Published in *Science of Computer Programming*, Vol. 15, No. 23, December 1990, pp. 217–233.

A systolic LRU processor and its top-down development

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

Geoffrey Brown
School of Electrical Engineering, Cornell University, Ithaca, New York 14853 USA

March 1990

Abstract. We present a novel systolic processor that implements the least-recently-used (LRU) policy for multi-level storage systems. The design is developed by successively refining a high-level description of the algorithm. The effect of varying the degree of pipelining on performance is discussed. We also show how the design methodology used for the LRU processor can be applied to the development of other systolic systems.

1 Introduction

In multi-level storage systems, data are partitioned into pages with frequently used pages kept in a small, fast primary storage and with less frequently used pages kept in a large, slower secondary storage. It may become necessary for performance reasons to move a page from secondary to primary storage, because the frequency with which pages are accessed varies with time. The replacement policy determines the best candidate for replacement among the pages in primary storage; LRU dictates that this candidate is the page that has been accessed least recently.

An LRU implementation needs to perform two tasks: first, to maintain a sequence of the pages, ordered by most recent time of access, that currently reside in the primary storage; and second, to provide a mechanism for detecting the least recently used page in this sequence. Given that the next page to be accessed is p, updating the sequence of pages consists of deleting p from the sequence and prepending p to the result. If p is not an element of the current sequence, then the prepending of p is accompanied by the removal of the last element of the sequence.

Our proposed LRU processor is based upon a non-systolic algorithm originally described, without proof, by Dijkstra [1]. It consists of a chain of identical components similar in both size and speed to the cells of a shift register. The novel aspects of our development of this design include:

- the architecture of the processor is obtained by successively refining a high-level description of the LRU algorithm;
- the design is captured as a succinct expression with parameters that can be varied to give implementations with different performance trade-offs; and
- the development method is quite general and has been used in deriving a number of word-level and bit-level systolic designs [4], [7].

We shall first introduce the notation for describing hardware and the associated algebraic theorems for transforming designs. We then indicate how the LRU algorithm, expressed as a set of recursion equations, can be recast in this notation. The resulting representation is further refined by algebraic transformations to produce a parametrised description from which a range of designs with different performance trade-offs can be generated. The application of this development method in deriving other systolic processors is also briefly discussed. Finally we summarise our work and provide the proofs of relevant theorems in the Appendix.

2 Notation

A simple notation for describing recursive algorithms and for expressing such algorithms using combinators will be presented. To deal with sequential systems some additional notions, such as relations and streams, will be considered.

2.1 Recursion equations and combinators

Objects in our notation are either atoms (such as numbers) or sequences of objects: for instance the object $\langle 0, \langle 1,2\rangle\rangle$ is a 2-sequence containing the number 0 and the sequence $\langle 1,2\rangle$. A sequence is an ordered collection of elements with the empty sequence denoted by $\langle\,\rangle$. Sequences are appended using the operator '$\hat{}$', so $\langle 1,2,3,4\rangle = \langle 1\rangle\hat{}\langle 2,3\rangle\hat{}\langle 4\rangle\hat{}\langle\,\rangle$. $\#x$ denotes the number of elements in sequence x. The function *last* is used to extract the last element of a sequence; so *last* $\langle x_0, x_1, x_2, x_3\rangle = x_3$. The k-th component of an N-element sequence can be extracted by the projection function π_k $(1 \leq k \leq N)$; for example $\pi_2\ \langle x, \langle y, z\rangle\rangle = \langle y, z\rangle$.

Notice that function application is denoted by juxtaposition, and this can be extended to two or more arguments. For instance, the value of a function f with arguments x and y is written as $f\ x\ y$ and means $(f\ x)\ y$. To illustrate this style of description, an algorithm for summing a given number and the elements of a sequence of numbers is as follows:

$$\begin{aligned} \mathit{sum}\ s\ \langle\,\rangle &\stackrel{\text{def}}{=} s, \\ \mathit{sum}\ s\ (\langle x\rangle\hat{}xs) &\stackrel{\text{def}}{=} \mathit{sum}\ (s+x)\ xs. \end{aligned}$$

While recursion equations like these are adequate for describing algorithms, a proliferation of such equations tends to produce unstructured descriptions. It is often useful to recast a recursive algorithm in combinators which are higher-order functions encapsulating common patterns of computation. For instance given the combinator **reduce** where

$$\begin{aligned} \mathsf{reduce}\, f\ a\ \langle\,\rangle &\overset{\text{def}}{=} a, \\ \mathsf{reduce}\, f\ a\ (\langle x\rangle \hat{}\, xs) &\overset{\text{def}}{=} \mathsf{reduce}\, f\ (f\ a\ x)\ xs, \end{aligned}$$

by matching these definitions with those for *sum* it is clear that

$$sum \quad = \quad \mathsf{reduce}\, add \quad \text{where} \;\; add\ x\ y \overset{\text{def}}{=} x + y.$$

There are two obvious benefits of expressing algorithms in combinators. The first benefit is that the absence of bound variables in combinatory expressions results in useful algebraic properties. These properties enable designs to be optimised by equational reasoning, and we shall illustrate that in the next section. The second benefit arises from the structure associated with a combinator which indicates how components can be connected together – for instance

$$\mathsf{reduce}\ add\ s\ \langle x_0, x_1, x_2, x_3\rangle$$

corresponds to the connection structure in Figure 1.

Two combinators will be needed in developing the LRU processor. The first combinator is (reverse) functional composition, $(f\ ;\ g)\ x \overset{\text{def}}{=} g\ (f\ x)$. The second combinator is **row**, a slight generalisation of **reduce**,

$$\begin{aligned} \mathsf{row}\, f\ \langle a, \langle\,\rangle\rangle &\overset{\text{def}}{=} \langle\langle\,\rangle, a\rangle && (1) \\ \mathsf{row}\, f\ \langle a, \langle x\rangle \hat{}\, xs\rangle &\overset{\text{def}}{=} \langle\langle y\rangle \hat{}\, ys, b\rangle && (2) \\ & \quad \text{where} \left\{ \begin{array}{lcl} \langle y, z\rangle & \overset{\text{def}}{=} & f\ \langle a, x\rangle, \\ \langle ys, b\rangle & \overset{\text{def}}{=} & \mathsf{row}\, f\ \langle z, xs\rangle, \end{array} \right. \end{aligned}$$

which corresponds to a linear array of components with connections on every side (Figure 2).

Combinators such as **reduce** and **row** provide a target template for recasting algorithms in the first phase of our development process.

2.2 Relations and streams

To deal with sequential circuits, a combinatory expression is promoted to a binary relation that relates a stream (an infinite sequence of data) in its domain to a stream in its range – an approach first suggested by Sheeran [7]. Our main motivation for using relations is that it allows a non-constructive description of circuits with feedback loops. We shall illustrate later how this description facilitates the statement and proof of useful transformation rules.

Different brackets will be used to indicate that the operations and data are 'lifted' to the corresponding stream versions; for instance $\prec a, b, c \succ$ denotes a stream of sequences formed by interleaving the sequence of streams $\langle a, b, c \rangle$. Hence if '$\frown$' denotes the stream version of '^' and y_t represents the value of stream y at time t, then for all t

$$\begin{aligned} \prec\,\succ_t &\overset{\text{def}}{=} \langle\,\rangle, \\ (\prec x \succ \frown xs)_t &\overset{\text{def}}{=} \langle x_t \rangle \hat{}\, xs_t . \end{aligned}$$

For instance, given two streams x and y, $\prec x, y \succ$ represents a stream of pairs such that for all t, $\prec x, y \succ_t = \langle x_t, y_t \rangle$.

We shall write binary relations in infix form, so that an adder can be defined by

$$\begin{aligned} \prec x, y \succ \; Add \; z &\overset{\text{def}}{=} \forall t \,.\; z_t \;=\; add\; x_t\; y_t \\ &\equiv \forall t \,.\; z_t = x_t + y_t . \end{aligned}$$

We shall follow the convention to denote a stream representation of a combinational circuit (such as *Add*) by capitalising the first letter of the corresponding 'static' expression (such as *add*).

We can also define combinators for relations on streams. Two components with a common interconnection can be described by the combinator *relational composition*, which is similar to the functional composition combinator defined earlier:

$$x\;(Q\;;R)\;z \;\overset{\text{def}}{=}\; \exists y \,.\; (x\;Q\;y) \wedge (y\;R\;z).$$

For combinational circuits Q and R, it is the case that $\forall t\,.\; z_t = (q;r)\; x_t$ implies $x\;(Q;R)\;z$.

A homogeneous *pipe* is obtained by repeatedly composing the same component using relational composition. Given that Id represents the identity relation such that $x\; Id\; y \overset{\text{def}}{=} x = y$, we have

$$R^0 \overset{\text{def}}{=} Id, \tag{3}$$

$$R^{n+1} \overset{\text{def}}{=} R\,;\,R^n. \tag{4}$$

Another common combinator is *parallel composition*, which describes two devices operating independently on the components of a stream of pairs,

$$\prec x, y \succ \;(Q \parallel R)\; \prec u, v \succ \;\overset{\text{def}}{=}\; (x\;Q\;u) \wedge (y\;R\;v).$$

We shall adopt the abbreviation $\mathsf{fst}\;Q \overset{\text{def}}{=} Q \parallel Id$ and $\mathsf{snd}\;Q \overset{\text{def}}{=} Id \parallel Q$.

The stream version of the row combinator will be needed, which can be defined as follows:

$$\begin{aligned}
&\prec a, x \succ \;(\mathsf{row}_0\, R)\; \prec y, b \succ \\
&\quad\overset{\text{def}}{=}\; (a = b) \wedge (x = y = \prec\,\succ),
\end{aligned} \tag{5}$$

$$\begin{aligned}
&\prec a,\, \prec x \succ ^\frown xs \succ \;(\mathsf{row}_{n+1}\, R)\; \prec\prec y \succ ^\frown ys,\; b \succ \\
&\quad\overset{\text{def}}{=}\; \exists z\,.\; (\prec a, x \succ \;R\; \prec y, z \succ) \wedge (\prec z, xs \succ \;(\mathsf{row}_n\, R)\; \prec ys, b \succ).
\end{aligned} \tag{6}$$

The subscript denotes the number of components in the row, and is omitted when the meaning is clear.

This stream version of row can be obtained from the non-lifted version of row (equation 1 and equation 2) with the appropriate decomposition of streams of sequences into sequences of streams, and vice versa. Some discussions of this method can be found in [3].

A *delay* is given by

$$x\;\mathcal{D}\;y \;\overset{\text{def}}{=}\; \forall t\,.\; x_{t-1} = y_t.$$

An *anti-delay*, $\mathcal{D}^{-1}$, is such that $\mathcal{D};\,\mathcal{D}^{-1} = \mathcal{D}^{-1};\,\mathcal{D} = Id$. This identity holds if t belongs to the set of positive and negative integers. A latch can be modelled by a delay with forward dataflow (such that x is the input and y is the output in the above definition for $\mathcal{D}$) and by an anti-delay with backward dataflow.

Note that delays and anti-delays can be used on signals of any type, for instance

$$\begin{aligned} \prec x, y \succ \ \mathcal{D} \ \prec u, v \succ \ &\equiv \ \forall t\,.\ \ \prec x, y \succ_{t-1} = \prec u, v \succ_t \\ &\equiv \ \forall t\,.\ \langle x_{t-1}, y_{t-1} \rangle \ = \prec u_t, v_t \succ \\ &\equiv \ (x \ \mathcal{D} \ u) \wedge (y \ \mathcal{D} \ v). \end{aligned}$$

Given that a component is delay-commutative, that is $R\,;\mathcal{D} \ = \ \mathcal{D}\,;R$, a theorem concerning pipes and delays is

$$R^{km} \ = \ (R^k; \mathcal{D})^m \ ; \ \mathcal{D}^{-m} \qquad (7)$$

which can be verified by induction on m. This theorem corresponds to pipelining clusters of k components by inserting latches between them. The expression $\mathcal{D}^{-m}$ corresponds to the latency (the number of clock cycles needed to produce the result) incurred as a consequence of pipelining. We shall use this theorem to pipeline our LRU design.

Another combinator that we need is a looping construct, defined by

$$x \ (\mathsf{loop}\, R) \ y \ \stackrel{\text{def}}{=} \ \exists z\,.\ \ \prec x, z \succ \ R \ \prec z, y \succ \qquad (8)$$

(Figure 3). $\mathsf{loop}\,(R;\ \mathsf{fst}\mathcal{D})$ corresponds to a circuit with a delay on the feedback path, a standard state machine configuration.

A useful result concerning rows, pipes and loops is

$$\mathsf{loop}\,(\mathsf{row}_n\ R) \ = \ (\mathsf{loop}\, R)^n, \qquad (9)$$

which can be verified by induction on n (see Appendix). An instance of this theorem is shown in Figure 4.

This theorem is important because it allows the designer to concentrate on developing the state-transition logic of a single state machine and subsequently decomposing it into a cascade of state machines. The alternative –

designing and synchronising individual state machines from the outset – is usually more complex.

Note. The loop combinator is defined such that equation 9 is expressed in its simplest form. In Ruby [7] the loop combinator is defined by

$$x\ (\mathit{loop}\, R)\ y \quad \stackrel{\text{def}}{=} \quad \exists z\,.\ \ \prec x, z \succ\ R\ \prec y, z \succ$$

so that, given that $u\ R^{-1}\ v \stackrel{\text{def}}{=} v\ R\ u$, the theorem $(\mathit{loop}\, R)^{-1} \ = \ \mathit{loop}\,(R^{-1})$ holds. The relationship between the two looping constructs is given by

$$\mathsf{loop}\, R \quad = \quad \mathit{loop}\,(R\,;\ \mathit{Swap})$$

where $\mathit{swap}\ \langle x, y\rangle \stackrel{\text{def}}{=} \langle y, x\rangle$.
(End of note.)

3 Developing the LRU processor

We are now ready to develop the LRU processor. There will be two phases in this development: in the first phase we specify the LRU processor and transform the specification into a combinatory expression to obtain a preliminary design; in the second phase we optimise the preliminary design by algebraic theorems to obtain a range of designs with different performance trade-offs.

3.1 Specifying the LRU processor

Our goal is to develop $LRU0$, a sequential implementation of the LRU algorithm. $LRU0$ should have the following characteristics: its state is the sequence being maintained, its output is the last element in this sequence, and its input is the page, if any, to insert on the next clock cycle. This circuit will be formed by adding latches and feedback paths to a purely combinational circuit, *InsImp*, which implements the state-transition logic. Hence we define $LRU0$ as follows:

$$LRU0 \quad \stackrel{\text{def}}{=} \quad \mathsf{loop}\,(\mathit{InsImp}\,;\ \mathsf{fst}\mathcal{D}). \qquad (10)$$

We adopt a specification which requires *insImp*, the static version of *InsImp*, to satisfy

$$\langle\langle p, b\rangle, xs\rangle \; insImp \; \langle xs', y\rangle \quad = \quad (xs' = ins\ p\ b\ xs) \wedge (y = last\ xs). \tag{11}$$

In this equation xs and xs' are respectively the current state and the next state and are sequences of pages. The output y is the last element – the least recently used page – of the current state xs. The boolean input b issues a request to insert page p in the current state xs,

$$\begin{aligned} ins\ p\ true\ xs &\overset{\text{def}}{=} insert\ p\ xs, \\ ins\ p\ false\ xs &\overset{\text{def}}{=} xs. \end{aligned}$$

The functions *insert* and *delete* capture the LRU algorithm: *insert p xs* prepends p to the sequence xs and calls *delete* to modify xs,

$$insert\ p\ \langle\,\rangle \overset{\text{def}}{=} \langle\,\rangle, \tag{12}$$

$$insert\ p\ (\langle x\rangle\hat{}\,xs) \overset{\text{def}}{=} \langle p\rangle\,\hat{}\; delete\ p\ (\langle x\rangle\hat{}\,xs) \tag{13}$$

and *delete p xs* removes the first instance of p from xs or the last element of xs if there is no such instance:

$$delete\ p\ \langle\,\rangle \overset{\text{def}}{=} \langle\,\rangle, \tag{14}$$

$$\begin{aligned} delete\ p\ (\langle x\rangle\hat{}\,xs) \overset{\text{def}}{=}\; & \textsf{if}\ (x = p) \vee (xs = \langle\,\rangle)\ \textsf{then}\ xs \\ & \square\ (x \neq p) \wedge (xs \neq \langle\,\rangle)\ \textsf{then}\ \langle x\rangle\,\hat{}\; delete\ p\ xs \\ & \textsf{fi}. \end{aligned} \tag{15}$$

Notice that *insert* preserves the size of the sequence of pages, since $\#(insert\ p\ xs) = \#xs$. This ensures that the system state is maintained at a constant size: that is $\#xs' = \#xs$ in equation 11.

Expanding the definition of *insert* using equation 12 and equation 13, we obtain

$$\begin{aligned} ins\ p\ true\ \langle\,\rangle &= \langle\,\rangle, \\ ins\ p\ true\ (\langle x\rangle\hat{}\,xs) &= \langle p\rangle\,\hat{}\; delete\ p\ (\langle x\rangle\hat{}\,xs), \\ ins\ p\ false\ xs &= xs. \end{aligned} \tag{16}$$

This completes the specification of the LRU processor.

3.2 Obtaining a preliminary design

There are many ways to implement the LRU algorithm. Since a systolic implementation is desired, we shall implement the state-transition logic specified by the function *ins* by a linear array of N identical cells, where N is the size of the system state. The state machine $LRU0$ (equation 10) can then be constructed by adding latches and feedback paths to the array of cells; developing a systolic version of this machine will consist of distributing latches between the cells.

In order to make use of the algebraic theorems for the combinators described in Section 2.2, we need to transform *ins* into a form compatible with the row combinator which describes a linear array structure. This is a crucial step that demands insight: like conducting other inductive proofs, the difficulty is to find an appropriate generalisation of the induction hypothesis. In this case we generalise *ins* to a function *update* by introducing a new argument q which replaces the second instance of p on the right-hand side of equation 16,

$$\begin{aligned} \mathit{update}\ p\ q\ \mathit{true}\ \langle\,\rangle &\stackrel{\text{def}}{=} \langle\,\rangle, & (17)\\ \mathit{update}\ p\ q\ \mathit{true}\ (\langle x\rangle\hat{}\,xs) &\stackrel{\text{def}}{=} \langle p\rangle\,\hat{}\ \mathit{delete}\ q\ (\langle x\rangle\hat{}\,xs), & (18)\\ \mathit{update}\ p\ q\ \mathit{false}\ xs &\stackrel{\text{def}}{=} xs, & (19) \end{aligned}$$

so that

$$\mathit{ins}\ p\ b\ xs \quad = \quad \mathit{update}\ p\ p\ b\ xs. \qquad (20)$$

We now show by induction that *update* can be implemented by a linear array of cells which will be called *insCell*. The definition of *insCell* will be chosen so that

$$\mathit{update}\ p\ q\ b\ xs \quad = \quad \pi_1\,(\mathsf{row}\ \mathit{insCell}\ \langle\langle p, q, b\rangle, xs\rangle). \qquad (21)$$

A schematic of the structure on the right-hand side of equation 21 is shown in Figure 5.

Consider first the base case of equation 21.

$$\begin{array}{ll}
& \mathit{update}\ p\ q\ b\ \langle\,\rangle \\
= & \{\text{equation 17 and equation 19: definition of } \mathit{update}\} \\
& \langle\,\rangle \\
= & \{\pi_1\ \langle x, y\rangle \overset{\text{def}}{=} x\} \\
& \pi_1\ \langle\langle\,\rangle, \langle p, q, b\rangle\rangle \\
= & \{\text{equation 1: definition of } \mathsf{row}\} \\
& \pi_1\ (\mathsf{row}\ \mathit{insCell}\ \langle\langle p, q, b\rangle, \langle\,\rangle\rangle).
\end{array}$$

Consider now the induction cases of *update*.

$$\begin{array}{ll}
& \mathit{update}\ p\ q\ \mathit{false}\ (\langle x\rangle \hat{}\, \mathit{xs}) \\
= & \{\text{equation 19: definition of } \mathit{update}\} \\
& \langle x\rangle \hat{}\, \mathit{xs} \\
= & \{\text{equation 19: definition of } \mathit{update}\} \\
& \langle x\rangle\ \hat{}\ \mathit{update}\ x\ q\ \mathit{false}\ \mathit{xs}.
\end{array}$$

In the Appendix we show that

$$\mathit{update}\ p\ q\ \mathit{true}\ (\langle x\rangle \hat{}\, \mathit{xs}) \quad = \quad \langle p\rangle\ \hat{}\ \mathit{update}\ x\ q\ (q \neq x)\ \mathit{xs},$$

hence for the induction case of equation 21,

$$\begin{array}{ll}
& \mathit{update}\ p\ q\ b\ (\langle x\rangle \hat{}\, \mathit{xs}) \\
= & \{\text{combining the two induction cases for } \mathit{update}\} \\
& \langle u\rangle \hat{}\, \mathit{us} \\
& \text{where} \left\{ \begin{array}{lcl} u & = & \mathsf{if}\ b\ \mathsf{then}\ p\ \ \mathsf{else}\ x\ \mathsf{fi}, \\ \mathit{us} & = & \mathit{update}\ x\ q\ ((q \neq x) \wedge b)\ \mathit{xs} \end{array} \right. \\
= & \{\text{induction hypothesis}\} \\
& \pi_1\ \langle\langle u\rangle \hat{}\, \mathit{us},\ v\rangle \\
& \text{where} \left\{ \begin{array}{rcl} u & = & \mathsf{if}\ b\ \mathsf{then}\ p\ \ \mathsf{else}\ x\ \mathsf{fi}, \\ \langle \mathit{us}, v\rangle & = & \mathsf{row}\ \mathit{insCell}\ \langle y, \mathit{xs}\rangle, \\ y & \overset{\text{def}}{=} & \langle x, q, (q \neq x) \wedge b\rangle \end{array} \right.
\end{array}$$

$$
\begin{aligned}
= \;& \{\text{define } y \text{ to be second output of } \mathit{insCell}\} \\
& \pi_1\ \langle \langle u\rangle\hat{}\,us,\ v\rangle \\
& \text{where } \left\{ \begin{array}{ccl} \langle u, y\rangle & = & \langle \textsf{if } b \textsf{ then } p \textsf{ else } x \textsf{ fi}, \langle x, q, (q \neq x) \wedge b\rangle\rangle \\ & \overset{\text{def}}{=} & \mathit{insCell}\ \langle\langle p, q, b\rangle, x\rangle, \\ \langle us, v\rangle & = & \textsf{row}\ \mathit{insCell}\ \langle y, xs\rangle \end{array}\right. \\
= \;& \{\text{equation 2: definition of } \textsf{row}\} \\
& \pi_1\ (\textsf{row}\ \mathit{insCell}\ \langle\langle p, q, b\rangle, \langle x\rangle\hat{}\,xs\rangle).
\end{aligned}
$$

From these calculations we have proved that

$$
\begin{aligned}
\mathit{update}\ p\ q\ b\ xs &= \pi_1\ (\textsf{row}\ \mathit{insCell}\ \langle\langle p, q, b\rangle, xs\rangle) \\
&= (\textsf{row}\ \mathit{insCell};\ \pi_1)\ \langle\langle p, q, b\rangle, xs\rangle \qquad (22)
\end{aligned}
$$

where

$$
\mathit{insCell}\ \langle\langle p, q, b\rangle, x\rangle \overset{\text{def}}{=} \langle \textsf{if } b \textsf{ then } p \textsf{ else } x \textsf{ fi}, \langle x, q, (q \neq x) \wedge b\rangle\rangle. (23)
$$

It remains for us to show that *insImp* is realised by appropriately combining instances of *insCell* so that *InsImp* is realised by combining instances of *InsCell* (the stream version of *insCell*).

$$
\begin{aligned}
& \mathit{insImp}\ \langle\langle p, b\rangle, xs\rangle \\
= \;& \{\text{equation 11: requirement for } \mathit{insImp}\} \\
& \langle \mathit{ins}\ p\ b\ xs, \mathit{last}\ xs\rangle \\
= \;& \{\text{equation 20: } \mathit{ins} \text{ generalised to } \mathit{update}\} \\
& \langle \mathit{update}\ p\ p\ b\ xs, \mathit{last}\ xs\rangle \\
= \;& \{\text{equation 22: } \mathit{update} \text{ expressed in } \textsf{row}\} \\
& \langle (\textsf{row}\ \mathit{insCell};\ \pi_1)\ \langle\langle p, p, b\rangle, xs\rangle,\ \mathit{last}\ xs\rangle \\
= \;& \{\text{given } \mathit{dupfst}\ \langle p, b\rangle \overset{\text{def}}{=} \langle p, p, b\rangle \text{ and } \textsf{fst}\, f\ \langle x, y\rangle \overset{\text{def}}{=} \langle f\ x, y\rangle\} \\
& \langle (\textsf{fst}\, \mathit{dupfst};\ \textsf{row}\ \mathit{insCell};\ \pi_1)\ \langle\langle p, b\rangle, xs\rangle,\ \mathit{last}\ xs\rangle.
\end{aligned}
$$

From equation 23 we obtain $\mathit{insCell};\pi_2;\pi_1 = \pi_2$, thus

$$
\mathit{last}\ xs \;=\; \pi_2\ \langle\langle p, p, b\rangle, \mathit{last}\ xs\rangle
$$

$$\begin{aligned} &= (\mathit{insCell};\pi_2;\pi_1)\ \langle\langle p,p,b\rangle, \mathit{last}\ xs\rangle \\ &= (\mathsf{fst}\,\mathit{dupfst};\ \mathsf{row}\ \mathit{insCell};\pi_2;\pi_1)\ \langle\langle p,b\rangle, xs\rangle. \end{aligned}$$

Given that $\mathsf{snd}\,f\ \langle x,y\rangle \overset{\text{def}}{=} \langle x, f\ y\rangle$ and since

$$\begin{aligned} &\langle(\mathsf{fst}f;\mathsf{row}g;\pi_1)\ x,\ (\mathsf{fst}f;\mathsf{row}g;\pi_2;\pi_1)\ x\rangle \\ &\quad= (\mathsf{fst}f;\mathsf{row}g;\mathsf{snd}\pi_1)\ x, \end{aligned}$$

we get

$$\begin{aligned} \mathit{insImp}\ \langle\langle p,b\rangle, xs\rangle &= \langle(\mathsf{fst}\,\mathit{dupfst};\ \mathsf{row}\ \mathit{insCell};\ \pi_1)\ \langle\langle p,b\rangle, xs\rangle,\ \mathit{last}\ xs\rangle \\ &= (\mathsf{fst}\,\mathit{dupfst};\ \mathsf{row}\ \mathit{insCell};\ \mathsf{snd}\,\pi_1)\ \langle\langle p,b\rangle, xs\rangle. \end{aligned}$$

Hence

$$\mathit{insImp} \overset{\text{def}}{=} \mathsf{fst}\,\mathit{dupfst};\ \mathsf{row}\ \mathit{insCell};\ \mathsf{snd}\,\pi_1 \qquad (24)$$

will satisfy equation 11.

To summarise, in this section we first captured the LRU algorithm as a set of recursion equations. These equations were then transformed into a combinatory expression, and during the process of transformation we determined the behaviour of the cells and the connection structure of the implementation.

It should be noted that only the implementation of the state-transition logic has been verified correct with respect to the LRU algorithm. In general the designer must also ensure that the system will be initialised to an appropriate state. Fortunately our LRU processor is self-initialising: it will give the correct result after N insertions where N is the number of cells in the processor.

3.3 Optimising the preliminary design

So far the LRU processor has been expressed as a single state machine with a single bank of latches and long feedback paths. Our next step is to decompose this state machine into a cascade of state machines, which can then be pipelined so that the clock speed is independent of the number of processors. In other words, we shall first construct a semi-systolic array that will subsequently be made fully systolic.

To make use of the theorems in Section 2.2, we promote *insImp* to work on streams by using the stream version of the components and combinators. We assume that there are N components in the row of *insCell*, and that $N = KM$ where $N \geq K \geq 1$.

$$
\begin{array}{cl}
 & LRU0 \\
= & \{\text{equation 10: definition of } LRU0\} \\
 & \mathsf{loop}\,(InsImp\,;\ \mathsf{fst}\mathcal{D}) \\
= & \{\text{equation 24: definition of } InsImp\} \\
 & \mathsf{loop}\,(\mathsf{fst}Dupfst;\ \mathsf{row}_N InsCell;\ \mathsf{snd}\pi_1;\ \mathsf{fst}\mathcal{D}) \\
= & \{\mathsf{snd}F; \mathsf{fst}G = \mathsf{fst}G; \mathsf{snd}F \text{ and } \mathsf{loop}(\mathsf{fst}F;\ G;\ \mathsf{snd}H) = F; \mathsf{loop}G;\ H\} \\
 & Dupfst;\ \mathsf{loop}\,(\mathsf{row}_N InsCell;\ \mathsf{fst}\mathcal{D});\ \pi_1 \\
= & \{\mathsf{row}F; \mathsf{fst}\mathcal{D} = \mathsf{row}(F; \mathsf{fst}\mathcal{D})\} \\
 & Dupfst;\ \mathsf{loop}\,(\mathsf{row}_N(InsCell;\ \mathsf{fst}\mathcal{D}));\ \pi_1 \\
= & \{\text{equation 9: looping a row expressed as a pipe of loops}\} \\
 & Dupfst;\ (\mathsf{loop}(InsCell;\ \mathsf{fst}\mathcal{D}))^N;\ \pi_1 \\
= & \{\mathsf{loop}(InsCell; \mathsf{fst}\mathcal{D}); \mathcal{D} = \mathcal{D}; \mathsf{loop}(InsCell; \mathsf{fst}\mathcal{D}) \\
 & \ \text{and equation 7: pipelining a pipe}\} \\
 & Dupfst;\ ((\mathsf{loop}(InsCell;\ \mathsf{fst}\mathcal{D}))^K;\ \mathcal{D})^M;\ \mathcal{D}^{-M};\ \pi_1 \\
= & \{\mathcal{D}; \pi_1 = \pi_1; \mathcal{D}\} \\
 & LRU1; \mathcal{D}^{-M}
\end{array}
$$

where

$$LRU1 \overset{\text{def}}{=} Dupfst;\ ((\mathsf{loop}(InsCell;\ \mathsf{fst}\mathcal{D}))^K;\ \mathcal{D})^M;\ \pi_1.$$

An instance of $LRU1$ is shown in Figure 6.

Note that $LRU1$ can be used to produce pipelined versions of $LRU0$. The parameter K controls the degree of pipelining: the array is fully pipelined when $K = 1$ and $M = N$, otherwise signal rippling through K cells will occur. Moreover $LRU1$ has a latency of $M + 1 = (N + K)/K$ cycles and requires $3M + N = N(K + 3)/K$ latches; hence a smaller K results in a faster circuit, but the latency and the number of latches in the design will increase.

A designer should therefore select the value of K to achieve the optimal trade-off in speed, latency and the amount of hardware for a particular LRU processor implementation. The readers are referred to [5] for additional discussions and examples on controlling pipelining in regular computational arrays.

3.4 Further refinement

Two observations will be reported in this section. First of all, one can check that a *true* value on the top horizontal output of the proposed architecture (Figure 6) indicates that the input page is not already residing in the primary storage. Hence our design can be used for generating requests for page replacements in the primary storage.

Next, we shall sketch how the number of latches in the LRU processor can be further reduced by adopting a two-phase non-overlapping clock scheme. In such a scheme a latch is made up of two half-latches – for instance in NMOS technology a half-latch is implemented by connecting together a pass transistor and an inverter. Two adjacent half-latches are activated in opposite phases of a two-phase clock,

$$\mathcal{D} \quad = \quad \mathcal{D}_{\phi 1} \, ; \, \mathcal{D}_{\phi 2}$$

such that $\mathcal{D}_{\phi 1}$ is activated during phase $\phi 1$ and $\mathcal{D}_{\phi 2}$ is activated during phase $\phi 2$. One can model this situation by regarding the system as containing two interleaved but independent computations, with the intermediate results of one computation stored in $\mathcal{D}_{\phi 1}$'s and those of the other computation stored in $\mathcal{D}_{\phi 2}$'s, forming a 2-slow system [7].

Now the core of $LRU1$ consists of the expression $(\mathit{Kloopcells}; \mathcal{D})^M$ where

$$\mathit{Kloopcells} \quad \stackrel{\mathrm{def}}{=} \quad (\mathsf{loop}(\mathit{InsCell};\ \mathsf{fst}\mathcal{D}))^K,$$

indicating that $LRU1$ is pipelined by every *Kloopcells*. Given that M is an even number, half of the pipelining latches can be saved if we are content to pipeline by every two *Kloopcells* instead, giving

$$\begin{aligned}(\mathit{Kloopcells}^2\,;\,\mathcal{D})^{M/2} &= (\mathit{Kloopcells}^2\,;\,\mathcal{D}_{\phi 1}\,;\,\mathcal{D}_{\phi 2})^{M/2}\\ &= (\mathit{Kloopcells}\,;\,\mathcal{D}_{\phi 1}\,;\,\mathit{Kloopcells}\,;\,\mathcal{D}_{\phi 2})^{M/2}.\end{aligned}$$

Of course, the speed of the system is halved as well.

Further discussions on n-slow systems can be found in [7].

4 Developing other systolic processors

Remember that the LRU processor has been developed in two steps: casting the algorithm in the combinator notation, and optimising the resulting combinatory expression using algebraic theorems. This is a general strategy for developing systolic processors [4]; and while the first step is usually problem-dependent, the algebraic theorems used in the second step can be applied to rewrite any expression in the required form provided that the preconditions associated with the theorems (such as delay commutativity of components for pipelining theorems) are satisfied.

Our optimisation of the LRU processor (Section 3.3) consists of a rewriting sequence for an expression in the form $\mathsf{loop}(\mathsf{row}\,F\,;\,\mathsf{fst}\mathcal{D})$. This optimisation can be applied to any design with its state-transition logic expressed in row. In the following we shall outline two examples, one involving a numerical algorithm and the other a non-numerical algorithm, which are amenable to this treatment.

Polynomial evaluation. The evaluation of a polynomial by Horner's rule can be described by the following recursive algorithm:

$$\begin{aligned}\mathit{peval}\,\langle s,x\rangle\,\langle\,\rangle &\overset{\text{def}}{=} s,\\ \mathit{peval}\,\langle s,x\rangle\,(\langle a\rangle\hat{\ }as) &\overset{\text{def}}{=} \mathit{peval}\,\langle s\times x+a,x\rangle\;as.\end{aligned}$$

(Notice that given $\mathit{mcell}\,\langle s,x\rangle\,a \overset{\text{def}}{=} \langle s\times x+a,x\rangle$, we could have expressed the algorithm as $peval = \textsf{reduce}\ mcell$.) It can be shown that

$$peval\ \langle s, x\rangle\ as \quad = \quad (\textsf{row}\, madd\ ;\ \pi_2\ ;\ \pi_1)\ \langle\langle s, x\rangle, as\rangle$$

where $madd\,\langle\langle s, x\rangle, a\rangle \overset{\text{def}}{=} \langle a, \langle s \times x + a, x\rangle\rangle$. Having expressed the algorithm in $\textsf{row}$ and checked that $Madd$ is delay-commutative, we can follow the optimisation steps described in Section 3.3 to obtain $((\textsf{loop}\,(Madd;\, \textsf{fst}\mathcal{D}))^K;\, \mathcal{D})^M$, a polynomial evaluator with a serial input and with constant coefficients. This description abstracts from the details of initialising the feedback latches with the sequence of polynomial coefficients.

Sorting. The function *insort a* takes a sorted sequence and inserts the element a at the appropriate place with respect to the ordering relation:

$$\begin{array}{rcl} insort\ a\ \langle\,\rangle & \overset{\text{def}}{=} & \langle a\rangle, \\ insort\ a\ (\langle x\rangle\hat{\ }xs) & \overset{\text{def}}{=} & \textsf{if}\ a \leq x\ \textsf{then}\ \langle a, x\rangle\hat{\ }xs \\ & & \square\ a \geq x\ \textsf{then}\ \langle x\rangle\hat{\ }insort\ a\ xs \\ & & \textsf{fi}. \end{array}$$

It can be shown that

$$last\ (insort\ a\ xs) \quad = \quad (\textsf{row}\, scell;\ \pi_2)\ \langle a, xs\rangle$$

where $scell\ \langle a, x\rangle \overset{\text{def}}{=} \textsf{if}\ a \leq x\ \textsf{then}\ \langle a, x\rangle\ \textsf{else}\ \langle x, a\rangle\ \textsf{fi}$. Again we can follow the rewriting steps described in Section 3.3, since $Scell$ is delay-commutative. This results in $((\textsf{loop}\,(Scell;\, \textsf{fst}\mathcal{D}))^K;\, \mathcal{D})^M$, a sorter with a serial input and a serial output, provided that the feedback latches are initialised with the greatest element given by the ordering relation.

5 Conclusion

Our implementation of the LRU algorithm consists of a regular array of components and is suitable for integrated circuit technology. The fully-pipelined version can accept page insertions at a very high rate, comparable to the speed of a shift register. Furthermore it is very compact: for a system with N pages of primary storage, it contains approximately $(3N \log_2 N + N)$ bits of storage (for feedback and pipelined latches) and $N \log_2 N$ exclusive-or gates for equality testing.

A survey of systematic methods for systolic array design can be found in [2]. In deriving the LRU processor we adopt a simple notation to express both the algorithm and its implementation. This approach allows designs to be transformed using 'traditional' mathematical manipulations such as inductive proofs and equational reasoning. The resulting expressions are concise and can be used to generate designs with different performance trade-offs; and it has been shown that the transformation strategy is general enough to optimise other systolic architectures. Currently tools [6] are being prototyped to support this style of systolic processor development.

Acknowledgements. We thank Michael Goldsmith, Graham Hutton, David Gries, Geriant Jones, Frank Luk, Fred Schneider and an anonymous referee for their careful reading of earlier drafts of this paper. The first author also expresses his gratitude to the Croucher Foundation and to Rank Xerox (UK) Limited for sponsoring his research.

Appendix

We shall first prove that

$$\mathsf{loop}\,(\mathsf{row}_n\, R) \quad = \quad (\mathsf{loop}\, R)^n. \tag{25}$$

The proof is by induction on n. Consider the base case of equation 25,

$$\begin{array}{ll}
 & x\ (\mathsf{loop}\,(\mathsf{row}_0\, R))\ y \\
\equiv & \{\text{equation 8: definition of } \mathsf{loop}\} \\
 & \exists z\,.\ \ \prec x, z\succ\ (\mathsf{row}_0\, R)\ \prec z, y\succ \\
\equiv & \{\text{equation 5: definition of } \mathsf{row}\} \\
 & \exists z\,.\ (x = y) \wedge (z = \prec\ \succ) \\
\equiv & \{\text{equation 3: definition of } \mathit{pipe}\} \\
 & x\ (\mathsf{loop}\, R)^0\ y
\end{array}$$

as required. Consider now the induction case of equation 25,

$$\begin{array}{ll}
 & x\ (\mathsf{loop}\,(\mathsf{row}_{n+1}\, R))\ y \\
\equiv & \{\text{equation 8: definition of } \mathsf{loop}\} \\
 & \exists z, zs\,.\ \ \prec x,\ \prec z\succ^\frown zs\succ\ (\mathsf{row}_{n+1}\, R)\ \prec\prec z\succ^\frown zs,\ y\succ \\
\equiv & \{\text{equation 6: definition of } \mathsf{row}\} \\
 & \exists u, z\,.\ \ \prec x, z\succ\ R\ \prec z, u\succ\ \wedge\ \exists zs\,.\ \ \prec u, zs\succ\ (\mathsf{row}_n\, R)\ \prec zs, y\succ \\
\equiv & \{\text{equation 8: definition of } \mathsf{loop}\} \\
 & \exists u\,.\ x\ (\mathsf{loop}\, R)\ u\ \wedge\ u\ (\mathsf{loop}\,(\mathsf{row}_n\, R))\ y \\
\equiv & \{\text{induction hypothesis}\} \\
 & \exists u\,.\ x\ (\mathsf{loop}\, R)\ u\ \wedge\ u\ (\mathsf{loop}\, R)^n\ y \\
\equiv & \{\text{equation 4: definition of } \mathit{pipe}\} \\
 & x\ (\mathsf{loop}\, R)^{n+1}\ y.
\end{array}$$

(End of proof.)

Next, we shall prove that

$$\mathit{update}\ p\ q\ \mathit{true}\ (\langle x\rangle^\frown xs) \quad = \quad \langle p\rangle^\frown\, \mathit{update}\ x\ q\ (q \neq x)\ xs.$$

The proof is by folding and unfolding the definition of *update* and *delete*.

$$
\begin{array}{ll}
 & \mathit{update}\ p\ q\ \mathit{true}\ (\langle x\rangle\hat{}\,xs) \\
= & \{\text{equation 18: definition of } \mathit{update}\} \\
 & \langle p\rangle\,\hat{}\ \mathit{delete}\ q\ (\langle x\rangle\hat{}\,xs) \\
= & \{\text{equation 15: definition of } \mathit{delete}\} \\
 & \langle p\rangle\,\hat{}\ \textsf{if}\ (q = x) \vee (xs = \langle\,\rangle)\ \textsf{then}\ xs \\
 & \qquad \square\ (q \neq x) \wedge (xs \neq \langle\,\rangle)\ \textsf{then}\ \langle x\rangle\,\hat{}\ \mathit{delete}\ q\ xs \\
 & \qquad \textsf{fi} \\
= & \{\text{equation 18: definition of } \mathit{update}\} \\
 & \langle p\rangle\,\hat{}\ \textsf{if}\ (q = x) \vee (xs = \langle\,\rangle)\ \textsf{then}\ xs \\
 & \qquad \square\ (q \neq x) \wedge (xs \neq \langle\,\rangle)\ \textsf{then}\ \mathit{update}\ x\ q\ \mathit{true}\ xs \\
 & \qquad \textsf{fi} \\
= & \{\text{expanding } \textsf{if}\,\} \\
 & \langle p\rangle\,\hat{}\ \textsf{if}\ q = x\ \ \textsf{then}\ xs \\
 & \qquad \square\ xs = \langle\,\rangle\ \ \textsf{then}\ xs \\
 & \qquad \square\ (q \neq x) \wedge (xs \neq \langle\,\rangle)\ \textsf{then}\ \mathit{update}\ x\ q\ \mathit{true}\ xs \\
 & \qquad \textsf{fi} \\
= & \{\text{equation 17 and equation 19: definition of } \mathit{update}\} \\
 & \langle p\rangle\,\hat{}\ \textsf{if}\ q = x\ \ \textsf{then}\ \mathit{update}\ x\ q\ (q \neq x)\ xs \\
 & \qquad \square\ xs = \langle\,\rangle\ \ \textsf{then}\ \mathit{update}\ x\ q\ (q \neq x)\ xs \\
 & \qquad \square\ (q \neq x) \wedge (xs \neq \langle\,\rangle)\ \textsf{then}\ \mathit{update}\ x\ q\ (q \neq x)\ xs \\
 & \qquad \textsf{fi} \\
= & \{\text{simplify}\} \\
 & \langle p\rangle\,\hat{}\ \mathit{update}\ x\ q\ (q \neq x)\ xs.
\end{array}
$$

(End of proof.)

References

[1] E. W. Dijkstra, Monotonic replacement algorithms and their implementation (EWD 465, 19 December 1974), in: E. W. Dijkstra, Ed., Selected writings on computing: a personal perspective (Springer-Verlag, 1982) 84–88.

[2] J. A. B. Fortes, K. S. Fu and B. W. Wah, Systematic approaches to the design of algorithmically specified systolic arrays, in: Proceedings of International Conference on Acoustics, Speech and Signal Processing (IEEE, 1985) 300–303.

[3] 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) 245–259.

[4] W. Luk and G. Jones, From specification to parametrised architectures, in: G. J. Milne, Ed., The fusion of hardware design and verification (North-Holland, 1988) 267–288.

[5] W. Luk and G. Jones, Parametrized retiming of regular computational arrays, in: P. M. Dew, R. A. Earnshaw and T. R. Heywood, Eds., Parallel processing for computer vision and display (Addison-Wesley, 1989) 50–63.

[6] W. Luk, G. Jones and M. Sheeran, Computer-based tools for regular array design, in: J. McCanny, J. McWhirter and E. Swartzlander, Eds., Systolic array processors (Prentice Hall, 1989) 589–598.

[7] M. Sheeran, Retiming and slowdown in Ruby, in: G. J. Milne, Ed., The fusion of hardware design and verification (North-Holland, 1988) 289–308.

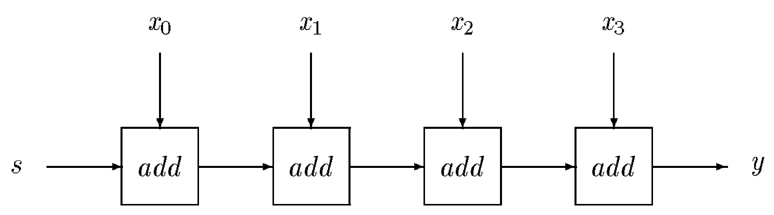


Figure 1: **reduce** $add\ s\ \langle x_0, x_1, x_2, x_3\rangle\ =\ y$.

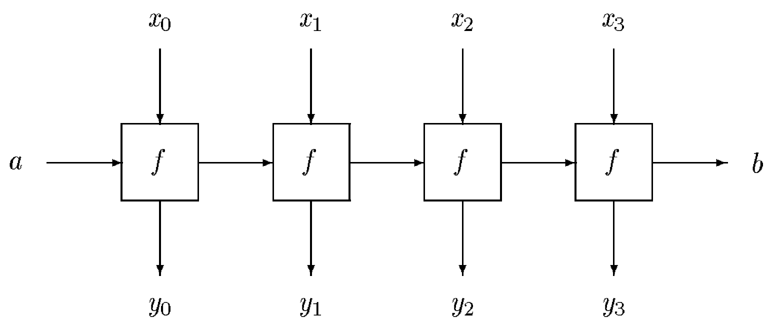
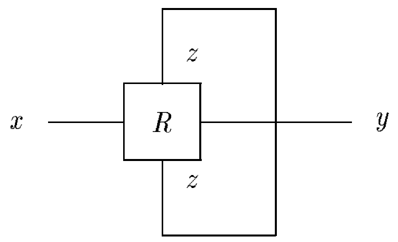
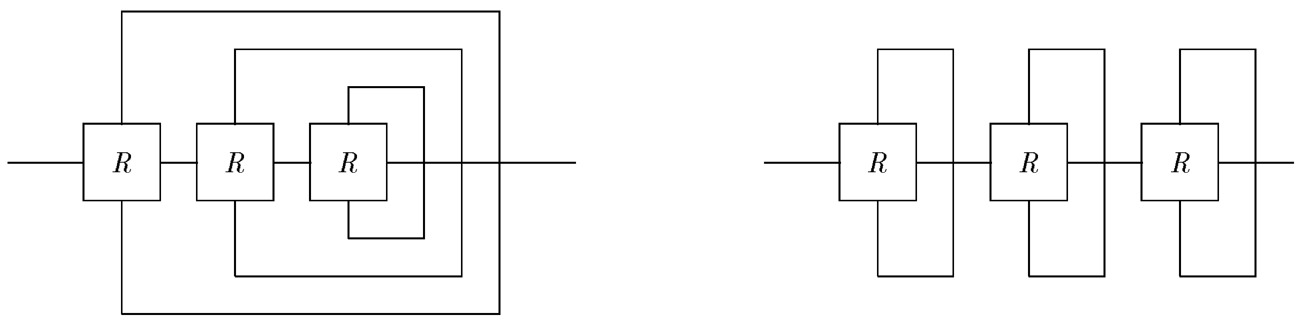


Figure 2: $\mathsf{row}\,f\ \langle a, \langle x_0, x_1, x_2, x_3\rangle\rangle\ =\ \langle\langle y_0, y_1, y_2, y_3\rangle, b\rangle$.



$$x\ (\mathsf{loop}\, R)\ y \ \stackrel{\text{def}}{=}\ \exists z\,.\ \ \prec x, z\succ\ \ R\ \ \prec z, y\succ$$

Figure 3: The loop combinator.



$$\mathsf{loop}\,(\mathsf{row}_3\,R) \quad = \quad (\mathsf{loop}\,R)^3$$

Figure 4: An instance of a theorem involving loop, row and pipe.

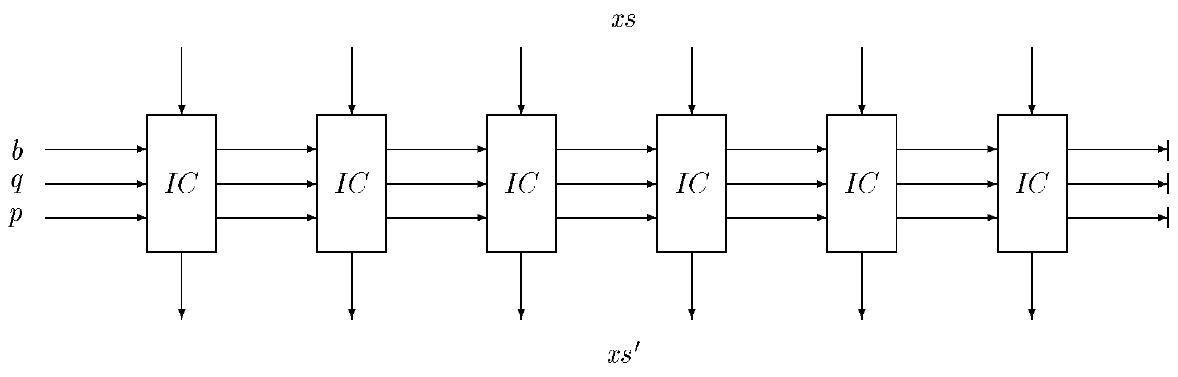


Figure 5: $xs' \ = \ \pi_1\,(\textsf{row}\ insCell\ \langle\langle p, q, b\rangle, xs\rangle)$. ($IC = InsCell$.)

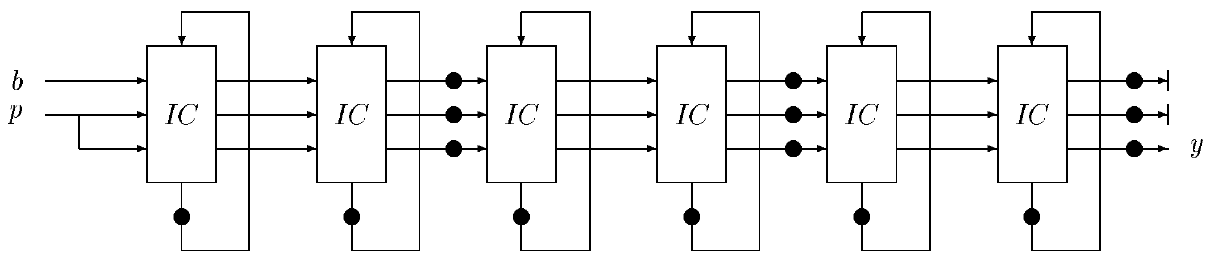


Figure 6: Design $LRU1$ ($N = 6$, $K = 2$, $M = 3$). ($\bullet = \mathcal{D}$ and $IC = InsCell$.)