Calculating a Round-Robin Scheduler

Matteo Vaccari* Roland Backhouse†

December 3, 1997

Abstract

Kropf [7] has collected together a number of problems that can be used as benchmarks for hardware verification, one of which is a bus arbiter. The function of the bus arbiter is to assign the use of a shared resource on each clock cycle to one out of N subsystems that may want to use it, and in such a way that no subsystem is denied access forever. The most significant component in its implementation is a round-robin scheduler. Rather than *verify* the existing implementation of the scheduler, this paper shows how to *construct* a correct implementation from the given requirements. We make use of both point-free and pointwise relation algebra.

1 Introduction

Kropf [7] has collected together a number of problems that can be used as benchmarks for hardware *verification*, one of which is a bus arbiter. The function of the bus arbiter is to assign the use of a shared resource on each clock cycle to one out of N subsystems that may want to use it, and in such a way that no subsystem is denied access forever.

The most significant component in the implementation of the bus arbiter is a round-robin scheduler. Rather than *verify* a given implementation of a scheduler, we consider in this paper the much more instructive problem of *constructing* a scheduler from its specification. The basis of our construction is the algebra of relations; we specify the problem within the algebra and then calculate a correct and efficient implementation.

In the next section we formulate the task and in the section thereafter we present the algebra in which our solution is presented. Subsequently, we outline our calculations and then we give the calculations in detail. The paper is concluded by a discussion of what has been achieved.

*Department of Computing Science, Università degli Studi di Milano, via Comelico 39/41, 20135 Milano, Italy. E-mail: `vaccari@dsi.unimi.it`

†Department of Mathematics and Computing Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands. E-mail: `rolandb@win.tue.nl`

2 The Task

A bus arbiter is a device that should assign the use of a resource at each clock cycle to at most one out of N subsystems that may want to use it, and it should do it so that no subsystem is denied access forever. More specifically, a bus arbiter is a circuit that maps a stream of N boolean inputs, representing requests to use the resource, to N boolean outputs, representing acknowledgements that the resource may be used[1].

Let us call the input stream *req* and the output stream *ack*. It is easier to think of $\mathbb{B}^N$ as a subset of $\{0,\ldots,N{-}1\}$, so we write $n \in req.t$ to mean that the n-th component of *req* is high at time t. The specification of the arbiter, as given by Kropf [7], is then:

1. No two output wires are asserted simultaneously: for each instant t,

$$|\ ack.t\ | \ \leq 1 \quad .$$

2. Acknowledge is not asserted without request:

$$ack.t \subseteq req.t \quad .$$

3. Every persistent request is eventually acknowledged: there is no pair (n, t) such that

$$\forall (t' :\ t \leq t' :\ n \in req.t' \ \wedge\ n \notin ack.t') \quad .$$

Kropf himself suggests an implementation. The arbiter should normally grant acknowledge to the request that is lowest in index, and fall back to a round-robin scheme when there are too many requests. This is accomplished as follows: at any given moment there is a privileged wire. For simplicity, we may take $t \bmod N$ to be the privileged wire at any time t. If wire n is privileged, and is asserting a request, and it was asserting its request the previous time it was privileged, then it is acknowledged. This way any wire will be acknowledged in less than $2N$ clock cycles.

One way to implement this arbiter is to construct two modules, called for instance LT and RR, the first one granting request to the lowest index asserted in its input, and the second one implementing the round-robin algorithm. The first module is combinational, while the second one has state. Module LT simply returns the lowest numbered signal that is asserted.

[1] Note that we are not interested in dealing with metastability problems. Metastability is an effect that may occur whenever a signal is sampled asynchronously; what may happen is that the signal floats in a state that cannot be interpreted as a logical "true" or as a logical "false". It should be clear that this effect cannot be modelled within the framework of this paper. What we call "wire" here is a mathematical abstraction of physical wires.

We focus in this paper on the development of the round-robin scheduler, RR. Suppose $\lhd^N$ is a function that delays an input stream by N clock cycles. That is,

$$(\lhd^N.b).t \quad = \quad b.(t{-}N) \quad .$$

Then, viewing RR as a binary relation between output ack and input req, its specification is

$$ack\langle RR\rangle req \quad \equiv \quad \forall(t{::}\quad ack.t \quad = \quad \{t \bmod N\} \cap req.t \cap (\lhd^N.req).t) \quad .$$

The task we undertake in this paper is to construct a circuit that implements RR as specified above.

3 Relation Algebra

We will write our specifications and our circuits in point-free relation algebra. A brief introduction to our style of relation algebra follows; for a more complete treatment see [1].

A (binary) relation over a set $\mathcal{U}$ is a set of pairs of elements of $\mathcal{U}$. For x,y in $\mathcal{U}$ and R a relation over $\mathcal{U}$, we write $x\langle R\rangle y$ instead of $(x,y){\in}R$. When a relation R satisfies $x\langle R\rangle y \wedge z\langle R\rangle y \;\Rightarrow\; x{=}z$ we say that the relation is *deterministic*. In that case it may be considered as a function with domain on the right side and target on the left side; we denote by $R.y$ the unique x such that $x\langle R\rangle y$ holds, if such an x exists. The reason for this name is that we usually interpret relations as programs taking input from the right and producing output on the left. In this way a deterministic relation is interpreted as a deterministic program. We usually use the letters f, g, h to stand for deterministic relations. We use the convention that "." associates to the right so that $f.g.x$ should be parsed as $f.(g.x)$. (This is contrary to the convention used in the lambda calculus.)

Relations are ordered by the usual set inclusion ordering. Hence the set of relations forms a complete lattice. The relation corresponding to the empty set is denoted by $\perp\!\!\perp$, and the relation that contains all pairs of elements of $\mathcal{U}$ is denoted by $\top\!\!\top$. The *identity relation*, ι, is defined by $x\langle\iota\rangle y \;\equiv\; x{=}y$. The composition of two relations R, S is denoted by $R{\circ}S$ and defined by $x\langle R\circ S\rangle y \;\equiv\; \exists(z :: \; x\langle R\rangle z \wedge z\langle S\rangle y)$. Composition is associative and has unit element ι. The converse of a relation R is written $R^{\cup}$ and is defined by $x\langle R^{\cup}\rangle y \;\equiv\; y\langle R\rangle x$.

A *monotype* is a relation A such that $A \subseteq \iota$. An example of a monotype is $\mathbb{N}$, defined by $n\langle\mathbb{N}\rangle m \quad\equiv\quad n{=}m \wedge (n$ is a natural number$)$. (Bird and De Moor [2] use the name "coreflexive" instead of monotype.) There is a one-to-one correspondence between the subsets of $\mathcal{U}$ and the monotypes; and this makes it possible to embed set calculus in relation calculus. The *left domain* of relation R, denoted $R{<}$, is the least monotype A such that $A \circ R = R$. As its name suggests, $R{<}$ represents the set of all x such that x is related by R to some y. Similarly, the *right domain* of relation R, denoted $R{>}$, is the least monotype A such that $R \circ A = R$.

The relation $R \vartriangle S$ (pronounced R *split* S) is defined as the least relation X such that for all x, y and z, $(x,y)\langle X\rangle z \;\equiv\; x\langle R\rangle z \wedge y\langle S\rangle z$. Note that the requirement that $R \vartriangle S$ be the *least* relation satisfying the above equation in X implies that there is no y such that $x\langle R \vartriangle S\rangle y$ when x is not a pair. That is, the left domain of $R \vartriangle S$ is a set of pairs.

In general, composition does not distribute through split. However, it is the case that

$$(1) \quad (R \vartriangle S) \circ T \;=\; (R \circ T) \vartriangle (S \circ T) \quad \Leftarrow \quad S \circ T \circ T^{\cup} \subseteq S \;\; .$$

The antecedent holds, for example, when T is a deterministic relation (since then $T \circ T^{\cup} \subseteq \iota$). It also holds if S is a so-called *left condition*: that is, if $S = S \circ \top\!\top$.

We define $R{\times}S$ (pronounced R *times* S) by

$$(x,y)\langle R{\times}S\rangle(z,v) \;\equiv\; x\langle R\rangle z \wedge y\langle S\rangle v \;\; .$$

The following properties are easily proved:

$$(2) \quad \begin{array}{rcl} (R{\times}S) \circ (T{\times}U) & = & (R \circ T) \times (S \circ U) \\ (R{\times}S) \circ (T \vartriangle U) & = & (R \circ T) \vartriangle (S \circ U) \;\; . \end{array}$$

Although the concepts of split and product are now standard, there is as yet no standard notation, Bird and De Moor [2] and the Ruby group [4, 5, 10] each using their own brand. The Ruby group, for example, would write

$$(3) \quad [R; S,\, T; U]$$

where we would write

$$(R \circ S) \times (T \circ U) \;\; .$$

This use of a comma to separate arguments is, in our view, highly inadvisable for several reasons, but in particular because of the relative size of the ",". In this example, because "," is smaller than ";" many people incorrectly read (3) as

$$[R; (S,T); V]$$

particularly when, as often happens, the author pays no consideration to spacing. The notation used in this document has been carefully chosen to enhance readability. We have also been careful to space formulae so as to automatically suggest the correct parsing of expressions to the human eye and thus minimise the need to refer to a table of precedences. Nevertheless the latter may on occasion be unavoidable, in which case the table on the following page should be consulted.

$\cdot$ $<$ $>$ $^{\sigma}$	all unary operators
$.$	function application
$\times$ $\vartriangle$	product, split
$\circ$	relational composition
$\cup$ $\cap$	union, intersection
$=$ $\subseteq$	equality, inclusion
$\wedge$ $\vee$	conjunction, disjunction
$\Rightarrow$ $\Leftarrow$	implication, consequence
$\equiv$	boolean equivalence

Table 1: Precedence of operators, from highest to lowest

4 Tuples and generalised products

We now generalise the relational product to n-wide products. A product of a single relation is the relation itself. A product of two relations is the usual relational product. The product of n relations R_0 through R_{n-1} is $R_0 \times (R_1 \times (\ldots \times R_{n-1}))$. By adopting the convention that $\times$ associates to the right, we can write the above product as simply $R_0 \times R_1 \times \ldots \times R_{n-1}$. Corresponding to n-wide products we have n-tuples. For instance, a 3-tuple has the shape $(a, (b, c))$ for some a, b, c. By adopting the convention that pairing associates to the right, we write the above tuple as simply (a, b, c).

We now define a notion of left and right *arity* for relations. First we introduce the (n) family of monotypes:

$$(4) \qquad \begin{array}{lcl} (1) & = & \iota \\ (n+1) & = & \iota \times (n) \quad \text{for } n \geq 1 \end{array} \quad .$$

For any expression E in the positive integers, we can assign a corresponding monotype (E) by means of definition (4). When the expression is just a numeral or a single letter, we can drop parentheses, and write n in place of (n).

We say that R has *right arity* n iff

$$R \;=\; R \circ n \quad .$$

Similarly, we say that R has *left arity* n whenever

$$R \;=\; n \circ R \quad .$$

We can now define new operations. The *map* operation generates the product of n copies of a circuit:

$$\begin{array}{lcl} map_1.R & = & R \\ map_{n+1}.R & = & R \times map_n.R \quad \text{for } n \geq 1 \end{array} \quad .$$

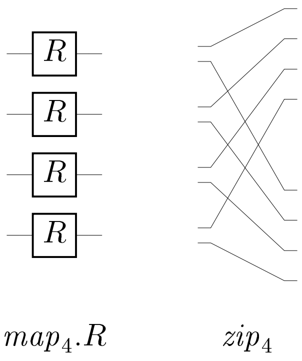

Figure 1: Instances of tuple operations

For every R, we have

$$map_n.R \quad = \quad n \circ map_n.R \quad = \quad map_n.R \circ n \quad .$$

Corresponding to the fusion law (2) we have the *map fusion* law:

$$(5) \qquad map_n.(R \circ S) \quad = \quad map_n.R \circ map_n.S \quad .$$

Generalising from making n copies of a circuit, we also use maps to combine n different circuits. The term

$$map.(k : \ 0 \leq k < n : \ R_k)$$

denotes a circuit with n inputs and n outputs, the relation between the kth input and output being determined by circuit R_k. Note that this more general map also obeys a fusion law like (5).

The *zip* operation is well-known in functional programming. Informally, zip_n transforms a pair of n-tuples into an n-tuple of pairs. One common way to define it is:

$$\begin{array}{lcl} zip_1.(a,b) & = & (a,b) \\ zip_{n+1}.((a,b),(c,d)) & = & ((a,c), zip_n.(b,d)) \quad . \end{array}$$

The above is an acceptable relational definition, since a function is also a relation.

The arity of zip_n is given by

$$zip_n \quad = \quad n \circ zip_n \quad = \quad zip_n \circ n{\times}n \quad .$$

A law about zip_n is

$$(6) \quad zip_n \circ map_n.R \vartriangle map_n.S \;=\; map_n.(R \vartriangle S) \quad ;$$

the proof is by induction on n.

5 About circuits

Following established practice (see [3, 5, 9]) we model a circuit as a relation between arbitrary collections of *streams*, a stream being a total function with domain the integer numbers. Abusing language somewhat, we will use the word "circuit" to mean an actual circuit, or a relation between streams as described above. Context should make clear which one is meant. We usually denote streams by small letters taken from the beginning of the alphabet.

Given a relation R, a relation between streams can be constructed by "lifting": $a\langle\dot{R}\rangle b \equiv \forall(t :: \; a.t \,\langle R\rangle\, b.t)$. Hence for any R, relation $\dot{R}$ is a circuit. Note that, for deterministic relation f, stream a and integer t, $f.a.t = (\dot{f}.a).t$. We refer to this property in our calculations by the hint "lifting". Circuits can be built by relational composition, and product: given R and S, two circuits, the relations $R \circ S$ and $R{\times}S$ are also circuits.

A particular relation on streams is the *primitive delay*, denoted by ∂ and defined by

$$a\langle\partial\rangle b \;\equiv\; \forall(t :: \; a.(t{+}1) = b.t) \quad .$$

The *delay* relation, written $\lhd$, is a generalisation of primitive delay to arbitrary pairings of streams. It is defined as the least fixed point of function $X \mapsto \partial \cup X{\times}X$:

$$\lhd \;=\; \mu(X \mapsto \partial \cup X{\times}X) \quad .$$

Delay can be thought of informally as the union of an infinite list of terms $\lhd \;=\; \partial \cup \partial{\times}\partial \cup \partial{\times}(\partial{\times}\partial) \cup (\partial{\times}\partial){\times}\partial \cup (\partial{\times}\partial){\times}(\partial{\times}\partial) \cup \; \ldots$.

Note that $\lhd$ is deterministic.

The delay relation is polymorphic in the sense that it applies the primitive delay ∂ to a collection of wires, independently of the shape of the collection. Formally,

$$(7) \quad \iota{\times}\iota \circ \lhd \;=\; \lhd{\times}\lhd \quad .$$

(Note that this and other properties of delays are proved in [11].)

Finally, the *feedback* of a circuit R, written R^σ, is defined by

$$a\langle R^\sigma\rangle b \;\equiv\; a\langle R\rangle(b,a) \quad .$$

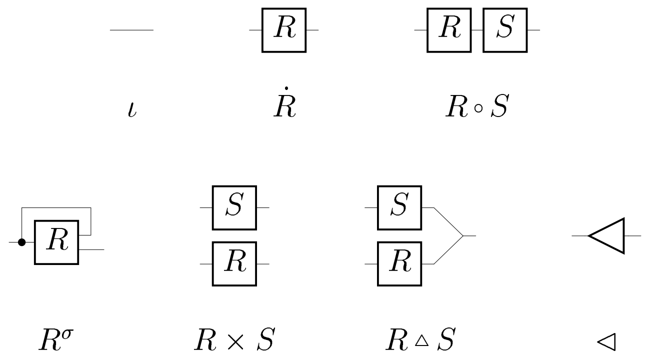

Figure 2: Circuits and their pictures

This definition is from Rietman [9, pages 23–25]; for f a deterministic relation, the above can be written as

$$(8) \quad a \;=\; f^\sigma.b \quad \equiv \quad a \;=\; f.(b,a) \quad .$$

We may now summarize our means of constructing circuits:

1. If R is a relation then $\dot{R}$ is a circuit.
2. If R, S are circuits, then $R \circ S$, $R{\times}S$ and $R \vartriangle S$ are circuits.
3. Delays are circuits.
4. If R is a circuit, then R^σ is a circuit.

A circuit R is said to be *combinational* if it is defined exclusively by means of the first two items in the above list; i.e., if delay and feedback do not appear in its definition.

A circuit term has an interpretation as a picture that is often useful as an aid to understanding how a circuit term is interpreted as a real circuit. Figure 2 shows the correspondence between pictures and circuit terms.

6 Design Steps

We recall from section 2 that our task is to construct a hardware circuit implementing the round robin scheduler RR where

$$a\langle RR\rangle b \quad \equiv \quad \forall(t :: \; a.t \;\;=\;\; \{t \bmod N\} \cap b.t \cap (\lhd^N.b).t) \quad .$$

Because our development of the round robin scheduler is quite long we begin first by giving an overview. Some of the terms used in this overview may not be completely clear at this stage. They will however be explained in full detail later.

The steps are as follows:

1. Low level specification.

   In the first step we reformulate the given specification of RR using tuples of booleans to represent sets. The new specification takes the form

   $$(9) \quad RR \;=\; \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \lhd^N \;\; .$$

   In this specification the definition of RR has been split into three components. Reading from right to left, the first component $\iota \vartriangle \lhd^N$ makes two copies of the input stream, one of which is delayed N time units with respect to the other. The second component *intersect* takes two streams b and c, each of which represents a stream of sets and computes the representation of $b \mathbin{\dot{\cap}} c$ (the stream whose tth element is $b.t \cap c.t$). In combination with the first component, this component maps input stream req to $req \mathbin{\dot{\cap}} \lhd^N.req$. Finally, the third component *filt* implements the function $(\{t \bmod N\}\cap)$. In this way the output value at time t is

   $$ack.t \;=\; \{t \bmod N\} \;\cap\; (req.t \cap (\lhd^N.req).t) \;\; .$$

   where the bracketing shows the order in which the individual terms are computed.

2. Analysis of implementability.

   There are two advantages of a modular specification. One is ease of understanding, which is of considerable help to ensuring that the informal requirements are correctly recorded in the formal specification. The other is that it is much easier to identify potential inefficiencies in the implementation. In the second step we analyse the three components with respect to implementability.

   The *filt* component is, at first sight, a potential bottleneck. However, this turns out not to be the case. It can in fact be implemented using what is called a "cyclic multiplexer".

   The problem with the implementability of RR as specified by (9) is the component $\iota \vartriangle \lhd^N$. The area required for its implementation is $O(N^2)$ since it consists of N delays each with arity N. The conclusion of this phase is thus that it is this component on which we should focus our attention.

3. Goal.

   Having analysed the source of inefficiency in (9) we can proceed to formulating the goal. Specifically, we wish to construct flip-flops $\mathit{ff}_{k,N}$ such that

   $$(10) \quad RR \;=\; \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \mathit{map}.(k:\; 0 \le k < N:\; \mathit{ff}_{k,N})$$

and such that each such component has at most one delay element. In this way the $O(N^2)$ area required by the component

$$\lhd^N$$

is replaced by the $O(N)$ area required by the component

$$map.(k:\ 0 \leq k < N:\ \mathit{ff}_{k,N})\ \ .$$

(The components are called "flip-flops" because this is a name that is commonly given to memory elements.)

4. Simplification of the goal.

   The first step in the achievement of the goal is to simplify it so that it becomes more manageable. The requirement on $\mathit{ff}_{k,N}$ is that

$$\begin{array}{ll} & \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \lhd^N \\ = & \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle map.(k:\ 0 \leq k < N:\ \mathit{ff}_{k,N}) \end{array}$$

   However, we show, in a series of steps, how to reduce it to

$$(11)\quad D_{k,N} \circ \lhd^N \ = \ D_{k,N} \circ \mathit{ff}_{k,N}$$

   (for each k) where

$$a\langle D_{k,N}\rangle b \ \equiv \ \forall(t:\ t \bmod N \ = \ k:\ a.t \ = \ b.t)\ \ .$$

   Note that (11) specifies the behaviour of the flip-flop $\mathit{ff}_{k,N}$ only at times t such that $t \bmod N \ = \ k$. At other times its behaviour is unspecified. This increased latitude (compared to the definition of $\lhd^N$ whose behaviour is specified at all instants) is what is needed to construct an efficient implementation of the circuit.

5. Construction of the flip-flops.

   The final step is the construction of the components $\mathit{ff}_{k,N}$. Again in a series of steps, we calculate that

$$(12)\quad \mathit{ff}_{k,N} \ = \ (\lhd \circ cmx_{k,N})^{\sigma}\ \ ,$$

   where $cmx_{k,N}$ is a cyclic multiplexer. Thus the implementation of the flip-flop does indeed require only one delay element. The combination of (10) and (12) is then the desired implementation of the round robin scheduler.

This then is the overview. Let us now present the full details.

7 Low Level Specification

7.1 Bit Representation

Let us recall the original specification of the round robin scheduler. For input stream b and output stream a,

$$a\langle RR\rangle b \quad\equiv\quad \forall(t :: \; a.t \;=\; \{t \bmod N\} \cap b.t \cap (\lhd^N.b).t) \quad .$$

As is usual we choose to represent a subset of a set of N elements by a sequence of N bits. A stream of subsets of $\{0, \ldots, N-1\}$ is represented by a N-tuple of boolean streams. Let a be such a N-tuple. We denote the k-th stream in a by a_k; the set that a represents at time t is $\{k \;\mid\; 0 \leq k < N \;\wedge\; a_k.t \;=\; \mathit{true}\}$.

The intersection operator on sets is then translated into the conjunction operator mapped over the N input wires. The implementation of the relation R where

$$a\langle R\rangle b \quad\equiv\quad \forall(t :: \; a.t \;=\; b.t \cap (\lhd^N.b).t)$$

is easily derived. Specifically, we have:

$$\begin{array}{cl}
& \forall(t :: \; a.t \;=\; b.t \cap (\lhd^N.b).t) \\
\equiv & \qquad \{ \qquad \text{representation of sets as sequence of } N \text{ bits} \quad \} \\
& \forall(t, k : \; 0 \leq k < N : \; a_k.t \;\equiv\; b_k.t \wedge (\lhd^N.b_k).t) \\
\equiv & \qquad \{ \qquad \text{definition of } \mathit{map} \quad \} \\
& \forall(t :: \; a.t \;=\; (\mathit{map}_N.\wedge).(b.t \;,\; (\lhd^N.b).t)) \\
\equiv & \qquad \{ \qquad \text{definition of } \mathit{zip} \quad \} \\
& \forall(t :: \; a.t \;=\; (\mathit{map}_N.\wedge).((\mathit{zip}_N.(b \;,\; \lhd^N.b)).t)) \\
\equiv & \qquad \{ \qquad \text{lifting, definition of split and composition} \quad \} \\
& a \;=\; (\mathit{map}_N.\dot{\wedge} \;\circ\; \mathit{zip}_N \;\circ\; \iota \vartriangle \lhd^N).b \quad .
\end{array}$$

Whence

$$R \quad=\quad \mathit{intersect} \;\circ\; \iota \vartriangle \lhd^N$$

where

$$\mathit{intersect} \;=\; \mathit{map}_N.\dot{\wedge} \;\circ\; \mathit{zip}_N \quad .$$

It follows that

$$(13) \quad RR \;=\; \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \lhd^{N}$$

where the component *filt* must satisfy the requirement:

$$a\langle \mathit{filt}\rangle b \;\equiv\; \forall(t :: \; a.t \;=\; \{t \bmod N\} \cap b.t) \;\; .$$

7.2 Implementing The Filter Component

With the exception of *filt*, it is clear that all components in (13) can be implemented directly. In this section we consider how *filt* is implemented.

In terms of the bit representation *filt* must satisfy:

$$(14) \quad \begin{array}{lll} a\langle \mathit{filt}\rangle b & \equiv & \forall(t,k : \; t \bmod N \;=\; k : \; a_k.t \;=\; b_k.t) \\ & \wedge & \forall(t,k : \; t \bmod N \;\neq\; k : \; a_k.t \;=\; \mathit{false}) \end{array}$$

In software, *filt* would be implemented with a straightforward if-then-else statement. In Ruby the full generality of an if-then-else statement is typically shunned. In this case, however, the full generality is not needed and the component can be implemented using a so-called cyclic multiplexer [8].

For each k, $0 \leq k < N$, the cyclic multiplexer $cmx_{k,N}$ has two input streams and one output stream. Its function is to copy the input value in the first stream to the output at times t such that $t \bmod N \;=\; k$ and to copy the input values from the second stream to the output stream at all other times. Formally,

$$\begin{array}{lll} a\langle cmx_{k,N}\rangle(b,c) & \equiv & \forall(t : \; t \bmod N \;=\; k : \; a.t \;=\; b.t) \\ & \wedge & \forall(t : \; t \bmod N \;\neq\; k : \; a.t \;=\; c.t) \;\; . \end{array}$$

In practice, a cyclic multiplexer can be implemented by a combination of a counter and a so-called demultiplexer, using techniques that can be found in any circuit design textbook; see for instance Katz [6]. More discussion on the implementation of *filt* is in [11].

Comparing the definition of *filt* with that of $cmx_{k,N}$ it is clear that *filt* can be implemented by pairing each of the N bits of the input stream with a stream of *false* bits and passing each pair of bits to the corresponding cyclic multiplexer. That is,

$$(15) \quad \mathit{filt} \;=\; \mathit{map}.(k : \; 0 \leq k < N : \; cmx_{k,N} \circ \iota \vartriangle K_F) \;\; .$$

where K_F is a circuit that ignores its input and constantly outputs the value F (false).

This concludes this section. The combination of (13) and (15) is a correct implementation of the round-robin scheduler.

8 Efficiency Analysis and the Goal

One advantage of a modular specification like (13) is that it simplifies the task of identifying potential inefficiencies. We need only examine each component in turn.

Assuming that a cyclic multiplexer has an efficient implementation, it is clear that the two components *filt* and *intersect* have efficient implementations. The bottleneck in the implementation is in fact the component $\lhd^N$. Note that the input arity of this component is N since the input stream is in fact a stream of N bits. The total area required for its implementation is thus $O(N^2)$. On the other hand, it seems plausible that an $O(N)$ implementation can be found for RR (although not the component $\lhd^N$) since at any stage only N bits need to be recorded. Specifically, at any time t it suffices to record the value of the kth input bit only at the last time that it was privileged.

We can express our intuition about the memory component that is required as follows. For each input bit k we replace the $\lhd^N$ component by a memory element $\mathit{ff}_{k,N}$, "ff" standing for "flip-flop" (this being the name often given by circuit designers to memory elements). That is, we wish to design $\mathit{ff}_{k,N}$ such that

$$
\begin{array}{ll}
 & \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \lhd^N \\
= & \mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \mathit{map}.(k:\ 0 \le k < N:\ \mathit{ff}_{k,N}) \quad .
\end{array}
$$

Moreover, the implementation of the flip-flops should involve at most one delay element.

9 Simplifying the Goal

In the following discussion it will be useful to introduce the name *specRR* for the term

$$\mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \lhd^N$$

and *impRR* for the term

$$\mathit{filt} \circ \mathit{intersect} \circ \iota \vartriangle \mathit{map}.(k:\ 0 \le k < N:\ \mathit{ff}_{k,N}) \quad .$$

The goal is to derive $\mathit{ff}_{k,N}$ such that $\mathit{specRR} = \mathit{impRR}$. In this section we simplify the goal by splitting it up into separate requirements for the individual flip-flops and by eliminating the "$\mathit{map}.\dot{\wedge}$" term in the definition of *intersect*. We begin by splitting the goal up.

Observe first that $\mathit{zip} \circ \iota \vartriangle \lhd^N$ and $\mathit{zip} \circ \iota \vartriangle \mathit{map}.(k:\ 0 \le k < N:\ \mathit{ff}_{k,N})$ can be written as maps. Specifically,

$$\begin{array}{ll}
& zip_N \circ \iota \vartriangle \lhd^N \\
= & \{ \quad \text{arity of } zip \quad \} \\
& zip_N \circ N{\times}N \circ \iota \vartriangle \lhd^N \\
= & \{ \quad \text{fusion, polymorphism of } \lhd\text{: (7)} \quad \} \\
& zip_N \circ map_N.\iota \vartriangle map_N.\lhd^N \\
= & \{ \quad (6) \quad \} \\
& map_N.(\iota \vartriangle \lhd^N) \quad .
\end{array}$$

Similarly,

$$\begin{array}{ll}
& zip \circ \iota \vartriangle map.(k:\ 0 \leq k < N:\ \mathit{ff}_{k,N}) \\
= & map.(k:\ 0 \leq i < N:\ \iota \vartriangle \mathit{ff}_{k,N}) \quad .
\end{array}$$

Substituting these terms back into the definitions of *specRR* and *impRR*, we conclude that $\mathit{ff}_{k,N}$ must satisfy

$$\begin{array}{ll}
& map.(k ::\ cmx_{k,N} \circ \iota \vartriangle K_F) \circ map.\dot{\wedge} \circ map.(\iota \vartriangle \lhd^N) \\
= & map.(k ::\ cmx_{k,N} \circ \iota \vartriangle K_F) \circ map.\dot{\wedge} \circ map.(k ::\ \iota \vartriangle \mathit{ff}_{k,N}) \quad .
\end{array} \tag{16}$$

Thus, by map fusion and introducing the abbreviation $\varphi_{k,N}$ where, by definition,

$$\varphi_{k,N} \ = \ cmx_{k,N} \circ \iota \vartriangle K_F \quad , \tag{17}$$

we have obtained individual requirements on each flip-flop. Specifically, we require that

$$\forall(k:\ 0 \leq k < N:\ \varphi_{k,N} \circ \dot{\wedge} \circ \iota \vartriangle \lhd^N \ = \ \varphi_{k,N} \circ \dot{\wedge} \circ \iota \vartriangle \mathit{ff}_{k,N}) \quad . \tag{18}$$

In order to remove the spurious conjunction we observe that $\varphi_{k,N}$ satisfies

$$\begin{array}{rl}
a\langle\varphi_{k,N}\rangle b \ \equiv & \forall(t:\ t \bmod N \ = \ k:\ a.t \ = \ b.t) \\
& \wedge\ \forall(t:\ t \bmod N \ \neq \ k:\ a.t \ = \ F) \quad .
\end{array}$$

From this definition, using the fact that $F \wedge F \ = \ F$, it is easy to see that

$$\varphi_{k,N} \circ \dot{\wedge} \ = \ \dot{\wedge} \circ \varphi_{k,N}{\times}\varphi_{k,N} \quad . \tag{19}$$

(We leave the proof to the reader.) Hence we may rewrite (18): for all k, $0 \leq k < N$:

$$
\begin{array}{cl}
& \varphi_{k,N} \circ \dot{\wedge} \circ \iota \vartriangle \lhd^N \;=\; \varphi_{k,N} \circ \dot{\wedge} \circ \iota \vartriangle \mathit{ff}_{k,N} \\
\equiv & \{ \quad \text{above property of } \varphi_{k,N} \quad \} \\
& \dot{\wedge} \circ \varphi_{k,N} \times \varphi_{k,N} \circ \iota \vartriangle \lhd^N \;=\; \dot{\wedge} \circ \varphi_{k,N} \times \varphi_{k,N} \circ \iota \vartriangle \mathit{ff}_{k,N} \\
\equiv & \{ \quad \text{fusion} \quad \} \\
& \dot{\wedge} \circ \varphi_{k,N} \vartriangle (\varphi_{k,N} \circ \lhd^N) \;=\; \dot{\wedge} \circ \varphi_{k,N} \vartriangle (\varphi_{k,N} \circ \mathit{ff}_{k,N}) \\
\Leftarrow & \{ \quad \text{Leibniz} \quad \} \\
& \varphi_{k,N} \circ \lhd^N \;=\; \varphi_{k,N} \circ \mathit{ff}_{k,N} \quad .
\end{array}
$$

So we have reduced the specification of $\mathit{ff}_{k,N}$ from (16) to

(20) $\varphi_{k,N} \circ \lhd^N \;=\; \varphi_{k,N} \circ \mathit{ff}_{k,N} \quad .$

One further simplification is possible. Recalling the definition of $\varphi_{k,N}$ (equation (17)):

$$\varphi_{k,N} \;=\; cmx_{k,N} \circ \iota \vartriangle K_F$$

and the definition of $cmx_{k,N}$

$$
\begin{array}{rcl}
a\langle cmx_{k,N}\rangle(b,c) & \equiv & \forall(t:\ t \bmod N \;=\; k:\ a.t \;=\; b.t) \\
& & \wedge\ \forall(t:\ t \bmod N \;\neq\; k:\ a.t \;=\; c.t)
\end{array}
$$

it is clear that $cmx_{k,N}$ ignores the first component of its input stream whenever $t \bmod N \;\neq\; k$. As a consequence, $\varphi_{k,N}$ ignores its input stream entirely whenever $t \bmod N \;\neq\; k$. We can express this formally by introducing the relation $D_{k,N}$ defined by

$$a\langle D_{k,N}\rangle b \;\equiv\; \forall(t:\ t \bmod N \;=\; k:\ a.t \;=\; b.t) \quad .$$

Two input streams are related by $D_{k,N}$ whenever they are equal for all times t such that $t \bmod N \;=\; k$; at all other times no relation between the two streams is required. Thus the fact that $cmx_{k,N}$ ignores the first input stream whenever $t \bmod N \;\neq\; k$ is expressed by the equation:

$$cmx_{k,N} \circ D_{k,N} \times \iota \;=\; cmx_{k,N} \quad ,$$

and the fact that $\varphi_{k,N}$ ignores its input stream entirely whenever $t \bmod N \;\neq\; k$ is expressed by the equation:

$$\varphi_{k,N} \circ D_{k,N} \;=\; \varphi_{k,N} \quad .$$

The verification of the former equation follows by straightforward pointwise reasoning. The derivation of the latter equation proceeds as follows:

$$
\begin{array}{ll}
& \varphi_{k,N} \\
= & \quad \{ \quad \text{definition of } \varphi_{k,N} \quad \} \\
& cmx_{k,N} \ \circ \ \iota \vartriangle K_F \\
= & \quad \{ \quad \text{above} \quad \} \\
& cmx_{k,N} \ \circ \ D_{k,N} \times \iota \ \circ \ \iota \vartriangle K_F \\
= & \quad \{ \quad \text{fusion} \quad \} \\
& cmx_{k,N} \ \circ \ D_{k,N} \ \vartriangle \ K_F \\
= & \quad \{ \quad K_F \ = \ K_F \ \circ \ D_{k,N} \quad \} \\
& cmx_{k,N} \ \circ \ D_{k,N} \ \vartriangle \ (K_F \ \circ \ D_{k,N}) \\
= & \quad \{ \quad (1),\ K_F \text{ is a left condition} \quad \} \\
& cmx_{k,N} \ \circ \ \iota \vartriangle K_F \ \circ \ D_{k,N} \\
= & \quad \{ \quad \text{definition of } \varphi_{k,N} \quad \} \\
& \varphi_{k,N} \ \circ \ D_{k,N} \quad .
\end{array}
$$

Substituting this equation in (20) we obtain the final simplification to the requirement on the flip-flops.

$$
\begin{array}{ll}
& \varphi_{k,N} \ \circ \ \lhd^N \ = \ \varphi_{k,N} \ \circ \ \mathit{ff}_{k,N} \\
\equiv & \quad \{ \quad \text{above} \quad \} \\
& \varphi_{k,N} \ \circ \ D_{k,N} \ \circ \ \lhd^N \ = \ \varphi_{k,N} \ \circ \ D_{k,N} \ \circ \ \mathit{ff}_{k,N} \\
\Leftarrow & \quad \{ \quad \text{Leibniz} \quad \} \\
& D_{k,N} \ \circ \ \lhd^N \ = \ D_{k,N} \ \circ \ \mathit{ff}_{k,N} \quad .
\end{array}
$$

In summary, the requirement on $\mathit{ff}_{k,N}$ is:

(21) $D_{k,N} \ \circ \ \lhd^N \ = \ D_{k,N} \ \circ \ \mathit{ff}_{k,N} \quad .$

10 Construction of the flip-flops

From the definition of $D_{k,N}$ it is clear that (21) specifies the behaviour of $\mathit{ff}_{k,N}$ only at times t such that $t \bmod N \ = \ k$; at all other times there is complete latitude in its behaviour. It is this latitude that we now exploit.

The component $\lhd^N$ can be seen as a memory element that stores N input values. This is because its implementation demands that the input value at each time t is recorded for use at time $t{+}N$ after which it can be discarded. From (21) it is clear, however, that it suffices to record the input value only at times t such that $t \bmod N = k$, that is once every N clock beats. The crucial step in the calculation of $\mathit{ff}_{k,N}$ below is thus to replace the function mapping t to $t{-}N$ by a function that is constant for N time intervals. A well known example of such a function is the function mapping t to $t \operatorname{div} N$. But this function does not suffice because of the additional requirement that the function's value should equal $t{-}N$ when $t \bmod N = k$. Noting that

$$(t \operatorname{div} N){*}N \quad = \quad t - t \bmod N$$

is the clue to discovering the appropriate function.

Since it occurs twice in the following calculation it is useful to begin by observing that, in general, for arbitrary function f

$$a\langle D_{k,N} \circ f\rangle b \quad \equiv \quad \forall(t\colon\ t \bmod N = k\colon\ a.t = (f.b).t) \ .$$

This is because

$$\begin{array}{ll}
 & a\langle D_{k,N} \circ f\rangle b \\
\equiv & \quad \{ \quad \text{composition and one point rule} \quad \} \\
 & a\ \langle D_{k,N}\rangle\ f.b \\
\equiv & \quad \{ \quad \text{definition of } D_{k,N} \quad \} \\
 & \forall(t\colon\ t \bmod N = k\colon\ a.t = (f.b).t) \ .
\end{array}$$

Now,

$$\begin{array}{ll}
 & a\langle D_{k,N} \circ \lhd^N\rangle b \\
\equiv & \quad \{ \quad \text{above, definition of } \lhd^N \quad \} \\
 & \forall(t\colon\ t \bmod N = k\colon\ a.t = b.(t{-}N)) \\
\equiv & \quad \{ \quad \text{This is the crucial step discussed above.} \\
 & \qquad\ \ \text{We replace ``}t{-}N\text{'' using the property of} \\
 & \qquad\ \ \text{modular arithmetic:} \\
 & \qquad\ \ t \bmod N = k \quad \Rightarrow \quad N{-}1 = (t{-}k{-}1) \bmod N \quad \} \\
 & \forall(t\colon\ t \bmod N = k\colon\ a.t = b.(t - 1 - (t{-}k{-}1) \bmod N)) \\
\equiv & \quad \{ \quad \text{Define } \mathit{ff}_{k,N} \text{ by} \\
 & \qquad\ \ (\mathit{ff}_{k,N}.b).t \quad = \quad b.(t - 1 - (t{-}k{-}1) \bmod N) \quad \} \\
 & \forall(t\colon\ t \bmod N = k\colon\ a.t = (\mathit{ff}_{k,N}.b).t) \\
\equiv & \quad \{ \quad \text{above} \quad \} \\
 & a\langle D_{k,N} \circ \mathit{ff}_{k,N}\rangle b \ .
\end{array}$$

We have thus calculated a functional specification of $\mathit{ff}_{k,N}$:

$$(\mathit{ff}_{k,N}.b).t \quad = \quad b.(t - 1 - (t{-}k{-}1) \bmod N) \quad .$$

The construction of an implementation for $\mathit{ff}_{k,N}$ amounts to verifying that the function mapping t to $t - 1 - (t{-}1{-}k) \bmod N$ is indeed constant over N time intervals. To be precise, we explore when $(\mathit{ff}_{k,N}.b).(t+1)$ equals $(\mathit{ff}_{k,N}.b).t$:

$$
\begin{array}{ll}
 & (\mathit{ff}_{k,N}.b).(t+1) \\
= & \qquad \{ \qquad \text{definition} \qquad \} \\
 & b.(t \ - \ (t-k) \bmod N) \\
= & \qquad \{ \qquad \bullet \quad \text{Suppose } (t-k) \bmod N \ \neq \ 0 \text{ . Then} \\
 & \qquad \qquad (t-k) \bmod N \quad = \quad (t-k-1) \bmod N + 1 \qquad \} \\
 & b.(t \ - \ ((t-k-1) \bmod N + 1)) \\
= & \qquad \{ \qquad \text{arithmetic} \qquad \} \\
 & b.(t - 1 - (t{-}k{-}1) \bmod N) \\
= & \qquad \{ \qquad \text{definition} \qquad \} \\
 & (\mathit{ff}_{k,N}.b).t \quad .
\end{array}
$$

Thus if $(t-k) \bmod N \ \neq \ 0$, $(\mathit{ff}_{k,N}.b).(t+1) \ = \ (\mathit{ff}_{k,N}.b).t$. Also, if $(t-k) \bmod N \ = \ 0$, it is obvious that $(\mathit{ff}_{k,N}.b).(t+1) \ = \ b.t$. So $\mathit{ff}_{k,N}.b$ is defined by the following equations:

$$
\begin{array}{lcll}
(\mathit{ff}_{k,N}.b).(t+1) & = & (\mathit{ff}_{k,N}.b).t & \text{if } (t-k) \bmod N \ \neq \ 0 \\
(\mathit{ff}_{k,N}.b).(t+1) & = & b.t & \text{if } (t-k) \bmod N \ = \ 0 \quad .
\end{array}
$$

We recognize in these equations a combination of the cyclic multiplexer and a feedback. Indeed,

$$\mathit{ff}_{k,N}.b \quad \equiv \quad m.(b \, , \mathit{ff}_{k,N}.b)$$

where

$$
\begin{array}{lcll}
(m.(b,c)).(t+1) & = & c.t & \text{if } (t-k) \bmod N \ \neq \ 0 \\
(m.(b,c)).(t+1) & = & b.t & \text{if } (t-k) \bmod N \ = \ 0 \quad .
\end{array}
$$

By (8) this equivales $\mathit{ff}_{k,N} = m^{\sigma}$ where, as is obvious from the definitions of the circuit multiplexer and delay, $m = \lhd \ \circ \ cmx_{k,N}$. Thus,

(22) $\mathit{ff}_{k,N} \quad = \quad (\lhd \ \circ \ cmx_{k,N})^{\sigma} \quad .$

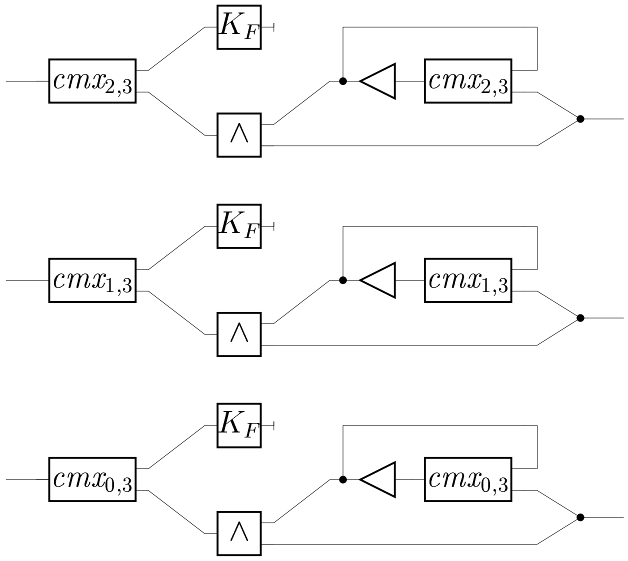

Figure 3: An instance of (23)

In summary, the implementation of RR we have come to is

$$RR \quad = \quad \mathit{filt} \circ \mathit{map}.\dot{\wedge} \circ \mathit{zip} \circ \iota \vartriangle \mathit{map}.(k:\ 0 \leq k < N:\ \mathit{ff}_{k,N}) \quad ,$$

or equivalently, exploiting (6)

$$(23) \quad RR \quad = \quad \mathit{filt} \circ \mathit{map}.\dot{\wedge} \circ \mathit{map}.(k:\ 0 \leq k < N:\ \iota \vartriangle \mathit{ff}_{k,N}) \quad ,$$

where $\mathit{ff}_{k,N}$ is defined by (22) and filt is defined by (15). A picture of (23) is in figure 3 above.

11 Discussion

Programming has been variously described as an "art", a "craft", a "discipline", a "logic" and a "science". In this paper we have tried to illustrate the calculational style of programming at work. The "art" of calculation lies in the way the original problem is dissected into manageable chunks, the "craft" is formulating the heuristics and goals that lead to a successful resolution of the individual calculations. The "discipline" is to maintain a uniform granularity in executing calculational steps, thus avoiding errors of oversight and omission. The "logic" lies in the way the individual pieces are combined and, the "science" is making the right choice of formalism, at the right level of detail, at the right time. In all these aspects of the programming task, the guarantee of quality is the combination of concision with precision.

In this paper we have done our utmost to add yet another clear illustration of the calculational method to the extensive literature on the topic, this time in an application area that is relatively unusual. We have shown how the natural decomposition in the original problem statement is preserved and indeed exploited in shaping the overall calculation. We have discussed at some length the heuristics governing the individual calculations so that the immediate and long-term goals are always clear. We have applied both pointwise and point-free relation algebra, the choice of which to use being governed by the concern for concision without loss of precision. And we have made all our calculational steps explicit and straightforward so that the reader is able to make an easy local check of their accuracy. In this way we hope to have made a modest but lasting contribution to the mathematics of program construction.

Acknowledgement

Thanks go to Tom Verhoeff and Peter Veltkamp for their comments on a draft of this paper.

References

[1] Chritiene Aarts, Roland Backhouse, Paul Hoogendijk, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes. Available at `ftp://ftp.win.tue.nl/pub/math.prog.construction/book.dvi`, December 1992.

[2] Richard S. Bird and Oege de Moor. *Algebra of Programming.* Prentice-Hall International, 1996.

[3] Avra Cohn and Mike Gordon. A mechanized proof of correctness of a simple counter. In K. McEvoy and J.V. Tucker, editors, *Theoretical Foundations of VLSI Design*, Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[4] Geraint Jones. Designing circuits by calculation. Technical Report PRG-TR-10-90, Programming Research Group, Oxford University Computing Laboratory, April 1990. Available at `http://www.comlab.ox.ac.uk/oucl/users/geraint.jones/publications`.

[5] Geraint Jones and Mary Sheeran. Circuit design in Ruby. In Jørgen Staunstrup, editor, *Formal Methods for VLSI Design. IFIP WG 10.5 Lecture Notes.* North-Holland, 1990. A revised version is available at `http://www.comlab.ox.ac.uk/oucl/users/geraint.jones/publications`.

[6] Randy H. Katz. *Contemporary Logic Design*. Addison-Wesley, 1994.

[7] Thomas Kropf. IFIP WG10.5 benchmark circuits for hardware verification. Available at `http://goethe.ira.uka.de/hvg`, 1996.

[8] Wayne Luk. Systematic serialisation of array-based architectures. *Integration*, 14(3):333–360, 1993. Available at `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Wayne.Luk`.

[9] Frans Rietman. *A Relational Calculus for the Design of Distributed Algorithms*. PhD thesis, University of Utrecht, 1995.

[10] Robin Sharp and Ole Rasmussen. An introduction to Ruby. 2nd edition. Technical report, Dept. of Computer Science, Technical University of Denmark, 1995. Available at `ftp://ftp.it.dtu.dk/pub/Ruby/intro.ps.Z`.

[11] Matteo Vaccari. *Calculational Derivation of Circuits*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Milano, forthcoming in 1998.