CMU-CS-84-145

REASONING ABOUT SYNCHRONOUS SYSTEMS

Stephen D. Brookes
Department of Computer Science
Carnegie-Mellon University
Pittsburgh
Pennsylvania 15213
March 1984

The research reported in this paper was supported in part by funds from the Computer Science Department of Carnegie-Mellon University, and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. The views and conclusions contained in it are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

0. Abstract.

In this paper we describe a simple semantic model for synchronous systems of processes, suitable for high level functional descriptions of VLSI designs, and use it to justify formal reasoning about the behaviour of systems. Treating a system as a directed graph in which the nodes represent computational units and the arcs indicate the communication links and time dependencies, we define the outputs of each node in a system as a function of its inputs. The inputs and outputs are regarded as data-valued functions of time. Basically the idea is to specify the semantics of a system as a set of (mutually recursive) function definitions. This set of function definitions amounts to a *fixed point equation* whose solution is the desired semantics of the system. A solution to these equations is guaranteed to exist under reasonable assumptions about the type of computational unit used in the system. For particularly regular or simple systems, the solutions will be explicitly determinable by standard methods such as substitution. However, even in cases when this is not possible, the solutions can be found by elementary fixed point techniques. This model of systems allows extremely easy and elegant proofs of some interesting results on retimings and other system transformations such as pipelining. A retiming is a transformation of the communication graph of a system which preserves the underlying graph but alters the internode delays in a uniform manner; retiming has a simple effect on the semantics of a system. These results were first obtained by Leiserson and Saxe [12], but only under certain assumptions on the underlying communication graphs of systems. We show that these assumptions are unnecessary. Moreover, these authors were able only to give a long and somewhat complicated proof, because of their choice of semantic model. The fixed point method also serves as mathematical basis for an "algebraic" approach to VLSI design, such as the one described by Kung and Lin [9]. Again, however, their results were obtained under certain assumptions ("well-definedness") which we show to be unnecessary. We demonstrate the use of our semantic model to develop and justify a design for a palindrome recogniser, beginning from a mathematical description of the problem.

1. Introduction.

The rapidly increasing use of VLSI technology and the massive potential offered by VLSI for parallel computation are well known. If we are to rely on VLSI systems for more and more applications, it is obviously important that we understand the intended behaviour of such systems, and develop methods of specifying and proving correctness properties. Establishing the correctness of large systems of synchronously operating chips, which may communicate by quite complex links, is generally regarded as a difficult problem. It is not clear how to provide a tractable semantic description for general VLSI designs. Ideally, we would like a semantic treatment which facilitates such tasks as specification, verification and design of correct systems, and which is mathematically manageable.

In this paper we show that a large class of VLSI systems, including the so-called "synchronous" or clocked systems [9,12], can be treated in a uniform and elegant way using elementary ideas of fixed point theory. Similar ideas on semantics have been applied in the work of Chen and Mead [2,3] and Cohen [4]. Gordon [5,6] also uses a functional model for VLSI systems. Our focus is on the use of fixed point semantics in the justification of formal treatments of the behaviour of synchronous systems, in the analysis of system transformations, and in the specification and design of systems. Fixed point theory has been well established for some time now and is widely used as the basis for formal semantic methods [15]. We are not claiming to have discovered a new method of giving semantics to programming languages. Nevertheless, we maintain that using a fixed point approach gives rise to a natural, elegant, and tractable mathematical model of complex synchronous systems and enables us to perform rigorous analysis of the behaviour of such systems as well as to derive or construct systems conforming to formal specifications. We will show that our methods, an application of well known ideas from another area, yield potential benefits in terms of clarity and tractability.

We model a VLSI system as a directed graph whose nodes represent the computational units and whose arcs represent the communication lines of the system. Each outgoing arc or edge has an associated output function which describes the output values transmitted along that edge by the node, as a function of the inputs to the node. All outputs and inputs are regarded as data-valued functions of time. Arcs are labelled by integers (usually, but not necessarily, non-negative) indicating a propagation delay. Given such a graph, one can write down a set of equations, in general mutually recursive, describing the dependencies between the functions computed at each node. This set of equations amounts to a fixed point definition of a set of functions, and the semantic functions describing the outputs of the nodes of the system should be a solution of this equation. Under natural (and realistic) assumptions about the functional behaviour of the nodes, we can show very easily that these equations will indeed have a solution. In the case of a regular system, by which we mean a system with a particularly regular or simple geometric structure, it will often be possible to solve the functional equations explicitly, typically by substitution. However, using standard results about fixed points we can explicitly find the solution, even when the equations are not solvable by substitution. The mathematics is quite straightforward and fully explained.

An advantage of this approach is that it is well suited to reasoning about *hierarchical* systems and facilitates a *modular* approach to system design and verification. We can replace any part of a system by another whose semantics is the same, without affecting the semantics of the whole system. This fact will be justified by an elementary property of sets of functional equations; indeed, there are elementary results about the solutions to fixed point equations that will guarantee the correctness of these transformations.

A simple class of *retiming* transformations which affect the communication graphs of systems were first suggested by Leiserson and Saxe [12]. The semantic effects of retimings are simple, but the proofs of these properties in [12] required extra assumptions about the retimings and graphs, and were overly complicated. This was largely because of their choice of semantic model. We show that an elegant and simple proof goes through in the more general fixed point setting, where no constraints need to be placed on the retimings and systems considered. Thus we achieve a simplification and a generalisation of the earlier results.

Retimings affect only the communication structure of a graph, leaving the underlying functional units the same. The retiming results therefore enable us only to compare the semantics of systems whose underlying graph and functional elements are identical. Since our methods also allow replacement of any subgraph by a semantically equivalent structure, we can effectively reason about systems whose underlying structure and combinational behaviour are different.

Our methods also serve to justify and extend the "algebraic" approach to VLSI system design advocated in [9]. We illustrate the use of our methods with an example of a complex VLSI design for recognising palindromes, similar to the design given in [12] (where, however, no correctness proof was supplied). Beginning with a mathematical description of the problem, we derive a correct design which meets the mathematical specification; the derivation uses both retiming and pipelining. The production of a high-level VLSI design is, in effect, accompanied by a correctness proof.

We expect to be able to apply our methods to other interesting problems in VLSI design and verification, such as the polynomial GCD algorithm of Brent and Kung [1] and various matrix applications such as matrix multiplication and LU decomposition [7].

2. Modelling VLSI systems.

For the purposes of formal analysis, a high-level description of a large VLSI system or circuit, or in general any synchronous system, can be regarded as a graph structure whose nodes represent functional or computational elements and whose arcs represent communication links. In order to model the effect of delay along communication paths we can assign integer weights to the arcs. Each arc also has an associated function symbol representing the output transmitted along the arc. We may thus model a system S as an edge-weighted graph (V, E, F). The vertices $v \in V$ of the graph represent functional elements, and the edges $e \in E$ correspond to connections between the functional elements, and each edge has an associated delay and function symbol. F is the set of function symbols used in the graph. We represent a typical edge, from source node u to destination v, with delay d and function symbol f in the diagrammatic form

$$u \xrightarrow[f]{d} v.$$

We may also write $f{:}(u, d, v)$ to describe such an edge. Note that there may be several edges between a pair of nodes, and each edge has a direction. We assume the usual graph-theoretic notions of path, cycle, and so on.

The *communication graph* (V, E) of a system $S = (V, E, F)$ is obtained by ignoring the function symbols. This specifies the data-flow paths of the system and their associated propagation delays. A path leading from node v_i to node v_j with total edge-weight w indicates that the output computed at node v_j at any time depends on the value output by node v_i at the w^{th} previous time step; note that the length of a time step is not specified, and that the delay corresponds to the number of time steps required to have elapsed during transmission of a value from one node to another. Since we are thinking about synchronous or *clocked* systems, the delay corresponds to the number of elapsed clock cycles.

A *cyclic* path in a graph indicates a dependence of the output computed at each node in the path on its own (presumably) earlier output; this may result in the system exhibiting feedback or race conditions, but we see no reason to exclude such systems from our analysis. Obviously in any realistic system it is unreasonable to require that an output value at some time should depend on a *future* output; thus, in most real systems (and all of our examples) the delays on arcs will be non-negative. In the interests of generality, however, we do not make this assumption about the systems we consider. Indeed, in some applications it may be profitable to think of a negative delay on an arc as equivalent to a positive delay in the reverse direction.

In order to specify precisely the semantics of a system we must provide an interpretation for the functional elements. In other words, we must provide a rule defining each output function of a node in terms of the inputs to that node. In general, however, we may think of an uninterpreted system as a schema which can be instantiated in many distinct ways by choosing appropriate interpretations for the nodes.

We will be especially interested in transformations on communication graphs which have a simple effect on the semantics of a system, regardless of the particular interpretation we place on the output functions, and thus, regardless of the particular choice of computational unit at the nodes. The retimings of Leiserson and Saxe fall into this category of transformations, and so does the pipelining operation.

Before launching into the general foundations of our approach, an example will serve to prepare the ground.

3. An Example: Finite Impulse Response.

As an example, the following system for computing the Finite Impulse Response function (FIR) appears in [9]. The FIR problem is to compute the terms of the sequence y_n, where these terms are given by a recurrence relation using a known sequence x_n :

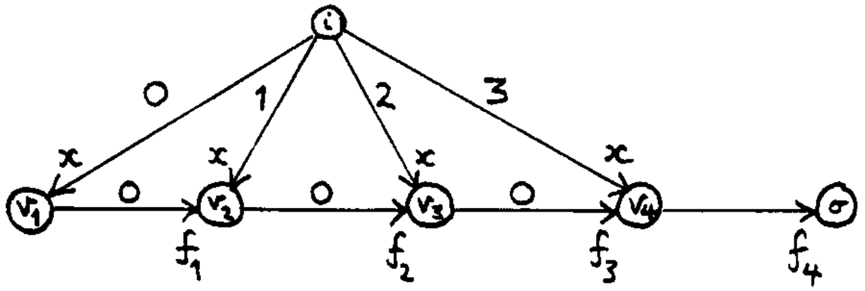
$$y_n = w_1 x_n + w_2 x_{n-1} + w_3 x_{n-2} + w_4 x_{n-3}, \qquad n \geq 3.$$

The w_i ($i = 1, \ldots, 4$), are integer weights and we assume that the x_n are integers. Thus, the problem is to compute weighted sums.

To implement this we can use a simple synchronous system whose structure is suggested directly by the mathematical formulation of the problem. We use a graph structure

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

Pictorially, this is represented in the following diagram:



Notice, for instance, that the output of node i is x, and one of the inputs to node v_2 is x delayed by one clock cycle.

The semantic description of this schema will be given as a set of functional equations. Since we are modelling time as a sequence of discrete steps, we use the set N of natural numbers to represent time. In order to keep in mind the distinction between integer times and integer data values, we use a distinct symbol $T = N$ for time, and let t range over T. We must provide a defining equation for each output function in terms of the corresponding inputs. For the node i, which has no incoming edges and represents an input to the system from outside, we will not specify the output except to require it to be an integer-valued function of time. The other outputs are built up in the obvious manner suggested by the recurrence relation. In describing the semantics we will assign to each node a computable operation on input values. Thus, in general, since in this system the nodes are computing integer values, a node such as v_2 with two inputs will correspond to a function $F_2 : N \times N \rightarrow N$.

Since the system is supposed to compute weighted sums, we choose the defining equations to be:

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

The desired functionalities are

$$f_i : T \rightarrow N, \qquad i = 1,2,3,4.$$

Note that the existence of functions f_i satisfying these equations and such that $f_i(t)$ is defined for all $t \geq i-1$ is easy (but tedious) to establish directly, if we are prepared to use an inductive argument on t. It is likewise possible to verify that the output y satisfies the equation:

$$y(t) = w_1 x(t) + w_2 x(t-1) + w_3 x(t-2) + w_4 x(t-3), \qquad t \geq 3.$$

Thus, if the terms of a sequence x_n are input along x, so that $x(t) = x_t$ for all t, the output $y_t = y(t)$ satisfies the recurrence relation above. The reason that $f_i(t)$ is not defined by the above equations when $t < i-1$ is that we are thinking of time as beginning at zero, immediately before the first clock cycle: the input x is not defined at any earlier time than 0. Thus we can see that the equations specify a set of *partial* functions f_i rather than total functions. The recurrence relations which defined the FIR problem do not specify the values of y_0, y_1 and y_2 and correspondingly the function f_4 is not specified by the functional equations at time $0, 1$ or 2. Alternatively, we can begin with a set of "initial conditions", specifying the values of the f_i at times earlier than $i-1$. Again it is clear that, for any set of initial conditions there is a solution to the above equations which agrees with the initial conditions. Note our slightly unusual usage here: the initial conditions do not correspond to time 0, but rather specify the outputs and inputs up to some moment in time.

All of the above informal justification for the existence of solutions can be established, as we noted, by a somewhat tedious inductive argument. We can, however, be more succinct. We can eliminate t from the defining equations by rewriting, in terms of the previously implicit F_i functions:

$$\begin{aligned} f_1 &= F_1(x) \\ f_i &= F_i(x \circ Z^{i-1}, f_{i-1}), \qquad i = 2,3,4 \\ y &= f_4, \end{aligned}$$

where the functionalities are:

$$\begin{aligned} F_1 &: (T \to N) \to (T \to N) \\ F_i &: (T \to N)^2 \to (T \to N) \qquad i = 2,3,4 \\ Z &: T \to T, \end{aligned}$$

and where we define

$$\begin{aligned} F_1(g) &= \lambda t.w_1 g(t) \\ F_i(g,h) &= \lambda t.(w_i g(t) + h(t)) \qquad i = 2,3,4 \\ Z(t) &= (t < 1 \to \perp, t-1). \end{aligned}$$

Here we have introduced the λ-notation; for example, the definition of F_1 above simply means that for all functions g and all times t, we have $(F_1(g))(t) = w_1 g(t)$. We use the notation $(p \to q, r)$ as a conditional expression. The function Z is a *delay* operator, similar to the one also used in [9,16]. Note in particular that for all $i \geq 0$

$$Z^i(t) = (t < i \to \perp, t-i).$$

By introducing the special symbol $\perp$ (pronounced "bottom") to represent an undefined value, we can write expressions explicitly denoting partial functions of time. If $f(t_0) = \perp$ we say that f is undefined at time t_0, or, equivalently, that f produces no result at time t_0. To be completely rigorous, we should also define the F_i to be *strict* in the sense that we want $F_i(g,h)(t)$ to be undefined when either $g(t)$ or $h(t)$ is undefined. With these implicit assumptions, it is clear that these equations define the same functions f_i as above. It is important to realise that we are using $\perp$ only as a convenient notation so that we can write expressions denoting partial functions; we are *not* thinking of $\perp$ as a data value to be passed around inside a system.

The advantage to be gained from the new formulation of the semantic equations is that we have effectively hidden the time parameter and the equations in this form are susceptible to algebraic manipulation without our having to keep track of the time dependencies. Indeed, we can explicitly solve the equations for f_i in this case to get, by substitution, the following closed forms for the functions:

$$\begin{aligned}
f_1 &= F_1(x)\\
f_2 &= F_2(x \circ Z^1, F_1(x))\\
f_3 &= F_3(x \circ Z^2, F_2(x \circ Z^1, F_1(x)))\\
f_4 &= F_4(x \circ Z^3, F_3(x \circ Z^2, F_2(x \circ Z^1, F_1(x)))).
\end{aligned}$$

In this form it is easy to see that provided x is a total function of time, so that $x \circ Z^i$ is defined from time i, then each f_i is defined from time $i-1$, as required. The point is that this fact emerges obviously from an elementary algebraic manipulation of the functional equations.

This example may not be a very convincing demonstration that there is much to be gained by using a higher-order formulation of the functional equations. Indeed, this particular example can be treated from first principles and the solutions to the functional equations can be found explicitly by substitution. However, in general this need not be the case. Many published VLSI designs do not at first sight appear to fall into the category of systems for which the solutions can be found by elementary means. In most cases proofs of well-definedness can be extremely tedious, if not difficult, because there is generally a lot of book-keeping to do in order to show that the outputs are transmitted around the system in such a way that they are always at the right place at the right time. Indeed, this last property can be quite difficult to formalise and prove, mixed in as it tends to be with concerns that the individual parts of the system are transforming their inputs correctly. In cases where the functions being computed by the system are given by equations which do not obviously have solutions, we would like to have a guarantee first of all that there *is* a solution and secondly a method of reasoning about the solution. If a function satisfying these equations cannot be guaranteed to exist it does not make sense to transform or manipulate the equations or to use the "solution" function to "prove" properties of the system in which the function appears.

In the next section we offer a rigorous semantic treatment of arbitrary synchronous systems. Essentially we regard a system as defining a set of fixed point equations whose solutions are the output functions computed by the nodes of the system. As long as each node is performing *computable* operations on its inputs, an eminently reasonable constraint which any realistic VLSI system is guaranteed to meet by its very definition, the standard fixed point theory applies here and tells us that the desired equations have a solution (technically, a *least fixed point*). Moreover, the solution can always be found (if all else fails) by a simple iterative method. This general material here is well known, but we give a full treatment in order to show how elegantly it can be applied to synchronous VLSI systems. The reader who is already familiar with the concepts of fixed point theory may want to skip this section.

4. Fixed point semantics.

The general system $S = (V, E, F)$ is described by a set of equations for the function symbols in F. Each function symbol appearing on an output arc of some node in V must have a defining equation, and in the definition the right-hand side can refer to the inputs to the node. A typical equation has the form

$$f_i = \Phi_i(f_1, \ldots, f_n)$$

where each $f_j \in F$. In our examples of systems the Φ_i will have the form

$$\Phi_i(f_1, \ldots, f_n) = F_i(f_1 \circ Z^{w_{i1}}, \ldots, f_n \circ Z^{w_{in}})$$

but this is not necessary for our treatment to work. Of course, some of the f_j will not appear explicitly on the right-hand side of an equation, since the outputs of a node only depend explicitly in these equations on f_j if f_j is an input function to that node.

Associated with these equations also are intended functionalities for the f_i, and for the Φ_i. Modelling time as a sequence of discrete clock steps, we represent time by the set T of natural numbers. Each f_i is intended to be a (partial) function from time into some set V_i of values, so we have

$$f_i : D_i = (T \rightarrow V_i).$$

We will use $\perp$ to represent an undefined value, so that for instance the totally undefined function is denoted $(\lambda t.\perp)$ The set V_i will depend, of course, on the problem which the system is to solve; as will the Φ_i. Each Φ_i is applied to a tuple of functions and produces a function:

$$\Phi_i : D_1 \times \ldots \times D_n \rightarrow D_i.$$

For any set V, the set $(T \rightarrow V)$ of partial functions is partially ordered by the relation

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

Using the symbol $\subseteq$ here should give rise to no confusion, because this relation coincides with the usual set-theoretical inclusion relation between the graphs of the functions. Under this ordering, the total functions are maximal. It is easy to check that every chain of functions $f_m \subseteq f_{m+1}$ converges to a limit function defined by

$$f = \bigcup_{m=0}^{\infty} f_m.$$

The functional equations for the f_i are intended to be specifications for the output functions computed by the nodes of the system. To be precise, we regard these equations as *defining* these output functions to be the *least defined* functions which satisfy them. We are now in a position to prove that such functions always exist.

We can define a sequence of partial functions $f_i^{(m)}$, $m \geq 0$, for each i, beginning in each case with the totally undefined function and at stage $m+1$ substituting in the right-hand side of the equation for f_i to get $f_i^{(m+1)}$. Intuitively, the m^{th} function $f_i^{(m)}$ represents all the information about the function f_i that can be obtained by expanding its definition m times. Thus, for each i we let

$$\begin{aligned} f_i^{(0)} &= \lambda t.\bot \\ f_i^{(m+1)} &= \Phi_i(f_1^{(m)}, \ldots f_n^{(m)}), \qquad m \geq 0. \end{aligned}$$

If we know that the Φ_i are *monotone, i.e.* ,

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

so that replacing any argument of Φ_i by a better-defined function produces a better-defined result function, then for each i the functions in the sequence $f_i^{(m)}$ become better defined as m gets larger:

$$f_i^{(m)} \subseteq f_i^{(m+1)}.$$

In other words, each function in the sequence agrees with all later functions on arguments at which it has a proper value, but later functions may be defined at more arguments. Thus for each i the sequence $f_i^{(m)}$ forms a *chain* of partial functions and converges to a limit function with definition

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

If we know that the operations Φ_i are *continuous* (so that they preserve limits of chains), we can show that the f_i defined as above satisfy the desired equations. Moreover, these f_i are the *least defined* functions which have this property: each f_i is only defined at those values of t at which the equations explicitly require it.

Note that the above argument can be modified to deal with the case when the iterations begin not with the totally undefined function but with a function representing the initial conditions. If we want to specify that the output function f_i agrees with a known function (k_i, say) for the first d time steps, we simply replace the above definition of the first term in the sequence of approximations by $f_i^{(0)} = \lambda t.(t < d \rightarrow k_i(t), \bot)$. Of course, it is necessary to check that this agreement property is *preserved* by the operations Φ_i, so that each approximation $f_i^{(m)}$ to the output function also agrees with k_i for the first d time steps. This will usually be a trivial condition to check.

The requirement that the Φ_i be continuous (and monotone) is a natural one that is met by any reasonable system in which the nodes are supposed to be performing *computable* operations on their inputs. We are representing the inputs and outputs as values in a flat domain obtained from a set V by adjoining a bottom element $\perp$.It is a standard argument that all computable functions on a flat domain are continuous. An operation $\Phi_i(f_1, \ldots, f_n)$ is, by definition, computable iff for all t in order to compute $\Phi_i(f_1, \ldots, f_n)(t)$ we need only compute a finite number of well-determined arguments (the precise number depending in general on t). More precisely, for all $f_1, \ldots, f_n$ and each $t \in N$, we require for computability that the value of $\Phi_i(f_1, \ldots, f_n)(t)$ depends on some set of values $f_1(t'), \ldots, f_n(t')$, for t' ranging over some finite set A. Although this does not look like the limit-preserving property we have associated with continuity, it is easily shown to be equivalent. The connection with continuity is possible because in the partially ordered set of functions $(T \to V)$ every function f is the union (limit) of the finite functions (those with finite graph) included in it. If an operation is computable, it follows that for each t the value $\Phi_i(f_1, \ldots, f_n)(t)$ can also be obtained by replacing the f_i by finite functions. For more details the reader is referred to [13,14,15].

It is now straightforward to check that, for instance, Z is a continuous operator. The composition operator $\circ$ on functions is also continuous. Continuous operations are closed under composition, so whenever we have a system whose semantic equations have the form

$$f_i = F_i(f_1 \circ Z^{w_{i1}}, \ldots, f_n \circ Z^{w_{in}})$$

we need only to verify that F_i is continuous in order to conclude that the fixed point methods are applicable. This is usually very straightforward.

Finite Impulse Response Example (revisited).

Let us now examine again the example of the previous section, this time in the context of our semantic definitions. It is trivial to verify that the operators F_i used here are continuous.

The semantic equations for the system are

$$\begin{aligned} f_1 &= F_1(x) \\ f_i &= F_i(x \circ Z^{i-1}, f_{i-1}) \qquad i = 2, 3, 4. \end{aligned}$$

The approximations to the solutions are thus

$$\begin{aligned} f_i^{(0)} &= \lambda t.\perp \qquad i = 1, 2, 3, 4 \\ f_1^{(m+1)} &= F_1(x) \\ f_i^{(m+1)} &= F_i(x \circ Z^{i-1}, f_{i-1}^{(m)}) \qquad i = 2, 3, 4. \end{aligned}$$

Trivially, we have $f_1^{(m)} = F_1(x)$ for all $m \geq 1$, so that the limit f_1 is indeed $F_1(x)$. Similarly, we see that

$$f_2^{(m)} = F_2(x \circ Z^1, f_1^{(m-1)}) = F_2(x \circ Z^1, F_1(x)), \qquad \text{for } m \geq 2.$$

The analyses for f_3 and f_4 are as easy. Thus for each i the approximations converge (as expected) to the functions derived earlier as closed forms for the f_i.

5. Transformations.

Retimings.

Now we will examine in our semantic framework the results obtained by Saxe and Leiserson on retiming synchronous systems. We will see that their results, obtained for a restricted class of system, generalise very easily and are provable by elementary arguments.

A retiming is a transformation on the communication graph of a system. Typically retiming is employed in order to optimise a system. As has been suggested by Kung et al., [7,9], it is particularly desirable to produce a system whose communication graph is *systolic:* all delays on internal arcs must have delay at least one. Systems satisfying the systolic condition exhibit the property that their clock cycle time need only be as long as the time taken by an *individual* node in the system to perform its operation, since there is no rippling through of inputs and outputs and each node requires at any time only inputs which have already been produced by a previous clock cycle. Retimings can often be used to achieve the systolic property.

Two particular and natural classes of retiming are introduced in [12]. The first type can be thought of as *node-delaying* and the second as *slowing*. In a node-delaying transformation, as formulated in [12], we assign an integer *lag* d_i to each node of the system and replace the edge weights in a uniform manner. An edge (v_i, w_{ij}, v_j) of the old system becomes $(v_i, w_{ij} + d_j - d_i, v_j)$. Intuitively, this transformation adds an extra lag delay to each node and effectively delays the outputs of that node by an extra fixed amount. We will see that this property can be formalised and proved very easily. The other type of transformation, slowing, multiplies all edge weights by the same integer; if the integer is k the new system is said to be a k-slowed version of the old. Again intuitively, this transformation has the effect of producing a system which outputs the same results as the old one but with a delay of k between successive results. Again this can be proved easily in our framework. Leiserson and Saxe discussed these forms of retiming in [12], and established a graph-theoretic condition on a system which guarantees that an "equivalent" systolic system can be found directly. Our methods will provide a rigorous basis for their results.

(i) Node-delaying.

Begin with a system $S = (V, E, F)$. Consider first a node-delaying transformation characterised by a mapping $d : V \to N$. The intuitive idea behind such a transformation is to introduce an additional *lag* $d_i = d(v_i)$ at each vertex v_i. An edge (v_i, w_{ij}, v_j) will be replaced by $(v_i, w_{ij} + d_j - d_i, v_j)$. A typical functional equation for the system S, say

$$f_i = F_i(f_1 \circ Z^{w_{i1}}, \ldots, f_n \circ Z^{w_{in}})$$

will correspond to the equation

$$g_i = F_i(g_1 \circ Z^{u_{i1}}, \ldots, g_n \circ Z^{u_{in}})$$

in the new system, where we have renamed the function symbols of the new system to distinguish them from the old ones, and where

$$u_{ij} = w_{ij} + d_j - d_i.$$

We claim that the following relations hold for each i :

$$g_i \circ Z^{d_i} = f_i,$$

i.e. , for all times t, $g_i(t) = f_i(t - d_i)$. In other words, the new system computes the same values as the old system, but each node lags behind by the appropriate number of time steps.

The proof is straightforward, relying only on the continuity of $\circ$ and the following property of the F_i : for all functions $h_1, \ldots, h_n$,

$$F_i(h_1, \ldots h_n) \circ Z = F_i(h_1 \circ Z, \ldots h_n \circ Z).$$

This will certainly be the case when the F_i simply apply their arguments to time and combine the results, *i.e.* when there is an associated function ϕ_i such that for all t

$$F_i(h_1, \ldots, h_n)(t) = \phi_i(h_1(t), \ldots, h_n(t)),$$

as is the case in our VLSI applications. This condition is therefore *sufficient* for the delay-distributing property to hold, but it is not *necessary*.

It is possible to use *fixed point induction rules* (see [14,15] for example) to prove this type of retiming property. However, to indicate how easily the arguments proceed in our application we give a standard inductive proof. The proof is by induction on m that for each pair of approximations $f_i^{(m)}$ and $g_i^{(m)}$ we have

$$g_i^{(m)} \circ Z_i^{d_i} = f_i^{(m)}.$$

The base case is simple, using elementary properties of functions:

$$\begin{aligned} g_i^{(0)} \circ Z^{d_i} &= (\lambda t.\bot) \circ Z^{d_i} \\ &= \lambda t.(t < d_i \rightarrow \bot, \bot) \\ &= \lambda t.\bot \\ &= f_i^{(0)}. \end{aligned}$$

For the inductive step, assume true for m and use the definitions:

$$\begin{aligned} g_i^{(m+1)} &= F_i(g_1^{(m)} \circ Z^{u_{i1}}, \ldots, g_n^{(m)} \circ Z^{u_{in}}) \\ &= F_i(g_1^{(m)} \circ Z^{d_1+w_{i1}-d_i}, \ldots, g_n^{(m)} \circ Z^{d_n+w_{in}-d_i}) \end{aligned}$$

Since F_i distributes over Z, we get

$$\begin{aligned} g_i^{(m+1)} \circ Z^{d_i} &= F_i(g_1^{(m)} \circ Z^{d_1} \circ Z^{w_{i1}}, \ldots, g_n^{(m)} \circ Z^{d_n} \circ Z^{w_{in}}) \\ &= F_i(f_1^{(m)} \circ Z^{w_{i1}}, \ldots, f_n^{(m)} \circ Z^{w_{in}}) \\ &= f_i^{(m+1)} \end{aligned}$$

and the result holds for $m+1$. That completes the inductive proof. Since $\circ$ is a continuous operation we can deduce that the limits f_i and g_i satisfy the desired relation

$$g_i \circ Z^{d_i} = f_i.$$

(ii) Slowing.

The k-slowed version of a system G is obtained by multiplying all edge weights of G by k. An edge of the form (u, d, v) becomes (u, kd, v). Thus, an equation

$$f_i = F_i(f_1 \circ Z^{w_{i1}}, \ldots, f_n \circ Z^{w_{in}})$$

corresponds to the equation

$$g_i = F_i(g_1 \circ Z^{kw_{i1}}, \ldots, g_n \circ Z^{kw_{in}}),$$

where again we rename the functions in the new system. Intuitively, the behaviour of this k-slowed design is that of the original, except that it takes its inputs and computes its results on every k^{th} clock cycle. We claim, therefore, that

$$g_i \circ \overline{k} = f_i,$$

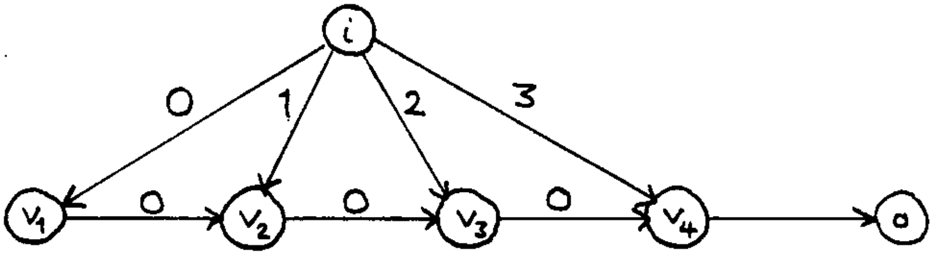
where $\overline{k} = \lambda t.kt$ is the obvious time-slowing function. In other words, for all t, $g_i(kt) = f_i(t)$. Again the proof is straightforward. It relies on the the identity

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

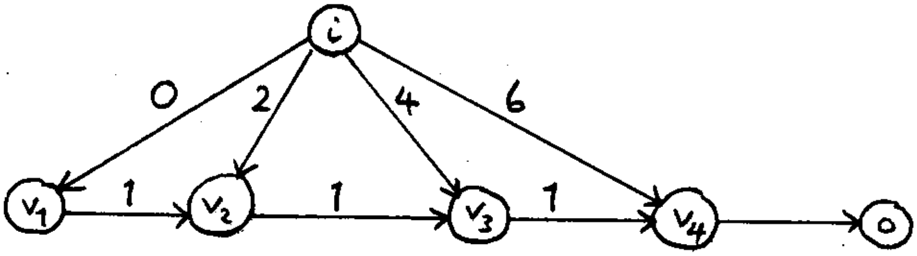
which obviously holds when F_i simply applies its arguments to time.

Example: Retiming in the FIR System.

To illustrate these retiming ideas, consider again the communication graph for the Finite Impulse Response problem:



If we retime by introducing lags 0 at the input node i, $k-1$ at v_k $(k = 1, 2, 3, 4)$, we get the following new communication graph:



Distinguishing the new output functions from the old by decorating them, this graph has the following semantic description:

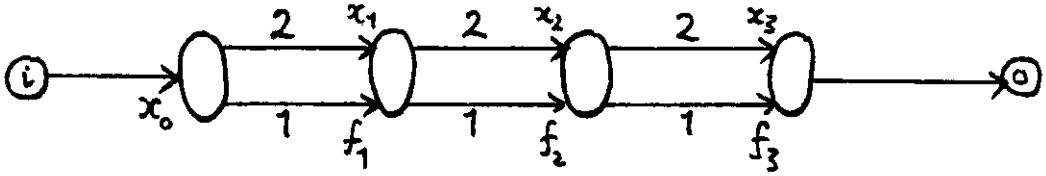
$$f_1{}' = F_1(x)$$
$$f_i{}' = F_i(x \circ Z^{2(i-1)}, f_{i-1}{}' \circ Z) \qquad i = 2, 3, 4.$$

Either by explicitly solving, or by the retiming results above, we see that

$$f_i{}' = f_i \circ Z^{i-1} \qquad i = 1, 2, 3, 4,$$

corresponding with the lags introduced at the nodes.

Pipelining. It is also easy to justify a further operation on this retimed system: *pipelining* the input x. Since this input is needed in several nodes of the system at different times, we can achieve this by sending the inputs to these nodes in sequence, with the appropriate inter-node delays. Formally, we produce a system with structure and semantics:



$$\begin{aligned} &x_0 = x \\ &f_1 = F_1(x_0) \\ &x_i = x_{i-1} \circ Z^2 \qquad i = 1, 2, 3 \\ &f_i = F_i(x_{i-1}, f_{i-1} \circ Z), \quad i = 2, 3, 4. \end{aligned}$$

It is obvious that the x_i satisfy the equations $x_i = x \circ Z^{2i}$. This is another *systolic* system computing the Finite Impulse Response function.

Justification of pipelining.

In this example, we replaced an output function which was replicated on several arcs by an "equivalent" set of output functions. To be specific, the effect on the semantic description was to introduce extra function symbols x_i defined in terms of the old x. Since the equations specifying the f_i in the two systems can be made identical by a simple substitution, it is obvious that the set of functions defined in each case is identical: the f_i computed by the two systems are the same.

6. Another Example: Palindromes.

In this section we apply the methods of the previous sections to a problem drawn from the literature [12]. Leiserson and Saxe described a systolic array for recognizing palindromes. The successive characters of a string 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 whether or not the characters input so far (up to time t) form a palindromic string. We show how a design for such an array can be derived directly from the mathematical specification of the problem, and as a consequence we are able to verify formally (and straightforwardly) a variant of the solution published in [12].

First of all, a string $x_0 x_1 \ldots x_n$ is a palindrome if and only if its first half is the reverse of its second half. Let us use the notation $x_{[i:j]}$ for the string $x_i x_{i+1} \ldots x_j$, when $i \leq j$; likewise we will write $x_{[j:i]}$ for the string $x_j x_{j-1} \ldots x_i$, when $j \geq i$. Then a string x of length $n+1$ is a palindrome if and only if $x_{[0:m]} = x_{[n:m+1]}$, where $m = \lfloor \frac{n}{2} \rfloor$. We want to design a system with input x and output p satisfying

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

From this definition it is clear that a palindrome recogniser can be built from two subsystems, one which inputs the sequence x and distributes the terms among outputs a_i, b_i so that

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

for $i \leq \lfloor \frac{t}{2} \rfloor$; and one with output p which tests two sequences of inputs a_i and b_i for equality:

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

To begin with the distribution of characters, observe that it is easy to implement the a_i by a simple pipeline. We want

$$a_i(t) = (t < 2i \rightarrow \bot, x(t-i)),$$

and this will be the case if we define:

$$\begin{aligned} a_0 &= x \\ a_{i+1} &= \lambda t.(t < 2i \rightarrow \bot, a_i(t-1)), \quad i \geq 0. \end{aligned}$$

Observe that the above specification for a_i and b_i does not define their values when $t < 2i$, but that we require

$$b_i(2i) = x(i) = a_i(2i)$$

and $b_i(t+1) = b_i(t)$ for $t \geq 2i$. Thus we can transform the functional definition of b_i :

$$\begin{aligned} b_i(t) &= (t < 2i \rightarrow \bot, x(i)) \\ &= (t < 2i \rightarrow \bot, t = 2i \rightarrow a_i(2i),\ b_i(t-1)) \\ &= (t < 2i \rightarrow \bot, t = 2i \rightarrow a_i(t),\ b_i(t-1)). \end{aligned}$$

The definition of b_i thus specifies that its output is undefined for $2i$ time steps and then becomes initialised to the output of a_i. It is easy to see that we can *implement* this behaviour by passing a *signal* to the node computing a_i and b_i which tells it when to do the initialisation. Thus, if we define

$$s_i(t) = (t < 2i \rightarrow \bot, t = 2i \rightarrow \text{true, false}),$$

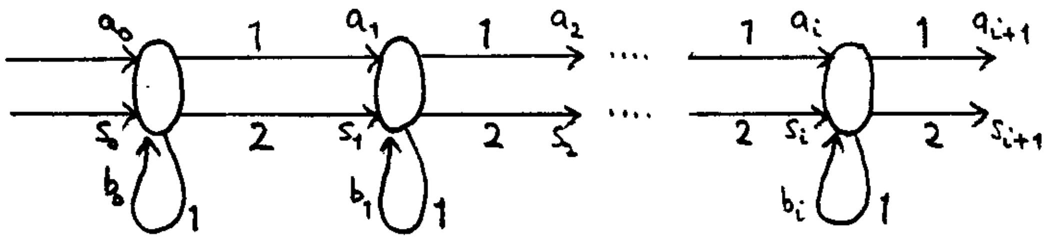
we can replace the above equations by:

$$\begin{aligned} a_{i+1}(t) &= (s_i(t) = \bot \rightarrow \bot, a_i(t-1)), \\ b_i(t) &= (s_i(t) \rightarrow a_i(t),\ b_i(t-1)). \end{aligned}$$

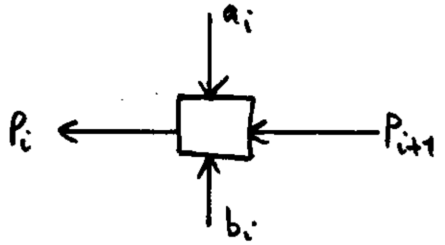
The signal s_i is undefined until time 2i, then becomes true for one time step and thereafter stays false. Clearly we can also implement this with a pipeline, setting:

$$\begin{aligned} s_0 &= \lambda t.(t = 0 \rightarrow \text{true, false}) \\ s_{i+1} &= s_i \circ Z^2 \qquad i \geq 0. \end{aligned}$$

Thus we can input a sequence of truth values on s_0 and pipeline them through the system in the obvious way. A communication graph for the a_i, b_i and s_i can be described by the following diagram:



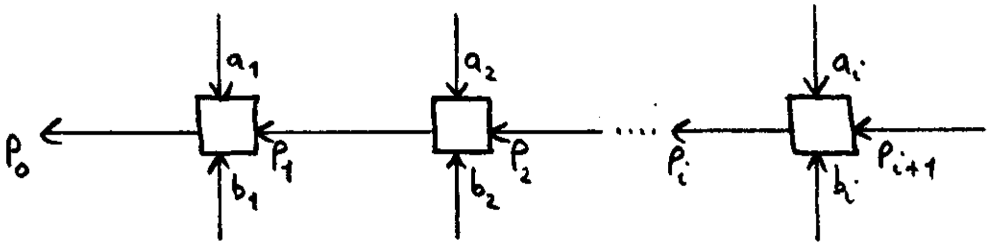
For the remaining part of the system, which is to test two sequences of inputs for termwise equality, we can clearly implement this as a sequence of connected single character comparators, in which the truth values ripple through accumulating character comparisons on the way. This diagram represents a single character comparator v_i with carry-in p_{i+1} and output p_i :



When this unit is activated, *i.e.* when both a_i and b_i contain characters of the input string, the output p_i should be

$$p_i(t) \;=\; (a_i(t) = b_i(t))\;\&\; p_{i+1}(t).$$

We need a system in which there are always $\lfloor \frac{t}{2} \rfloor$ active character comparators at time t. Equivalently, the i^{th} comparator must wait for $2i$ clock cycles before making any comparisons. Again we can use the signal functions s_i to implement this idling. It is convenient to set the default output of an inactive unit to "true." The following structure then describes a system which tests at time t two strings, each of length $\lfloor \frac{t}{2} \rfloor$, for equality.

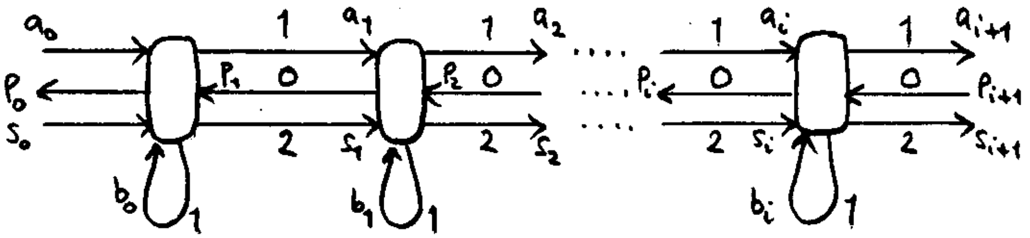


$$p_i(t) = (s_i(t) = \bot \rightarrow \text{ true, } (a_i(t) = b_i(t))\;\&\; p_{i+1}(t)), \quad i \geq 0.$$

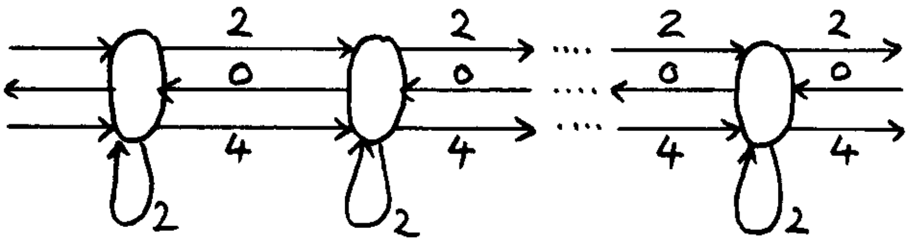
In fact, given the definitions of the s_i, we have

$$p_i(t) = (t < 2i \;\rightarrow\; \text{true, } (a_i(t) = b_i(t))\;\&\; p_{i+1}(t)).$$

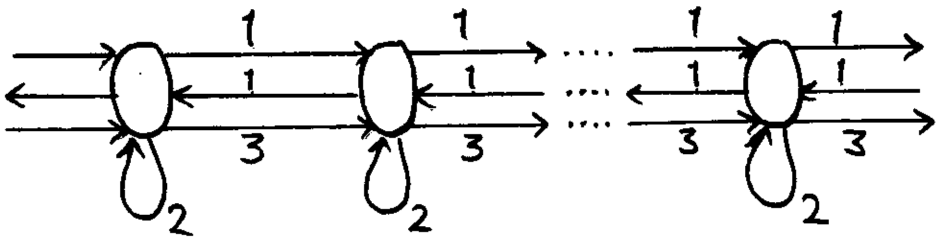
Note that the two subsystems are *compatible,* in the sense that the first system *defined* the outputs a_i, b_i and s_i, while the second system *inputs* these and uses them to define the p_i. This means that we may superimpose the two systems, forming a new system as above whose defining equations are simply obtained by combining those of the separate systems. If we combine the two parts of our system we produce the following design:



As it stands, this design is not systolic; the computation to be performed at time t by the leftmost node must wait until the result of the rightmost node at that same time step has filtered or rippled through. This corresponds to a path of total delay zero. It is not possible to find a simple node-delaying systolic retiming for this design, because the communication graph contains cycles of length 2 but with total delay only 1. In any node-delaying operation on the graph the total delay on a cycle is preserved, so that in this case we cannot produce delays of at least 1 on the arcs of such cycles. We can, however, transform to get a systolic version if we first *slow* the entire system by a factor of 2. The slowed version is:



Now, if we introduce lags $-i$ at node i, we produce a systolic design:



The semantic definitions to go with this design are obtainable from those of the earlier design by applying the two retiming operations in the same way. The correctness of the final design is guaranteed because of the rather more obvious correctness of the initial design, based as it was on the mathematical description of the problem.

7. Conclusions.

We have described a mathematical semantics for synchronous VLSI systems, based on fixpoint equations. We showed how this semantics allows easy manipulation of VLSI systems and facilitates a variety of activities such as hierarchical decomposition (as also argued in [2] and [3]), modular analysis and synthesis of VLSI designs, and verification of correctness properties. The framework we have described supports extremely easy proofs of results on retiming of systems, and delineates precisely the conditions under which such transformations are valid. We made no assumptions in our proofs about the particular d_i chosen in a node-delaying operation or about the slowing factor k. Likewise we did not have to assume any structural properties of the communication graph. Contrast this with the results as given in [12], where the authors assumed that the communication graph had non-negative weights on all edges and that there were no cycles in the graph whose total weight was zero; in addition they assumed that the d_i in a node-delaying operation were chosen so that the new weights $w_{ij} + d_j - d_i$ are also non-negative. Of course, these restrictions are arguably natural ones to impose on a system; but the proofs go through in a more general setting, and are arguably more elegant than the proofs in [12].

The authors of [9] assumed a condition ("well-definedness") of their systems of equations which we find is unnecessary. Their condition amounts to saying that at each sufficiently late time t the outputs $f_i(t)$ are completely determined by the (global) inputs $x(0), \ldots, x(t)$ and outputs $f_i(0), \ldots, f_i(t-1)$. Because this definition refers to the global inputs to the system in which the individual outputs are being computed, it does not seem well suited to constructing larger systems from smaller ones by standard methods such as feeding outputs from one system into inputs of another. The problem is that one cannot check for well-definedness at a node by examining its immediate neighbors, since the definition involves the distinguished global input nodes. It seems that in any case the task of verifying that a system is well-defined requires us to find an explicit solution (in a particular form) to the functional equations; this may in general be very difficult. In contrast, we have shown that solutions do exist, but that it is not really necessary to find closed forms for the solutions in order to reason about a system. It is also evident that this notion of well-definedness is a special case of *continuity,* since it states that the outputs at any time can be computed from a particular (time-dependent) finite set of arguments. This is the sense in which we feel that our techniques provide a more uniform approach to the problem of reasoning about systems.

In addition to the use to which we have put our methods in this paper, we believe that they are just as readily applicable to more complicated problems in the literature, where so far very few correctness proofs have been produced. We are thinking in particular of systems in which the propagation delays may be highly dependent on the data flowing through the system. Such is the case, for example, with the polynomial GCD systolic array of Kung and Brent [1]. This particular algorithm seems at first sight to be complicated by the dependence of the delays on the coefficients of the two polynomials. However, we believe it will be possible to *derive* a correct GCD algorithm from a mathematical description of the problem, in much the same way as was done here for the palindrome problem. This and other more complicated examples will be the subject of future work. Our semantic model and our techniques are also of use in establishing the correctness of pipelining operations, and can be used to prove the correctness of the so-called "Cut Theorem" of [8], showing that it may be interesting to adapt fixed point semantics in a wider setting than that discussed here. The application of our methods to analysing fault tolerance behavior of systems is an interesting possibility for further research.

8. References.

[1] R. P. Brent and H. T. Kung, Systolic VLSI Arrays for Polynomial GCD Computation, CMU Technical Report CMU-CS-82-118 (March 1982).

[2] M. C. Chen, Doctoral Dissertation, Computer Science Department, California Institute of Technology (1983).

[3] M. C. Chen and C. A. Mead, A Hierarchical Simulator Based on Formal Semantics, Proc. Third Caltech Conference on VLSI, pp 207-223, Computer Science Press (March 1983).

[4] D. Cohen, Mathematical Approach to Computational Networks, Technical Report ISI/RR-78-73, University of Southern California, Information Sciences Institute (November 1978).

[5] M. J. C. Gordon, A Very Simple Model of Sequential Behaviour of nMOS, Dept of Computer Science Internal Report, University of Cambridge.

[6] M. J. C. Gordon, A Model of Register Transfer Systems with Applications to Microcode and VLSI Correctness, Department of Computer Science Internal Report CSR-82-81, University of Edinburgh (1981).

[7] H. T. Kung, Let's Design Algorithms for VLSI Systems, Proc. Conference on VLSI: Architecture, Design, Fabrication. California Institute of Technology, pp 65-90 (January 1979).

[8] H. T. Kung and M. Lam, Fault-Tolerance and Two-Level Pipelining in VLSI Systolic Arrays, Proc. Conference on Advanced Research in VLSI,MIT, January 1984.

[9] H. T. Kung and W. L. Lin, An Algebra for Systolic Computation, in: Elliptical Problem Solvers II, ed. Birkhoff, G., and Schoenstadt, A, pp 141-160, Academic Press (1984).

[10] H. T. Kung and C. E. Leiserson, Systolic Arrays (for VLSI), in: Duff, I.S., and Stewart, G.W., (eds), Sparse Matrix Proceedings 1978, Society of Industrial and Applied Mathematics (1979).

[11] M. Lam and J. Mostow, A Transformational Model of VLSI Systolic Design, in: Uehara, T., and Barbacci, M. (eds), Procedings of the 6th International Symposium on Computer Hardware Description Languages and their Applications, pp 65-77, IFIP (May 1983).

[12] C. E. Leiserson and J. B. Saxe, Optimizing Synchronous Systems, Proc. 22^{nd} Annual IEEE Symposium on Foundations of Computer Science, pp 23-36, IEEE Computer Society (October 1981); final version in: Journal of VLSI and Computer Systems, Vol. 1 no. 1, (1983).

[13] D. S. Scott, Some Ordered Sets in Computer Science, Proc. NATO Advanced Study Institute (September 1981).

[14] D. S. Scott, Notes on a Mathematical Theory of Computation, Oxford University, Programming Research Group, Technical Report.

[15] J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics, MIT Press.

[16] U. Weiser and A. Davis, A Wavefront Notational Tool for VLSI Array Design, in: Kung, H.T., Sproull, R.F., and Steele, G.L.,Jr.,(eds), VLSI Systems and Computations, pp 226-234, Computer Science Press (October 1981).