Formal Methods in System Design, 2: 73–91 (1993)
© 1993 Kluwer Academic Publishers

Using Fixed-Point Semantics to Prove Retiming Lemmas

STEPHEN BROOKES
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract. Algorithms designed for VLSI implementation are commonly described by directed graphs, in which the nodes represent functional units and the arcs indicate communication links. We give a denotational semantics for such a graph in terms of the least fixed point of a set of (mutually recursive) function definitions, describing the outputs produced at each node as a function of time. This semantics is consistent with the conventional clocked operational semantics of the system. A retiming is a systematic modification of the internode delays of a design, often used to convert an algorithm design into a systolic form. The utility of such retimings in optimizing the behavior of designs is well known. We use fixed-point semantics to provide simple proofs of the correctness of certain retiming transformations from the literature and to justify other design transformations such as pipelining.

Keywords: systematic arrays, fixed-point semantics, pipelining, retiming

1. Introduction

The rapidly increasing use of VLSI technology and the potential offered by VLSI technology for parallel computation are well known. It is therefore important to provide rigorous methods of specifying and proving correctness properties of algorithms designed for VLSI implementation. It seems to be common to use a high-level abstract model of algorithm designs in which a design is presented as a labeled directed graph whose nodes are regarded as standing for combinational units and whose edges represent dataflow connections. Such graphs (or related models) have been used (either explicitly or implicitly) by various people in specification and verification of the correctness of VLSI designs, often by using a semantic model in which the behavior of each node is modeled as a function (typically, from time to data values). For instance, functional models are used by Leiserson and Saxe [1], by Kung and Lin [2], in Chen's work on space–time equations [3], in Chen and Mead's hierarchical simulation [4], and in the work of Gordon [5].

In most of these earlier papers, correctness arguments were based (usually implicitly) on operational reasoning, analyzing the "state" of a system at each time step and arguing by induction on the number of elapsed time steps or clock cycles. This reliance on operational reasoning can result in rather complicated proofs [1]. In this article, we show that some elementary ideas from denotational semantics [6, 7] can be used to give simple correctness proofs. We focus on

a transformational approach to VLSI design based on notions such as retiming and pipelining, much as described in [1, 2, 8]. Essentially, we use a denotational semantics using fixed-point theory to describe the behavior of VLSI designs. The use of fixed points and related proof methods such as fixed-point induction [9] in formal reasoning about program behavior is common in branches of theoretical computer science, but we believe that the relevance of these ideas to VLSI theory is not widely realized. Our use of fixed points is inspired by the Kahn–MacQueen treatment of networks of lazy asynchronous functional processes [10], although we employ an underlying model of synchronous computation.

Of this earlier work, ours is most closely related to the algebraic approach of Kung and Lin [2]. We adopt a graphical model of system designs closely related to their z-graphs (and also to the *communication graphs* of Leiserson and Saxe [1]). We provide such graphs with a formal denotational semantics.[1] One of our contributions is a formal justification of the kind of algebraic manipulations performed in transformational approaches to VLSI design (see, for example, [2, 11]), based on a natural underlying denotational semantics rather than by using operational semantics.

An advantage of the denotational approach is that it is well suited to reasoning about hierarchically designed systems and facilitates a modular approach to system design, analysis, and synthesis. In a denotational semantics, the meaning of a compound object is built from the meanings of its parts, so that replacing any part by another with the same meaning leaves the overall meaning unchanged. This justifies certain common design transformations such as pipelining, and more general transformations involving the replacement of a subsystem by an equivalent one. It is also easy to use denotational semantics to analyze the effect of overlaying one design on top of another (with the same underlying network topology).

We illustrate the use of our ideas in the derivation of a systolic design for a palindrome recognizer, similar to the design given in [1], which was itself related to an earlier design by Cole [12]. Beginning with a mathematical specification of the problem, we derive a correct, systolic design. The derivation uses retiming, pipelining, and overlaying.

2. Modeling VLSI systems

It is common to describe algorithms for VLSI implementation by means of directed graphs, in which nodes stand for combinational elements or registers and arcs indicate the dataflow paths between the nodes. While this type of graphical representation can be very useful, it is not ideal if we want to discuss the effect of retiming on VLSI designs. At this level of abstraction, a retiming transformation has the effect of systematically adjusting the number of registers between each pair of combinational nodes. Thus, the graph of a VLSI design may differ considerably from the graph of a retimed version of the same design,

and this can make it unnecessarily complicated to formalize the relationship between the behavior of the system and the behavior of its retimed version.

Instead, we model a VLSI system as a directed graph, each node representing a combinational unit and each arc labeled by a (nonnegative) integer weight and a function symbol. Intuitively, the weight on an arc indicates the number of registers along a data path between two combinational units, or (equivalently) the number of clock cycles required for data to move along a data path. We represent a typical edge, from source u to destination v and with weight d and function symbol f by the notation $f : (u, d, v)$ or, pictorially,

$$u \xrightarrow{d,f} v.$$

We assume the usual graph-theoretical notions of path and cycle. A path leading from node u to node v on which the total accumulated weight is w indicates that the output(s) produced by node v at any time depend on a value output by node u at the wth previous time step. Since we are modeling synchronous (or clocked) systems, we do not specify the exact length of a time step; the delay w corresponds to the number of clock cycles that it takes for data to move along the path.

A cyclic path in a graph indicates a cyclic dependency: the output produced at each node on the cycle may depend on its own output from some earlier time. This kind of path obviously occurs in systems with feedback, and the special case when the total delay on a cycle is zero corresponds to a race condition. For obvious reasons, race conditions in designs are often considered bugs. Intuitively, the presence of a cycle of zero delay may lead to unpredictable operational behavior.

In order to specify a semantics for a system we need to provide an interpretation for its combinational elements. We do this by giving a rule, for each output function of each node, describing how that output is produced from the input values supplied to the node. All outputs and inputs are regarded as functions of time, and since we are modeling synchronous systems, time is regarded as integer-valued. We therefore specify the combinational behavior of the nodes by giving a functional equation for each output function, describing the output functions produced by each node in terms of the input functions at that node and the input delays. This amounts to a fixed-point definition of a set of functions, and the intended semantics of the system is just the least fixed point, which always exists (under certain general and realistic assumptions about the nature of the computations that take place at the nodes). In the case of a system with particularly regular or simple structure, it may be possible to find the intended semantics of the system by inspection. However, even when a solution may not be so obvious, elementary fixed-point techniques can be used to calculate the solution and to reason about it.

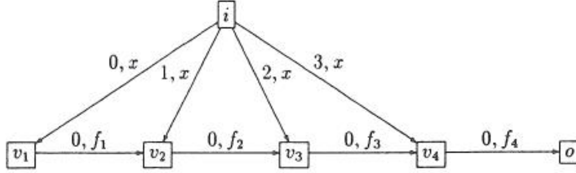

Figure 1. Finite impulse response.

Before giving the mathematical foundations in more detail, we will discuss a simple example.

2.1. An example: finite impulse response

The finite impulse response function (FIR) appears in [2]. The problem is to compute the terms of the sequence $y_n (n \geq 3)$, assuming a given sequence $x_n (n \geq 0)$, corresponding to the recurrence relation

$$y_n = w_1 x_n + w_2 x_{n-1} + w_3 x_{n-2} + w_4 x_{n-3} \qquad (n \geq 3).$$

The $w_k (1 \leq k \leq 4)$ are given integers, and we assume that the x_n are all integers. Thus, the problem is to compute certain weighted sums.

The system pictured in figure 1 solves this problem. The graph structure is

$$\begin{aligned} G &= (V, E, F) \\ V &= \{i, v_1, v_2, v_3, v_4, o\} \\ E &= \{x : (i, 0, v_1), x : (i, 1, v_2), x : (i, 2, v_3), x : (i, 3, v_4), \\ &\quad f_1 : (v_1, 0, v_2), f_2 : (v_2, 0, v_3), f_3 : (v_3, 0, v_4), f_4 : (v_4, 0, o)\} \\ F &= \{x, f_1, f_2, f_3, f_4, y\}. \end{aligned}$$

We are modeling time as a sequence of discrete steps, so we use the natural numbers to represent time instances. In order to keep the distinction between integer times and integer data values, we use T for the set of times, N for the set of integer values, and we let t range over T, n over N. In this example, each function symbol is to be interpreted as a function from T to N; since each $f_k(t)$ is only defined for $t \geq k - 1$, it is natural to regard these as *partial functions* rather than total functions. The function definitions are

$$\begin{aligned} f_1(t) &= w_1 x(t), \\ f_k(t) &= w_k x(t - k + 1) + f_{k-1}(t) \qquad (k = 2, 3, 4), \\ y(t) &= f_4(t). \end{aligned}$$

It is obvious and easy to show formally that these equations have a closed-form solution

$$
\begin{aligned}
f_1(t) &= w_1 x(t),\\
f_2(t) &= w_1 x(t) + w_2 x(t-1),\\
f_3(t) &= w_1 x(t) + w_2 x(t-1) + w_3 x(t-2),\\
f_4(t) &= w_1 x(t) + w_2 x(t-1) + w_3 x(t-2) + w_4 x(t-3).
\end{aligned}
$$

Thus, if the terms of a sequence x_t are input along x, so that for each time t we have $x(t) = x_t$, then the output sequence $y_t = y(t)$ satisfies the desired recurrence relation.

We now move towards a more general treatment, as follows. Firstly, we can be more succinct and eliminate t from the defining equations by introducing functionals $\Phi_k (1 \leq k \leq 4)$ defined by

$$
\begin{aligned}
\Phi_1(g) &= \lambda t. w_1 g(t),\\
\Phi_k(g, h) &= \lambda t. (w_k g(t) + h(t)) \qquad (k = 2, 3, 4).
\end{aligned}
$$

Note the use of lambda-notation: for instance, $\Phi_1(g)$ is simply the function mapping each time t to the value $w_1 g(t)$ (provided this expression has a defined value at time t).

Now let $Z : T \rightarrow T$ be the "delay function," described by

$$Z(t) = (t < 1 \rightarrow \perp, t-1).$$

Here, as is common, we use the special symbol $\perp$ to indicate undefinedness, so that $Z(0)$ is undefined, and $Z(t) = t - 1$ if $t > 0$. The notation $(p \rightarrow q, r)$ is a form of conditional expression, and is intended to denote the value of q if p is true and the value of r if p is false, and is undefined otherwise. We also define Z^n, the n-fold iteration of Z, for $n \geq 0$, given by

$$Z^n(t) = (t < n \rightarrow \perp, t-n).$$

Note that $Z^1 = Z$, and $Z^0(t) = t$ for all t. Note also the obvious identities $Z^m \circ Z^n = Z^{m+n}$ for all $m, n \geq 0$.

Now the equations defining the f_k can be rewritten as

$$
\begin{aligned}
f_1 &= \Phi_1(x),\\
f_k &= \Phi_k(x \circ Z^{k-1}, f_{k-1}),\\
y &= f_4.
\end{aligned}
$$

In this form, the functional equations are susceptible to algebraic manipulation without keeping such explicit track of the time parameter. By substitution, we

obtain immediately the following closed-form solution:

$$\begin{aligned}
f_1 &= \Phi_1(x),\\
f_2 &= \Phi_2(x \circ Z^1, \Phi_1(x)),\\
f_3 &= \Phi_3(x \circ Z^2, \Phi_2(x \circ Z^1, \Phi_1(x))),\\
f_4 &= \Phi_4(x \circ Z^3, \Phi_3(x \circ Z^2, \Phi_2(x \circ Z^1, \Phi_1(x)))).
\end{aligned}$$

Of course, this represents the same closed-form solution given above.

This example is rather simple, since the system behavior involves an extremely straightforward flow of data. In more complex designs, it may be more difficult to prove correctness: there may be a lot of bookkeeping to do to show that data are transmitted around the system in such a way that the correct values are always at the right place at the right time. Moreover, the nature of the functions computed by a system may not always be so obvious. We now show that under certain very general conditions, elementary fixed-point techniques provide a guarantee that a system has a well-defined semantics and provide a means of calculating effectively what functions are computed in a system.

3. Fixed-point semantics

A system $S = (V, E, F)$ is described by a set of equations for the function symbols in F. For each node $v \in V$ with inputs $f_1, \ldots, f_n$ weighted $d_1, \ldots, d_n$, respectively, for each output symbol f of v, there is an equation of the form

$$f = \Phi(f_1 \circ Z^{d_1}, \ldots, f_n \circ Z^{d_n}),$$

where Φ is a functional that captures (an aspect of) the combinational behavior of node v: this equation describes how the output function f depends on the input functions $f_1, \ldots, f_n$.

Let T be the set of natural numbers. Suppose that each arc transmits data of a certain type (for example, integer or boolean). Then each function symbol is intended to denote a partial function of type $T \to V$, where V is the type associated with the arc labeled by the function symbol. With the obvious association of types to arcs, the functional Φ in this equation has type

$$(T \to V_1) \times \cdots \times (T \to V_n) \to (T \to V).$$

Using $\perp$ to represent undefinedness, any set $T \to V$ of partial functions may be partially ordered by the relation $f \subseteq g$, defined as follows:

$$f \subseteq g \quad \Leftrightarrow \quad (\forall t \in T)(f(t) \neq \perp \Rightarrow f(t) = g(t)).$$

We use the symbol $\subseteq$ for this relation because it coincides with the usual notion of set inclusion on the graphs of the two partial functions. If $f \subseteq g$, then at all times t for which $f(t)$ is defined, $g(t)$ is also defined, and the two values are equal.

This partial order is complete in that every increasing sequence of partial functions has a limit (or least upper bound): this is because whenever f_m is a partial function and $f_m \subseteq f_{m+1}$ for each $m \geq 0$, then $f = \bigcup_{m=0}^{\infty} f_m$ is again a partial function.

Since each node in a VLSI design is a combinational element, we lose no generality if we make the following *Combinational Assumption:* there is a function ϕ on data values such that, for all $t \in T$, and all partial functions $g_1, \ldots, g_n$,

$$\Phi(g_1, \ldots, g_n)(t) = \phi(g_1(t), \ldots, g_n(t)).$$

Several important and useful properties follow from this assumption:

- Each Φ is *monotone*, in that

$$g_1 \subseteq h_1, \ldots, g_n \subseteq h_n \Rightarrow \Phi(g_1, \ldots, g_n) \subseteq \Phi(h_1, \ldots, h_n).$$

- Each Φ is *continuous*, or preserves limits: if $g_i^{(m)} \subseteq g_i^{(m+1)}$ for all m and each i between 1 and n, then

$$\Phi\left(\bigcup_{m=0}^{\infty} g_1^{(m)}, \ldots, \bigcup_{m=0}^{\infty} g_n^{(m)}\right) = \bigcup_{m=0}^{\infty} \Phi\left(g_1^{(m)}, \ldots, g_n^{(m)}\right).$$

- Each Φ *distributes over composition*, in that for all partial functions $g_1, \ldots, g_n$ and h,

$$\Phi(g_1, \ldots, g_n) \circ h = \Phi(g_1 \circ h, \ldots, g_n \circ h).$$

We define for each function symbol f a chain $f^{(m)} (m \geq 0)$ of partial functions as follows, by induction on m, using the functional equation for f:

$$f^{(0)} = \lambda t.\, \bot$$
$$f^{(m+1)} = \Phi\left(f_1^{(m)} \circ Z^{d_1}, \ldots, f_n^{(m)} \circ Z^{d_n}\right).$$

Intuitively, each $f^{(m)}$ is obtained by expanding the functional definitions m times and replacing all remaining recursive calls by the everywhere undefined function $\lambda t.\, \bot$. The fact that $f^{(m)} \subseteq f^{(m+1)}$ for each $f \in F$ and each $m \geq 0$ follows by monotonicity of each Φ.

Let $\hat{f}$ be the limit of this sequence of partial functions:

$$\hat{f} = \bigcup_{m=0}^{\infty} f^{(m)}.$$

Then by continuity of Φ, we obtain the equations

$$\hat{f} = \Phi(\hat{f}_1 \circ Z^{d_1}, \ldots, \hat{f}_n \circ Z^{d_n}).$$

Moreover, the $\widehat{f}$ are the least solutions to these equations, in that for every collection of functions f^* such that

$$f^* = \Phi(f_1^* \circ Z^{d_1}, \ldots, f_n^* \circ Z^{d_n}),$$

we get $\widehat{f} \subseteq f^*$. Thus, each $\widehat{f}$ function has a defined value at time t if and only if that value can be deduced from the defining equations.

Note that in a design with a cycle of zero weight, the fixed-point semantics still makes sense, and the operational unpredictability of such a design would correspond to the persistent undefinedness of an output function produced along such a cycle, since the semantics will only ascribe a proper output value at any time when this is truly implied by the functional equations. As an extreme example, the system defined by the equation $f = f$ (with a graph containing a single node and a cyclic arc with zero delay) has semantics $\widehat{f} = \lambda t.\ \bot$.

3.1. Example: finite impulse response, revisited

The defining equations for this system were

$$f_1 = \Phi_1(x),$$
$$f_k = \Phi_k(x \circ Z^{k-1}, f_{k-1}) \qquad (k = 2, 3, 4).$$

The sequences of approximations are therefore simply

$$f_k^{(0)} = \lambda t.\ \bot \qquad (k = 1, 2, 3, 4),$$
$$f_1^{(m+1)} = \Phi_1(x),$$
$$f_k^{(m+1)} = \Phi_k\left(x \circ Z^{k-1}, f_{k-1}^{(m)}\right) \qquad (k = 2, 3, 4).$$

Trivially it follows that $f_1^{(m)} = \Phi_1(x)$, for all $m \geq 1$, so that the limit function $\widehat{f_1}$ is also $\Phi_1(x)$. Similarly,

$$f_2^{(m)} = \Phi_2(x \circ Z^1, f_1^{(m-1)}) = \Phi_2(x \circ Z^1, \Phi_1(x)),$$

for all $m \geq 2$. The analyses for f_3 and f_4 are also easy. This shows that the approximations converge, as expected, to the functions derived earlier as closed-form solutions for the f_k.

4. Retiming transformations

A retiming is a systematic transformation of the internode delays in the communication graph of a system. The utility of retimings in improving the timing behavior of systems is argued cogently in [1, 13]. In particular, a design is *systolic* if each (internal) arc has weight at least one. Systolic designs are especially

appealing because the clock period in a system is independent of the size of the system and is typically short, since it only depends on the time needed for a signal to propagate through an individual combinational unit (equivalently, the time taken by the slowest individual node in the system to perform its output calculations). In a general system, the clock period may need to be longer to allow for rippling of inputs along paths consisting of arcs with zero delay. Leiserson and Saxe [1] show how to transform a wide variety of system designs into an equivalent systolic version by means of retimings. Other people have used retimings to support the development of fault-tolerant systolic designs [8]. We discuss the main notions of retiming introduced by these authors, and show how easy it is in our framework to prove the correctness of their retiming methods.

Two simple yet powerful notions of retiming were introduced in [1]. The first may be thought of as a *lagging* operation, the second as a slowing. In a lagging transformation, we assign an integer "lag" to each node of the system and adjust the weights on each edge according to a simple formula: if each node v_i is given lag l_i, then each edge (v_i, w_{ij}, v_j) of the original system becomes $(v_i, w_{ij} + l_j - l_i, v_j)$ in the retimed system. Intuitively the effect is to delay each output of each node by a fixed amount. Like Leiserson and Saxe, we assume that the delays are chosen so that the new weights are all nonnegative. A slowing transformation multiples each edge weight of a system by a constant (integer) factor k; the resulting system is a k-slowed version of the original system. Intuitively the effect is to produce a system that computes the same outputs as before, but with a delay of length k between successive outputs. We now formalize these two types of retiming and prove that the effect of a retiming on the semantics of a design is as expected.

4.1. Lagging

Given a system $S = (V, E, F)$, suppose we have an integer lag l_i to be associated with each node v_i. Let v be a typical node of V with an output arc labeled by the function symbol f. Suppose that the incoming arcs for this node are $f_i : (v_i, d_i, v)$, for $i = 1 \ldots n$. The output equation for f then has the form

$$f = \Phi(f_1 \circ Z^{d_1}, \ldots, f_n \circ Z^{d_n}).$$

For clarity, we will prime all the function symbols in the retimed system (so that f becomes f', and each f_i becomes f_i'). The lagging operation replaces each d_i by $d_i + l - l_i$, where l is the lag assigned to node v. The corresponding equation in the retimed system S' is thus

$$f' = \Phi(f_1' \circ Z^{d_1+l-l_1}, \ldots, f_n' \circ Z^{d_n+l-l_n}).$$

We assume that the delays are chosen so that each new weight $d_i + l - l_i$ is nonnegative.

We claim that the least functions satisfying the equations for S' are just the appropriately lagged versions of the least functions satisfying the equations for S. More precisely, for each output function f of a node v with lag l, $\hat{f'} = \hat{f} \circ Z^l$. Hence, at all times $t \geq l$, $\hat{f'}(t) = \hat{f}(t-l)$.

This is easy to prove, using the following property of the functionals Φ: for all partial functions $h_1, \ldots, h_n$, and all $l \geq 0$,

$$\Phi(h_1, \ldots, h_n) \circ Z^l = \Phi(h_1 \circ Z^l, \ldots, h_n \circ Z^l),$$

which is implied by the Combinational Assumption.

The desired result can be proved using *fixed-point induction* [9], but we will give a direct proof. First we show by induction on m that for each $m \geq 0$ and all output symbols $f, f'^{(m)} \circ Z^l = f^{(m)}$. Note that the property to be proven concerns *all* output functions at once. The base case is trivial: For each $f \in F$ we have

$$f'^{(0)} = (\lambda t.\ \bot) = (\lambda t.\ \bot) \circ Z^l = f^{(0)}.$$

For the inductive step, let f and f' be defined by the equations

$$\begin{aligned} f &= \Phi(f_1 \circ Z^{d_1}, \ldots, f_n \circ Z^{d_n}), \\ f' &= \Phi(f'_1 \circ Z^{d_1+l-l_1}, \ldots, f'_n \circ Z^{d_n+l-l_n}). \end{aligned}$$

By the induction hypothesis, $f'^{(m)} = f^{(m)} \circ Z^l$ and for each i, $f_i'^{(m)} = f_i^{(m)} \circ Z^{l_i}$. Hence,

$$\begin{aligned} f'^{(m+1)} &= \Phi(f_1'^{(m)} \circ Z^{d_1+l-l_1}, \ldots, f_n'^{(m)} \circ Z^{d_n+l-l_n}) \\ &= \Phi(f_1'^{(m)} \circ Z^{d_1-l_1} \circ Z^l, \ldots, f_n'^{(m)} \circ Z^{d_n-l_n} \circ Z^l) \\ &= \Phi(f_1'^{(m)} \circ Z^{d_1-l_1}, \ldots, f_n'^{(m)} \circ Z^{d_n-l_n}) \circ Z^l \\ &= \Phi(f_1^{(m)} \circ Z^{l_1} \circ Z^{d_1-l_1}, \ldots, f_n^{(m)} \circ Z^{l_n} \circ Z^{d_n-l_n}) \circ Z^l \\ &= \Phi(f_1^{(m)} \circ Z^{d_1}, \ldots, f_n^{(m)} \circ Z^{d_n}) \circ Z^l \\ &= f^{(m+1)} \circ Z^l. \end{aligned}$$

Hence, for all $m \geq 0$, we have $f'^{(m)} = f^{(m)} \circ Z^l$. It follows, since composition with the fixed function Z^l is a continuous operation, that the equation also holds in the limit: $\hat{f'} = \hat{f} \circ Z^l$, as required.

Note that in Leiserson and Saxe's paper [1], the proof only applies to systems in which there are no cycles of total weight zero. The existence in a system of such a cycle means intuitively that there is a race condition (some output function is defined at time t in terms of its own result at time t), and therefore one might reasonably want to rule out such occurrences in system designs. Nevertheless, our fixed-point treatment may still be used even in such circumstances, and the retiming proof goes through. Note that if a design has a cycle of zero weight, then so does its retimed version.

4.2. Slowing

The k-slowed version of a system ($k \geq 1$) is obtained by replacing each function equation

$$f = \Phi(f_1 \circ Z^{d_1}, \ldots, f_n \circ Z^{d_n})$$

by (again, priming the function symbols):

$$f' = \Phi(f'_1 \circ Z^{kd_1}, \ldots, f'_n \circ Z^{kd_n}).$$

Let $\overline{k}$ be the function $\lambda t.kt$. We claim that

$$\widehat{f'} \circ \overline{k} = \widehat{f},$$

so that for all $t \geq 0, \widehat{f'}(kt) = \widehat{f}(t)$.

Note first the following identity, which holds for all $h_1, \ldots, h_n$ and all $k \geq 1$:

$$\Phi(h_1, \ldots, h_n) \circ \overline{k} = \Phi(h_1 \circ \overline{k}, \ldots, h_n \circ \overline{k}).$$

This again is a consequence of the Combinational Assumption. Again one can show by induction on m that for all $m \geq 0$, $f'^{(m)} \circ \overline{k} = f^{(m)}$. The proof is similar to the previous proof, but uses the identity

$$Z^{kw} \circ \overline{k} = \overline{k} \circ Z^{w}.$$

which holds for all $w, k \geq 0$. The result follows by continuity of composition with the fixed function $\overline{k}$: we get $\widehat{f'} \circ \overline{k} = \widehat{f}$.

Again, if a design has a cycle of zero weight, then so does its k-slowed version. Again, our proof is applicable even in such degenerate cases.

4.3. The cut theorem

Kung and Lam [8] give a simple yet useful Cut Theorem concerning the design of fault-tolerant systolic arrays without feedback cycles. We first recall their definitions and then give a simple proof of their theorem; then we make some remarks about the generality under which the result can be used.

A *cut* is a set of edges that partitions a system graph into two disjoint sets of nodes, the *source set* and the *destination set*, with the property that the edges crossing the boundary between these sets are the cut edges, and all cut edges are directed from source set to destination set.

The Cut Theorem states the following: adding the same delay d to all edges entering the destination set of a cut produces an equivalent system. To be precise, if the cut partitions a system S into source set A and destination set B, and if S' is the retimed system, with A' and B' as the subsystems that correspond to A and B, then all output functions computed in A' are identical to their counterparts

in A, and each output function computed in B' lags behind its counterpart in B by d time units.

These properties are intuitively obvious, and Kung and Lam [8] gave a proof that analyzed the flow of data among the registers linking arcs, which they regarded as much simpler than the proof supplied by Leiserson and Saxe [1] for the (more general) retiming lemma. In fact the Cut Theorem is itself a special case of the retiming lemma. A cut corresponds simply to a lagging transformation in which each node in the source set gets lag 0 and each node in the destination set gets lag d; this has the intended effect of adding d to the weights of all edges leading into the destination set, and the effect predicted by the retiming lemma is exactly as required.

Again, if one prefers a direct proof of the Cut Theorem, it is easy to provide one using elementary fixed-point properties. In a cut partition, it is evident that the functions defined by nodes in A are independent of the functions defined in B, because none of the functions computed by B is an input to any of the nodes in A. This is again true for A' and B'. Moreover, apart from renaming the function symbols to avoid confusion, the subsystems A and A' are identical (and therefore, so are the functions computed). If we add d to the weights of all arcs leading into B we get B'. The fact that for each output function f of a node in B the output function f' of the corresponding node in B' satisfies the condition $\hat{f'} = \hat{f} \circ Z^d$ is easy to prove, either by repeating the (relevant part of the) argument given above for the general lagging transformation, adapted to the special case used here, or by using fixed-point induction.

Note also that although Kung and Lam [8] stated the Cut Theorem only for systems without feedback, the result (and our proof) applies even in systems permitting feedback, provided there is no feedback from the destination set to the source set. That is, the result holds if each feedback cycle is contained entirely in either the source set or the destination set.

4.4. Pipelining

In many system designs the same input value is needed at several nodes at different times. Instead of inputting directly to each such node, when the nodes can be linearly ordered by the relative times at which they need to operate on the data, we can arrange for each node to pass on the input value to its successor, with the appropriate extra delay: this is known as *pipelining*. The effect on the functions computed by a system when the system is reorganized to employ pipelining is trivial. In essence, what used to be several different input arcs each labeled (say) x becomes a sequence of arcs, successively labeled x_k, where each x_k computes a lagged version of x. It is then obvious that such changes make no difference to the other functions computed by a system, provided that the internode delays along the pipeline are chosen correctly.

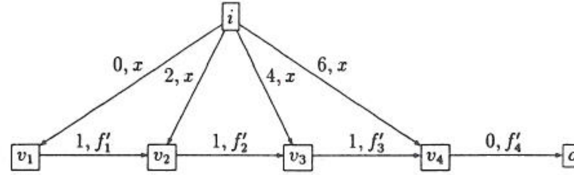

Figure 2. Retimed finite impulse response.

4.5. *Refinement*

In a hierarchical development of a system to meet a desired specification, one may want to begin with a design based on certain types of node and later to refine these nodes, replacing them with an equivalent combination or subsystem. In the FIR system, for instance, we might replace each multiply–add unit by the obvious sequence (multiply first, then add). Provided we do not insert any delay between these successive operations, the overall effect is obviously going to be the same.

5. Example: retiming and pipelining in the FIR system

To illustrate the ideas, consider again the graph corresponding to the finite impulse response (FIR) problem (figure 1). If we retime using the lagging transformation that delays each v_k by $k-1$ (with zero delay at the input node i), we get the graph of figure 2.

Again using primed function symbols, the function equations in the retimed system are

$$f'_1 = \Phi_1(x),$$
$$f'_k = \Phi_k(x \circ Z^{2(k-1)}, f'_{k-1} \circ Z^1) \quad (k = 2,3,4).$$

Either by inspection, or by the retiming lemma, the following relationships hold:

$$\widehat{f'_k} = \widehat{f_k} \circ Z^{k-1} \quad (k = 1,2,3,4).$$

Since x is required at each of the v_k nodes successively, we may pipeline x by sending it first to v_1 and then to v_2 with a delay of 2; v_2 then sends it with a further delay of 2 to v_3, and so on. The result is the system defined by figure 3 and the equations

$$x_0 = x,$$
$$f_1 = \Phi_1(x_0),$$
$$x_k = x_{k-1} \circ Z^2 \qquad (k = 1,2,3),$$
$$f_k = \Phi_k(x_{k-1}, f_{k-1} \circ Z) \quad (k = 2,3,4).$$

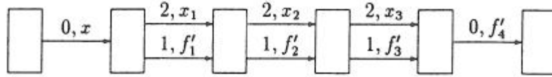

Figure 3. Pipelined, retimed finite impulse response.

It is obvious that the equations $x_k = x \circ Z^{2k}$ hold for $k = 0, 1, 2, 3$, and hence that the f_k' defined by these equations are identical to the f_k' defined by the retimed FIR system. This system is systolic, since the delays on all internal arcs are all positive.

6. A palindrome recognizer

In this section we apply the methods of the previous section to a problem drawn from the literature: Leiserson and Saxe [1] described a systolic array for recognizing palindromes. The successive characters of a string or sequence are to be input to the array, one at each clock step, and the output of the array at time $t + 1$ is to be a boolean value indicating if the characters input up to time t form a palindrome (i.e., if the input string reads the same backwards as forwards). We show how to derive a correct design directly from the mathematical specification of the problem, and as a consequence we obtain a formal correctness proof of (a variant of) the solution proposed in [1].

Informally, a nonempty string is a palindrome if and only if its first half is the reverse of its second half. We specify this more rigorously as follows. We use the notation $x_{[i:j]}$ for the substring $x_i \ldots x_j$ when $i \leq j$ (and likewise, $x_{[j:i]}$ for $x_j \ldots x_i$ when $i \geq j$). Two strings are equal if they have the same length and identical components. A string $x_0 \ldots x_n (n \geq 0)$ of length $n + 1$ is a palindrome if and only if $x_{[0:m]} = x_{[n:m+1]}$, where $m = \lfloor n/2 \rfloor$, or, equivalently, for all $i \leq \lfloor n/2 \rfloor, x_{n-i} = x_i$.

We want to build a system with input x and output p satisfying the following condition: for all $t \geq 0$,

$$p(t) = \forall i \leq \lfloor t/2 \rfloor . (x(t - i) = x(i)).$$

Clearly such a system can be built from two subsystems, one to input x and distribute its values through a suitably arranged sequence of outputs (a_i and b_i), and one to perform equality tests on single characters: for each $i \leq \lfloor t/2 \rfloor$ we want

$$a_i(t) = x(t - i), \qquad b_i(t) = x(i),$$

and then we will set

$$p(t) = \forall i \leq \lfloor t/2 \rfloor . (a_i(t) = b_i(t)).$$

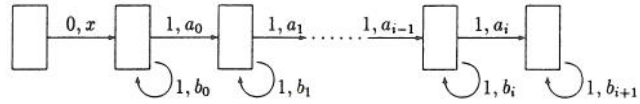

Figure 4. Input routing.

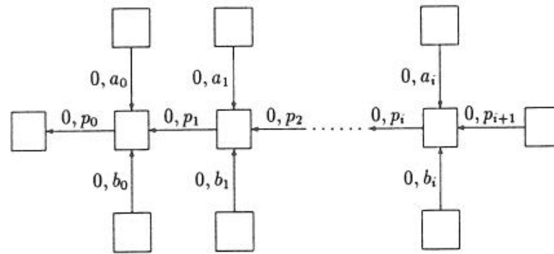

Figure 5. Combining character tests.

The problem specification does not define values for $a_i(t)$ or $b_i(t)$ when $t < 2i$. The following pipelining definitions for the a_i are obvious:

$$a_0 = x,$$

$$a_{i+1} = \lambda t.(t < 2i + 2 \rightarrow \perp, a_i(t-1)) \qquad (i \geq 0).$$

The least functions satisfying these equations are given by

$$a_i(t) = (t < 2i \rightarrow \perp, x(t-i)) \qquad (i \geq 0).$$

For the b_i we can use the definition:

$$b_i(t) = (t < 2i \rightarrow \perp, (t = 2i \rightarrow a_i(t), b_i(t-1))) \qquad (i \geq 0).$$

In other words, b_i is undefined for $2i$ time steps, gets initialized on step $2i$, and remains constant thereafter. Equivalently, we may use the equations:

$$b_0 = \lambda t.(t = 0 \rightarrow x(t), b_0(t-1))$$

$$b_{i+1} = \lambda t.(t < 2i + 2 \rightarrow \perp, (t = 2i + 2 \rightarrow a_i(t-1), b_{i+1}(t-1)).$$

This leads to the design of figure 4. Note the fact that the labels on the arcs of this design fit the form of the defining equations for the output functions.

We can implement the part of the system that computes the palindrome function using character comparators, linked in a sequence, and conjoining their results. We need there to be $\lfloor t/2 \rfloor$ "active" comparators at time t, and an "inactive" comparator should output the default value T. One possible design appears in figure 5. For each i, the equation defining p_i is

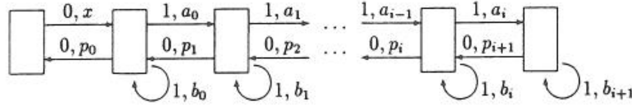

Figure 6. Palindrome recognizer.

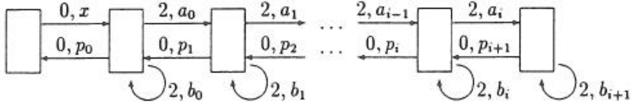

Figure 7. Palindrome recognizer, 2-slowed.

$$p_i(t) = (t < 2i \rightarrow \mathrm{T}, (a_i(t) = b_i(t)) \wedge p_{i+1}(t)).$$

Note that these two systems are compatible, in that the first system defines the a_i and b_i outputs while the second system inputs these functions and uses them to define the p_i. We may therefore superpose the two designs, with the effect of combining the sets of equations. This produces the design of figure 6.

This design is not systolic: the leftmost node, trying to compute $p_0(t)$, must wait until the result of the rightmost active comparator has filtered through before giving its answer, because of the sequence of zero delay arcs linking the successive nodes. It is also impossible to find a lagging that will make the system systolic, because every lagging retiming preserves the total delay on all cycles in a system, and this system has cycles of length 2 with total delay 1: in a systolic system the delay on a cycle must be at least the cycle's length. However, we can produce a systolic version if we first slow the system by a factor of 2. The result of this slowing is the system of figure 7.

If we now introduce, for each i, lag $-i$ at the node computing a_i and b_i, we obtain the systolic design of figure 8.

The functional equations and semantic definitions to go with this design are easy to derive from the original definitions by applying the retiming lemmas. The correctness of this final design is guaranteed because of the (more obvious) correctness of the initial design, based as it was on the mathematical specification of the problem.

7. Conclusions

We have described how elementary fixed-point techniques and a simple functional semantic interpretation for VLSI designs may be used to justify certain transformations such as retimings and pipelining. The fixed-point proofs make clear the underlying assumptions on which these proofs depend and the degree of

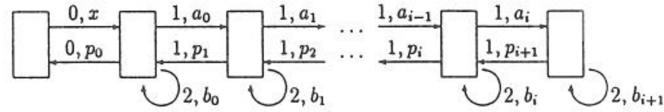

Figure 8. Palindrome recognizer, 2-slowed and lagged.

generality behind these results. In particular, our proofs of the retiming results work provided the functionals associated with each node obey certain natural delay-respecting properties:

$$\Phi(h_1, \ldots, h_n) \circ Z^l = \Phi(h_1 \circ Z^l, \ldots, h_n \circ Z^l),$$
$$\Phi(h_1, \ldots, h_n) \circ \overline{k} = \Phi(h_1 \circ \overline{k}, \ldots, f_n \circ \overline{k}).$$

These properties will certainly hold for the kinds of combinational units normally used in implementing systems. Although race conditions may be regarded as undesirable in system designs, our results do not depend on their absence. Our proof of the Cut Theorem does not require that the cut be chosen so that there is no feedback even within the source set and within the destination set, although of course there must be no flow back across the cut into the source.

It is perhaps worth pointing out that one of the main reasons why Leiserson and Saxe [1] needed a rather longer proof for the retiming lemma concerns their reliance on operational reasoning about the flow of data from register to register on each clock cycle. They were, in effect, using an operational semantics as the basis for their reasoning. Moreover, their graph representation for a system design includes nodes for registers as well as for combinational units. The effect of a retiming transformation on such a representation is to add or delete registers along the paths between combinational units, and thus to alter the graphical structure. To show equivalence of a design with a retimed version then requires comparison of the operational behavior of two graphs with different structure: the proof given in [1] establishes a time-dependent correspondence between the contents of certain pairs of registers, one from each graph, and the details are rather intricate.

In contrast, in our graphical representation (and in the z-graph notation of Kung and Lin [2]) nodes are always combinational units, and only the number of registers between successive combinational units is retained, as a weight label on an arc. The effect of a retiming is just to modify weights, leaving graph structure the same. We use, instead, a denotational semantics. The main difference in semantic descriptions here is stylistic, but the use of a functional semantics and fixed-point theory to provide a satisfactory treatment or recursion has the advantage of providing us with a powerful technique based on fixed-point induction. It would be straightforward to present a fully fledged operational semantics for a language of VLSI designs, and to show that the fixed-point

semantics used in this article essentially coincides with the behavior predicted by the operational semantics. However, the natural style of proof for an operational model involves induction on the length of a computation (here, on the number of time steps), and it seems to be easier to use fixed-point induction for analyzing retimings.

The methodology illustrated here can be used to support incremental and hierarchical development and analysis of systems. Although the palindrome example is not very profound in itself, it does demonstrate the joining together of two compatible systems; this is one of the keys to designing systems incrementally. The effect this has on the semantics of the system is obvious, and the justification is trivial. In a hierarchical development of a system, one might want to first assume as given a node with certain combinatorial nature, and design a system built from such nodes. Later one might implement these nodes themselves in terms of a system built from "smaller" nodes; provided that this system computes the same function(s) as the node was assumed to compute, replacement of the node by the newly designed system will not alter the behavior of the overall system. Again, the justification for this will be easy.

Notes

1. Kung and Lin alluded to the relevance of fixed points in a footnote to their paper; however, they did not elaborate on this, and they based their reasoning on operational arguments.

References

[1] Charles E. Leiserson and James B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1): 41–68 (1983).

[2] H.T. Kung and W.T. Lin. An algebra for VLSI computation. In *Elliptic Problem Solvers II*, G. Birkhoff and A.L. Schoenstadt (eds.). Academic Press, New York, January 1983.

[3] M.C. Chen. *Space-time Algorithms: Semantics and Methodology*. Ph.D. thesis, Department of Computer Science, California Institute of Technology, May 1983.

[4] M.C. Chen and C.A. Mead. A hierarchical simulator based on formal semantics. In *Proceedings of the Third Caltech Conference on Very Large Scale Integration*, R. Bryant (ed.). Computer Science Press, Rockville, Maryland, March 1983, pages 207–223.

[5] M.J.C. Gordon. A model of register transfer systems with applications to microcode and VLSI correctness. Department of Computer Science Technical Report CSR-82-81, University of Edinburgh, 1981.

[6] J.E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press, Cambridge, MA, 1977.

[7] D.S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In *Proceedings of the Symposium on Computers and Automata*, J. Fox (ed.). Polytechnic Institute of Brooklyn Press, New York, 1971, pp. 19–46.

[8] H.T. Kung and M. Lam. Fault-tolerance and two-level pipelining in VLSI systolic arrays. In *Proceedings of the Conference on Advanced Research in VLSI*, MIT, Cambridge, MA, January 1984.

[9] J.W. de Bakker and D.S. Scott. A theory of programs. *Seminar on Programming Theory*, Vienna, 1969.

[10] G. Kahn and D.B. MacQueen. Coroutines and networks of parallel processes. In *Information Processing 1977*. North Holland, Amsterdam, 1977, pp. 993–998.

[11] M. Lam and J. Mostow. A transformational model of VLSI systolic design. In *Proceedings of the 6th International Symposium on Computer Hardware Description Languages and their Applications*, T. Uehara and M. Barbacci (eds.). IFIP, Pittsburgh, PA, May 1983.

[12] Stephen N. Cole. Real-time computation by n-dimensional iterative arrays of finite-state machines. *IEEE Transactions on Computers*, C-18:349–365 (April 1969).

[13] H.T. Kung and Charles E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings 1978*, I.S. Duff and G.W Stewart (eds.). Society for Industrial and Applied Mathematics, 1978, Philadelphia, PA, pp. 256–282.