev};\ \mathit{abs};\ \mathit{sm} & = & \mathit{smbs}, \\ \mathit{abs};\ \mathit{sm} & = & \propto\mathit{rev};\ \mathit{smbs} \\ & = & \propto\mathit{rev};\ [!0, \mathit{zip}];\ /_H \mathit{smb}. \end{array}$$

Therefore a satisfactory *smc* is

$$\begin{array}{rcl} \mathit{smc} & \stackrel{\text{def}}{=} & [\propto\mathit{rev};\ [!0, \mathit{zip}];\ /_H \mathit{smb}, \mathit{id}2] \\ & = & [\propto\mathit{rev}, \mathit{id}2];\ \mathsf{fst}([!0, \mathit{zip}];\ /_H \mathit{smb}) \quad \text{since } [f;\ g,\ h] = [f, h];\ \mathsf{fst}g, \\ & = & \mathit{zip};\ \propto\mathit{tj}2;\ \mathit{zip};\ (\mathit{rev};\ [!0, \mathit{id}];\ /_H \mathit{smb}) \parallel \mathit{zip} \\ & & \text{since } [\propto\mathit{rev}, \mathit{id}2] = \mathit{zip};\ \propto\mathit{tj}2;\ \mathit{zip};\ (\mathit{rev};\ \mathit{zip}) \parallel \mathit{zip}, \\ & = & \mathit{zip};\ [\mathit{id}, !0];\ \backslash_V \mathit{smcell};\ \mathsf{snd}\mathit{zip} \quad \text{where } \mathit{smcell} \stackrel{\text{def}}{=} [\mathit{rev}2;\ \mathit{smb},\ \mathit{el}_1] \\ & & \text{since } \propto\mathit{tj}2;\ \mathit{zip};\ \mathsf{fst}(\mathit{rev};\ [!c, \mathit{id}];\ /_H f) = [\mathit{id}, !c];\ \backslash_V [\mathit{rev}2;\ f, \mathit{el}_1]. \end{array}$$

addc, the bit level implementation of *add*, is similar to *adds* developed in the section on data refinement except that the inputs are assumed to be small enough not to affect the most significant bit which will be stripped off: $\mathit{abs}\,(\mathit{addc}\langle \mathit{xs}, \mathit{y}\rangle) = \mathit{abs}\ \mathit{xs} + \mathit{y}$ where $\mathit{addc} \stackrel{\text{def}}{=} \backslash_V \mathit{hadd};\ \mathit{el}_2$.

Given that S and A are the number of rows of *Smcell* and *Hadd* cells respectively, and using $Tj_V^{(A)};\ \mathit{El}_2;\ \propto!0$ to implement at bit level the constant generator !0 in the definition of *Re*1, a bit level circuit corresponding to *Re*1 is given by

$$[Tj_V^{(A)};\ \mathit{El}_2;\ \propto!0,\ \mathit{Tj}2];\ (\mathsf{snd}(\mathsf{fst}\mathcal{D});\ \mathit{Rel cellc})^N;\ \mathit{El}_1$$

which can be transformed to *Reb*1 where

$$\begin{array}{rcl} \mathit{Reb}1 & \stackrel{\text{def}}{=} & [\mathit{Id}, [\propto \mathit{Tj}2, \mathit{Id}]];\ \mathit{Sm1array} \mathbin{\underline{\uparrow}} \mathit{Hadd1array};\ \mathit{El}_1, \\ \mathit{Sm1array} & \stackrel{\text{def}}{=} & \mathsf{snd}(\,Tj_H^{(N)};\ \mathit{El}_1;\ \propto!0);\ \backslash^{(S,N)} \mathit{Sm1cell}, \\ \mathit{Sm1cell} & \stackrel{\text{def}}{=} & \mathsf{fst}(\mathsf{fst}\mathcal{D});\ \mathit{Smcell}, \\ \mathit{Hadd1array} & \stackrel{\text{def}}{=} & \mathsf{fst}(\,Tj_V^{(A)};\ \mathit{El}_2;\ \propto!0);\ \backslash^{(A,N)} \mathit{Hadd};\ \mathit{El}_2. \end{array}$$

Using retiming theorem (6), *Reb*1 can be pipelined by every K by K cluster of cells ($K > 0$, $P \times K = S$, $Q \times K = A$ and $M \times K = N$), giving

$$\begin{array}{rcl} \mathit{Reb}2 & \stackrel{\text{def}}{=} & [\mathit{Id}, [\triangle_L\mathcal{D};\ \propto\propto \mathit{Tj}2, \mathit{Id}]];\ \mathit{Sm2array} \mathbin{\underline{\uparrow}} \mathit{Hadd2array};\ \mathit{El}_1;\ \mathcal{D}^{-(P+Q)};\ \triangle_L\mathcal{D}^{-1}, \\ \mathit{Sm2array} & \stackrel{\text{def}}{=} & \mathsf{snd}\ Tj0_H;\ \backslash^{(P,M)}(\mathsf{trail}^{(K)}(\mathit{Sm1cell}, \mathit{Sm1cell};\ \mathcal{D})), \\ Tj0_H & \stackrel{\text{def}}{=} & (/_V^{(M)}(/_V^{(K)}\ Tj2^{-1}))^{-1};\ \mathit{El}_1;\ \propto\propto!0, \\ \mathit{Hadd2array} & \stackrel{\text{def}}{=} & \mathsf{fst}\ Tj0_V;\ \backslash^{(Q,M)}(\mathsf{trail}^{(K)}(\mathit{Hadd}, \mathit{Hadd};\ \mathcal{D}));\ \mathit{El}_2, \\ Tj0_V & \stackrel{\text{def}}{=} & (/_H^{(Q)}(/_H^{(K)}\ Tj2^{-1}))^{-1};\ \mathit{El}_2;\ \propto\propto!0. \end{array}$$

A bit level circuit corresponding to $Re4$ is given by

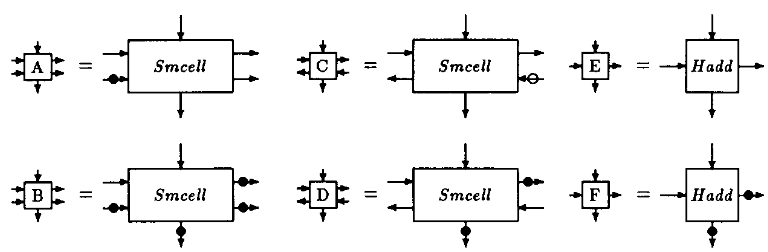
$$\begin{aligned}
Reb3 &\stackrel{\text{def}}{=} [Id, [\propto El_2^{-1}, Id]];\ Sm3array \mathbin{\underline{\downarrow}} Hadd3array;\ El_1,\\
Sm3array &\stackrel{\text{def}}{=} \mathsf{snd}(Tj_H^{(N)};\ El_1;\ \propto !0);\ \backslash^{(S,N)} Sm3cell;\ \mathsf{snd}\propto Tj2^{-1},\\
Sm3cell &\stackrel{\text{def}}{=} Smcell;\ \mathsf{snd}(\mathsf{fst}\mathcal{D}^{-1}),\\
Hadd3array &\stackrel{\text{def}}{=} \mathsf{fst}(Tj_V^{(A)};\ El_2;\ \propto !0);\ \backslash^{(A,N)} Hadd;\ El_2.
\end{aligned}$$

$Reb3$ can be fully pipelined by making it 2-slow and then applying retiming theorem (3):

$$\begin{aligned}
Reb4 &\stackrel{\text{def}}{=} [Id, [\triangle_L\mathcal{D};\ \propto El_2^{-1}, Id]];\ Sm4array \mathbin{\underline{\downarrow}} Hadd4array;\ El_1;\ \mathcal{D}^{-(S+A)};\ \triangle_L\mathcal{D}^{-1},\\
Sm4array &\stackrel{\text{def}}{=} \mathsf{snd}(Tj_H^{(N)};\ El_1;\ \propto !0);\ \backslash^{(S,N)} Sm4cell;\ \mathsf{snd}\propto Tj2^{-1},\\
Sm4cell &\stackrel{\text{def}}{=} Smcell;\ \mathcal{D} \parallel (\mathcal{D}^{-1} \parallel \mathcal{D}),\\
Hadd4array &\stackrel{\text{def}}{=} \mathsf{fst}(Tj_V^{(A)};\ El_2;\ \propto !0);\ \backslash^{(A,N)}(Hadd;\ \mathcal{D});\ El_2.
\end{aligned}$$

Alternatively, using retiming theorem (6) $Reb3$ can be pipelined by every K by K cluster of cells ($K > 1, P \times K = S, Q \times K = A$ and $M \times K = N$), giving

$$\begin{aligned}
Reb5 &\stackrel{\text{def}}{=} [Id, [\triangle_L\mathcal{D};\ \propto\propto El_2^{-1}, Id]];\ Sm5array \mathbin{\underline{\downarrow}} Hadd5array;\ El_1;\ \mathcal{D}^{-(P+Q)};\ \triangle_L\mathcal{D}^{-1},\\
Sm5array &\stackrel{\text{def}}{=} \mathsf{snd}\ Tj0_H;\ \backslash^{(P,M)}(\mathsf{trail}^{(K)}(Sm3cell, Sm5cell));\ \mathsf{snd}\propto Tj2^{-1},\\
Sm5cell &\stackrel{\text{def}}{=} Smcell;\ \mathcal{D} \parallel (\mathsf{snd}\mathcal{D}),\\
Hadd5array &\stackrel{\text{def}}{=} \mathsf{fst}\ Tj0_V;\ \backslash^{(Q,M)}(\mathsf{trail}^{(K)}(Hadd, Hadd;\ \mathcal{D}));\ El_2.
\end{aligned}$$



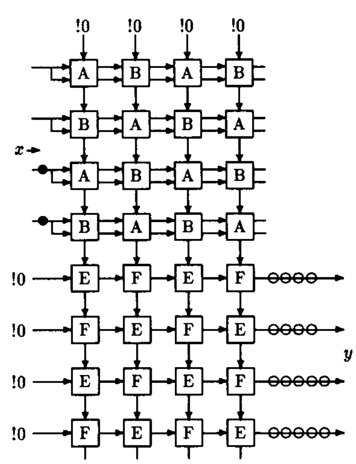


Figure 7. Design $Reb2$ ($K = M = P = Q = 2$)

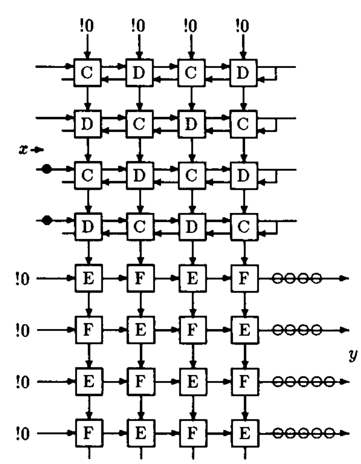


Figure 8. Design $Reb5$ ($K = M = P = Q = 2$)

Some simple measures of design tradeoff for these bit level circuits are given in Table 2.

Table 2. Comparison of bit level rank evaluator designs

Data flow	*Design*	*Min. clock period (for a particular computation)*	*Latency*	*Slow-down by 2*	*Number of latches in array*
Unidir. flow	*Reb2* $(K > 0)$	KT_{max}	$\frac{N(K+1)+S}{K}$	No	$\frac{SN(K+3)+2AN}{K}$
Contra-flow	*Reb4*	$2T_{max}$	$3N + S$	Yes	$3SN + 2AN$
	Reb5 $(K > 1)$	KT_{max}	$\frac{N(K+1)+S}{K}$	No	$\frac{SN(K+1)+2AN}{K}$

N: number of columns of *Smcell*
S: number of rows of *Smcell*
A: number of rows of *Hadd*
K: clustering cells in groups of K by K cells
T_{max}: $max\ (T_{Smcell}, T_{Hadd})$

CONCLUSION

The importance of specifying and verifying regular array designs has been clear [1]. We have presented a framework for the coherent development of a parametrised architecture from the specification of a system. It has the advantage of being modular, separating concerns of accurate specification and of adequate performance. Precision and correctness can be maintained and checked at every stage of the design, and successive transformations summarise the design decisions made at each step. The tradeoff resulting from different design options can be evaluated.

Our work also provides the basis for computer-aided design tools facilitating the construction of regular array circuits. Simulators, floorplanners and transformation assistants have been prototyped; we intend to develop them further to cater for more sophisticated designs.

Acknowledgement. We thank Andrew McCabe for suggesting the rank evaluation problem and Stuart Wilson for discussions on rank evaluator designs. This work has been undertaken as part of the UK Alvey Programme (Project ARCH 013) whose support is gratefully acknowledged. The first author also expresses his gratitude to St. Edmund Hall, Oxford for a Brockhues Senior Scholarship.

References

[1] H.T. Kung, "Why systolic architectures?" *Computer*, vol. 15, no. 1, p. 37, 1982.

[2] W. Luk and G. Jones, "Structuring and reasoning about regular array designs," submitted for publication.

[3] I.N. Parker, "VLSI architecture," in *VLSI image processing*, R.J. Offen (ed.), Collins, London, p. 99, 1985.

[4] M. Sheeran, "Describing and reasoning about circuits using relations," to appear in *Proc. 1986 Leeds workshop on theoretical aspects of VLSI architectures.*

[5] M. Sheeran and G. Jones, "Relations + higher order functions = hardware descriptions," *Proc. CompEuro*, Hamburg, p. 303, 1987.