TECHNICAL REPORT NO. 221

A Tactical Framework for Hardware Design

by

Steven D. Johnson, Bhaskar Bose & C. David Boyer

May, 1987

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

A Tactical Framework for Hardware Design

by

Steven D. Johnson, Bhaskar Bose, and C. David Boyer

Computer Science Department
Indiana University
Bloomington, IN 47405

Appears in *VLSI Specification, Verification and Synthesis,* Graham Birtwistle and P. A. Subramanyam (eds.), Kluwer Academic Publishers, pp. 349–384.

Research reported herein was supported in part by the National Science Foundation, under grants numbered DCR 84-05241 and DCR 85-21497, and by the Westinghouse Educational Foundation.

A Tactical Framework for Hardware Design*

Steven D. Johnson, Bhaskar Bose, and C. David Boyer
Computer Science Department
Indiana University
Bloomington, Indiana

Introduction

This work explores an algebraic approach to digital design. A collection of transformations has been implemented to derive circuits, but "hardware compilation" is not a primary goal. Paths to physical implementation are needed to explore the approach at practical levels of engineering. The potential benefits are more descriptive power and a unified foundation for design automation. However, these prospects mean little when the algebra is done manually. Hence, a basic, automated algebra is prerequisite to our experimentation with design methods.

The transformation system discussed in this paper supports a basic design algebra. At this early stage of its development, the system is essentially an editor for deriving digital system descriptions from certain applicative specifications and for interactively manipulating them. It is integrated with existing assemblers for programmable hardware packages, such as the PAL components used to implement the examples presented in Sections 3 and 5. The transformation system and back-end assemblers are implemented in Scheme [18], a Lisp dialect, by Bose [2].

The coherent manipulation of design descriptions is a significant problem. Digital engineering is governed by a variety of theories addressing orthogonal problem aspects. It is not enough to unify these treatments in some

*Research reported herein was supported, in part, by the National Science Foundation, under grants numbered DCR 84-05241 and DCR 85-21497, and by the Westinghouse Educational Foundation.

metalanguage, for they must also be integrated. In any design instance, independent aspects become dependent facets, and hence, designs simultaneously decompose in distinct hierarchies. In this paper, we look at the separation of algorithm and architecture, the logical hierarchy of functional units, and the organization of physical components. The direction of this work is to find ways to maintain orthogonal views of a description as it is manipulated toward an implementation.

The examples in this paper are implemented as externally clocked systems, often called "synchronous systems" or "clocked sequential systems." The underlying model of sequential behavior is simply realized by a global clock. There is nothing to preclude other realizations of timing, although we have not done so. The examples show a transformational approach to synchronous design. They follow a structured digital design strategy to obtain comparable implementations. They also reflect a more general design methodology.

Section 1 is an informal discussion of how the approach is related to standard methods. The key idea, and central subject of this paper, is a class of transformations called *system factorizations*. These are used to isolate a level of detail while maintaining the global coherence of a circuit description.

We use an algebra on applicative notation. Section 2 gives a brief look at this language and its use to describe sequential systems. Functional recursion equations are used as specifications. Part of the algebra is to correctly translate these specifications into sequential-system descriptions, and Section 2 concludes with a simple characterization relating the two forms of description.

Section 3 is a first example of design derivation, and it serves several purposes. Most important is the illustration of the transformation algebra, and in particular, the factorization of subsystems. At the same time, a generic treatment of synchronous communication is discussed. We then explain how the transformation system is integrated with PAL assemblers to obtain physical implementations. Integration with VLSI tools is in progress. Section 3.3 is a brief discussion promoting the free use of functional abstraction in describing sequential systems. This notion is not explicitly used later, so Section 3.3 may be skipped.

Section 4 explores system factorizations in more detail, laying out basic laws from which transformations are composed. The parameters in factorization are discussed.

Section 5 reviews the derivation and implementation of a garbage collector. This example shows how the basic algebra presented in this paper applies

to a non-trivial design exercise. The accompanying figures (5–1 through 5–6) are developed directly from representations used by the transformation system. It is the form of these figures, not their content, that is of interest. They are included to convey the sense of engineering in this syntactic framework. Boyer gives details of the collector's design [4].

The garbage collector's derivation is targeted to a bit-slice implementation in PAL components. Repeated factorizations lead to an appropriate physical decomposition and bring a substantial portion of the design into programmable technology. This derivation strategy also bypasses the problem of incorporating representations in higher level descriptions. This issue is discussed in the conclusion.

1. The Relationship to Conventional Synchronous Design

Synchronous design isolates the algorithmic aspect of a problem—often specified as a finite-state machine, flowchart, or procedure—and refines a preliminary architecture once control is understood. We view this as a translation between dialects of recursive expression, a perspective that is projected in the ideogram

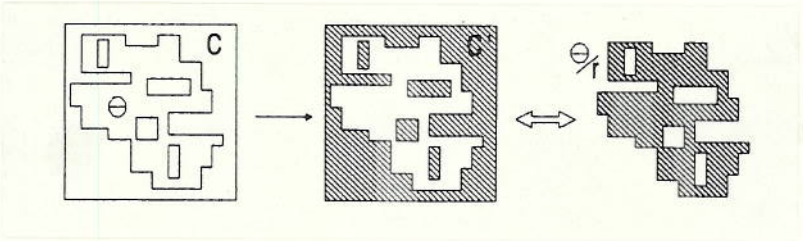
$$S_{\Theta/\Gamma} \longrightarrow I_{\Theta/\Gamma} \longrightarrow C_{\Theta/\Gamma} \longrightarrow C'_{\Gamma} \| \Theta_{\Gamma} \longrightarrow \cdots$$

This formula expresses nothing formal beyond the notion of transformation. S is a problem specification, which for us is a system of function definitions. S is expressed in terms of a vocabulary, Θ/Γ, of primitive constants, operations, predicates, and so forth, called the *basis*, or "ground type." The basis is an aggregate of complex objects (Θ) and concrete objects (Γ). Some examples are arrays of integers, queues of characters, sets of points. The complex/concrete distinction is subjective and hierarchical; Γ represents an intended level of description.

The first engineering task is transforming S to a form conducive to hardware implementation. Previous papers develop this phase of design derivation [12, 14, 13]. I stands for "*iterative*"—that class of recursion schemata that characterizes sequential control. In software, $S \rightarrow I$ is sometimes called "recursion removal," but implementing recursion in Θ is not precluded. An explicit treatment of control is one argument for the power of a functional algebra, but that is apart from the subject here. In this paper, all specifications are iterative to begin with.

C is a sequential-system description. We have shown that such descriptions are mechanically derivable from iterative specifications [14]. The transformation $I \rightarrow C$ is implemented and this phase of design derivation is automatic. C has the same ground type as I, but its interpretation is "behavioral." Where before, variables ranged over values, in C they denote sequences of values. We are interested in other interpretations of C as well, such as its description of physical connectivity or graphical layout [16, 3].

C is an abstract description because its sequences hold complex objects. It is refined to a more concrete description by factoring Θ as an *abstract component*. The formula $C' \| \Theta$ is meant to suggest a communicating system. C' is subject to further refinement and Γ to further decomposition. What makes this kind of transformation a factorization is that terms inside the *sub-system* Θ may be retained in C'.



One purpose of a *system factorization* is to segregate problem dependent concrete terms from terms that are intrinsic to the type abstraction. The entailed analysis may depend both on the structure of the basis and also on prior assumptions of how Θ is implemented. Our present concern is exploring how to parameterize syntactic transformations in support of such analysis.

For the purpose of contrast, we would describe standard synchronous design methods as

$$I_\Gamma \| \Theta_\Gamma \longrightarrow C'_\Gamma \| \Theta_\Gamma \rightarrow \cdots$$

The engineer makes an estimate of the architecture required to solve a problem, then develops control algorithm I to govern that architecture. A hardware realization of control is then compiled or systematically derived; that is, C' is Θ's controller. Communication is implicit in the control specification

language. For example, in finite-state [10] and ASM [22] notations, variables denote the presence of a value *now*; and this is also true in higher level hardware description languages.

The basis Θ/Γ of a functional specification corresponds to the estimate of architecture. The initial transformations secure the global design description as control is isolated. That is, a correct description of connectivity is maintained as the control is incorporated in architecture. However, complex (i.e., complex typed) behaviors arise in descriptions thus derived. Factorizations isolate concrete terms that are actually implemented from those that serve to preserve the global coherence of a description. It is a thesis of this work that many aspects of digital engineering can be addressed in this manner.

2. Notation and Background

Descriptions are built from applicative *terms*, such as $f(x, \mathsf{c})$, where operation symbols like f and constant symbols like c come from a given basis and x is a variable. Functional values are allowed; the lambda-expression $\lambda\mathcal{X}.\mathcal{E}$ denotes the function with rule $\mathcal{E}$ and parameter $\mathcal{X}$. In function definitions it is conventional to suppress the lambda, writing

$$\mathit{AndNot}(u, v) \stackrel{\mathrm{d}}{=} \mathit{and}(u, \mathit{not}(v))$$

instead of

$$\mathit{AndNot} = \lambda(u, v)\ .\ \mathit{and}(u, \mathit{not}(v)).$$

Function names are written in *slanted* font, variables in *italic* font and constants in sans serif font. The names of defined functions, such as *AndNot*, are capitalized. Product formation is expressed with brackets. For instance, we might describe a multiple output function

$$\mathit{BitSum}(a, b, c_{\mathrm{in}}) \stackrel{\mathrm{d}}{=} [s, c_{\mathrm{out}}] \quad \text{where} \quad \begin{cases} s = \mathit{parity}(a, b, c_{\mathrm{in}}) \\ c_{\mathrm{out}} = \mathit{majority}(a, b, c_{\mathrm{in}}) \end{cases}$$

A *recursion equation* is a simultaneous system of function-defining equations. There are examples throughout this paper. Function definitions usually involve *conditional* expressions, sometimes written as

$$P \rightarrow E,\ E'$$

and sometimes as case-statements such as

$$\begin{array}{rl} \textit{per instruction viz} & \mathsf{Q} : E_1 \\ & \mathsf{R} : E_3 \\ & \mathsf{W} : E_2 \end{array}$$

This form expresses a simple selection; the *per*-expression is always a variable and the case labels are always constants.

An expression is *simple* if it contains no recursively defined function symbols; otherwise it is *recursive*. A simple function definition is called a *combination*. A recursive term is *tail-recursive* if its only defined symbol is in the outermost applied position. A recursion equation is *iterative* if each branch of every conditional is either simple or tail-recursive. All but one of the examples in this paper are iterative.

Circuits are also described with applicative expressions, but the interpretation is sequential. The *signal* $\boxed{f}(X, \boxed{\mathsf{c}})$ denotes the sequence S in which $S_i = f(X_i, \mathsf{c})$. The boxes are a convention to distinguish the sequential interpretation and are omitted later. Signal variables are upper case.

Abstraction is extended so that, for example

$$\boxed{AndNot}(A, B) \equiv \boxed{and}(A, \boxed{not}(B))$$

and

$$\boxed{\lambda(u, v) \,.\, f(u, g(u, v))}(X, Y) \equiv \boxed{f}(X, \boxed{g}(X, Y))$$

The two-way selector $\boxed{if}$ is a sequential conditional.

$$\text{If} \quad X = \boxed{if}(P, S, S') \quad \text{then} \quad X_k = \begin{cases} S_k, & \text{if } P_k \text{ is true,} \\ S'_k, & \text{if } P_k \text{ is false.} \end{cases}$$

The operator '!' prefixes a value to a signal. It expresses storage or delay.

$$\text{If} \quad X = \check{v} \,!\, S \quad \text{then} \quad X_0 = \check{v} \text{ and } X_{k+1} = S_k.$$

A *system description* is a collection of simultaneous signal-defining equations. For example,

$$\begin{array}{l} \mathit{Toggle}(C, T) \stackrel{\mathrm{d}}{=} Q \\ \quad \text{where} \\ \qquad Q = \phi \,!\, \boxed{if}(C, \boxed{\mathsf{false}}, Z) \\ \qquad Z = \boxed{if}(T, \boxed{not}(Q), Q) \end{array}$$

describes a storage element that is cleared to false whenever signal C is true and otherwise inverts its content whenever signal T is true. The symbol ϕ stands for an unspecified value; it can mean "don't know" or "don't care" or "don't care to say," depending on the context. In *Toggle* $\phi \in \{\text{true, false}\}$. According to the sequential interpretation, the recursively defined signal Q is given by

$$Q_0 = \phi \quad \text{and} \quad Q_{k+1} = \begin{cases} \text{false} & \text{if } C_k = \text{true} \\ Z_k & \text{if } C_k = \text{false} \end{cases}$$

$$Z_k = \begin{cases} not(Q_k) & \text{if } T_k = \text{true} \\ Q_k & \text{if } T_k = \text{true} \end{cases}$$

The left-hand sides of signal equations may be groupings when the defining expression involves multiple-output operations. For example, one might describe an adder using the *BitSum* combination:

$$Adder([A_n \ldots A_0],\ [B_n \ldots B_0],\ C_0) \stackrel{\text{d}}{=} [C_{n+1}\ S_n\ \cdots\ S_0]$$

$$\text{where} \quad [C_1,\ S_0] = BitSum(A_0, B_0, C_0)$$

$$\vdots$$

$$[C_{n+1},\ S_n] = BitSum(A_n, B_n, C_n)$$

The *Adder* combination is "combinational" and could stand in either the discrete or sequential interpretation (From here on, the boxes are omitted). Our system currently has no means to recognize ellipses, or to understand making n a parameter in *Adder*. Such constructs are desirable and are used throughout this paper.

At the core of this work is a proposition relating recursion equations to system descriptions. If $\mathcal{P}$, $\mathcal{T}$, and $\mathcal{R}_i$ are all simple expressions, the function definition

$$\mathbb{F}(x_1, \ldots, x_n) = \mathcal{P} \rightarrow \mathcal{T}, \mathbb{F}(\mathcal{R}_1, \ldots, \mathcal{R}_n)$$

is equivalent to the system description

$$\mathbf{C}(\check{x}_1, \ldots, \check{x}_n) \stackrel{\text{d}}{=} [P\ T]$$

$$\text{where} \quad X_1 = \check{x}_1\ !\ \mathcal{R}_1$$

$$\vdots$$

$$X_n = \check{x}_n\ !\ \mathcal{R}_n$$

$$P = \mathcal{P}$$

$$T = \mathcal{T}$$

That is, in $\mathbf{C}(v_1, \ldots, v_n)$ the signal T contains $\mathbf{F}(v_1, \ldots, v_n)$ as soon as P contains true [14]. Thus, a function with $\mathbf{F}$'s form *transliterates* to a sequential-system description with the same terms.

The system $\mathbf{C}$ has the characteristics of externally clocked synchronous hardware: all combinational feedback passes through storage. Under minimal assumptions, an instance of $\mathbf{C}$ can be constructed from any iterative recursion equation. Thus, iterative schemata serve as a conservative specification language for synchronous-system designs.

3. Derivation of a Single Pulser

The *Single Pulser* is from a textbook by Winkel and Prosser on digital design [22]. This first example of design derivation is presented to give an intuitive idea of the algebra and to show how implementations are assembled from lower level system descriptions.

A second purpose is to develop a treatment of communication. In the general system description $\mathbf{C}$ at the end of Section 2, the "ready" signal P is scant acknowledgment that circuits coordinate with external behaviors. The Single Pulser, being almost purely communicative, sheds a harsh light on that characterization, making it plain that more expressiveness is needed. We attack the problem by introducing communication as an attribute of the ground type. This approach is self serving because it is on the abstraction of communication that we illustrate the notion of system factorization.

We begin with a specification of a two state process that outputs a unit pulse for every pulse on its synchronized input i.

$$\begin{aligned} ACK(i) = now?(i) \;\rightarrow\; & put(\mathsf{true}, NAK(next(i))), \\ & put(\mathsf{false}, ACK(next(i))) \\ NAK(i) = now?(i) \;\rightarrow\; & put(\mathsf{false}, NAK(next(i))), \\ & put(\mathsf{false}, ACK(next(i))) \end{aligned} \tag{1}$$

Let us call the basis of this specification PORT, whose intended model is boolean communication, and let P be its domain of values. Input involves two operations: the predicate $now?: P \rightarrow \{\mathsf{true}, \mathsf{false}\}$ gives the current value and $next: P \rightarrow P$ gives subsequent behavior. $put: \{\mathsf{true}, \mathsf{false}\} \times P \rightarrow P$ builds output. Two laws for PORT, used later, are $now?(put(b, p)) \equiv b$ and $put(now?(p), next(p)) = p$.

Each branch of ACK and NAK performs input and output exactly once. This is essential since (1) is to be considered a specification of synchronous control.

3.1. Construction of a System Description

In order to construct a system description, an iterative specification is is needed. Let us abuse the notation by moving the *puts* inside the calls to *ACK* and *NAK*. The assertion "$\{now?(o)\}$" stands as a reminder that this was done.

$$\begin{aligned} ACK(i,o) &= \{now?(o)\}\ now?(i) \rightarrow NAK(next(i), put(\mathsf{true}, o)), \\ &\qquad\qquad\qquad\qquad\quad ACK(next(i), put(\mathsf{false}, o)) \\ NAK(i,o) &= \{now?(o)\}\ now?(i) \rightarrow NAK(next(i), put(\mathsf{false}, o)), \\ &\qquad\qquad\qquad\qquad\quad ACK(next(i), put(\mathsf{false}, o)) \end{aligned} \tag{2}$$

A single function is built by first introducing a parameter, $c \in \{\mathsf{ack}, \mathsf{nak}\}$, to indicate whether *ACK* or *NAK* is in control; then extracting a combination for the conditional; and distributing it across recursive calls. This transformation is discussed in [12], [13], and [14], with many examples; it is an instance of Harel's compendium of folk theorems [9]. The result is

$$\begin{aligned} &Select(p,q,v_0,v_1,v_2,v_3) \stackrel{\mathrm{d}}{=} per\ p\ viz\ \mathsf{ack} : q \rightarrow v_0,\ v_1 \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{nak} : q \rightarrow v_2,\ v_3 \\ &SP(c,i,o) = \{now?(o)\} \\ &\quad SP\big(Select(c, now?(i), \mathsf{nak}, \mathsf{ack}, \mathsf{nak}, \mathsf{ack}), \\ &\qquad Select(c, now?(i), next(i), next(i), next(i), next(i)), \\ &\qquad Select(c, now?(i), put(\mathsf{true}, o), put(\mathsf{false}, o), \\ &\qquad\qquad\qquad\qquad put(\mathsf{false}, o), put(\mathsf{false}, o))\big) \end{aligned} \tag{3}$$

For any i and o, $SP(\mathsf{ack}, i, o) \equiv ACK(i, o)$. Since *SP* is an instance of **C**, according to the characterization in Section 2 the corresponding sequential-system description is

$$\begin{aligned} &SP(\check{\imath}, \check{o}) = now?(O) \\ &\quad \text{where} \\ &\qquad C = \mathsf{ack}\ !\ Select(C, now?(I), \mathsf{nak}, \mathsf{ack}, \mathsf{nak}, \mathsf{ack}) \\ &\qquad I = \quad \check{\imath}\ !\ Select(C, now?(I), next(I), next(I), next(I), next(I)) \\ &\qquad O = \quad \check{o}\ !\ Select(C, now?(I), put(\mathsf{true}, O), put(\mathsf{false}, O), \\ &\qquad\qquad\qquad\qquad\qquad\qquad put(\mathsf{false}, O), put(\mathsf{false}, O)) \end{aligned} \tag{4}$$

The term $now?(O)$ replaces $[P,\ T]$ in the characterization. The signals clearly simplify, but it serves our purpose not to simplify them too much. I's

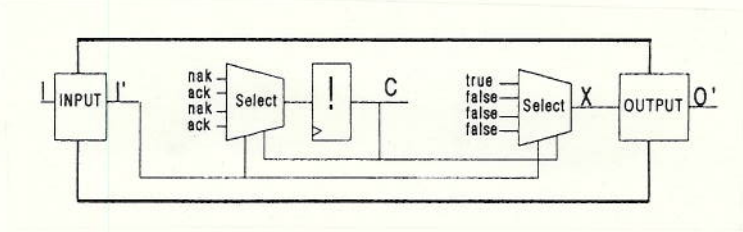
selector isn't needed and O's is needed only for *put*'s first argument.

$$\begin{array}{l} SP(\check{\imath},\check{o}) = now?(O) \\ \quad \text{where} \\ \qquad C = \mathsf{ack} \mathbin{!} Select(C, now?(I), \mathsf{nak}, \mathsf{ack}, \mathsf{nak}, \mathsf{ack}) \\ \qquad I = \quad \check{\imath} \mathbin{!} next(I) \\ \qquad O = \quad \check{o} \mathbin{!} put(X, O) \\ \qquad X = Select(C, now?(I), \mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{false}) \end{array} \tag{5}$$

I and O range over ports; they are complex signals that do not describe physical hardware. However, the surrounding system is concerned not with their content but only with the result of the operation *now?*. Therefore, let us encapsulate the defining expressions as combinations presenting the desired signals. In general, there are problem dependent subterms that should be retained, but the one instance in this case has already been identified as the signal X. Defining appropriate combinations *INPUT* and *OUTPUT* yields

$$\begin{array}{l} SP(\check{\imath},\check{o}) = O' \\ \quad \text{where} \\ \qquad C = \mathsf{ack} \mathbin{!} Select(C, I', \mathsf{nak}, \mathsf{ack}, \mathsf{nak}, \mathsf{ack}) \\ \qquad I' = INPUT(\check{\imath}) \\ \qquad O' = OUTPUT(\check{o}, X) \\ \qquad X = Select(C, I', \mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{false}) \\ \quad \text{and} \\ \qquad INPUT(\check{p}) \stackrel{\mathrm{d}}{=} now?(P) \text{ where } P = \check{p} \mathbin{!} next(P) \\ \quad \text{and} \\ \qquad OUTPUT(\check{p}, B) \stackrel{\mathrm{d}}{=} now?(P) \text{ where } P = \check{p} \mathbin{!} put(B, P) \end{array} \tag{6}$$

A schematic can be drawn from these equations:



Component abstractions *INPUT* and *OUTPUT* hide the complex behaviors over ports from the concrete boolean signals in (6), leaving a description that

could reasonably be implemented. This encapsulation is one kind of *system factorization.*

From the laws of PORT in the sequential interpretation, $INPUT(p) \equiv p$ and $OUTPUT(p, B) \equiv B$. Hence, we may rewrite (6) as

$$SP(I) = O \quad \text{where} \quad \begin{cases} C = \mathsf{ack} \,!\, Select(C, I, \mathsf{nak}, \mathsf{ack}, \mathsf{nak}, \mathsf{ack}) \\ O = Select(C, I, \mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{false}) \end{cases} \tag{7}$$

Description (7) is obtained more directly from the specification

$$\begin{aligned} ACK(i, o) = i \;&\rightarrow\; NAK(i, \mathsf{true}), \\ &\quad\; ACK(i, \mathsf{false}) \\ NAK(i, o) = i \;&\rightarrow\; NAK(i, \mathsf{false}), \\ &\quad\; ACK(i, \mathsf{false}) \end{aligned} \tag{8}$$

That is, the treatment of communication in PORT can be incorporated in the specification notation, as surely it should be. A refined semantics—at the very least a typing discipline—is needed to make sense of specifications that implicitly use PORT. However, this understanding is not needed in the transformation process; the same construction used to build (4) from (1) yields (7) from (8).

3.2. PAL Implementation of the Single Pulser

From (7), boolean equations are generated and targeted both to a PAL programmer and into VLSI layout facilities. Both (7) and the garbage collector in Section 5 are built with PALs. "PAL" stands for Programmable Array Logic, a general purpose logic device that also contains storage elements. Once programmed, a PAL circuit has the description

$$\begin{aligned} &\lambda(Y_1, \ldots, Y_n) \,.\, [X_1^c, \ldots, X_m^c, X_1^d, \ldots, X_r^d] \\ &\quad \text{where} \\ &\qquad X_1^c = \quad f_1(Y_1, \ldots, Y_n, X_1^c, \ldots, X_m^c, X_1^d, \ldots, X_r^d) \\ &\qquad\quad \vdots \\ &\qquad X_m^c = \quad f_m(Y_1, \ldots, Y_n, X_1^c, \ldots, X_m^c, X_1^d, \ldots, X_r^d) \\ &\qquad X_1^d = \phi \,!\, f_{m+1}(Y_1, \ldots, Y_n, X_1^c, \ldots, X_m^c, X_1^d, \ldots, X_r^d) \\ &\qquad\quad \vdots \\ &\qquad X_r^d = \phi \,!\, f_{m+r}(Y_1, \ldots, Y_n, X_1^c, \ldots, X_m^c, X_1^d, \ldots, X_r^d) \end{aligned}$$

Each f_i is a boolean function subject to certain restrictions, such as the number of or's it can do. The values of n, m, and r vary in different packagings. Outputs X^c are combinational and signals X^d are stored in clocked, D-type flip-flops. A PAL "program" is a description of the functions f_i.

Recall that the selector combination for the single pulser is

$$\begin{aligned} Select(p, q, v_0, v_1, v_2, v_3) \stackrel{\mathrm{d}}{=} \textit{per } p \textit{ viz } & \mathsf{ack} : q \rightarrow v_0,\ v_1 \\ & \mathsf{nak} : q \rightarrow v_2,\ v_3 \end{aligned}$$

The method for implementing (7) is to generate a state assignment for each selection branch. This is a standard synthesis technique following [22]. The state associated with alternative v_i is the binary value of i; represented in signals $[S_1, S_0]$. Boolean equations are developed in conjunction with an assignment of truth values for the symbolic constants. For (7), with nak represented as true, the following is essentially the input generated for the A+PLUS simplifier [1].

$$\begin{aligned} C^d &= (\overline{S}_1 \wedge \overline{S}_0) \vee (S_1 \wedge \overline{S}_0) \\ O^d &= \overline{S}_1 \wedge \overline{S}_0 \\ S_1^c &= (C \wedge \overline{I}) \vee (C \wedge I) \\ S_0^c &= (\overline{C} \wedge \overline{I}) \vee (C \wedge \overline{I}) \end{aligned}$$

The simplifier reduced the system to

$$\begin{aligned} S_0^c &= \overline{I} \\ S_1^c &= C \\ O^d &= \overline{S}_1 \wedge \overline{S}_0 \\ C^d &= \overline{S}_0 \end{aligned}$$

This is directed to the Altera LE-2 fuse burner to obtain a working single pulser in an Altera EP310 PAL [1].

The simplifier is understandably reluctant to remove signals, but the equations could clearly reduce further to

$$SP(I) = O \quad \text{where} \quad \begin{cases} C = \mathsf{ack}\ !\ I \\ O = and(I, not(C)) \end{cases} \tag{9}$$

as was the outcome in Winkel's and Prosser's text. Version (9) might be developed by a direct symbolic expansion of (7). We have not implemented this kind of analysis and make do with a state encoding passed through available simplifiers.

3.3. A Comment about Modeling

Since the ideas discussed below are not explicitly used later, this section can be skipped or read lightly.

The Single Pulser presents an opportunity to discuss the use of functional abstraction in bridging levels of hardware description. The underlying principle is that describing something as a function is the purest form of specification, carrying no suggestion of representation. A more complicated mathematical foundation is needed if functions are allowed in a *basis,* and Boute makes credible claims that functions aren't needed in this way [3]. Further examination of the issue is warranted; it is too early to preclude the free use of functions in hardware modeling.

The version of SP below uses $AndNot$, from Section 2, to suppress details of O's defining equation in (9).

$$SP(I) = O \text{ where } \begin{cases} C = \phi \,!\, I \\ O = AndNot(I, C) \end{cases} \tag{10}$$

Now, consider $AndNot$'s truth table with the idea of abstracting its second argument to a function on the first.

u	v	$AndNot(v,u)$	$u: v \to \{\text{true, false}\}$
true	true	false	$u(v) = \text{false}$
true	false	false	
false	true	true	$u(v) = v$
false	false	false	

The right-most column shows that the functional abstraction of u is

$$Abstr(u) \stackrel{\text{d}}{=} u \to (\lambda v.\text{false}),\ (\lambda v.v)$$

The version of SP below uses the abstraction, treating signal C as a component whose function is governed by I.

$$SP(I) = O \text{ where } \begin{cases} C = \phi \,!\, Abstr(I) \\ O = C(I) \end{cases} \tag{11}$$

For this system description to be meaningful, it must make sense to apply signals. Let us revise the sequential interpretation of application to allow it.

$$\text{If} \quad X = F(\cdots S^i \cdots) \quad \text{then} \quad X_k = F_k(\cdots S^i_k \cdots).$$

```
                                    |-----------------------
L = \[v].v                          | Voltage abstractions
H = \[u].\[v].v                     |-----------------------
                                    | Symbolic interface
repr = \[A]. (A:<\v."L">):<"H">     |  repr: value --> name
REPR = <repr *>                     |  REPR: values --> names
                                    |-----------------------
SP = \I. <H ! I>:<I>                | SP(I) = (delay I)(I)
                                    |-----------------------

REPR:< SP:<L L L H H H H L L H L L H H L L H H H L L L> >
          [L L L H L L L L L H L L H L L L H L L L L L ■]
```

The language is Daisy, a lazy dialect of Pure Lisp, whose use for hardware description is discussed in [14,16]. Sequences are represented as stream-like lists. `<* ! *>`, a CONS-expression, evaluates to the list `[* ! *]`; ':' means application; '\' is a lambda. REPR maps repr over a list. SP, taken from equation (12), maps I delayed over I.

FIGURE 3–1. EXECUTABLE DESCRIPTION OF SP

The interpretation of a base operation is then a constant sequence

$$\boxed{f} = (f, f, \ldots)$$

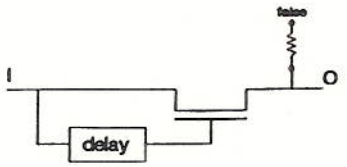
The values $\{(\lambda v.v), (\lambda v.\text{false})\}$ suggest a pass transistor with a pull-false resistor. If I were already abstracted, and taking

$$Delay_x(I) \stackrel{d}{=} x \,!\, I$$

we get

$$SP(I) = (Delay_\phi\ I)(I) \tag{12}$$

Hence, we might consider implementing SP in NMOS as



However, it remains to solve for independent basis of values, in order to eliminate the false in $\{(\lambda v.v), (\lambda v.\mathsf{false})\}$. A set that works is $\{(\lambda v.v), (\lambda u\,.\,\lambda v.v)\}$ as is demonstrated by the program in Figure 3–1. Whether this is a useful transistor model is a matter for study. Also of interest is the manner by which a logic description transforms to a gate description, and in general, the role of function abstractions in developing such transformations.

4. Factorization of System Descriptions

A system factorization encapsulates a subsystem in order to remove some collection of operations from the surrounding description. This is called "abstract component factorization" in [12] referring to the use of abstract data types in decomposing software systems into information hiding modules. After building a basic vocabulary of elementary transformations, we look at a small example to get a sense of what a factorization does.

Identification is giving a name to an expression by adding a signal equation for it. Identification of like terms by a single equation has the effect of eliminating redundant circuitry when the occurrence of a term corresponds to an instance of physical hardware.

Grouping is the collection of terms into a single equation. A set of equations $\{X_i = S_i\}$ may be rewritten as

$$[\cdots X_i \cdots] = [\cdots S_i \cdots]$$

Grouping is sometimes done to develop physical decompositions. The expression $[\cdots S_i \cdots]$ might describe a physical package; recall the n-bit *Adder* in Section 2, in which *BitSum* combinations stand for groupings.

As before, *combination* is the formation of a lambda-combinator, or "macro," or "repeated-pattern symbol" in drafting terminology. Examples are *AndNot*, *BitSum*, and *Adder* in Section 2; *Select*, *INPUT*, *OUTPUT*, and *SP* in Section 3.

Distribution is the symbolic application of a distributive law. A key instance is the law for selection that says

$$if[p, f(x), g(y)] \quad \equiv \quad (if[p, f, g])\,(if[p, x, y])$$

This rule extends to *Select* combinations—it was used in equation (5) of Section 3.1 to get the signal X— but we shall explore its use with the binary selector, as above.

Generalization is the introduction of extraneous don't-care arguments. It is typically done in order to distribute selection over non-uniform alternatives.

$$if[p,\ f(x,y),\ g(u,v,w)]$$

is rewritten as

$$(if[p,f',g])(if[p,x,u],\ if[p,y,v],\ if[p,\phi,w])$$

with f extended to

$$f'(a,b,c) \stackrel{\mathrm{d}}{=} f(a,b).$$

Collation is a merging of signals by instantiating ϕs. An example is to replace

$$\left\{\begin{matrix} X = if[p,s,\phi] \\ Y = if[p,\phi,t] \end{matrix}\right\} \quad \text{with} \quad \{XY = if[p,s,t]\}$$

We sketch this elementary algebra in a small example involving two signals with a common selection predicate.

$$\begin{aligned} U &= if[p,\ add(A,B),\ inc(C)] \\ X &= if[p,\ dcr(D),\ add(E,F)] \end{aligned} \tag{1}$$

The operations' names suggest arithmetic functions and it is assumed that all signals enjoy a uniform representation. The goal is to reduce the number of components by distributing selection in the system. With this in mind, (1) is expanded to

$$\begin{aligned} U &= if[p,V,W] \\ V &= add(if[p,A,\phi],\ if[p,B,\phi]) \\ W &= inc(if[p,\phi,C]) \\ X &= if[p,Y,Z] \\ Y &= dcr(if[p,D,\phi]) \\ Z &= add(if[p,\phi,E],\ if[p,\phi,F]) \end{aligned} \tag{2}$$

We get a single *add* by collating V and Z.

$$\begin{aligned} U &= if[p,\ VZ,\ W] \\ W &= inc(if[p,\phi,C]) \\ VZ &= add(if[p,A,E],\ if[p,B,F]) \\ X &= if[p,Y,\ VZ] \\ Y &= dcr(if[p,D,\phi]) \end{aligned} \tag{3}$$

The arguments to *inc* and *dcr* also merge and we can express W and Y as a single signal

$$WY = (if[p, dcr, inc])(if[p, D, C])$$

The component-expression $if[p, dcr, inc]$ is represented in concrete terms by encoding *dcr* and *inc* in an *instruction signal* ranging over symbols $\{\mathsf{up}, \mathsf{dn}\}$.

$$\begin{aligned} U &= if[p,\ VZ,\ WY] \\ X &= if[p,\ WY,\ VZ] \\ VZ &= add(if[p, A, E],\ if[p, B, F]) \\ WY &= UPDN(if[p, \mathsf{dn}, \mathsf{up}],\ if[p, D, C]) \end{aligned} \tag{4}$$

Instructions are interpreted by a synthesized abstraction of *WY's* behavior.

$$\begin{aligned} UPDN(I, S) \stackrel{\mathrm{d}}{=} \textit{per } I \textit{ viz } & \mathsf{up}\text{: } inc(S) \\ & \mathsf{dn}\text{: } dcr(S) \end{aligned}$$

Of course, there are alternative decompositions. For instance, one could introduce a component abstraction for each of U and X. Unary operations in (1) are generalized to get a uniform actual parameter:

$$\begin{aligned} U &= if[p,\ add(A, B),\ inc'(C, \phi)] \\ X &= if[p,\ dcr'(D, \phi),\ add(E, F)] \end{aligned} \tag{5}$$

and two combinations are introduced, with synthesized instruction signals.

$$\begin{aligned} U &= ADDINC(if[p, \mathsf{add}, \mathsf{up}],\ if[p, A, C],\ B) \\ X &= DCRADD(if[p, \mathsf{dn}, \mathsf{add}],\ if[p, D, E],\ F) \end{aligned}$$

where

$$\begin{aligned} ADDINC(I, S, T) \stackrel{\mathrm{d}}{=} \textit{per } I \textit{ viz } & \mathsf{add}\text{: } add(S, T) \\ & \mathsf{up}\text{: } inc'(S, T) \end{aligned} \tag{6}$$

and

$$\begin{aligned} DCRADD(I, S, T) \stackrel{\mathrm{d}}{=} \textit{per } I \textit{ viz } & \mathsf{dn}\text{: } dcr'(S, T) \\ & \mathsf{add}\text{: } add(S, T) \end{aligned}$$

Abstract components *UPDN*, *ADDINC*, and *DCRADD* are *system factorizations*. These kinds of transformations are motivated by a variety of

global considerations, such as physical packaging, routing, and logical decomposition. One function of the transformation system is to execute the steps of a factorization correctly. The process has several implicit steps, including

(A) Determining a set of *subject terms.*

(B) Isolating these terms in the surrounding description.

(C) Executing some collation tactic.

(D) Generating a symbolic instruction signal.

(E) Synthesizing a correct component specification.

(F) Incorporating the component in the description.

The subject terms are those that apply an operation which is to be encapsulated. We have implemented two ways of doing (A). The first, called a *general factorization,* is to state the set of operations that are to be encapsulated. The subject terms are those in which one of the set is applied. The second, called a *signal factorization,* encapsulates a signal; in this case the subject terms are those in which that signal's name occurs as an argument.

Descriptions (3) and (4), above, are general factorizations. In (3), the set of subject terms are those using *add*; in (4), subject terms develop from the set $\{inc, dcr\}$. Description (6) results from two general factorizations of (1), each restricted to a single signal equation. Subject terms are determined by $\{add, inc\}$ for signal U and $\{add, dcr\}$ for signal X.

In a signal factorization, the encapsulated operations are those applied to a named signal. The combinations *INPUT* and *OUTPUT* in Section 3.1 are examples. The purpose is often to isolate (more) concrete signals from (more) complex signals. Subject terms $\{d_i(T, \ldots)\}$ are identified and grouped with the signal of interest, which typically has the form

$$T = \check{t}\,!\; Select(\cdots c_j(T, \ldots) \cdots)$$

If the basis is Θ/Γ, then T has type Θ; the operations c_j: $(\Theta \times \Gamma^r) \rightarrow \Theta$, inside T's equation, build new complex values; and those d_i: $(\Theta \times \Gamma^n) \rightarrow \Gamma^m$ in the surrounding system extract concrete values. The component abstraction encloses all references to and operations on T, taking an appropriate instruction signal as an input:

$$\Theta(\check{t}, I, \ldots) \stackrel{\mathrm{d}}{=} [\cdots d_i(T, \ldots) \cdots]$$
$$\text{where } T = \check{t}\,!\ \textit{per } I \textit{ viz } \cdots \text{ instruction}_j\text{: } c_j(T, \ldots) \cdots$$

Θ specifies how *any* implementation must behave in relation to the surrounding circuit. However, Θ is itself an abstract behavioral description. We do not derive implementations from such specifications and would like a compatible means of verification for establishing the correctness of implementations.

The factorizations now done by the transformation system are entirely syntactic and depend on convention. Factorization steps (A) through (F) raise theoretical and analytical issues. For example, generalization depends on assumptions of type-compatibility among the subject operations. It is not always possible to form reasonable combinations from a set of subject terms, due to clashes in collation. Factorizations are often motivated by unstated assumptions about the implementation. Despite these problems, we find this way of manipulating design descriptions to be natural and powerful. Fairly simple factorizations seem adequate for decomposing designs at levels of description that are typical in digital design.

5. Derivation of a Garbage Collector

This section sketches a non-trivial design exercise that has been carried to hardware. The accompanying figures are intended to be *impressionistic* and are included mainly to give a sense of the derivation process. Figures 5–1 through 5–5, at the end of this paper, are developed directly from the representations used in the transformations system, but they are highly condensed. They convey little of the logical design, whose details are explained by Boyer [4]. The transformation system is due to Bose [2].

The derivation strategy is governed by architectural considerations. The design is targeted for a bit-slice implementation in PALs. We have the means to prototype quickly in this technology, using a Logic Engine [17], which provides a large wire-wrap panel, secure power and clocking, and good digital display facilities. In addition, the Logic Engine houses a general purpose microprogramming environment, which is used to implement the peripheral system needed to exercise the design. The collector has been tested against heaps generated by a Scheme implementation [5].

PALs have a good switching capacity, and a bit-slice implementation of the collector eliminates the need for an internal register bus. On the other hand, bit-slicing in PALs precludes a direct implementation of arithmetic, due to intolerable propagation delays. Hence, conventional arithmetic components are used in the design.

Figure 5–1 is a specification for a *stop-and-copy* garbage collector [7]. This kind of collector divides memory into two semispaces, only one being

active at any time. When the allocator exhausts available space, the collector copies and compresses the heap image into the inactive space. The roles of the two semispaces are then exchanged. The specified collector compresses a heap that includes binary list cells, vectors, and relocatable segments of data; there is also a provision for bypassing non-relocatable segments. Details of the collection algorithm and representations of collected objects are not important to what follows. They are explained in [4]. Relevant details can be seen in the following portion of Figure 5–1.

$$\begin{aligned} &DRIVER(M_1, M_2, H, D, C, U, A, GO, R) = \\ &\quad eq?(U, A) \rightarrow IDLE(M_2, M_1, H, D, C, U, A, GO, \text{true}), \\ &\qquad\qquad NEXT(M_1, M_2, read(M_1, U), D, C, U, A, GO, R) \end{aligned}$$

The formal argument describes the "registers" at this level of description. Parameters M_1 and M_2 have type MEMORY and are subject to operations $read(*,*)$ and $write(*,*,*)$. The design uses two distinct physical memories for the semispaces, and operates simultaneously on both during collection. Registers H and D each hold a CONTENT, with field manipulations $ptr(*)$, $tag(*)$, and $cell(*,*)$. C, U, and A are of type ADDRESS with arithmetic operations $inc(*)$, $dcr(*)$, $add(*,*)$, $btow(*)$, $addinc(*,*)$, and $incadd(*,*)$. The *btow* operation rounds byte offsets to word boundaries; *addinc* and *incadd* are combinations of addition and incrementing. The remaining parameters, GO and R, are boolean communication ports. Ports are treated in the manner of Section 3.1.

Figure 5–1 is an unstructured register-transfer description. Each recursive function is a point of control, whose invocation could be thought of as a parallel assignment to the registers and a transfer of control. This suggests that the *NEXT* branch in *DRIVER* literally exchanges M_1 and M_2. Since this is unreasonable in standard technology, the alternative used in this design makes explicit the fact that memory paths are switched, and not memories themselves. A boolean register W records which memory is active. For instance, *DRIVER* becomes

$$\begin{aligned} &DRIVER(M_1, M_2, H, D, C, U, A, GO, R, W) = \\ &\quad eq?(U, A) \rightarrow \\ &\qquad IDLE(M_1, M_2, H, D, C, U, A, GO, \text{true}, not(W)), \\ &\qquad W \rightarrow \\ &\qquad\quad NEXT(M_1, M_2, read(M_2, U), D, C, U, A, GO, R, W), \\ &\qquad\quad NEXT(M_1, M_2, read(M_1, U), D, C, U, A, GO, R, W) \end{aligned}$$

and similarly for the whole of Figure 5–1. Since all memory operations are predicated by a test on W, the size of the specification is nearly doubled. The

manner of introducing W reflects the planned implementation. W might be seen as an input to a TWOMEMORY component abstraction and would then be added as an argument to *read* and *write*. However, the PALs provide enough gates to switch the data paths internally. Making W a selection predicate leads to this implementation.

The design's selector, *SEL*, in Figure 5–2, and the initial system description, in Figure 5–3, are both mechanically derived. The grouping of predicates, called *STATUS* in Figure 5–2, was done manually to condense Figure 5–3. This combination is automatically encapsulated later in the derivation process.

Five system factorizations are applied to Figure 5–3, resulting in Figures 5–4 and 5–5. These are signal factorizations of M_1, M_2, and C; and two general factorizations of arithmetic. Subject terms for one of the general factorizations are indicated by boxes in Figure 5–3. The transformation system must correctly draw these terms into a single combination and synthesize an abstract component specification.

Factorization of signal M_1 (and M_2 similarly) leaves residual signals $M_1.I$, $M_1.A$, and $M_1.D$ (instruction, address, and data).

$$M_1 = MEMORY(\phi, M_1.I, M_1.A, M_1.D)$$

$$M_1.I = SEL(P, \mathtt{@}, \mathtt{@}, \mathtt{@}, \mathtt{@}, \mathtt{R}, \mathtt{R}, \mathtt{@}, \mathtt{@}, \mathtt{@}, \mathtt{@}, \mathtt{W}, \mathtt{W}, \mathtt{W}, \mathtt{@}, \mathtt{W}, \mathtt{@}, \mathtt{W}, \mathtt{@}, \mathtt{R}, \mathtt{W}, \mathtt{@}, \mathtt{W}, \mathtt{R}, \mathtt{W}, \mathtt{W}, \mathtt{@}, \mathtt{R}, \mathtt{W}, \mathtt{R}, \mathtt{W}, \mathtt{W}, \mathtt{@}, \mathtt{R}, \mathtt{W})$$

$$M_1.A = SEL(P, \phi, \phi, \phi, \phi, U, H, \phi, \phi, \phi, \phi, U, H, A, \phi, U, \phi, A, \phi, Z_2, U, \phi, A, Z_1, A, H, \phi, Z_2, Z_1, Z_1, U, H, \phi, Z_2, Z_1)$$

$$M_1.D = SEL(P, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, cell(H, D), cell(\mathsf{fwd}, A), D, \phi, cell(H, A), \phi, D, \phi, \phi, cell(H, A), \phi, D, \phi, D, cell(\mathsf{fwd}, A), \phi, \phi, D, \phi, cell(H, A), cell(\mathsf{fwd}, A), \phi, \phi, D)$$

These connect to the abstract component *MEMORY*, whose description in Figure 5–5 is a byproduct of the factorization. Instructions to a *MEMORY* are @ (do nothing), R (read), and W (write).

$$MEMORY(\check{m}, Inst, Addr, Data) \stackrel{d}{=} read(M)$$
where
$$M = \check{m}\,!\,Interpret(M, Inst, Addr, Data)$$
and
$$Interpret(m, inst, addr, data) \stackrel{d}{=} per\ inst\ viz\ \begin{array}{l}\mathtt{@} : m \\ \mathtt{R} : m \\ \mathtt{W} : write(m, addr, data)\end{array}$$

Factorization of signal C encapsulates the combinational operations on register C; these constitute a counter.

Next, two general factorizations of arithmetic yield three abstract components. The operation *btow* is hidden in a component *BYTETOWORD*. Finally, subject terms are developed from the set

$$\{add, inc, dcr, addinc, incadd\}.$$

As is illustrated in Section 4, generalizations of the subject terms must be collated into signal equations. In this case, the attempt to collate leads to clashes, which means that more than one adder is needed. When this happens, the transformation system must partition the subject terms to generate additional component abstractions. Partitioning is interactive because it involves tactical design considerations. For example, the residual signal $Z_2.B$ in Figure 5–4 simplifies to C as a result of the partitioning choice.

Two issues of concern in this work are the degree to which specifications must anticipate factorizations, and the effort needed to revise descriptions when factorizations don't satisfy engineering intentions. More serialization of addition might be coded in the specification, but two adders are tolerated because the intent is to keep the memories busy. On a design of this size, it is reasonable to experiment rather than to analyze. In fact, the first derivation attempt led to *three* adders, and the specification was subsequently changed. Re-execution of the derivation went through quickly. A more timely serialization, through transformations on Figure 5.3 for example, is worth study but is beyond the current capabilities of our system.

Subsequent algebra is directed toward a physical organization and consequently toward a boolean representation of the symbolic basis. One reason for directing this derivation toward a bit-slice implementation is the lack of mechanisms for imposing representations. The derivation strategy minimizes the impact of this gap in automation. This is a central issue in the continuation of this work, which we discuss briefly in Section 6.

Figure 5–4 is a description in which all substantial operations are factored out. The remaining equations are in terms of selection and record manipulation, and the latter are realized implicitly in the architecture. For example, the operation *ptr* is simply the extraction of a 24-bit ADDRESS field from a 32-bit CONTENT. Each bit slice represents the projection of an n-bit quantity to a single bit. That is, the signals in Figure 5–4, excluding Q, can be interpreted as single-bit projections; field manipulations project to identity

functions and can be ignored. What remains is a system of boolean selections that assemble to PAL fuse maps (Recall the generic PAL description in Section 3.2).

In this interpretation, a bit-sliced collector is described by n instances of Figure 5–4, connected in parallel to the component abstractions. However, some equations are irrelevant in certain slices. The projection of bit $N + m$ of an N-bit vector is vacuous; equations for N-bit signals need not appear in the description of slice $N + m$.

Figure 5–6 shows the physical composition of the collector. Six distinct PAL descriptions are developed, each implementing a subset of equations in Figure 5–4, with some differing only in their representation of constants. Internal descriptions of components *PALa* through *PALf* can be inferred from the equations that use them. For instance, the equation

$$[H_{24}, D_{24}, M1.D_{24}, M2.D_{24}] = PALb(CMD, M1, M2, R)$$

indicates that *PALb* is described by equations for H, D, $M1.D$, and $M2.D$ in Figure 5–4. A broadcast command signal is specified as:

$$CMD = SEL(Q, 0,\ 1,\ \cdots,\ 33)$$

The encoding is automatically synthesized to obtain description *PALsel*. The decoding of CMD is manifested in the remaining components, as illustrated in Section 3.2. Signals S, W, R, $M1.I$, $M2.I$, $C.I$, $Z1.I$, and $Z2.I$ are packaged in a single component *PALinst*. The equations for multiple-bit signals are manually replicated and constants assigned. The boolean description of *PALinst* is then automatically generated.

All PAL descriptions are generated, reduced, and translated to Altera Design Files. Simplification is done by Espresso [19]. Thirty-four instances of eight descriptions, *PALsel*, *PALinst*, *PALa*, ..., *PALf* are assembled to Altera PALs to implement the majority of the collector's architecture. Figure 5–7 shows a custom PLA realization of the *PALa* bit slice, the largest of eight boolean descriptions. It is generated from the same design file by the `MPLA` function of the Berkeley VLSI tools [19]. Four register feedback paths are added manually in `Magic`.

The *STATUS* combination (Figure 5–2) and two dynamic-RAM memories are built by hand. Standard LSI is used to realize specifications of the arithmetic abstract components. The prototype is wire-wrapped on a Logic Engine, where a microprogrammed serial interface cross-loads heap images from an Apollo workstation. The workstation hosts the benchmark Scheme implementation.

6. Summary and Directions

A digital engineer balances many considerations when developing an implementation. A need to decompose descriptions in orthogonal hierarchies is fundamental, and system factorizations are a basic mechanism for syntactic decomposition. We are just beginning to explore what parameters are involved in factorization. This paper looks at impositions of logical and physical structure, including the control/architecture decomposition, the use of complex types in specification, and physical partitionings.

The transformation system discussed here establishes a path to physical implementation. It is envisaged as a vehicle for exploring how functional modeling techniques can be brought to bear on a wider range of target technologies. We see a working relationship between this approach and hardware verification research. At the same time, a transformational discipline seems practical without being tied to formal correctness. The frontier of our work is to confront the problem of representing abstract bases in low level digital descriptions.

Derived implementations are correct provided that (1) their specifications are correct, (2) the transformations preserve correctness, and (3) the physical elements used accurately model the theory of description. These are three ways that formal verification methods integrate with a transformational design. In the other direction, hardware verification needs automation to compose "theorems" because hardware proofs (e.g., [8], [11], [15], [6]), almost necessarily, follow the logical structure of design. We use Hunt's FM8501 proof [11] as an example of the relationship. The proof has two levels, one addressing the implementation of an arithmetic basis, the other a realization of instruction semantics. To get a physical description of FM8501, it remains to derive a sequential-system description from its register-transfer specification, to incorporate the representation of the ground type, and to develop an appropriate physical organization of the whole. It is toward these kinds of manipulations that our transformations are directed. For the garbage collector derived in Section 5, integrating Hunt's proof of arithmetic primitives would secure a "more correct" implementation.

There was no digital debugging of the garbage collector derived in Section 5. Though mistakes were made in assembling the prototype, none were found in the programmed hardware. The collector worked on its first full test. The control algorithm was well understood. The specification and intermediate forms (e.g., Figures 5–1 through 5–5) are executable; implementation proceeded directly from a tested specification. The later stages of derivation could also have been used to explore the design, but were not in this exercise.

In addition, Figure 5-1 served almost immediately as a simulation of the algorithm. On representative heaps, the derived collector is 62 to 98 times faster than a compatible 68000 implementation. The hardware collects at a rate of about 5M bytes per second. The main factors in speed improvement are hardware support of tagged data, broader data paths, parallel activity on two memories, and the usual gains over software implementation of an algorithm [4]. The derivation process exposed optimizations in physical organization and these led to revisions (i.e., manual transformations) on the initial description. One instance discussed was the decision to switch memory paths internally in PALs, which lead to a more conventional memory abstraction. The derived collector is comparable to a design developed manually using structured digital design techniques for the same class of components. The experience convinces us that engineering can be done in the linear notation exhibited in this paper. That is, the transformation system presented an adequate global view of the design while factorizations isolated facets of immediate interest.

Targeting the garbage collector for a bit-slice PAL implementation was an expedient derivation strategy. A number of practical problems were circumvented. At the same time, a fundamental formal issue was also bypassed; it is the problem of incorporating representations. In the figurative sense of Section 1, the "concrete" type Γ is replaced by a suitable representation, G

$$\begin{array}{l} S_{\Theta/\Gamma} \longrightarrow I_{\Theta/\Gamma} \longrightarrow C_{\Theta/\Gamma} \longrightarrow C'_{\Gamma} \| \Theta_{\Gamma} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Downarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad C'_{G} \| \Theta_{G} \longrightarrow \cdots \end{array}$$

The derivation in Section 5 is tailored to make this step superficial. Bit slices are an extreme example of orthogonal logical and physical decompositions. They project the logical structure into each physical component. For a glimpse at this issue, let us consider incorporating addition in the garbage collector's data path. Using the *Adder* defined in Section 2, it is not hard to see that *BitSum* combinations would be incorporated in each projection. A different physical organization, such as partitioning for 4-bit slices, would also have to be imposed on *Adder*. One PAL description, *PALinst* in Figure 5–6, was interactively developed for a similar reason.

Incorporating representations can be highly exacting and therefore needs automation. However, a general treatment should not be restricted to the simple types involved in this paper's examples. Discussions in Section 3 suggest that abstraction and hierarchy are useful in this dimension of derivation. Considered as a component for a list processing system, the garbage collector implements the complex type HEAP in a processor specification [21]. Theories

for representation are developing that will have an impact for both hardware and software engineering methodology.

This work reflects a view of digital engineering as a creative endeavor. Automated support should present intuitive facets of design while correctly maintaining the relationships among them. We seek characterizations of basic abstractions, to be codified in an algebraic framework. Our early experience with this approach suggests that reasonable and correct implementations can be derived through a manageable set of general transformations. The immediate challenge is to broaden the range of implementations while preserving generality in the algebra.

Acknowledgments

Will Clinger made numerous contributions to the design exercise reported in Section 5, including his implementation of the Scheme test bed. Eric Ost helped establish the test environment. Bob Wehrmeister provided essential support in the integration of design and fabrication tools. We are grateful to David Winkel and Franklin Prosser for their insights in methodology and their help in constructing the prototype.

References

[1] Altera Corporation, Altera Programmable Logic User System User Guide (Version 4.0), Altera Corporation, Santa Clara, 1985,

[2] Bose, Bhaskar, *Hardware Derivation from a High Level Specification: an Automated Transformation System Implementation,* in progress.

[3] Boute, R. T., Current Work on the Semantics of Digital Systems, in G. Milne and P. Subrahmanyam (eds.) *Formal Aspects of VLSI Design,* North-Holland, Amsterdam, 1986, 99–112.

[4] Boyer, C. David, *A Transformationally Correct Hardware Garbage Collector Implementation,* in progress.

[5] Clinger, William C., The Scheme 312 Version 4 Reference Manual, 1985 (unpublished).

[6] Cohn, Avra, A Proof of Correctness of the Viper Microprocessor: The First Level, *this volume.*

[7] Fenichel, R., and J. Yochelson, A LISP garbage-collector for virtual-memory computer systems, *Comm. ACM* **12**(11):611–612 (1969).

[8] Gordon, Mike, Proving a Computer Correct, University of Cambridge Computer Laboratory Technical Report No. 42, Cambridge, 1983.

[9] Harel, David, On folk theorems *Comm. ACM* **23** (7):379–389, (1980)

[10] Hill, F. J. and G. R. Peterson, *Introduction to Switching Theory and Logical Design* (Third Ed.), John Wiley&Sons, New York, 1981.

[11] Hunt, Warren A., Jr., *FM8501: A verified Microprocessor,* Ph.D. dissertation, Technical Report 47, Institute for Computing Science, The University of Texas at Austin, 1985.

[12] Johnson, Steven D., Digital Design in a Functional Calculus, in G. Milne and P. Subrahmanyam (eds.) *Formal Aspects of VLSI Design,* North-Holland, Amsterdam, 1986, 153–178.

[13] Johnson, Steven D., Applicative Programming and Digital Design, *Proc. Eleventh Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (1984), 218–227.

[14] Johnson, Steven D., *Synthesis of Digital Designs from Recursion Equations, The ACM Distinguished Dissertation Series,* The MIT Press, 1984.

[15] Joyce, Jeffrey, Formal Verification and Implementation of a Microprocessor, *this volume.*

[16] O'Donnell, John T., Hardware Description with Recursion Equations, *Proc. 8th International Symposium on Computer Hardware Description Languages and their Applications,* Amsterdam, April, 1987.

[17] Prosser, Franklin P., and David E. Winkel, The Logic Engine Development System—Support for Microprogrammed Bit-Slice Development, *Proc. Micro 16,* 84–91.

[18] Rees, Jonathan and William C. Clinger (eds.), Revised^3 Report on the Algorithmic Language Scheme, Indiana University Computer Science Department Technical Report No. 174, Bloomington, December, 1986.

[19] Scott, Walter S., Robert N. Mayo, Gordon Hamachi, and John K. Ousterhout (eds.), 1986 VLSI Tools, Report No. UCB/CSD 86/272, Computer Science Division (EECS), University of California at Berkeley, 1985.

[20] Winkel, David E., What Next for PALs. Technical Report No. 188, Indiana Univ. Computer Science Dept., Bloomington, Indiana, February, 1986.

[21] Winkel, David E. and Christopher T. Haynes, Hardware Design Using Functionally Connected Units, Indiana University Computer Science Dept. Technical Report No. 219, *submitted for publication.*

[22] Winkel, David E., and Franklin P. Prosser. *The Art of Digital Design, 2nd Edition* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

```
IDLE(M1, M2, H, D, C, U, A, GO, R) =
  GO → NEXT(M1, M2, *H*, D, C, 0, 0, GO, false),
       IDLE(M1, M2, H, D, C, U, A, true)

DRIVER(M1, M2, H, D, C, U, A, GO, R) =
  eq?(U, A) → IDLE(M2, M1, H, D, C, U, A, GO, true),
              NEXT(M1, M2, read(M1, U), D, C, U, A, GO, R)

NEXT(M1, M2, H, D, C, U, A, GO, R) =
  pointer?(H) → OBJ(M1, M2, H, read(M2, H), C, U, A, GO, R),
  bvec?(H) → DRIVER(M1, M2, H, D, C, addinc(U, btow(ptr(H))), A, GO, R),
             DRIVER(M1, M2, H, D, C, inc(U), A, GO, R)

OBJ(M1, M2, H, D, C, U, A, GO, R) =
  eq?(fwd, tag(D)) →
    DRIVER(write(M1, U, cell(H, D)), M2, H, D, C, inc(U), A, GO, R),
  eq?(pair, tag(H)) →
    PAIR1(write(M1, A, D), write(M2, H, cell(fwd, A)), H, D, C, U, A, GO, R),
  eq?(vec, tag(H)) →
    VEC(write(M1, U, cell(H, A)), M2, H, D, ptr(D), inc(U), A, GO, R),
  eq?(bvec, tag(H)) →
    BVEC(write(M1, A, D), M2, H, cell(D, btow(ptr(D))),
         btow(ptr(D)), U, A, GO, R)
  eq?(fbvec, tag(H)) →
    DRIVER(M1, M2, H, D, C, inc(U), A, GO, R),


PAIR1(M1, M2, H, D, C, U, A, GO, R) =
  PAIR2(write(M1, U, cell(H, A)), M2, H, read(M2, inc(ptr(H))),
        C, U, inc(A), GO, R)

PAIR2(M1, M2, H, D, C, U, A, GO, R) =
  DRIVER(write(M1, A, D), M2, H, D, C, inc(U), inc(A), GO, R)

VEC(M1, M2, H, D, C, U, A, GO, R) =
  VLOOP(write(M1, A, D), M2, H, read(M2, add(ptr(H), C)), dcr(C), U, A, GO, R)

VLOOP(M1, M2, H, D, C, U, A, GO, R) =
  eq?(C, -1) →
    DRIVER(M1, write(M2, H, cell(fwd, A)), H, D, C, U, incadd(A, ptr(D)), GO, R),
    VLOOP(write(M1, incadd(C, A), D), M2, H, read(M2, add(ptr(H), C)),
          dcr(C), U, A, GO, R)

BVEC(M1, M2, H, D, C, U, A, GO, R) =
  BLOOP(write(M1, U, cell(H, A)), M2, H, read(M2, add(ptr(H), ptr(D))),
        dcr(C), inc(U), A, GO, R)

BLOOP(M1, M2, H, D, C, U, A, GO, R) =
  eq?(C, -1) →
    DRIVER(M1, write(M2, H, cell(fwd, A)), H, D, C, U,
           incadd(A, btow(ptr(D))), GO, R)
    BLOOP(write(M1, incadd(C, A), D), M2, H, read(M2, add(ptr(H), C)),
          dcr(C), U, A, GO, R)
```

FIGURE 5–1. GARBAGE COLLECTOR SPECIFICATION

$$
\begin{aligned}
&SEL([p_0, p_1, p_2, W, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}, p_{11}],\\
&\quad v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}, v_{16}, v_{17},\\
&\quad v_{18}, v_{19}, v_{20}, v_{21}, v_{22}, v_{23}, v_{24}, v_{25}, v_{26}, v_{27}, v_{28}, v_{29}, v_{30}, v_{31}, v_{32}, v_{33})\\
&\quad \stackrel{\mathrm{d}}{=} \textit{per } p_0 \textit{ viz} \quad \text{idle} : p_1 \rightarrow v_0,\ v_1\\
&\qquad\qquad\qquad \text{driver} : p_2 \rightarrow v_2,\\
&\qquad\qquad\qquad\qquad\qquad W \rightarrow v_3,\ v_4\\
&\qquad\qquad\qquad\quad \text{next} : p_4 \rightarrow W \rightarrow v_5,\ v_6,\\
&\qquad\qquad\qquad\qquad\quad p_5 \rightarrow v_7,\ v_8\\
&\qquad\qquad\qquad\qquad \text{obj} : p_6 \rightarrow W \rightarrow v_9,\ v_{10},\\
&\qquad\qquad\qquad\qquad\qquad p_7 \rightarrow W \rightarrow v_{11},\ v_{12},\\
&\qquad\qquad\qquad\qquad\qquad p_8 \rightarrow W \rightarrow v_{13},\ v_{14},\\
&\qquad\qquad\qquad\qquad\qquad p_9 \rightarrow W \rightarrow v_{15},\ v_{16},\\
&\qquad\qquad\qquad\qquad\qquad p_{10} \rightarrow v_{17},\ \phi\\
&\qquad\qquad\qquad\qquad \text{pair1} : W \rightarrow v_{18},\ v_{19}\\
&\qquad\qquad\qquad\qquad \text{pair2} : W \rightarrow v_{20},\ v_{21}\\
&\qquad\qquad\qquad\qquad\quad \text{vec} : W \rightarrow v_{22},\ v_{23}\\
&\qquad\qquad\qquad\qquad \text{vloop} : p_{11} \rightarrow W \rightarrow v_{24},\ v_{25},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad W \rightarrow v_{26},\ v_{27}\\
&\qquad\qquad\qquad\qquad\quad \text{bvec} : W \rightarrow v_{28},\ v_{29}\\
&\qquad\qquad\qquad\qquad \text{bloop} : p_{11} \rightarrow W \rightarrow v_{30},\ v_{31},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad W \rightarrow v_{32},\ v_{33}
\end{aligned}
$$

$$
\begin{aligned}
&STATUS(S, H, D, C, U, A, GO, W) \stackrel{\mathrm{d}}{=} [\ S\\
&\qquad\qquad\qquad\qquad\qquad\quad GO\\
&\qquad\qquad\qquad\qquad\qquad\quad eq?(U, A)\\
&\qquad\qquad\qquad\qquad\qquad\quad W\\
&\qquad\qquad\qquad\qquad\qquad\quad pointer?(H)\\
&\qquad\qquad\qquad\qquad\qquad\quad bvec?(H)\\
&\qquad\qquad\qquad\qquad\qquad\quad eq?(\text{fwd}, tag(D))\\
&\qquad\qquad\qquad\qquad\qquad\quad eq?(\text{pair}, tag(H))\\
&\qquad\qquad\qquad\qquad\qquad\quad eq?(\text{vec}, tag(H))\\
&\qquad\qquad\qquad\qquad\qquad\quad eq?(\text{bvec}, tag(H))\\
&\qquad\qquad\qquad\qquad\qquad\quad eq?(\text{fbvec}, tag(H))\\
&\qquad\qquad\qquad\qquad\qquad\quad eq?(C, \text{-1})\\
&\qquad\qquad\qquad\qquad\qquad]
\end{aligned}
$$

FIGURE 5–2. SELECTOR COMBINATION AND STATUS GROUPING

$$P = STATUS(S, H, D, C, C, U, A, GO, W)$$

$$S \overset{\phi!}{=} SEL(\text{P, next, idle, idle, next, next, obj, obj, driver, driver, driver, driver, pair1, pair1, vec1, vec1, bvec1, bvec1, driver, pair2, pair2, driver, driver, vloop, vloop, driver, driver, vloop, vloop, bloop, bloop, driver, driver, bloop, bloop})$$

$$M1 \overset{\phi!}{=} SEL(\text{P}, M1, M1, M1, M1, M1, M1, M1, M1, M1, M1, write(M1, U, cell(H, D)), write(M1, H, cell(\text{fwd}, A)), write(M1, A, D), M1, write(M1, U, cell(H, A)), M1, write(M1, A, D), M1, M1, write(M1, U, cell(H, A)), M1, write(M1, A, D), M1, write(M1, A, D), write(M1, H, cell(\text{fwd}, A)), M1, M1, write(M1, \boxed{incadd}(C, A), D), M1, write(M1, U, cell(H, A)), write(M1, H, cell(\text{fwd}, A)), M1, M1, write(M1, \boxed{incadd}(C, A), D))$$

$$M2 \overset{\phi!}{=} SEL(\text{P}, M2, M2, M2, M2, M2, M2, M2, M2, M2, write(M2, U, cell(H, D)), M2, write(M2, A, D), write(M2, H, cell(\text{fwd}, A)), write(M2, U, cell(H, A)), M2, write(M2, A, D), M2, M2, write(M2, U, cell(H, A)), M2, write(M2, A, D), M2, write(M2, A, D), M2, M2, write(M2, H, cell(\text{fwd}, A)), write(M2, \boxed{incadd}(C, A), D), M2, write(M2, U, cell(H, A)), M2, M2, write(M2, H, cell(\text{fwd}, A)), write(M2, \boxed{incadd}(C, A), D), M2)$$

$$H \overset{\phi!}{=} SEL(\text{P, *H*, H, H}, read(M2, U), read(M1, U), \text{H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H})$$

$$D \overset{\phi!}{=} SEL(\text{P, D, D, D, D, D}, read(M1, H), read(M2, H), \text{D, D, D, D, D, D, D, D}, cell(D, btow(ptr(D))), cell(D, btow(ptr(D))), \text{D}, read(M1, \boxed{inc}(ptr(H))), read(M2, \boxed{inc}(ptr(H))), \text{D, D}, read(M1, \boxed{add}(ptr(H), C)), read(M2, \boxed{add}(ptr(H), C)), \text{D, D}, read(M1, \boxed{add}(ptr(H), C)), read(M2, \boxed{add}(ptr(H), C)), read(M1, \boxed{add}(ptr(H), ptr(D))), read(M2, \boxed{add}(ptr(H), ptr(D))), \text{D, D}, read(M1, \boxed{add}(ptr(H), C)), read(M2, \boxed{add}(ptr(H), C)))$$

$$C \overset{\phi!}{=} SEL(\text{P, C, C, C, C, C, C, C, C, C, C, C, C, C}, ptr(D), ptr(D), btow(ptr(D)), btow(ptr(D)), \text{C, C, C, C, C}, dcr(C), dcr(C), \text{C, C}, dcr(C), dcr(C), dcr(C), dcr(C), \text{C, C}, dcr(C), dcr(C))$$

$$U \overset{\phi!}{=} SEL(\text{P, 0, U, U, U, U, U, U}, \boxed{addinc}(U, btow(ptr(H))), \boxed{inc}(U), \boxed{inc}(U), \boxed{inc}(U), \text{U, U}, \boxed{inc}(U), \boxed{inc}(U), \text{U, U}, \boxed{inc}(U), \text{U, U}, \boxed{inc}(U), \boxed{inc}(U), \text{U, U, U, U, U, U}, \boxed{inc}(U), \boxed{inc}(U), \text{U, U, U, U})$$

$$A \overset{\phi!}{=} SEL(\text{P, 0, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A}, \boxed{inc}(A), \boxed{inc}(A), \boxed{inc}(A), \boxed{inc}(A), \text{A, A}, \boxed{incadd}(A, ptr(D)), \boxed{incadd}(A, ptr(D)), \text{A, A, A, A}, \boxed{incadd}(A, btow(ptr(D))), \boxed{incadd}(A, btow(ptr(D))), \text{A, A})$$

$$R \overset{\phi!}{=} SEL(\text{P, false, true, true, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R})$$

$$W \overset{\phi!}{=} SEL(\text{P, W, W}, not(W), \text{W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W})$$

FIGURE 5–3. INITIAL SYSTEM DESCRIPTION

$$S \stackrel{\phi!}{=} \mathit{SEL}(P, \text{next}, \text{idle}, \text{idle}, \text{next}, \text{next}, \text{obj}, \text{obj}, \text{driver}, \text{driver}, \text{driver}, \text{driver}, \text{pair1}, \text{pair1}, \text{vec1}, \text{vec1}, \text{bvec1}, \text{bvec1}, \text{driver}, \text{pair2}, \text{pair2}, \text{driver}, \text{driver}, \text{vloop}, \text{vloop}, \text{driver}, \text{driver}, \text{vloop}, \text{vloop}, \text{bloop}, \text{bloop}, \text{driver}, \text{driver}, \text{bloop}, \text{bloop})$$

$$H \stackrel{\phi!}{=} \mathit{SEL}(P, \text{*H*}, H, H, M_2, M_1, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H, H)$$

$$D \stackrel{\phi!}{=} \mathit{SEL}(P, D, D, D, D, D, M_1, M_2, D, D, D, D, D, D, D, D, \mathit{cell}(D, Z_3), \mathit{cell}(D, Z_3), D, M_1, M_2, D, D, M_1, M_2, D, D, M_1, M_2, M_1, M_2, D, D, M_1, M_2)$$

$$U \stackrel{\phi!}{=} \mathit{SEL}(P, 0, U, U, U, U, U, U, Z_1, Z_1, Z_1, Z_1, U, U, Z_1, Z_1, U, U, Z_1, U, U, Z_2, Z_2, U, U, U, U, U, U, Z_2, Z_2, U, U, U, U)$$

$$A \stackrel{\phi!}{=} \mathit{SEL}(P, 0, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, A, Z_1, Z_1, Z_1, Z_1, A, A, Z_1, Z_1, A, A, A, A, Z_1, Z_1, A, A)$$

$$R \stackrel{\phi!}{=} \mathit{SEL}(P, \text{false}, \text{true}, \text{true}, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R)$$

$$W \stackrel{\phi!}{=} \mathit{SEL}(P, W, W, \mathit{not}(W), W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W)$$

$$C = \mathit{COUNT}(\phi, C.I, C.D)$$

$$C.I = \mathit{SEL}(P, @, @, @, @, @, @, @, @, @, @, @, @, @, \text{L}, \text{L}, \text{L}, \text{L}, @, @, @, @, @, \text{D}, \text{D}, @, @, \text{D}, \text{D}, \text{D}, \text{D}, @, @, \text{D}, \text{D})$$

$$C.D = \mathit{SEL}(P, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \mathit{ptr}(D), \mathit{ptr}(D), Z_3, Z_3, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi)$$

$$M_1 = \mathit{MEMORY}(\phi, M_1.I, M_1.A, M_1.D)$$

$$M_1.I = \mathit{SEL}(P, @, @, @, @, \text{R}, \text{R}, @, @, @, @, \text{W}, \text{W}, \text{W}, @, \text{W}, @, \text{W}, @, \text{R}, \text{W}, @, \text{W}, \text{R}, \text{W}, \text{W}, @, \text{R}, \text{W}, \text{R}, \text{W}, \text{W}, @, \text{R}, \text{W})$$

$$M_1.A = \mathit{SEL}(P, \phi, \phi, \phi, \phi, U, H, \phi, \phi, \phi, \phi, U, H, A, \phi, U, \phi, A, \phi, Z_2, U, \phi, A, Z_1, A, H, \phi, Z_2, Z_1, Z_1, U, H, \phi, Z_2, Z_1)$$

$$M_1.D = \mathit{SEL}(P, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \mathit{cell}(H, D), \mathit{cell}(\text{fwd}, A), D, \phi, \mathit{cell}(H, A), \phi, D, \phi, \phi, \mathit{cell}(H, A), \phi, D, \phi, D, \mathit{cell}(\text{fwd}, A), \phi, \phi, D, \phi, \mathit{cell}(H, A), \mathit{cell}(\text{fwd}, A), \phi, \phi, D)$$

FIGURE 5–4. FACTORED SYSTEM DESCRIPTION
(continued on next page)

$$M_2 = MEMORY(\phi, M_2.I, M_2.A, M_2.D)$$

$$M_2.I = SEL(P, \mathsf{Q}, \mathsf{Q}, \mathsf{Q}, \mathsf{R}, \mathsf{Q}, \mathsf{Q}, \mathsf{R}, \mathsf{Q}, \mathsf{Q}, \mathsf{W}, \mathsf{Q}, \mathsf{W}, \mathsf{W}, \mathsf{W}, \mathsf{Q}, \mathsf{W}, \mathsf{Q}, \mathsf{Q}, \mathsf{W}, \mathsf{R}, \mathsf{W}, \mathsf{Q}, \mathsf{W}, \mathsf{R}, \mathsf{Q}, \mathsf{W}, \mathsf{W}, \mathsf{R}, \mathsf{W}, \mathsf{R}, \mathsf{Q}, \mathsf{W}, \mathsf{W}, \mathsf{R})$$

$$M_2.A = SEL(P, \phi, \phi, \phi, U, \phi, \phi, H, \phi, \phi, U, \phi, A, H, U, \phi, A, \phi, \phi, U, Z_2, A, \phi, A, Z_1, \phi, H, Z_1, Z_2, U, Z_1, \phi, H, Z_1, Z_2)$$

$$M_2.D = SEL(P, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, cell(H, D), \phi, D, cell(\mathsf{fwd}, A), cell(H, A), \phi, D, \phi, \phi, cell(H, A), \phi, D, \phi, D, \phi, \phi, cell(\mathsf{fwd}, A), D, \phi, cell(H, A), \phi, \phi, cell(\mathsf{fwd}, A), D, \phi)$$

$$Z_1 = ALU1(Z_1.I, Z_1.A, Z_1.B)$$

$$Z_1.I = SEL(P, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{AI}, \mathsf{I}, \mathsf{I}, \mathsf{I}, \mathsf{C}, \mathsf{C}, \mathsf{I}, \mathsf{I}, \mathsf{C}, \mathsf{C}, \mathsf{I}, \mathsf{I}, \mathsf{I}, \mathsf{I}, \mathsf{I}, \mathsf{A}, \mathsf{A}, \mathsf{IA}, \mathsf{IA}, \mathsf{IA}, \mathsf{IA}, \mathsf{A}, \mathsf{A}, \mathsf{IA}, \mathsf{IA}, \mathsf{IA}, \mathsf{IA})$$

$$Z_1.A = SEL(P, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, U, U, U, U, \phi, \phi, U, U, \phi, \phi, U, A, A, A, A, ptr(H), ptr(H), A, A, C, C, ptr(H), ptr(H), A, A, C, C)$$

$$Z_1.B = SEL(P, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, Z_3, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, C, C, ptr(D), ptr(D), A, A, ptr(D), ptr(D), Z_3, Z_3, A, A)$$

$$Z_2 = ALU2(Z_2.I, Z_2.A, Z_2.B)$$

$$Z_2.I = SEL(P, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{I}, \mathsf{I}, \mathsf{I}, \mathsf{I}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{A}, \mathsf{A}, \mathsf{I}, \mathsf{I}, \mathsf{C}, \mathsf{C}, \mathsf{A}, \mathsf{A})$$

$$Z_2.A = SEL(P, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, ptr(H), ptr(H), U, U, \phi, \phi, \phi, \phi, ptr(H), ptr(H), U, U, \phi, \phi, ptr(H), ptr(H))$$

$$Z_2.B = SEL(P, \phi, C, C, \phi, \phi, \phi, \phi, C, C)$$

$$Z_3 = BYTETOWORD(Z_3.I, Z_3.B)$$

$$Z_3.I = SEL(P, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{B}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{B}, \mathsf{B}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{B}, \mathsf{B}, \mathsf{C}, \mathsf{C})$$

$$Z_3.B = SEL(P, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, ptr(H), \phi, \phi, \phi, \phi, \phi, \phi, \phi, ptr(D), ptr(D), \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, \phi, ptr(D), ptr(D), \phi, \phi)$$

FIGURE 5–4 (CONTINUED). FACTORED SYSTEM DESCRIPTION

$$
\begin{array}{l}
MEMORY(\check{m}, Inst, Addr, Data) \stackrel{d}{=} read(M) \\
\quad \textbf{where} \\
\qquad M = \check{m}\ !\ Interpret(M, Inst, Addr, Data) \\
\quad \textbf{and} \\
\qquad Interpret(m, inst, addr, data) \stackrel{d}{=} \\
\qquad\quad per\ inst\ viz\ \ @ : m \\
\qquad\qquad\qquad\qquad\quad R : m \\
\qquad\qquad\qquad\qquad\quad W : write(m, addr, data)
\end{array}
$$

$$
\begin{array}{l}
ALU1(Inst, A, B) \stackrel{d}{=} X \\
\quad \textbf{where} \\
\qquad X = per\ Inst\ viz\ \ C : 0 \\
\qquad\qquad\qquad\qquad\quad AI : addinc(A, B) \\
\qquad\qquad\qquad\qquad\quad\ I : inc(A) \\
\qquad\qquad\qquad\qquad\quad\ A : add(A, B) \\
\qquad\qquad\qquad\qquad\quad IA : incadd(A, B)
\end{array}
$$

$$
\begin{array}{l}
ALU2(Inst, A, B) \stackrel{d}{=} X \\
\quad \textbf{where} \\
\qquad X = per\ Inst\ viz\ C : 0 \\
\qquad\qquad\qquad\qquad\quad\ I : inc(A) \\
\qquad\qquad\qquad\qquad\quad\ A : add(A, B)
\end{array}
$$

$$
\begin{array}{l}
COUNT(\check{x}, Inst, Data) \stackrel{d}{=} X \\
\quad \textbf{where} \\
\qquad X = \check{x}\ !\ Interpret(X, Inst, Data) \\
\quad \textbf{and} \\
\qquad Interpret(x, inst, data) \stackrel{d}{=} per\ inst\ viz\ @ : x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad L : data \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad D : dcr(x)
\end{array}
$$

$$
\begin{array}{l}
BYTETOWORD(Inst, Byte) \stackrel{d}{=} X \\
\quad \textbf{where} \\
\qquad X = per\ Inst\ viz\ C : 0 \\
\qquad\qquad\qquad\qquad\quad B : btow(Byte)
\end{array}
$$

FIGURE 5–5. FACTORED COMPONENT SPECIFICATIONS

$$\textbf{COLLECTOR}(GO) \stackrel{\mathrm{d}}{=} R \quad \text{where}$$

$$P = STATUS(S,\ H,\ D,\ C,\ U,\ A,\ GO,\ W)$$
$$CMD = PALsel(P)$$
$$[\,S, R, W, M_1.I,\ M_2.I,\ Z_1.I,\ Z_2.I,\ C.I,\ Z_3.I\,] = PALinst(CMD)$$
$$M_1 = MEMORY(\phi, M_1.I,\ [\,M_1.A_{23} \cdots M_1.A_0\,],\ [\,M_1.D_{31} \cdots M_1.D_0\,])$$
$$M_2 = MEMORY(\phi, M_2.I,\ [\,M_2.A_{23} \cdots M_2.A_0\,],\ [\,M_2.D_{31} \cdots M_2.D_0\,])$$
$$Z_1 = ALU1(Z_1.I,\ [\,Z_1.A_{23} \cdots Z_1.A_0\,],\ [\,Z_1.B_{23} \cdots Z_1.B_0\,])$$
$$Z_2 = ALU2(Z_2.I,\ [\,Z_2.A_{23} \cdots Z_2.A_0\,],\ [\,Z_2.B_{23} \cdots Z_2.B_0\,])$$
$$C = COUNT(\phi, C.I,\ [\,C.D_{23} \cdots C.D_0\,])$$
$$Z_3 = BYTETOWORD(Z_3.I,\ [\,Z_3.B_{23} \cdots Z_3.B_0\,])$$
$$H \stackrel{\phi!}{=} [\,H_{31} \cdots H_0\,]$$
$$D \stackrel{\phi!}{=} [\,D_{31} \cdots D_0\,]$$
$$U \stackrel{\phi!}{=} [\,U_{23} \cdots U_0\,]$$
$$A \stackrel{\phi!}{=} [\,U_{23} \cdots U_0\,]$$
$$[\,H_0, D_0, U_0, A_0, M_1.A_0, M_1.D_0, M_2.A_0, M_2.D_0, Z_1.A_0, Z_1.B_0, Z_2.A_0, Z_2.B_0, C.D_0, Z_3.B_0\,] = PALa(CMD, M_1, M_2, Z_1, Z_2, C, Z_3, R)$$
$$\vdots$$
$$[\,H_{23}, D_{23}, U_{23}, A_{23}, M_1.A_{23}, M_1.D_{23}, M_2.A_{23}, M_2.D_{23}, Z_1.A_{23}, Z_1.B_{23}, Z_2.A_{23}, Z_2.B_{23}, C.D_{23}, Z_3.B_{23}\,] = PALa(CMD, M_1, M_2, Z_1, Z_2, C, Z_3, R)$$
$$[\,H_{24}, D_{24}, M_1.D_{24}, M_2.D_{24}\,] = PALb(CMD, M_1, M_2, R)$$
$$\vdots$$
$$[\,H_{27}, D_{27}, M_1.D_{27}, M_2.D_{27}\,] = PALb(CMD, M_1, M_2, R)$$
$$[\,H_{28}, D_{28}, M_1.D_{28}, M_2.D_{28}\,] = PALc(CMD, M_1, M_2, R)$$
$$[\,H_{29}, D_{29}, M_1.D_{29}, M_2.D_{29}\,] = PALd(CMD, M_1, M_2, R)$$
$$[\,H_{30}, D_{30}, M_1.D_{30}, M_2.D_{30}\,] = PALe(CMD, M_1, M_2, R)$$
$$[\,H_{31}, D_{31}, M_1.D_{31}, M_2.D_{31}\,] = PALf(CMD, M_1, M_2, R)$$

FIGURE 5–6. GARBAGE COLLECTOR CIRCUIT DESCRIPTION

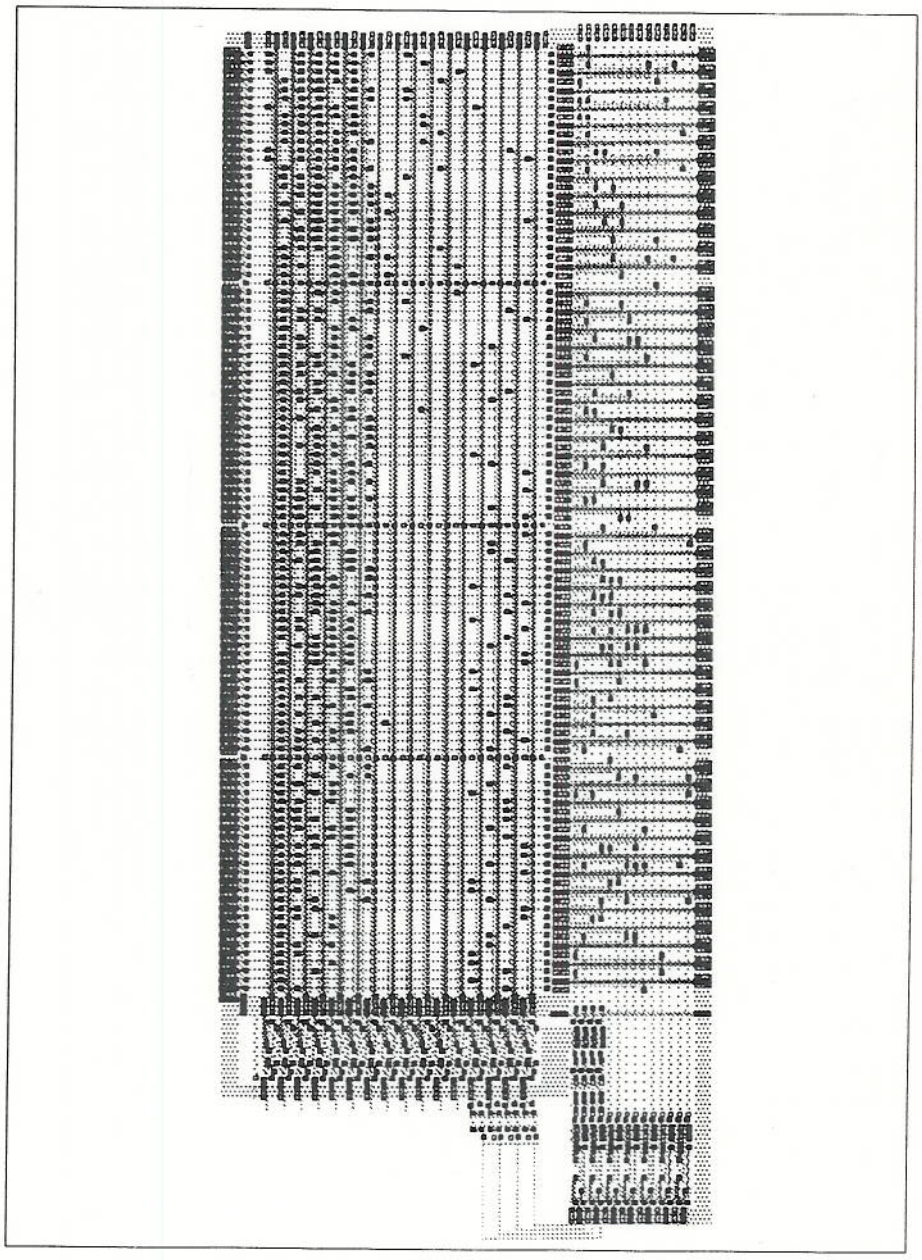


FIGURE 5–7. PLA IMPLEMENTATION OF COMPONENT *PALa*

VLSI SPECIFICATION, VERIFICATION AND SYNTHESIS

edited by

Graham Birtwistle
University of Calgary

and

P.A. Subrahmanyam
AT&T Bell Laboratories

KLUWER ACADEMIC PUBLISHERS
Boston/Dordrecht/Lancaster

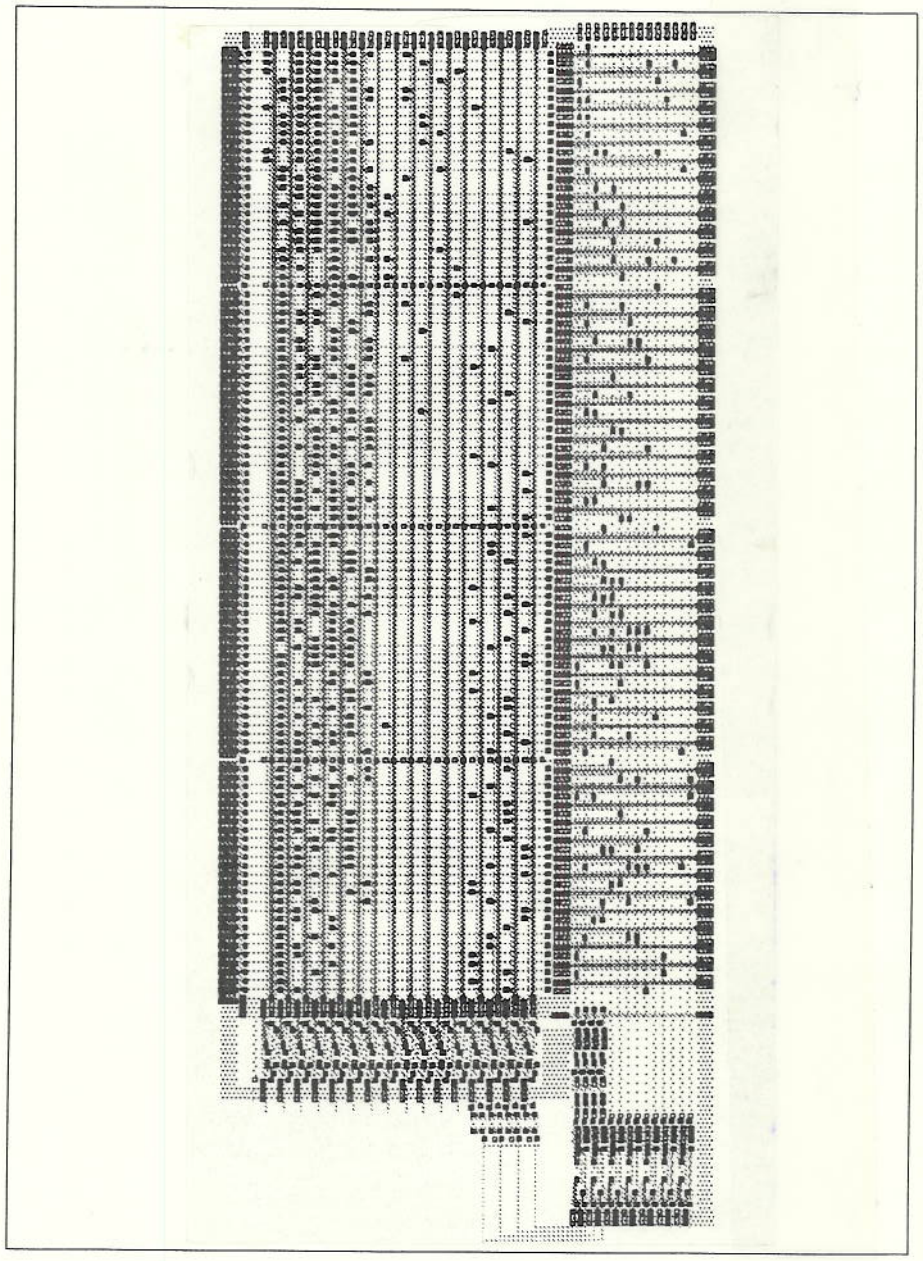


FIGURE 5–7. PLA IMPLEMENTATION OF COMPONENT *PALa*