Hardware Description with Recursion Equations

by

John T. O'Donnell

Computer Science Department
Indiana University
Bloomington, Indiana 47405

TECHNICAL REPORT NO. 212

Hardware Description
With Recursion Equations

by

John T. O'Donnell

December, 1986

To appear in the *IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications,* April 1987.

Hardware Description with Recursion Equations

John T. O'Donnell
Computer Science Department
Indiana University
Bloomington, Indiana 47405

Using a stream to represent the full history of the sequence of values on a wire in a circuit makes it possible to treat components and their ports as first-class objects in a hardware description program. Therefore the program can directly operate on the systems of recursion equations that define the inputs and outputs of components. This provides the foundation for a simple, yet powerful, hardware description methodology that can be implemented in many existing programming languages simply by defining a library of basic functions. A hardware designer doesn't need to write equations for all the components in a circuit; it is often better to write higher order functions that generate the lexically nested systems of recursion equations that correspond to the levels of abstraction of the circuit. Furthermore, a programming language implementation can directly interpret the hardware description, using a set of function definitions for the primitive components in the hardware. The designer can extract alternative meanings from a single hardware description — such as a simulation function and a component/wiring list — simply by providing alternative sets of primitive function definitions. Generation of geometric layouts requires additional information, which can be specified by the designer or generated automatically. The methodology will work in any programming language that supports first class functions, recursive functions and data, and streams or closures, and it has been implemented in Daisy and used in several hardware designs.

1. Introduction

A circuit designer is largely concerned with two goals: first, to develop a design and ensure that it is correct; and second, to implement the design in a real piece of hardware. Any circuit design methodology and "computer hardware description language" (CHDL) must support both of these activities. This has lead to the idea of CHDLs that can describe both the behavior and the structure of a circuit, allowing the designer to work toward both goals within one language [3].

The original CHDL — and still one of the best — is the schematic diagram. Schematic diagrams contain a great amount of information, some of it explicit (what components are present, how they are connected), and some of it implicit (how many components of each type will be needed to construct the circuit, what the fanout of each output is, how the components might be placed on a chip or circuit board so as to simplify the wiring). Most importantly, a schematic diagram tells an experienced designer both the behavior of the circuit and its structure. As we move toward more sophisticated design languages, it is important to retain these advantages of schematics. In particular, it is important to describe as many aspects of the circuit as possible using one document, rather than allowing a complex description to become fragmented into many partial descriptions.

Most CHDLs that can describe both behavior and structure are based on conventional imperative programming languages, so they use language objects like variables, assignments and imperative control statements to describe objects in the circuit's behavior and structure. Unfortunately, this leads to subtle difficulties that must be overcome by adding new features to the underlying programming language. Therefore the CHDL becomes a completely new

language, requiring its own compiler and tools, and designing such a CHDL combines all the complexities of programming language design with those of circuit description.

A better alternative is to describe each component of a circuit with a programming language construct that correctly captures all the relevant aspects of that component. This leads to a much more elegant combination of behavioral and structural descriptions, and it has many practical advantages as well. The designer can carry out all aspects of the circuit description, simulation and implementation using just one general purpose language. This unification reduces errors and inconsistencies that creep in when a set of design languages and tools are used. Any existing programming language with the right set of primitive data structures and operations will support the design methodology, and a wide range of design tools can easily be added to the system, since the underlying language has general purpose computing facilities. Finally, this approach can be combined gracefully with architecture synthesis and hardware correctness proof techniques.

The remainder of this paper develops such a scheme, called "hardware description with recursion equations" (abbreviated HDRE and pronounced as hydra). A designer using HDRE may describe a circuit using a simple set of primitive functions written in an underlying general purpose programming language, and the description itself is just a function written in that language. Executing a circuit description function provides its *meaning* — its semantic content.

What is the meaning of a circuit description? While the designer is pursuing the first goal — developing the design and ensuring that it does what it is supposed to — the meaning of a circuit description is the *behavior* that the corresponding hardware circuit will have. Thus the meaning of the description should be a simulation function that receives inputs and produces outputs, just as the real circuit will when it is fabricated. But when it comes time to construct the circuit using real hardware, the meaning of the circuit description is *the hardware itself* — the set of components, their interconnections, and their placement. At this point, executing the circuit description should produce a set of instructions that can drive a wire wrapper, a VLSI mask generator, or some other automatic hardware fabrication tool.

It is straightforward to implement the HDRE method in any programming language that provides the right facilities, just by defining a small library of functions that will call, and be called by, the circuit description functions. The necessary language features are first-class functions, lists, streams (or unbounded lists), lexically scoped blocks with equations, function recursion and data recursion (or the ability to represent arbitrary graphs). Lazy (or normal order) evaluation is useful because it supports correct equational behavior inside lexical blocks, as well as providing streams as a special type of list. However, it is possible to implement HDRE without lazy evaluation. Many programming languages provide the necessary features, or their equivalents, including Common Lisp [33], Scheme [29], ML [23], SASL [34], Lucid [35], etc. I have implemented HDRE in Daisy [18, 19], which is ideal because of its suspended list constructor. One of the advantages of HDRE is its ability to run in many existing languages.

HDRE is based on previous work by Gordon [8] and Johnson [14, 15, 16, 17]. A related approach, which considers many aspects of a circuit in addition to its structure and behavior, is "functional descriptions of systems" [4]. The main innovations in HDRE are its exploitation of sets of recursion equations inside **letrec** expressions to describe levels of abstraction within a circuit, its use of alternative sets of primitive functions to provide the various circuit meaning functions, its support of simulation at multiple levels of abstraction, and its use of functional geometry [13] for generating circuit layouts.

2. The basic idea

The basic idea of HDRE is to attach multiple meanings to a single circuit description through the use of alternative sets of primitive functions. This allows the designer to specify all aspects of the circuit with one semantic description. In order to achieve that, however, we must find a way to represent directly the inputs and outputs of components and the values carried on wires (called *signals*).

To illustrate the problem, consider a circuit that contains an AND gate with inputs A and B and output Y. A conventional hardware description language based on an imperative programming language would represent the signals A, B and Y as Boolean variables. However, each variable only represents the value of a signal during one clock cycle. The hardware signal itself consists of the entire history of values represented at that point in the circuit, one value per clock cycle. Thus the signal is partly represented by its variable, and partly by the sequence of values assigned to that variable during the course of a simulation. The circuit description functions cannot directly manipulate signals — they can only manipulate data structures supported by the programming language.

The real problem with conventional hardware description techniques is that they do not use "first class" objects to represent signals. A first class object is one that the language can manipulate directly (e.g., numbers, pointers and arrays are first class objects in most languages). However, the sequence of values that are assigned to a variable during the execution of a program is certainly not a first class object — and this is the fundamental reason why hardware descriptions using conventional programming language constructs end up with separate specifications of the structure and function of the circuit.

Stream recursion equations provide a way to represent the full history of a signal as a first-class language object. This allows the hardware description to treat the inputs and outputs of each component as ordinary objects. By selecting different sets of functions to operate on components, HDRE can produce either a simulation function or a circuit wiring specification from the same circuit description, and the designer can add geometric information in a straightforward manner in order to obtain a circuit layout.

3. Stream recursion equations and circuit behavior

This section gives an introduction to stream recursion equations and shows how they can describe the behavior of a circuit. A more formal explanation is in [15]. The following section shows how to construct a systematic library of behavioral descriptions using this technique.

Consider a combinational component with one input and one output, such as an inverter. During each cycle the component produces an output y which is a function f of its input x. Using an infix colon to denote function application, we can say that the behavior of the component is $y = f : x$, and f is called the *instantaneous behavior function.*

A realistic behavioral description must account for the sequence of states that the machine goes through, and a simple equation like $y = f : x$ is not sufficient. On the other hand, the form of this equation is clear; we just need a way to extrapolate the equation to handle the component's behavior through time. One way to do this is to use subscripts that specify when a value exists. Thus x_3 is the value of the input during clock cycle 3, and we have $y_i = f : x_i$ for $i \geq 0$.

It is possible to combine all the x_i values into a single stream X, where X=[x_0 x_1 x_2 ...]. A stream is a list whose length is unspecified, and possibly infinite. There are three functions that operate on streams, called "head", "tail" and "[... ! ...]":

- head:[x_0 x_1 x_2 ...] returns x_0. The head function is called `car` in Lisp.
- tail:[x_0 x_1 x_2 ...] returns [x_1 x_2 ...]. The tail function is called `cdr` in Lisp.

- $[a\ !\ [x_0\ x_1\ x_2\ \ldots]]$ returns $[a\ x_0\ x_1\ x_2\ \ldots]$. The notation [p ! q] corresponds to (cons p q) in Lisp.

Now we can write a function F which describes the behavior of the combinational component over time, assuming that f gives its instantaneous behavior during one cycle. The input X and the output Y are streams of individual values, and $Y = F : X$ describes the component's behavior, where

$$F = \lambda s\,.\,[f : \text{head} : s\ !\ F : \text{tail} : s].$$

The notation $f = \lambda x.y$ means "f is a function which returns the value of y when given input x". Solving this equation for the first few stream elements yields

$$y_0 = f : \text{head} : s = f : x_0$$

$$y_1 = f : \text{head} : \text{tail} : s = f : x_1$$

$$y_2 = f : \text{head} : \text{tail} : \text{tail} : s = f : x_2$$

A component with two inputs has a similar description. Let g be its instantaneous behavior function (e.g., $g = \lambda[a\ b]\,.\,a \cdot b$ describes the "and" function). Now the component's description equation is $Y = G : [A\ B]$ where

$$G = \lambda[r\ s]\,.\,\big[g : [\text{head} : r\ \ \text{head} : s]\ \ !\ \ G : [\text{tail} : r\ \ \text{tail} : s]\big].$$

The next section generalizes this to construct stream behavior functions for components that have an arbitrary number of inputs and outputs.

Clocked components (the various kinds of flip flops) behave differently from combinational components because they don't have instantaneous behavior functions. Instead, their output during clock cycle i depends on the memory they have of inputs during clock cycle $i-1$. Therefore we must write equations that directly describe the entire stream behavior of a clocked component. The simplest case is the D flip flop, or latch, whose output is just the value that its input had during the previous clock cycle. The stream behavior function for a latch is simply $Y = [0\ !\ X]$, where 0 is the initial value stored in the latch. It is easy to see that $y_i = x_{i-1}$ for $i > 0$.

The circuit C (Figure 1) combines a latch with a combinational exclusive or gate, and it contains a feedback loop.

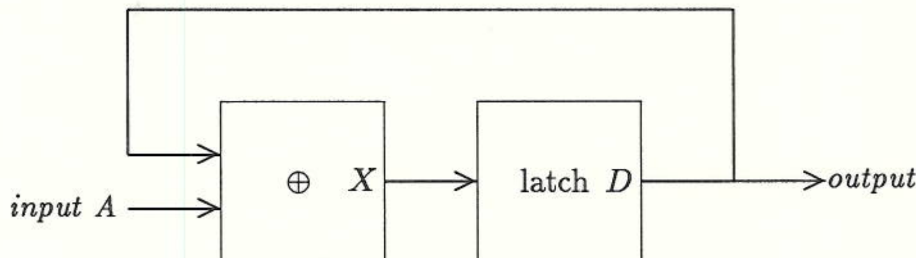


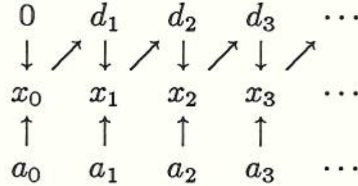
Figure 1. Circuit C

We assume that the initial value of the latch is 0. (In a real design, of course, there must be a provision for forcing all the flip flops into a valid initial configuration, but we will ignore such details here.) The value d_i which the latch outputs during clock cycle i for $i > 0$ is just the value x_{i-1}, which was on its input during cycle $i-1$. The exclusive or gate defines x_i, which is a function of its inputs a_i and d_i. This leads to the following recurrence equations, which fully describe the synchronous behavior of the circuit:

$$d_0 = 0, \quad d_i = x_{i-1} \text{ for } i > 0,$$

$$x_i = d_i \oplus a_i \text{ for } i \geq 0.$$

These equations illustrate the key difference between combinational components and clocked components in synchronous design. The output of a combinational component (the exclusive or gate) depends on the values of its inputs *during the same clock cycle*, but the output of a clocked component (the latch) depends on the values of its inputs *during the previous clock cycle*. The following diagram shows the data dependencies that are implied by the recurrence equations:



The recurrence equations can be solved by computing the values x_i in order to calculate the values of the circuit's output.

$$output_0 = d_0 = 0$$

$$output_1 = d_1 = x_0 = a_0 \oplus d_0 = a_0 \oplus 0 = a_0$$

$$output_2 = d_2 = x_1 = a_1 \oplus d_1 = a_1 \oplus a_0$$

$$output_3 = d_3 = x_2 = a_2 \oplus d_2 = a_2 \oplus (a_1 \oplus a_0)$$

$$\vdots$$

Instead of explicitly programming a simulator to work out the equations, we can simply write a stream function that specifies the circuit's behavior.

$C = \lambda\ in$.
letrec $X = \text{XOR} : [D\ in]$
$D = [0 \ !\ X]$
in D

Since the circuit C contains feedback, its streams are defined in terms of each other. The **letrec** expression implements recursive definitions of data structures. Although D is defined using X and X is defined using D, none of the stream elements are defined circularly. Programming languages that use lazy evaluation [12] can solve systems of recursion equations by calculating the values of stream elements one by one, as they are needed. Therefore no one needs to write a circuit simulator algorithm in order to solve the recurrence equations.

Notice that the function C encapsulates the circuit's behavior. The function takes an input and produces an output, but the streams X and D are defined locally within the **letrec** expression, so they are not visible outside the scope of the function definition. In effect, this turns the description of C into a "black box". Nested **letrec** expressions support a structured, hierarchical design style. Section 8 illustrates this with the description of a tree architecture.

4. Building primitive behavioral descriptions

In the previous section we built a stream behavior function F for each component using its instantaneous behavior function f. This section shows how to generalize that operation and build a set of primitive component behavior definitions.

Consider an instantaneous behavior function f that takes j inputs and produces k outputs. Instead of manually constructing the corresponding stream behavior function, it is more convenient to use a general functional *star* that maps f over the i'th elements of each of the input streams in order to compute the k output streams. Thus the equation

$$[Y] = (star : and) : [A\ B]$$

will define $y_i = a_i \cdot b_i$ for $i \geq 0$.

Each component in a circuit description will have a behavioral definition similar to this, so it is convenient to introduce a function *behavior* that creates stream functions from instantaneous behavior functions.

$$behavior = \lambda f\,.\,\lambda inputs\,.\,(star : f)\ :\ inputs$$

In the lambda calculus, the *behavior* function is just *star* (by two eta conversions), so we can define *behavior=star*.

The expression $behavior : f$ produces a function from j input streams that produces k output streams. For example, an adder produces two outputs — a sum and a carry — so a circuit containing an adder might define

$$[S\ C] = (behavior : \mathrm{add}) : [X\ Y\ Z]$$

Many functional programming languages (including Daisy and SASL) allow equations with structured left sides. A component that produces only one output will have a stream behavior function that produces a list containing just the output stream. It is convenient to define such a component's function using head : *behavior* : f, which allows composition of functions. For example, it is simpler to write

$$Y = AND : [A\ \ NOT : B]$$

than to use

$$[\mathrm{DUMMY}] = NOT : B \qquad Y = AND : [A\ \mathrm{DUMMY}]$$

HDRE uses *behavior* to define a standard library of primitive components called *behavior_primitives*. Some of the components (e.g. *AND*) return a stream, allowing them to be composed with other functions, while others (e.g. ADD) return a list of streams representing their multiple outputs.

$$
\begin{aligned}
AND &= \mathrm{head} : behavior : \lambda[a\ b]\,.\,a \wedge b \\
OR &= \mathrm{head} : behavior : \lambda[a\ b]\,.\,a \vee b \\
NOT &= \mathrm{head} : behavior : \lambda a\,.\,\overline{a} \\
ADD &= behavior : \lambda[a\ b\ c]\,.\,[\mathrm{sum} : [a\ b\ c]\quad \mathrm{carry} : [a\ b\ c]] \\
CMP &= \mathrm{head} : behavior : \lambda[a\ b]\,.\,(a = b \rightarrow 1,\ 0) \\
MUX1 &= \mathrm{head} : behavior : \lambda[p\ a\ \ b]\,.\,(p = 0 \rightarrow a,\ b) \\
MUX2 &= \mathrm{head} : behavior : \lambda[p\ a\ b\ c\ d]\,.\,(p = 0 \rightarrow a,\ p = 1 \rightarrow b,\ p = 2 \rightarrow c,\ d) \\
LATCH &= \lambda A\,.\,[0\ !\ A]
\end{aligned}
$$

behavior_primitives

It is straightforward to implement the *behavior* function in Daisy using functional combination [7], but other languages can achieve the same effect with closures and mapping functions.

5. Systems of recursion equations

To illustrate how to describe a circuit with a system of recursion equations, it is important to use an example that contains both state and feedback. Figure 2 shows the schematic diagram of a three-bit shift register *SR* that has has inputs *OP*, *L* and *R*, (the opcode, left input and right input respectively), and outputs from each of the three bits *ALPHA*, *BETA* and *GAMMA*. The system has four operations:

opcode	*mnemonic*	*description*
0	CLR	each bit stores 0
1	NOP	each bit retains its previous value
2	SHL	shift left
3	SHR	shift right

The hardware description will be at an intermediate level of abstraction, which shows how the individual bits of the register are interconnected while suppressing the internal details of each bit. (An example in a later section shows how to give both intermediate and low level descriptions; this is useful in hierarchical design and simulation.)

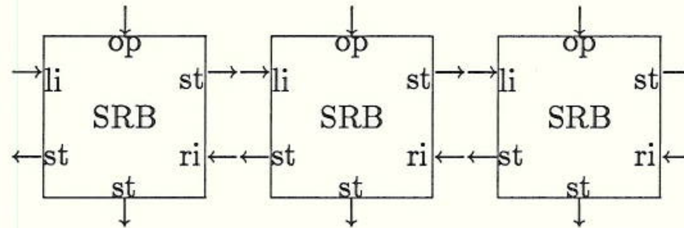


Figure 2. Shift Register Schematic

Each bit in *SR* is a component of type *SRB*, and the function *srbf* gives the behavior of an *SRB* during one clock cycle. The inputs *OP*, *LI* and *RI* stand for "opcode", "left input", and "right input" respectively, and *ST* is the "state" of the flip flop.

$$
\begin{aligned}
&SRB = \lambda[OP\ LI\ RI].\\
&\quad \textbf{letrec}\ ST = LATCH : (behavior : srbf) : [OP\ LI\ RI\ ST]\\
&\qquad \textbf{in}\ ST
\end{aligned}
$$

$$
\begin{aligned}
srbf = \lambda[op\ li\ ri\ st].\ &(op = 0 \rightarrow 0,\\
&\ op = 1 \rightarrow st,\\
&\ op = 2 \rightarrow ri,\\
&\ op = 3 \rightarrow li)
\end{aligned}
$$

The ST stream must be defined inside a **letrec** because its value appears on the right side of the defining equation. However, this is not a circular definition, because $LATCH$ causes ST_k to depend on OP_{k-1}, LI_{k-1}, RI_{k-1} and ST_{k-1}, so ST_k is not defined in terms of ST_k. In a synchronous circuit all feedback loops must pass through a clocked (sequential) component. If there is a purely combinational feedback loop, the circuit is asynchronous and it may not settle down to a stable state in a bounded amount of time. A system of

recursion equations describing such a circuit models its behavior by diverging: the value of S would be $[\bot\ \bot\ \bot\ \ldots]$.

The shift register is a function with inputs OP, L and R which contains a system of recursion equations defining the individual bits and their interconnections:

$$
\begin{aligned}
&SR = \lambda[OP\ L\ R].\\
&\quad \textbf{letrec}\ A = SRB : [OP\ L\ B]\\
&\quad\quad\quad\ B = SRB : [OP\ A\ C]\\
&\quad\quad\quad\ C = SRB : [OP\ B\ R]\\
&\quad\quad \textbf{in}\ [LABEL : [\text{“ALPHA”}\ A]\ LABEL : [\text{“BETA”}\ B]\ LABEL : [\text{“GAMMA”}\ C]]
\end{aligned}
$$

The $LABEL$ component is just used to label the outputs with names, which is necessary in order to generate a circuit wiring list (see the following sections). The behavioral specification ignores the labels, so $LABEL$ is an identity function on its second input as far as the behavior of SR is concerned:

$$LABEL = \lambda[n\ s]\ .\ s$$

The functions described in this section, including SR, are not written in a new hardware description language. They are just ordinary functions that can be written in any programming language that supports first class functions, **letrec** and streams. The next section shows how to execute hardware description functions like SR in order to extract their meanings.

6. The meanings of a circuit description

The SR function is meaningless until the primitive SRB is defined. Evaluating SR in an environment where SRB is defined as in the previous section should yield a *simulation function* which takes streams OP, L and R as inputs and produces streams $ALPHA$, $BETA$ and $GAMMA$ as outputs. The higher order function *meaning* does this transformation.

$$
\begin{aligned}
&meaning = \lambda[c\ p]\ .\ \lambda inputs\ .\\
&\quad \textbf{letrec}\ p\\
&\quad\quad \textbf{in}\ c : inputs
\end{aligned}
$$

The value bound to c must be a circuit description (like SR), and p must be bound to a set of equations defining the primitives used by c. The result is a function similar to c except that all the free variables referring to primitive components have been defined. Since *meaning* constructs a **letrec** expression on-the-fly and then evaluates it, the underlying programming language needs an operation like the Lisp `eval`. Daisy can interpret the definition of *meaning* directly.

To demonstrate the shift register's simulation function, we must apply it to a set of test data. It is easy to read the data in the form of a sequence of lists of values, where each list gives all the inputs or outputs for one clock cycle. However, the simulation function requires its inputs and outputs to be a list of streams. The trivial *transpose* function converts between these data formats, while *lines* inserts line breaks between the lists. The '|' character makes the rest of an input line into a comment. Executing

```
SR_test_data =
 [[3 11 21]   | SHR  Shift right           [OPCODE LI RI]
  [3 12 22]   | SHR  Shift right
  [3 13 23]   | SHR  Shift right
  [3 14 24]   | SHR  Shift right
  [2 15 25]   | SHL  Shift left
  [1 16 26]   | NOP  Stay with previous value
```

```
    [0 17 27]    | CLR  Clear
    [2 18 28]    | SHL  Shift left
    [1 19 29]    | STA  Stay with previous value
  ]

"Shift Register Simulation:"

  lines : transpose :
     (meaning : <SR behavior_primitives>) :
        transpose : inputs
```

yields the correct output:

```
Shift Register Simulation:
[[0 0 0]
[11 0 0]
[12 11 0]
[13 12 11]
[14 13 12]
[13 12 25]
[13 12 25]
[0 0 0]
[0 0 28]
[0 0 28] ]
```

The simulation function helps a designer work toward the first goal — developing a correct circuit description. However, someone trying to fabricate the circuit will interpret *SR* entirely differently: it is a specification of the set of components in the circuit and their interconnections. *Given the right set of primitives, the meaning function can extract this information from SR, instead of extracting the simulation function.*

The key observation needed to write a set of "structural primitives" is that no component knows where its outputs go, but each component does know what its inputs are — it is applied to them. Thus if we represent signals by symbolic names rather than streams of behavior values, any component receiving an input signal can build a list representing that connection. A system of recursion equations inside a **letrec** will result in a circular graph structure that is isomorphic to the graph structure of the actual circuit. A straightforward graph traversal function can then print the graph in a readable form.

Each component in the connection list has a type, and other identifying information, and each of its inputs and outputs has a local name. For example, the *and* gate has type "AND", input ports named 'a' and 'b', and an output port named 'x'. Since all components have a similar overall structure, it is convenient to write a function *blackbox* that produces the component's internal graph representation from its type, a list of its inputs and a list of its outputs. We can then use *blackbox* to provide a set of structure primitives analogous to the behavior primitives given earlier.

$$\begin{aligned}
AND &= \textit{blackbox} : [\text{“AND”}\ [a\ b]\ [x]] \\
OR &= \textit{blackbox} : [\text{“OR”}\ [a\ b]\ [x]] \\
NOT &= \textit{blackbox} : [\text{“NOT”}\ [a]\ [x]] \\
ADD &= \textit{blackbox} : [\text{“ADD”}\ [a\ b\ c]\ [s\ co]] \\
CMP &= \textit{blackbox} : [\text{“CMP”}\ [a\ b]\ [x]] \\
MUX1 &= \textit{blackbox} : [\text{“MUX1”}\ [p\ a\ b]\ [x]] \\
MUX2 &= \textit{blackbox} : [\text{“MUX2”}\ [p\ a\ b\ c\ d]\ [x]]
\end{aligned}$$

structure_primitives

An additional primitive *mk_source* constructs a graph representing a system input, which can be bound to an input of the circuit description. The *connectivity* function traverses the graph that results from executing a circuit description, and produces the component and wiring lists. Executing

```
SR_structure =
   (meaning:<SR structure_primitives>) :
        <mk_source:"OPin" mk_source:"Lin" mk_source:"Rin">
connectivity : SR_structure
```

produces a low-level description of the circuit that could directly drive a placement and routing machine. The specification consists of a component list and a wire list. The first element of each component specification is its unique component number; the second element contains the component type; the third element lists the inputs; and the last element lists each output along with its fanout (the number of inputs that are driven by that output). Each wire is represented by a list of the form `(source --> sink)`, where the source and sink of the wire consist of the component number and type and the port name. For example, the first wire connects the `Rin` input to the `ri` input port of the rightmost shift register, which is component number 5. Note that the center shift register bit has a fanout of 3 from its state, while the other two bits have a fanout of only 2. The reason is that only one neighbor reads the state from the leftmost and rightmost bits, but both neighbors read the center bit.

```
Components:
[0 [*OUTPORT*] [[ALPHA]] [<"outside_interface"> 0]]
[1 [*OUTPORT*] [[BETA]] [<"outside_interface"> 0]]
[2 [*OUTPORT*] [[GAMMA]] [<"outside_interface"> 0]]
[3 [SRB] [[op] [li] [ri]] [[st] 2]]
[4 [SRB] [[op] [li] [ri]] [[st] 3]]
[5 [SRB] [[op] [li] [ri]] [[st] 2]]
[6 [*INPORT*] [] [[OPin] 3]]
[7 [*INPORT*] [] [[Lin] 1]]
[8 [*INPORT*] [] [[Rin] 1]]

Wires:
[*INPORT* 8 Rin --> SRB 5 ri]
[SRB 4 st --> SRB 5 li]
```

```
[*INPORT* 6 OPin --> SRB 5 op]
[SRB 5 st --> SRB 4 ri]
[SRB 3 st --> SRB 4 li]
[*INPORT* 6 OPin --> SRB 4 op]
[SRB 4 st --> SRB 3 ri]
[*INPORT* 7 Lin --> SRB 3 li]
[*INPORT* 6 OPin --> SRB 3 op]
[SRB 5 st --> *OUTPORT* 2 GAMMA]
[SRB 4 st --> *OUTPORT* 1 BETA]
[SRB 3 st --> *OUTPORT* 0 ALPHA]
```

7. Implementation of the wire list extractor

Extracting a wire list from a circuit description takes place in two stages. First, the *meaning* function interprets the description with a set of structural primitives. That results in an internal graph structure that is isomorphic to the interconnection structure of the circuit. Second, the *connectivity* function traverses the internal graph structure and prints it in a readable fashion.

Consider the circuit $Y = F : G : X$. The output of G becomes the input of F. Therefore the graph structure that represents F will contain a pointer to G. Similarly, the value of Y is simply a pointer to the output port of F. This leads to a surprising phenomenon in the internal graph representations: the *outputs* of a component representation are the targets of pointers from devices that read those outputs, while the *inputs* of the component are represented by pointers to the sources that provide the input values.

The representation of a component must contain the type of component, a pointer to each of its inputs, and a port for each of its outputs. Each output port points to the component, and other circuits may contain pointers to whichever output ports they need. For example, consider a component F which has three inputs, x, y and z, and which produces two outputs, a and b. Figure 3 shows the notation for F that would be used in a schematic diagram.

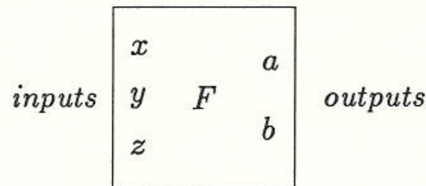


Figure 3. Notation for component F

Figure 4 shows the internal graph representation for F. The structure consists of a number of cons boxes linked by their tail fields and containing information in their head fields. There are two output ports, whose heads contain the corresponding output port names (a and b). Both output ports point to the component description, which specifies the type F and points to the list of input ports. Note that a component must have each input port connected to exactly one source (we are prohibiting "wired or"). On the other hand, there may be any number of other circuits that use each output of the component. This does not lead to difficulties, because the component representation doesn't contain any pointers to the users of its outputs.

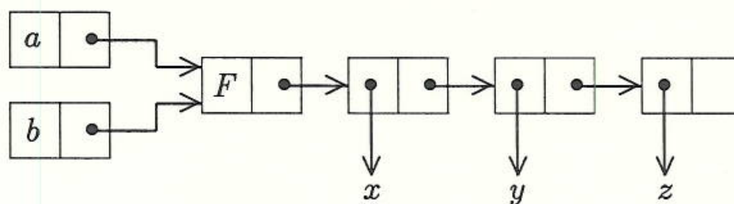


Figure 4. Internal graph representation of component F

To illustrate how this representation works, Figure 5 shows the complete internal graph representation for Circuit C (Figure 1).

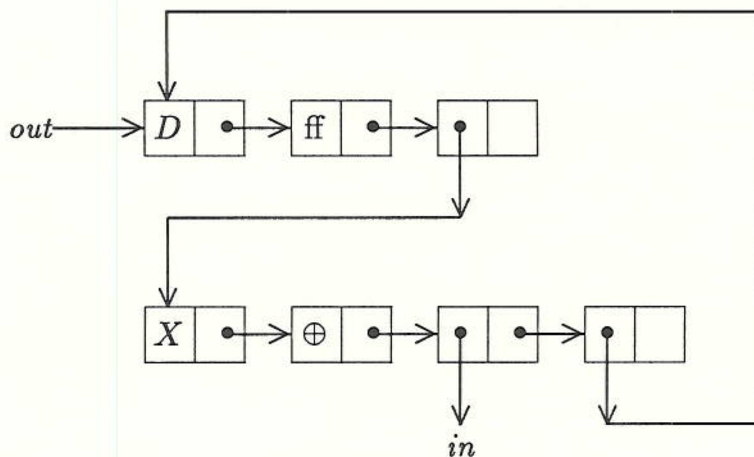


Figure 5. Graph representation of circuit C

The *connectivity* function starts from pointers to the output ports of a circuit graph, and traverses the graph in order to build the wiring list. This means that any components in the circuit whose outputs are unused will not appear in the wiring list. The implementation of *connectivity* is similar to Lisp pretty-printing functions, except it must handle circular structures. As the function goes through the graph it keeps a list of components that it has already looked at, so that it can detect when a circular pointer brings it back to a point in the graph a second time.

8. Hierarchical circuit descriptions

Since circuit description functions are first-class objects in the programming language, other functions can create and combine them. This allows a designer to write a function that gives a hierarchical circuit description. The description can be evaluated (with either structural or behavioral primitives) at any level in the hierarchy, or even at different levels in different parts of the design. For example, if the designer is having trouble with one part of the circuit, he or she can simulate that part at a low level, while all the rest of the circuit is simulated at a high level. Again, this advantage is not a special feature tacked on to a hardware description language; it results directly from the representation of components by first-class functions.

A generalization of the shift register *SR* to a tree architecture *TREE* (Figure 6) illustrates the elegance and power of functions that generate circuit descriptions. *TREE* contains two kinds of components. Its leaves are called *cells*, and each cell is connected to both of its neighbors as well as to its parent in the tree. Therefore the sequence of cells is a shift register like *SR*, except for the extra data path. Each cell contains both sequential

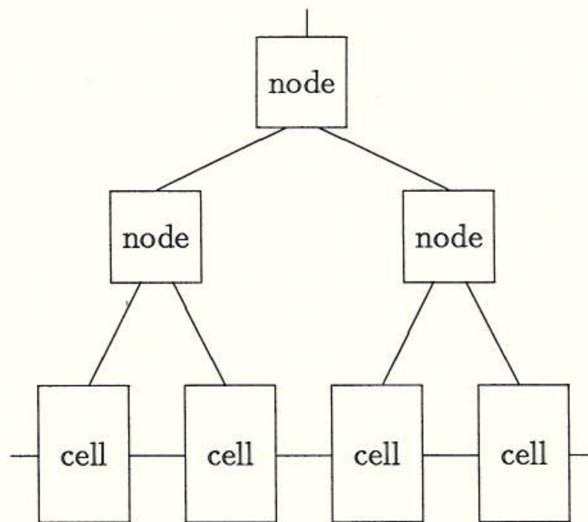


Figure 6. Tree Architecture

and combinational components, just like the shift register bits. The non-leaf components, called *nodes*, contain no sequential components; they are purely combinational.

The *TREE* architecture is a simplified version of a very powerful data structure memory [25, 26, 27], which generalizes the facilities of associative memories. Each cell contains two storage fields: "M" (for match), and "V" (for value). The system can access data either by shifting it through the left and right ports, or by performing associative matches that compare cell values with an operand. The nodes perform two services: (1) they broadcast instructions from the top port to all the cells, and collect the cells' responses back to the top port, and (2) they carry out a priority resolution for multiple matches. Each instruction (which comes in through the top input TI) contains three fields: the opcode *TI_op*, *TI_m*, and the operand *TI_v*. There are four instructions which cause each cell to perform the following actions:

opcode	*mnemonic*	*description*
0	SHFTR	shift right
1	MATCH	set M if V = operand
2	LEFTM	leftmost cell with M is unchanged; others clear M
3	CSTOR	conditionally store operand iff M

The tree nodes and cells are intermediate level components. We can define their internal structure by:

$$
\begin{aligned}
node = \lambda[[&TI_op\ TI_m\ TI_v]\ [LI_m\ LI_v]\ [RI_m\ RI_v]]\,.\\
&[[OR : [LI_m\ RI_m]\ ADD : [LI_v\ RI_v]]\\
&\ [TI_op\ TI_m\ TI_v]\\
&\ [TI_op\ AND : [TI_m\ NOT : LI_m]\ TI_v]\\
&]
\end{aligned}
$$

$$
\begin{aligned}
cell = \lambda[[&TI_op\ TI_m\ TI_v]\ [LI_m\ LI_v]\ [RI_m\ RI_v]]\,.\\
&\mathbf{letrec}\ ST_m = LATCH : MUX2 :\\
&\qquad [TI_op && \text{(opcode selects result value)}\\
&\qquad\ LI_m && \text{(Shift Right instruction)}\\
&\qquad\ CMP : [ST_v\ TI_v] && \text{(Match instruction)}\\
&\qquad\ AND : [ST_m\ TI_m] && \text{(Select Leftmost instruction)}\\
&\qquad\ ST_m && \text{(Conditional Store instruction)}
\end{aligned}
$$

```
                 ]
      ST_v = LATCH : MUX2 :
                 [TI_op                      (opcode selects result value)
                  LI_v                       (Shift Right instruction)
                  ST_v                       (Match instruction)
                  ST_v                       (Select Leftmost instruction)
                  MUX1 : [ST_m ST_v TI_v]    (Conditional Store instruction)
                 ]
   in [ST_m STv] [ST_m ST_v] [ST_m ST_v]]
```

The tree architecture consists of a number of nodes and cells connected together. However, we don't need to write out all these components in a **letrec**; we can exploit the regular structure of the architecture by a high order function *tree* that builds the systems of recursion equations using recursive descent. The *tree* function's formal parameter n specifies the depth of the tree. If $n = 0$ then the tree is simply a cell, but if $n > 0$ then the tree consists of two subtrees and a node connected together in a **letrec**. The subtrees are not primitive components; they result from recursive calls to *tree*. The naming convention in the *tree* function indicates the relative location of each port: thus *N_to* is the top output from the node, while *L_ro* is the right output from the left subtree.

```
tree = λn . λ[TI LI RI] .
         (n = 0  →  cell : [TI LI RI],
              letrec [N_to N_lo N_ro] = node : [TI L_to R_to]
                     [L_to L_lo L_ro] = (tree : n − 1) : [N_lo LI R_lo]
                     [R_to R_lo R_ro] = (tree : n − 1) : [N_ro L_ro RI]
                 in [N_to L_lo R_ro]
```

As with the shift register, we can extract the meaning of the tree circuit with either behavior primitives or structure primitives. In addition, we can either treat nodes and cells as primitive components themselves, yielding a high level connection list, or we can expand them into lower level components such as and gates. The high level interpretation of *tree*:2 produces:

```
Components:
[0 [*OUTPORT*] [[A]] [<"outside_interface"> 0]]
[1 [*OUTPORT*] [[B]] [<"outside_interface"> 0]]
[2 [*OUTPORT*] [[C]] [<"outside_interface"> 0]]
[3 [node] [[ti] [li] [ri]] [[to] 1 [lo] 1 [ro] 1]]
[4 [cell] [[ti] [li] [ri]] [[to] 1 [lo] 1 [ro] 1]]
[5 [cell] [[ti] [li] [ri]] [[to] 1 [lo] 1 [ro] 1]]
[6 [*INPORT*] [] [[AA] 1]]
[7 [node] [[ti] [li] [ri]] [[to] 1 [lo] 1 [ro] 1]]
[8 [node] [[ti] [li] [ri]] [[to] 1 [lo] 1 [ro] 1]]
[9 [*INPORT*] [] [[BB] 1]]
[10 [cell] [[ti] [li] [ri]] [[to] 1 [lo] 1 [ro] 1]]
[11 [cell] [[ti] [li] [ri]] [[to] 1 [lo] 1 [ro] 1]]
[12 [*INPORT*] [] [[CC] 1]]

Wires:
[cell 5 lo --> cell 11 ri]
[cell 10 ro --> cell 11 li]
[node 8 lo --> cell 11 ti]
```

```
[cell 11 lo --> cell 10 ri]
[cell 4 ro --> cell 10 li]
[node 7 ro --> cell 10 ti]
[cell 5 to --> node 8 ri]
[cell 11 to --> node 8 li]
[node 3 ro --> node 8 ti]
[cell 10 to --> node 7 ri]
[cell 4 to --> node 7 li]
[node 3 lo --> node 7 ti]
[*INPORT* 12 CC --> cell 5 ri]
[cell 11 ro --> cell 5 li]
[node 8 ro --> cell 5 ti]
[cell 10 lo --> cell 4 ri]
[*INPORT* 9 BB --> cell 4 li]
[node 7 lo --> cell 4 ti]
[node 8 to --> node 3 ri]
[node 7 to --> node 3 li]
[*INPORT* 6 AA --> node 3 ti]
[cell 5 ro --> *OUTPORT* 2 C]
[cell 4 lo --> *OUTPORT* 1 B]
[node 3 to --> *OUTPORT* 0 A]
```

With the low level description of nodes and cells, the meaning of *TREE* with the structural primitives expands into a connection list that contains 50 components and 103 wires (which will not be shown here!). The meaning of *tree* : n with the behavior primitives is a simulation function that illustrates the operation of the machine.

9. Layouts require geometric information

Neither the shift register nor the tree circuit descriptions contain any knowledge about *where* the components and wires are supposed to go; they simply give the connectivity of a circuit without saying anything about its geometry. Thus Figure 2, the shift register schematic, contains more information than the *SR* function. In order to produce either a rough schematic or a refined VLSI mask layout, it is necessary to know where to put all the components.

Three general approaches to this problem appear in various hardware description methodologies:

- Make the system automatically figure out the placement of components and wires.
- Give the designer a restricted set of operations that build up larger circuits from smaller ones, and associate a specific geometric placement for each of these operations.
- Require the designer to specify where everything goes.

Most hardware description languages combine several of these approaches. Silicon compilers [1] often allow the designer to specify part of the placement, while automating the rest. It is frequently useful to have the designer specify the component placements, while the system then routes the wires. Several hardware description systems are based on the FP language [2]. The muFP language [30, 31, 32] associates a geometric organization with each of its circuit combining forms. A related system, νFP [28], extracts circuit layouts from functional descriptions. The Escher system [5] allows the user to describe geometric layouts recursively, which makes it possible for a designer to write a layout specification that has

the same hierarchical structure as the abstract circuit design. There is increasing interest in using expert systems to aid in automatic component placement.

HDRE allows the user to describe the geometric structure of a circuit by providing several high order combining functions similar to **letrec**. These functions specify the geometric combination of several smaller components into a larger one. Two components are connected in a VLSI layout or a schematic simply by placing them next to each other. If their input and output ports line up properly on their common edge, then the components are connected to each other. In HDRE, the designer may combine two components with *hbox*, which lines them up horizontally, or *vbox*, which lines them up vertically. (These functions are named after similar ones in the TEX typesetting system [20], which also allows the user to specify geometric placement of components.) Each structural primitive in HDRE is rectangular, and its description specifies which edge, and where on that edge, every port lies. When the user combines two components with *hbox* or *vbox*, the behavioral version of the combining function creates a **letrec** expression containing the equations needed to connect the ports that lie on the components' common edge, and it appends the ports on the other edges to synthesize the port description for the complete circuit. Of course, the structural version of each combining function simply generates a larger layout from the smaller ones.

This method of generating layouts is similar to Henderson's functional geometry [13], except that HDRE does not currently support rotations or reflections. Rotating a building block in a VLSI design is a tricky problem, because all the transistors must be correctly connected up to the power supplies (VDD and GND) both before and after rotation. This means that some layouts will happen to work properly after rotation, while others will not. Further work is needed on these problems.

Functional geometry works extremely well on regular layouts, although it is awkward for "random logic" circuits. Systolic arrays [22] can be generated easily with primitives like *hbox* and *vbox*. However, many computer aided design systems will perform automatic component placement and wire routing for small sections of random logic, while functional geometry requires the designer to specify the relative position of all the components. It would probably be best for the system to automatically do the lowest levels of component placement and wire routing, while allowing the designer to specify the higher levels with functional geometry. This would exploit the designer's insight and experience, while making the process of describing a circuit less tedious.

The current HDRE system builds TEX code to describe the geometric layout of a circuit, as long as the designer uses *hbox* and *vbox* to connect the components. The TEX code can then be included in an ordinary document to be typeset, which is very convenient for documenting digital designs. This system generated the tree architecture layout in Figure 7. The tree nodes are labelled *ncl* (for "node combinational logic") and the cells have subsections labelled *ccl* (for "cell combinational logic") and *cst n* (for "cell *n* storage"). The organization of the layout is similar to the tree layout generator in [21].

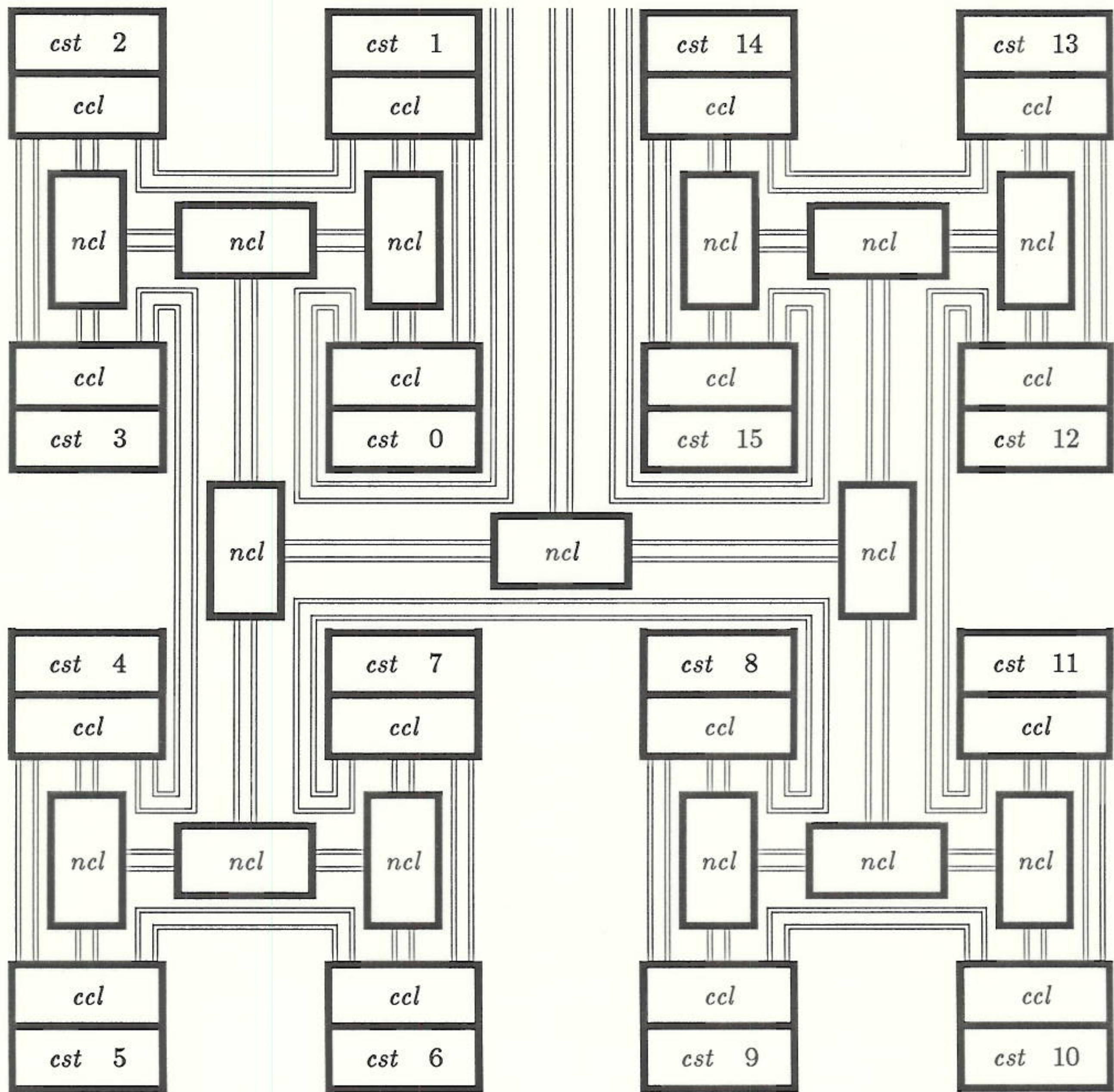


Figure 7. Tree architecture layout

10. Conclusion

One of the problems with modern hardware design is the proliferation of tools and languages for using them. A typical VLSI design system will contain layout editors, logic simulators, design rule checkers, etc. As a circuit design spawns more and more independent descriptions of different aspects of the circuit, inconsistencies among them become almost inevitable. The recursion equation approach to circuit description unifies three key aspects of any circuit: its behavior, its connectivity and its geometry.

HDRE is not a programming language. It is a technique for describing hardware using an existing programming language. The current implementation consists of a set of function definitions written in Daisy, comprising about five pages of code. It is by no means a complete design environment. The opposite is true: a major purpose of HDRE is to provide a unified basis for a set of design tools.

Several people have used HDRE to design circuits ranging from small to intermediate in size. These include an associative memory, a RISC processor, and a distributed real-time graphics controller. A general observation from this experience is that using HDRE is very

slow in terms of computer time but very fast in terms of how long it takes a designer to experiment with a circuit and modify it quickly if necessary. (This is the reason that "rapid prototyping" systems are becoming popular.) One designer found a bug in a circuit design that was about to be burned into a PAL chip. It is clearly easier and cheaper to develop and debug a circuit with HDRE than it would be with a series of physical fabrications.

An alternative to describing digital hardware with functions is to use mathematical logic. Temporal logic can express phenomena that vary with time, and Moszkowski shows how to use temporal logic to describe the behavior of digital circuits [24]. Temporal logic can describe the behavior of a clock in a synchronous circuit, but HDRE cannot (HDRE's clock is implicitly described by the data dependencies among the stream elements). Thus temporal logic gives a lower level description of hardware than stream functions do.

Gordon shows how to use higher order logic as an elegant hardware description language [11]. This also leads to a lower level of abstraction than HDRE. For example, higher order logic can express the bidirectional nature of a VLSI pass transistor [22]. For a transistor T with gate g and diffusion ports a and b, the expression

$$T(g, a, b) \equiv \big(g \supset (a = b)\big)$$

specifies exactly what the transistor does, and no more: if the value on g is 1, then we can deduce that $a = b$. But there is no assumption about which of a and b is the input and which is the output. In contrast, HDRE treats every component as a function, forcing each of the component's ports to be identified as an input or as an output. This means that mathematical logic is better suited than HDRE for describing low-level VLSI techniques like bidirectional ports and precharged wires.

Although it cannot easily describe the lowest levels of circuit behavior, HDRE has a number of advantages. It provides a very effective way of describing circuits hierarchically, and supports mixed levels of simulation. It is easy to describe and modify a circuit design with HDRE. When a user is satisfied with the behavior of a circuit description, HDRE will automatically provide the structural specification needed to fabricate it , and the structural specification is guaranteed to produce exactly the same circuit that was being simulated.

A major unresolved problem is finding the best way for the user to express the geometry of a circuit in a description that also must express behavior. It is easiest for a designer to use a recursive graphic language, such as Escher [5], rather than a textual notation. On the other hand, HDRE uses applicative programming language notation for circuit description, and this leads it to a textual specification of the geometry as well. A good solution must provide the user with a natural notation, without separating the circuit's behavioral and structural descriptions. Davie discusses this and related problems [6].

The set of tools that run under HDRE needs to be extended. For example, a designer might want to specify component placement while allowing an automatic wire router to fill in the interconnections. HDRE doesn't address some of the low level problems that arise in VLSI design, as discussed above. It would be useful to combine HDRE with circuit verification techniques [9, 10]. Further research and experimentation are necessary to discover its potential and its limitations.

Acknowledgements

I would like to thank Steve Johnson, who introduced me to recursion equations and motivated this work, and Rosalee Nerheim and Timothy Bridges, who experimented with the system.

References

1. Ayres, Ronald F., *VLSI Silicon Compilation and the Art of Automatic Microchip Design*, Prentice-Hall, 1983.
2. Backus, John, "Can Programming be Liberated from the von Neumann Style?", *CACM* Vol. 21 No. 8, 1978.
3. Barbacci, Mario R., "Structural and behavioral description of digital systems", pp. 139–223 in J. Tiberghien (ed.), *New Computer Architectures*, Academic Press, 1984.
4. Boute, Raymond T., "Current Work on the Semantics of Digital Systems", pp. 99–112 in G. J. Milne and P. A. Subrahmanyam (ed.), *Formal Aspects of VLSI Design*, North-Holland, 1986.
5. Clarke, Edmund and Feng, Yulin, "Escher—A Geometrical Layout System for Recursively Defined Circuits", CMU-CS-85-150, Department of Computer Science, Carnegie-Mellon University, 1985.
6. Davie, B. S., "Hardware Description Languages: Some Recent Developments", Internal Report CSR-198-86, Department of Computer Science, University of Edinburgh, April 1986.
7. Friedman, Daniel P. and Wise, David S., "Functional Combination", *Computer Languages*, Vol. 3 No. 1, pp. 31–35, 1978.
8. Gordon, Mike, "A Very Simple Model of Sequential Behavior of nMOS", in *VLSI 81: Very Large Scale Integration*, John P. Gray (ed.), Academic Press, 1981.
9. Gordon, Mike, "LCF–LSM", Technical Report No. 41, University of Cambridge Computer Laboratory.
10. Gordon, Mike, "Proving a Computer Correct", Technical Report No. 42, University of Cambridge Computer Laboratory.
11. Gordon, Mike, "Why higher-order logic is a good formalism for specifying and verifying hardware", pp. 153–177 in G. J. Milne and P. A. Subrahmanyam (ed.), *Formal Aspects of VLSI Design*, North-Holland, 1986.
12. Henderson, Peter, *Functional Programming, Application and Implementation*, Prentice-Hall, 1980.
13. Henderson, Peter, "Functional Geometry", *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming.*
14. Johnson, Steven D.,"Circuits and Systems: Implementing Communication with Streams", *Proceedings of the 10th IMACS World Congress on Systems Simulation and Scientific Computation*, pp. 311–319, 1982.
15. Johnson, Steven D., *Synthesis of Digital Designs from Recursion Equations*, The MIT Press, Cambridge, 1984.
16. Johnson, Steven D., "Applicative Programming and Digital Design", *Eleventh Annual ACM Symposium on Principles of Programming Languages*, pp. 218–227, 1984.
17. Johnson, Steven D., "Digital Design in a Functional Calculus", in G. J. Milne and P. A. Subrahmanyam (ed.), *Proceedings of the Workshop on Formal Aspects of VLSI Design*, North-Holland, Amsterdam, 1985.
18. Johnson, Steven D., "Daisy Language Summary", Indiana University Computer Science Department, to appear.
19. Kohlstaedt, Anne T., "Daisy 1.0 Reference Manual", Technical Report 119, Indiana University Computer Science Department, Bloomington, 1981.
20. Knuth, Donald E., *The TEXbook*, Addison-Wesley, 1984.

21. Luk, W. K. and Vuillemin, J. E., "Recursive Implementation of Optimal Time VLSI Integer Multipliers", in F. Anceau and E. J. Aas (eds.), *VLSI 83: VLSI Design of Digital Systems*, North-Holland, 1983.

22. Mead, Carver and Conway, Lynn, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

23. Milner, Robin, "The Standard ML Core Language", Report CSR-168-84, Department of Computer Science, University of Edinburgh, 1984.

24. Moszkowski, Ben C., *Executing Temporal Logic Programs*, Cambridge University Press, 1986.

25. O'Donnell, John T., *A Systolic Associative LISP Computer Architecture with Incremental Parallel Storage Management*, Technical Report 81-5, Computer Science Department, University of Iowa, Iowa City, 1981.

26. O'Donnell, John T., "An Architecture that Efficiently Updates Associative Aggregates in Applicative Programming Languages", *Functional Programming Languages and Computer Architecture*, pp. 164–189, *Lecture Notes in Computer Science 201*, Springer-Verlag, 1985.

27. O'Donnell, John T., "An efficient architecture for implementing sparse array variables," *Twenty-third Allerton Conference on Communication, Control and Computing*, October 1985.

28. Patel, Dorab; Schlag, Martine and Ercegovac, Milos, "νFP: An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms", *Functional Programming Languages and Computer Architecture*, pp. 238–255, Lecture Notes in Computer Science 201, Springer-Verlag, 1985.

29. Rees, Jonathan and Clinger, William (eds.), et. al., "Revised^3 Report on the Algorithmic Language Scheme", *ACM SIGPLAN Notices*, Vol. 21 No. 12, pp. 37–79, 1986.

30. Sheeran, Mary, *μFP — An Algebraic VLSI Design Language*, Technical Monograph PRG-39, Programming Research Group, Oxford University Computing Laboratory, Nov. 1983.

31. Sheeran, Mary, "muFP, A Language for VLSI Design", *Proc. ACM Symposium on Lisp and Functional Programming*", pp. 104–112, 1984.

32. Sheeran, Mary, "Designing Regular Array Architectures using Higher Order Functions", *Functional Programming Languages and Computer Architecture*, pp. 220–237, *Lecture Notes in Computer Science 201*, Springer-Verlag, 1985.

33. Steele, Guy Lewis Jr., *Common Lisp: the Language*, Digital Press, 1984.

34. Turner, David A., "SASL Reference Manual", Computer Science Department, University of Kent, 1979.

35. Wadge, William W. and Ashcroft, Edward A., *Lucid, the Dataflow Programming Language*, Academic Press, 1985.