

INDIANA UNIVERSITY  
COMPUTER SCIENCE DEPARTMENT

TECHNICAL REPORT NO. 383

## **An Example of Interactive Hardware Transformation**

Zheng Zhu and Steven D. Johnson

MAY 1993\*

\*This paper was presented at the *ACM/SIGDA Workshop on Formal Methods in VLSI design*, Maimi, Florida, January 1991. Due to publication problems the proceedings of this workshop has not yet appeared.

# An Example of Interactive Hardware Transformation\*

Zheng Zhu, Steven D. Johnson  
Department of Computer Science  
Indiana University  
Bloomington, Indiana, USA

## Abstract

This article presents an example of correct circuit design through interactive transformation. Interactive transformation differs from traditional hardware design transformation frameworks in that it focuses on the issue of finding suitable hardware architecture for the specified system and the issue of architecture correctness. The transformation framework divides every transformation in designs into two steps. The first step is to find a proper architecture implementation. Although the framework does not guarantee existence of such an implementation, nor its discovery, it does provide a characterization of architectural implementation so that the question “is this a correct implementation?” can be answered by equational rewriting. The framework allows a correct architecture implementation to be automatically incorporated with control descriptions to obtain a new system description.

The significance of this transformation framework lies in the fact that it requires simpler mechanism of verification (equational rewriting versus first order logic, functional calculus, or higher logic) to guarantee correctness of transformations while gives reasonable expressive power to describe a wide range of hardware systems.

## 1 Introduction

This article presents an abstract data oriented approach to hardware design transformations. Our goal is to lay a foundation for a general treatment of abstraction in hardware description, verification, and derivation so that the virtues of verification and derivation can be combined in designs. We develop extensions to the algebraic characterization of abstract data types which permit us to specify hardware architectures, and we give a corresponding characterization of hardware controls as recursive functions. The organization of the article is the following: first, we discuss the motivation and goals of our research, then we introduce a formal framework for hardware design transformation and explain a group of correctness preserving transformations for hardware designs through an example.

Throughout the article, the following conventions are observed:

- “ $a = b$ ” means that “ $a$  equals  $b$ ” or “ $a$  is defined as  $b$ ”;

---

\*This research was supported, in part, by the National Science Foundation under grants numbered MIP87-07067, DCR85-21947, and MIP89-21842

- “ $a \equiv b$ ” is a logical expression which means that “ $a$  is equivalent to  $b$ ”. “ $a \equiv_E b$ ” means “ $a$  is equivalent to  $b$  in some model of  $E$ ” where  $E$  is a set of equations.
- $[a_1, \dots, a_n]$  denotes a vector whose  $i$ th element is  $a_i$  where  $1 \leq i \leq n$ .

## 2 Architecture Oriented Interactive Transformation

*Architecture oriented interactive transformation* attempts to provide a unified formal framework for both verification and derivation. It restricts verification to architecture transformations and derivation to control synthesis. This section briefly discusses the motivation, basic ideas, and goals of the approach.

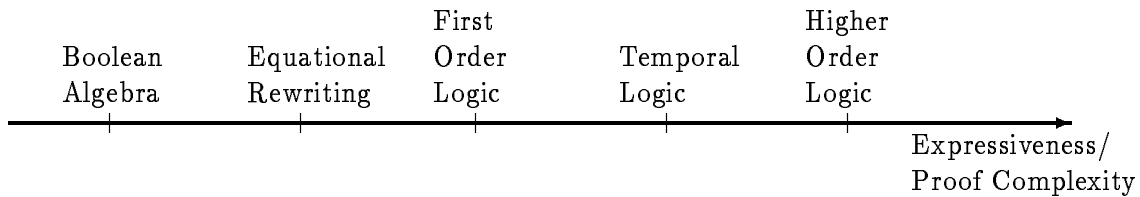
### 2.1 Motivation

Both VLSI design transformation and verification are confronted by unsolvable or computationally intractable problems. Although both of them have been drawing more attention and are successful in some designs, research in both areas has provided strong evidence that neither of them alone can give a satisfactory resolution to the fast growing complexity in VLSI design. Transformation and verification complement each other in the sense that some “hard” derivation problem can be solved by verification techniques relatively easily and some “hard” verification problems can be solved by transformation relatively easily. To take advantage of this fact, we should look to combining verification and transformation. In order to achieve this, we need to address at least two crucial problems: A unified formal framework for both verification and transformation; and a proper division of the roles of verification and derivation in a hardware design process.

### 2.2 Hardware Verification and Derivation

Formal hardware design has two different approaches: design verification and correctness preserving transformation. Design verification starts from an implementation and establishes a validation path to a given specification in some formal framework. As a by-product, the validation path serves as a design because it is actually a path between the specification and the implementation. On the other hand, design transformation starts from a specification, using a set of predefined transformations to establish a transformation path to a description which satisfies certain design constraints. The correctness of the design is guaranteed if transformations used in the path are *meaning preserving*. Although these two approaches differ in several important ways, researchers have recognized the necessity of combining them.

The power of verification systems is usually measured by their expressiveness. But a trade-off for expressiveness is proof complexity in those systems [1]. The following diagram gives a list of formal systems in the order of their expressive power versus the degree of proof complexity. A compromise between expressiveness and proof complexity has always been an important consideration in designing verification systems.



Transformations in hardware designs can be classified by two categories: control transformation and architecture (or abstract basis) transformation. The control transformations are those which change control structures of hardware systems to achieve certain design goals. To preserve the meaning of hardware description, formal systems which support control transformations usually allow *free interpretation* of abstract basis of hardware descriptions. Although this property provides an opportunity to borrowing results from other mature research areas in computer science, such as  $\lambda$ -calculus and program transformation, it is not sufficient when transformation between abstract bases becomes necessary. Architecture transformations are those which change underlying abstract basis of hardware systems. They are necessary because the abstract data types in specifications can not be, in general, be directly implemented. As a matter of fact, finding an appropriate basis is usually a major part of design tasks. Lack of architecture transformations either forces a transformation system to work on fixed bases or fails to preserve meaning of specifications. Currently, few transformation systems have the ability of architecture transformation in this more general sense.

The theoretical research in abstract data types and equational rewriting already lays a foundation of architecture transformation but its strength has rarely been integrated with the high level synthesis and control transformation systems. It is one of our main purposes to explore the possibility of combining the power of verification (equational rewriting) and virtues of control transformation into a correctness preserving transformation framework in order to avoid verification complexity and to maintain correctness of control transformations.

### 2.3 Proposed Framework for Verification and Derivation

The proposed framework can be informally characterized as the followings:

- A hardware specification is a pair  $[A, C]$  where  $A$  is a description of architecture and  $C$  is a control operating on  $A$ . See Figure 1(a);
- Hardware architecture  $A$  is specified as an extended abstract data type using equational specification. Hardware controls are specified as recursive functions on abstract data types representing the corresponding architecture;
- Assume there are definitions of “implementation of architecture” and of “hardware system”. The proposed design procedure (shown in Figure 1(b)) will be: given  $[A_1, C_1]$  as a specification, an oracle (designer, an heuristic guided algorithm, etc) will propose  $\gamma_a$  which maps  $A_1$  to  $A_2$  as an architecture implementation. Then a verifier will be called to test “does  $A_2$  implement  $A_1$ ?”. If the verifier validates the test, an algorithm  $\gamma_c$  which is dependent on  $\gamma_a$  and original specification  $D_1$ , synthesizes a  $C_2$  such that  $[A_2, C_2]$  is an implementation of  $[A_1, C_1]$ . This process repeats until a satisfactory design is achieved.

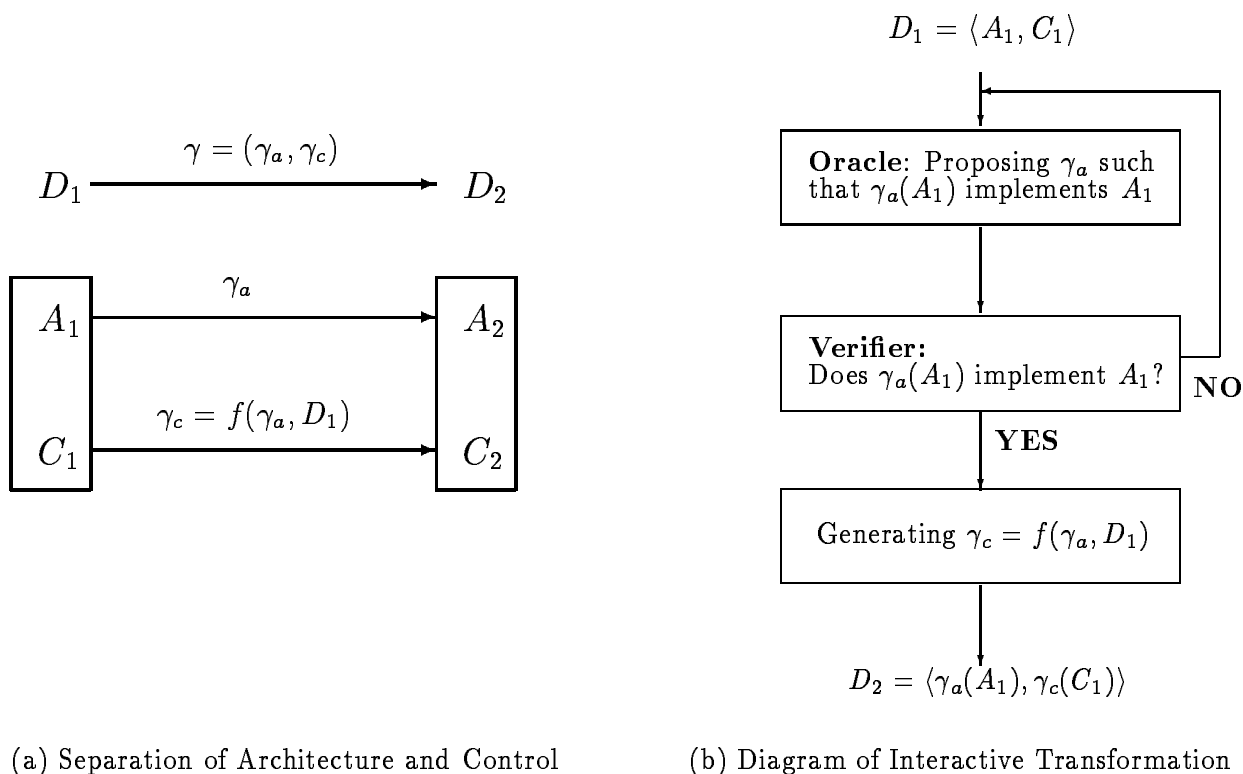


Figure 1: Illustration of Interactive Transformation

## 2.4 Evaluation of The Proposed Design Framework

The proposed framework attempts to combine the virtues of verification and transformation. It has the following features:

1. It has reasonable expressive power to specify a wide range of hardware systems;
2. The correctness of transformations can be established by equational rewriting;
3. It can address the issue of multi-level data abstraction which is rarely touched in current high level synthesis research.
4. Since the transformations require synthesis of hardware control after the correctness of architecture implementation is established, they can take advantage of fruitful results in high level synthesis research. At the same time, our functional specification allows us to benefit from research in program transformation which has been a cornerstone of our research.

## 3 A Formal Framework of Hardware Specification

This section introduces a formal framework for hardware architecture. The hardware architecture specification is based on equational specification and universal algebra. Although equational specification and universal algebra are the bases of abstract data types specification and semantics, we

do not attempt to associate our hardware specification with any of the particular abstract data type semantics proposed in [2, 3, 4, 5, 6, 7]. Some of the definitions were inspired by [2]. More detailed discussion on the specifying hardware architecture in an algebraic framework can be found in [8, 9].

### 3.1 Specifying Hardware Architecture and System

Let  $S$  be a set of sorts. An  $S$ -sorted *signature*  $\Sigma$  is an  $S^* \rightarrow S$ -sorted family  $\langle \Sigma_{w \rightarrow s} \mid w \in S^* \text{ and } s \in S \rangle$ . An element  $\sigma$  of  $\Sigma_{w \rightarrow s}$  is a function symbol of *arity*  $w$  and of sort  $s$ . The arity of a function symbol expresses what sorts of data it expects to take as its inputs and in what order. The sort of a function symbol expresses the sort of data it returns. Constant symbols are considered function symbols of *empty* arity  $\lambda$ . Let  $\Sigma$  be  $S$ -sorted and  $X = \langle X_s \mid s \in S \rangle$  where each pair of  $X_{s_1}, X_{s_2}$  are disjoint if  $s_1 \neq s_2$ . Each  $X_s$  is a set of ( $s$ -sorted) variables. We use  $T_\Sigma(X)$  to denote the set of *terms of generated from  $\Sigma$  and  $X$*

Traditionally,  $T_\Sigma$  is used to denote  $T_\Sigma(\phi)$ . Elements in  $T_{\Sigma_s}(X)$  are called  $s$ -sorted terms. Let  $Y$  be an  $S$ -sorted class of variables which is disjoint with  $X$ . A  $Y$ -equation of  $T_\Sigma(X)$  is a tuple  $(l, r)$  where  $l, r \in T_\Sigma(X \cup Y)$ . We usually write  $(l, r)$  as “ $l \equiv r$ ”. Set  $Y$  is called a set of specification variables. Let  $E$  be a set of  $Y$ -equations of  $T_\Sigma(X)$ , then  $E$  specifies an equational theory. It can be shown that  $E$  defines an equivalence relation on the set  $T_\Sigma(X)$ ; hence we can define  $T_\Sigma(X)/E$  as the set of equivalence classes of  $T_\Sigma(X)$  under  $E$ . If  $t/inT_\Sigma(X)$ ,  $[t]_E$  denotes the equivalent class under  $E$  which contains  $t$ .

Let  $\Sigma$  be an  $S$ -sorted signature. A  $\Sigma$ -*algebra*  $A$  consists of a set of carriers  $\langle A_s \mid s \in S \rangle$  together with a set of functions  $F = \{f_\sigma \mid \sigma \in \Sigma\}$  among them.

An equational specification is  $\langle S, \Sigma, X, E \rangle$  or simply  $\langle \Sigma, X, E \rangle$  when  $S$  is known. It defines an algebra  $T(\Sigma, E) = \langle T_\Sigma(X)/E, F_\Sigma \rangle$  (or  $T(\Sigma)$  if  $E = \phi$ ) where  $F_\Sigma = \{f_\sigma \mid \sigma \in \Sigma\}$  and for every  $f_\sigma \in F_\Sigma$ ,  $f_\sigma([t_1]_E, \dots, [t_n]_E) = [\sigma(t_1, \dots, t_n)]_E$ . As pointed out in the literature (*e.g.* [2, 3] etc), equational specifications can be used as specifications for abstract data types. We now extend equational specifications to define abstract hardware architecture syntactically.

**Definition 1.** A specification of architecture is a quadruple

$$\langle S, \Sigma, R, E, C \rangle$$

where  $\langle S, \Sigma, R, E \rangle$  is an equational specification, called the specification of *underlying (abstract data) type*.  $R$  is a set of  $n$  variables, called *the set of registers of the architecture*;  $H$  is a set of functions  $R \rightarrow T_\Sigma(R)$  containing the identity function:  $id(r) = r$  for every  $r \in R$ .

$id$  of  $H$  is the identity function in terms of behavior. However, if we take timing into consideration, it represents a delay operation. The set  $H$  is also called a set of *constraints* because each function in  $H$  describes a specific data transfers. For example, let  $R = \{x, y\}$  and  $c \in H$  be the function  $c(x) = add(x, y)$  and  $c(y) = y$ . This  $c$  “connects” the output of an adder to the register  $x$ . Data transfers not explicitly mentioned in  $H$  are prohibited. We may view  $H$  as something more than just a set of constraints. As we introduce an interpretation of an architecture specification,  $H$  characterizes all the possible computations in one machine cycle.

### 3.2 Semantics of Hardware Architecture

Let  $\langle S, \Sigma, R, E, H \rangle$  be a specification of some architecture where  $R = \{r_1, r_2, \dots, r_n\}$ . Assume that  $r_1, r_2, \dots, r_n$  are of sorts  $s_1, s_2, \dots, s_n \in S$  respectively. The following definition defines an operator  $\circ$  which is analogous to function composition:

**Definition 2..** Let  $f, g$  be functions from  $R$  to  $T_\Sigma$ .  $h = f \circ g$  is defined as for every  $i : 1 \leq i \leq n$ ,  $h(r_i) = f(r_i)[r_j \leftarrow g(r_j) : 1 \leq j \leq n]$  where the right hand side denotes a term generated by substituting every occurrence of  $r_j$  in  $f(r_i)$  with  $g(r_j)$  for all  $j : 1 \leq j \leq n$ .

Definition 3 defines a set  $\mathcal{B}_H$  which is shown to form a universal algebra and is used later to model the behavior of the architecture specification.

**Definition 3..** Given a set of data paths  $H$ , the set of *computations by  $H$* , denoted by  $\mathcal{B}_H$ , is the smallest set containing  $H$  and closed under  $\circ$ .

The equivalence relation  $E$  can be naturally extended to an equivalence relation on  $\mathcal{B}_H$ . Let  $f, g \in \mathcal{B}_H$ ,  $f \equiv_E g$  if and only if  $f(r) \equiv_E g(r)$  for all  $r \in R$ . Let  $b \in \mathcal{B}_H$ , and  $f \in H$ . If we define  $f(b) = f \circ b$  then  $B(H) = \langle \mathcal{B}_H, H \rangle$  is a  $H$ -algebra. The universe of the algebra is  $\mathcal{B}_H$ . Furthermore,  $E$  is a congruence relation on  $B(H)$ . Therefore,  $B(H, E) = \langle \mathcal{B}_H/E, H \rangle$  is an  $H$ -algebra.

**Definition 4..**  $B(H, E)$  is called the functional behavior of the architecture  $\langle S, \Sigma, R, E, H \rangle$ .

### 3.3 Correct Implementation of Hardware Architectures

Given an architecture specification, we would like to define the notion of an implementation of the architecture. Since we have shown that an architecture can be defined in terms of  $\Sigma$ -algebra, we borrow the definition of  $\Sigma$ -algebras from [2].

**Definition 5..** Let  $\Sigma$  and  $\Omega$  be  $S_1$ -sorted and  $S_2$ -sorted operator domains respectively. A *derivator*  $d: \Sigma \rightarrow \Omega$  is a pair of functions  $\langle \xi, \kappa \rangle$  where  $\xi: S_1 \rightarrow S_2^*$  and  $\kappa: \Sigma \rightarrow \lambda(T_\Omega(X))$ , the abstractions of elements in  $T_\Omega(X)$ , such that for every  $\sigma \in \Sigma_{w,s}$ ,  $\kappa(\sigma)$  is a  $\xi(s)$ -sorted function symbol of arity  $\xi(w)$ . The  $d$ -derived term algebra of  $T(\Sigma, E_1)$ ,  $dT(\Sigma, E_1)$ , is defined as  $\langle T_\Omega/E_2, d(\Sigma) \rangle$ .

**Definition 6..** Let  $A_\Sigma = \langle S_1, \Sigma, R_1, E_1, H_1 \rangle$  and  $A_\Omega = \langle S_2, \Omega, R_2, E_2, H_2 \rangle$  be two specifications of architecture.  $A_\Omega$  is said to be an implementation of  $A_\Sigma$  if there exists a derivator  $d$  and an equivalence relation  $E$  on  $\mathcal{B}_{H_2}$  such that  $B(H_1, E_1) \subseteq \langle \mathcal{B}_{H_2}/E, d(H_1) \rangle/E$  where  $\subseteq$  means “is a subalgebra of”.

Specifically, we regard a derivator  $d$  of an architecture as a family of three functions  $d = \langle \xi, \gamma, \hbar \rangle$ :

1.  $\xi: S_1 \rightarrow S_2$ ;
2.  $\gamma: R_1 \rightarrow R_2$ ;
3.  $\hbar: H_1 \rightarrow H_2$  which is consistent with  $\xi$ .

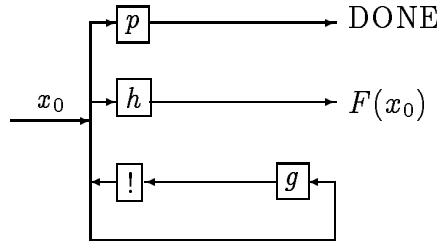
## 4 Specification of Hardware Systems

This section gives a simple specification of circuits. It is a simplified version of Johnson's functional notation [10] with an emphasis on the idea of separating architecture from control in circuit specifications. As we can see in this section that a control of a hardware system is a recursive structure over the underlying hardware architecture.

Let  $H$  be set of datapaths defined in section 4 and  $A_H$  be an  $H$ -algebra. A set of recursive functions on  $A_H$ , denoted by  $\mathcal{R}_H$  is the set of all iterative recursive functions which are in the form of

$$F(x) = \text{if } p(x) \text{ then } h(x) \text{ else } F(g(x)) \quad (1)$$

where  $x$  is a variable ranging over  $A_H$  and  $h = h_l \circ h_{l-1} \circ \dots \circ h_1$ ,  $g = g_k \circ g_{k-1} \circ \dots \circ g_1$  for some  $h_1, \dots, h_l, g_1, \dots, g_k$  belonging to  $A_H$ . Johnson [10] shows that such an iterative recursion function characterizes a sequential control of circuit as shown in the circuit schematic below. In the schematic,  $\boxed{g}$ ,  $\boxed{h}$ ,  $\boxed{p}$  denote functional units implementing  $g$ ,  $h$ , and  $p$  respectively.  $\boxed{!}$  denotes a unit delay component.  $\boxed{!}$  denotes a register or a vector of registers.



**Circuit Schematic Defined by  $F(x)$**

The interpretation of the schematic is that when the output from  $\boxed{p}$  is true (or high), then the output from  $\boxed{h}$  is the result of the computation defined by  $F(x_0)$ .

Let  $A_{H_1}$  and  $A_{H_2}$  be  $H_1$ -algebra and  $H_2$ -algebra respectively and  $F_1 \in \mathcal{R}_{H_1}$  and  $F_2 \in \mathcal{R}_{H_2}$  be the functions:

$$F_1(x) = \text{if } p_1(x) \text{ then } h_1(x) \text{ else } F_1(g_1(x))$$

$$F_2(y) = \text{if } p_2(y) \text{ then } h_2(y) \text{ else } F_2(g_2(y))$$

The implementation relation among them is traditionally characterized by input-output relations:  $F_2$  is an implementation of  $F_1$  if there exist two functions  $\alpha: A_{H_1} \rightarrow A_{H_2}$  and  $\beta: A_{H_2} \rightarrow A_{H_1}$  and an equivalence relation  $E$  of  $A_{H_1}$ , such that for every  $x \in A_{H_1}$ ,

$$F_1(x) \equiv_E \beta \circ F_2 \circ \alpha(x)$$

Let  $\langle A_{H_2}, d, E \rangle$  be an implementation of  $A_{H_1}$ ,  $d$  is one to one,  $p_2 = d(p_1)$ ,  $h_2 = d(h_1)$ , and  $g_2 = d(g_1)$ , then it can be proven that  $F_2$  is an implementation of  $F_1$  through  $\alpha = d$ ,  $\beta = d^{-1}$  and  $E$ . That is, hardware controls (recursive functions) preserve the implementation relation of their underlying architectures.

## 5 Deriving A Machine Implementation From Specification

This section presents an example of hardware transformation. The transformation starts from a specification of a simple machine with abstract architecture on abstract data structures such as stack. As we see in this section, four transformations are needed in order to obtain our goal. Three of these four are of our major interest. They are: implementation of one architecture by another; merge of two objects into one; and replacing combinational signals by sequential ones. One common property of these three transformations is that although they are transformations on architectures, they induce transformations of system descriptions.

```

machine (M-i a-i S)
  case rd(M-i, a-i)
    ('stop' machine(M-i, a-i, S))
    ('add' machine(M-i, dcr(a-i), push(add(top(S),top(pop(S))), pop(pop(S))))))
    ('sub' machine(M-i, dcr(a-i), push(sub(top(S),top(pop(S))), pop(pop(S))))))
    ('mul' machine(M-i, dcr(a-i), push(mul(top(S),top(pop(S))), pop(pop(S))))))
    ('div' machine(M-i, dcr(a-i), push(div(top(S),top(pop(S))), pop(pop(S))))))
    ('eq?' machine(M-i, dcr(a-i), push(eq?(top(S),top(pop(S))), pop(pop(S))))))
    ('le?' machine(M-i, dcr(a-i), push(le?(top(S),top(pop(S))), pop(pop(S))))))

```

**Specification 1** *The Machine*

Specification 1 is a simple stack machine specification. The first transformation is the *abstraction or encapsulation* of all the arithmetic operations into an arithmetic unit [11, 12]. The result of the transformation is Specification 2.

```

machine (M-i, a-i, S) =
  if rd(M-i, a-i)=='stop'
  then machine (M-i, a-i, S)
  else machine (M-i, dcr(a-i), push(arith(rd(M-i, a-i), top(S), top(pop(S))), pop(pop(S))))

```

**Specification 2** *The Modified Machine*

The next transformation is to implement the stack by a memory. Informally, we intend to implement abstract stack object by a memory-address pair. Assume that  $m_0$  is the initial element of memory and  $\phi$  be the initial element of address. A standard mapping between the stack and [memory address] pair is

$$\begin{aligned}
 \text{empty} &\rightarrow [m_0 \ \phi] \\
 \text{push}(x_s, x_d) &\rightarrow [wr(x_m, inc(x_a), x_d) \ inc(x_a)] \\
 \text{top}(x_s) &\rightarrow rd(x_m, x_a) \\
 \text{pop}(x_s) &\rightarrow [x_m \ dcr(x_a)]
 \end{aligned}$$

Replacing function symbol of stack in specification 2 transforms it to specification 3:

```

machine (M-i, a-i, M-s, a-s) =
  if rd(M-i, a-i)=='stop'
    then machine (M-i, a-i, M-s, a-s)
    else machine (M-i,
                  dcr(a-i),
                  wr(arith(rd(M-i, a-i), rd(M-s a-s), rd(M-s, dcr(a-s))), M-s, dcr(a-s)),
                  dcr(a-s))

```

**Specification 3** *A Machine Without Stack*

The next transformation is to implement two memories ( $M-s$ ,  $M-i$ ) by one memory object  $M$ . A typical way of doing this is to map  $M-s$  to the odd-address half of  $M$  and  $M-i$  to the even-address half of  $M$ . The specification 4 is the result of this transformation.

```

machine (M, a-i, a-s) =
  if rd(M, a-i)=='stop'
    then machine(M, rd(M, dcr2(a-i)), a-s)
    else machine(wr(arith(rd(M, a-i), rd(M a-s), rd(M, dcr2(a-s))), M, dcr2(a-s)),
                  dcr2(a-i),
                  dcr2(a-s))

```

**Specification 4** *The Machine With One Memory*

In general, we must consider two questions. The first one is “What constitutes a correct implementation of hardware components?” This requires a proper definition of implementation which reasonably reflects our intuition of implementation; The second question is “Can the definition of implementation be tested with some mechanical help?” In the above transformations, we need to test whether (memory,address) under the mapping is an implementation of stack and so on.

The last transformation replaces combinational signal  $rd(M, a-i)$  by a register  $I$ . This transformation requires the following steps:

- Adding a new element  $I$  to the set of registers of the architecture  $R$ .
- Each function in  $H$  will be extended from  $R \rightarrow T_{\Sigma}(R)$  to  $R \cup \{I\} \rightarrow T_{\Sigma}(R \cup \{I\})$ .

The result of the transformation is Specification 5.

```

machine (M, a-i, a-s, I) =
  if I=='stop'
    then machine (M, rd(M, dcr2(a-i)), a-s, rd(M, rd(M, dcr2(a-i))))
    else machine (wr(arith(rd(M, a-i), rd(M, a-s)), rd(M, dcr2(a-s))), M, dcr2(a-s)),
                dcr2(a-i),
                dcr2(a-s),
                rd(M, dcr2(a-i)))

```

**Specification 5** *The Machine With One Memory*

One subtle thing about this circuit description is that, in order to achieve the same behavior as that of Specification 4, register  $I$  needs to be initiated properly when circuit is started. As we can see in next section, our method guarantees that  $I$  is initiated properly.

In order to achieve a realizable circuit description, it is necessary to conduct other transformations from Specification 5 but those transformations are beyond the scope of this paper. To summarize, this example of transformation involves 4 transformations: Abstraction, implementing stack by memory-address pair, merging two memories into one, and replacing combinational signal by register. It is not claimed that those transformations are complete in any sense. They are selected because they demonstrate some interesting aspects of design transformation.

## 6 Correctness of the Transformations

The transformation from Specification 1 to Specification 2 belongs to the class of control transformations. It is called *encapsulation* since it encapsulates all the arithmetic operations to an arithmetic unit [11]. It has been implemented in Indiana University's digital design system DDD [12]. The other transformations in the example are transformations concerning abstract bases of hardware systems such as implementing one architecture by another. These transformations are our major interest since they are rarely discussed in the research of digital design transformation.

This section reviews the results concerning correct implementation of architecture [8] and examines the correctness of the transformations presented in the last section.

### 6.1 Correctness of Architectural Implementations

As we stated in the last section, in order to obtain a correctness preserving hardware transformation, the correctness of the implementation of the underlying architecture must be assured when it is given. Theorem 1 asserts the condition under which a given function  $d$  constitutes a derivor of an implementation (See Definition 5). [8] provides a formal proof of the theorem. Furthermore, the theorem implies that the proof of the correctness of implementation can be conducted with the help of a partially effective procedure.

**Definition 7.** Let  $E_1$ ,  $E_2$  and  $E_3$  be definite sets of equations.

1.  $E_1 \vdash E_2$  means that for every  $(l, r) \in E_2$ , there exist  $(l_1, r_1), \dots, (l_n, r_n) \in E_1$  such that  $l_1 \equiv l$ ,  $r_n \equiv r$  and for every  $i = 1, \dots, n-1$ ,  $r_i \equiv l_{i+1}$ ;

2.  $E_1/E_2 \vdash E_3$  means that for every  $(l, r) \in E_3$ , there exists  $t$  such that  $E_1 \vdash (l, t)$  and  $E_2 \vdash (t, r)$ ;
3. Let  $d$  be a derivor  $H_1 \rightarrow H_2$  and  $E$  be a set of equations. Define  $d(E) = \{ (d(l), d(r)) \mid (l, r) \in E \}$ .

**Theorem 1.** Let  $A_\Sigma = \langle S_1, \Sigma, R_1, E_1, H_1 \rangle$  and  $A_\Omega = \langle S_2, \Omega, R_2, E_2, H_2 \rangle$  be two specifications of architectures. Let  $d : H_1 \rightarrow H_2$  be a derivor as defined in Definition 5 and  $E$  be a set of equations of  $\mathcal{B}_{H_2}$ . If  $E_2/E \vdash d(E_1)$  then  $\langle A_\Omega, d, E \rangle$  is an implementation of  $A_\Sigma$ .

The significance of this theorem is that it characterizes implementation in terms of equational rewriting mechanisms. Therefore, the correctness of implementations can be examined with a help of theorem provers such as OBJ [13] and AFFIRM [14].

In the example, the first derivor  $d$  was to transfer an architecture with *stack* to one with *memory* (From Specification 2. to Specification 3.). It is specified as follows:

1. The function  $\xi$  is:  $\xi$  is identity function except  $\xi(S) \rightarrow [M \ A]$  where  $M$  and  $A$  are the sorts for *memory* and *address of memory*. What this  $\xi$  indicates is that a stack is implemented by a pair of memory and memory address.
2. The function  $\tilde{h} : H_1 \rightarrow H_2^*$  is defined by the following mapping:

$$\begin{aligned} \text{empty} &\rightarrow [m_0 \ \phi] \\ \text{push}(x_s, x_d) &\rightarrow [wr(x_m, inc(x_a), x_d) \ inc(x_a)] \\ \text{top}(x_s) &\rightarrow rd(x_m, x_a) \\ \text{pop}(x_s) &\rightarrow [x_m \ dcr(x_a)] \end{aligned}$$

$x_s, x_m, x_a$ , and  $x_d$  are variables for stack, memory, address and data respectively.  $m_0$  is the predefined *initial memory* and  $\phi$  is the predefined *initial address*. It should be pointed out this mapping is actually the one for an abstract data type (stack) to another (memory/address) rather than  $H_1 \rightarrow H_2$ . But since we can prove that the product preserves implementation relation [15], we consider that the mapping of  $H_1 \rightarrow H_2$  as the *natural extension* of that of the abstract data types.

Let  $E_s$  and  $E_m$  denote the sets of equations specifying stack and memory respectively<sup>1</sup> and  $E_a$  denotes the set of equations specifying memory address type. Among other equations in  $E_a$  are  $x_a \equiv dcr(inc(x_a))$  and  $x_a \equiv inc(dcr(x_a))$ .  $E_s$  and  $E_m$  are as follows:

$$\begin{aligned} E_s &= \{ \begin{array}{l} x_s \equiv pop(push(x_s, x_d)) \\ x_d \equiv top(push(x_s, x_d)) \\ x_s \equiv push(pop(x_s), top(x_s)) \end{array} \} \\ E_m &= E_a \cup \{ \begin{array}{l} x_d \equiv rd(wr(x_m, x_a, x_d), x_a) \\ x_m \equiv wr(x_m, x_a, rd(x_m, x_a)) \end{array} \} \end{aligned}$$

and  $d(E_s)$ , the image of  $E_s$  under the derivor  $d$ , is:

$$\begin{aligned} d(E_s) &= \{ \begin{array}{l} [x_m \ x_a] \equiv [wr(x_m, inc(x_a), x_d) \ dcr(inc(x_a))] , \\ x_d \equiv rd([wr(x_m, inc(x_a), x_d) \ inc(x_a)]) , \\ [x_m \ x_a] \equiv [wr(x_m, x_a, rd(x_m, x_a)), inc(dcr(x_a))] \} \end{array} \end{aligned}$$

Let  $bot = [m_0 \ \phi]$ ,  $pu = [wr(x_m, inc(x_a), x_d) \ inc(x_a)]$ ,  $to = rd(x_m, x_a)$ , and  $po = [x_m \ dcr(x_a)]$ , then the underlying abstract operator set of the derived architecture is  $\{bot, pu, to, po\}$ .

<sup>1</sup>Equations pertaining to error handling are not included.

Let  $s = s_1 s_2 \dots s_k$  and  $E$  by a set of equations. Define

$$E(s) = \{ [t_{s_1} \dots t_{s_k}] \equiv [t'_{s_1} \dots t'_{s_k}] \mid \forall i : 1 \leq i \leq k. t_{s_i} \equiv t'_{s_i} \in E \}$$

called the extension of  $E$  over  $s$ . Then the set of equations for the architecture composed of memory, address, and memory data is  $E_{ma} = E_m(s_d) \cup E_m([s_m s_a])$ :

$$\begin{aligned} E_{ma} &= E_m(s_d) \cup E_m([s_m s_a]) \\ &= \{ x_d \equiv rd(wr(x_m, inc(x_a), x_d), inc(x_a)) \} \cup \\ &\quad \{ [x_m x_a] \equiv [wr(x_m, x_a, rd(x_m, x)) inc(dcr(x_a))] \\ &\quad [x_m x_a] \equiv [wr(x_m, x_a, rd(x_m, x_a)) dcr(inc(x_a))] \\ &\quad [x_m x_a] \equiv [x_m dcr(inc(x_a))] \\ &\quad \dots \} \end{aligned}$$

Let  $E = \{ ([x_m x_a] \equiv [wr(x_m inc(x_a) x_d) x_a]) \}$ . Then we can easily prove that  $E_{ma}/E \vdash d(E_s)$ . All the equations of  $d(E_s)$  are either equations of  $E_{ma}$  or can be derived directly from  $E_{ma}/E$ . By **Theorem 1**, we assert that we obtain a correctness preserving transformation from stack to memory/address.

## 6.2 Sharing Hardware Resource

It is common in digital designs to share available hardware to minimize the system cost. However, in a specification of system, it is ideal to use different names for different abstract components in order to increase the clarity of the specification. It is often the case that the possibility of sharing a resource is not evident until a certain level of detail has been explored. There are two types of resource sharing. The first one is sharing functional units such as arithmetic unit, buses etc. This type of problem has been extensively studied and successfully applied to an area of design automation. It is called *scheduling and allocation* in high level synthesis literature [16, 17]. The main techniques used to solve the problem are control and data flow analysis. The second kind of resource sharing is complicated components such as memory or registers. Since these components have their own internal structures, flow analysis alone is insufficient to achieve the goal. In our example, two abstract memories ( $M-i$  and  $M-s$ ) are implemented by one ( $M$ ) through a transformation of merging two memory objects into one (Specification 3 to Specification 4).

In what follows, we discuss the issue of merging two identical components and a precondition under which a merge preserves the meaning of the original specification.

Let  $S$  be a set of sorts and  $\bar{S}$  be the transitive closure of  $S$  under the character string concatenation. Let  $s \in S$  and  $s_1, s_2 \in \bar{S}$  and  $s$  is the common prefix of both  $s_1$  and  $s_2$ . Without losing generality, we assume that  $s_1 = ss'_1$  and  $s_2 = ss'_2$ . Let  $A = \langle S, \Sigma, E \rangle$ ,  $H_i \subseteq [R_{s_i} \rightarrow T_\Sigma]$  for  $i = 1, 2$  where  $R_{s_i}$  is a set of register variables such that for every  $r \in R_{s_i}$ , there is one and only one  $s' \in S$  and  $s'$  is in  $s_i$  such that  $r$  is of sort  $s'$  and  $H_3 \subseteq [R_{s_1 s_2} \rightarrow T_\Sigma]$ . Assume that  $\mathcal{A}_{s_1, H_1}$ ,  $\mathcal{A}_{s_2, H_2}$  and  $\mathcal{A}_{s_1 s_2, H_3}$  are architectures generated from  $R_{s_1}$  and  $H_1$ ,  $R_{s_2}$  and  $H_2$ ,  $R_{s_1 s_2}$  and  $H_3$ , respectively. Also assume that  $\mathcal{A}_{ss'_1 s'_2, H}$  is an architecture with  $R_{ss'_1 s'_2} = \{r_s\} \cup R_{s'_1} \cup R_{s'_2}$  and  $H$  where for every  $h \in H$ , there exist  $h_1 \in H_1$  and  $h_2 \in H_2$  such that

$$h(r) = \begin{cases} h_1(r) & r \in R_{s_1} \\ h_2(r) & r \in R_{s_2} \\ h_1(r) \circ h_2(r) & r = r_s \end{cases}$$

Now, the question of whether  $r$  can be shared by  $\mathcal{A}_{s_1, H_1}$  and  $\mathcal{A}_{s_2, H_2}$  becomes one of whether  $\mathcal{A}_{s_1, H_1} \times \mathcal{A}_{s_2, H_2}$  is equivalent to  $\mathcal{A}_{s, s'_1, s'_2, H}$ . Intuitively, we know that this is not always the case. We are interested in knowing that under what conditions this is true.

We define two homomorphisms  $g_1$  and  $g_2$  as: for  $a = [a_1 \ a_2 \ a_3] \in A_{s, s'_1, s'_2, H}$  where  $a_1, a_2, a_3$  are of  $s, s'_1, s'_2$  sorted respectively,  $g_1(a) = [a_1 \ a_2]$  and  $g_2(a) = [a_1 \ a_3]$ . Let  $\theta_1$  and  $\theta_2$  be the congruence relations induced by  $g_1$  and  $g_2$ , then  $A_{s, s'_1, s'_2, H}$  is an implementation of  $\mathcal{A}_{s_1, H_1} \times \mathcal{A}_{s_2, H_2}$  iff the following conditions hold:

1.  $\theta_1 \cap \theta_2 = \omega$  where  $\omega$  is the identity relation;
2.  $\theta_1 \cup \theta_2 = \iota$  where  $\iota$  is the complete relation.

This result is a variation of Birkhoff's. The proof of original result can be found in [18], page 120. Furthermore, it can be proven that  $\theta_1$  and  $\theta_2$  defined above automatically satisfy condition 2 and condition 3. They also satisfy condition 1 if and only if the algebras corresponding to  $s'_1$  and  $s'_2$ , denoted as  $A_{s'_1}$  and  $A_{s'_2}$ , are disjoint,  $A_{s'_1} \cap A_{s'_2} = \phi$ . Ideally, this condition should be effectively decidable. Unfortunately, we have not found an effective algorithm to test the given conditions.

### 6.3 Replacing Combinational Signals

Delay caused by the propagation of combinational signals may significantly affect the performance of circuits. One of the classical solutions to this problem is to pre-fetch some combinational signals into registers in the clock cycle before the signals are used and use the contents of those registers to generate the signals. In our example (Specification 4 to Specification 5), in order to save instruction coding time (needed to generate the signal (rd  $M$   $a$ -i)) in each clock cycle, an instruction register is introduced in the place of signal and the register is loaded a cycle before it is referenced. Such a technique is also widely used in pipelined architectural designs and other areas of hardware designs.

In order to replace combinational signals by registers, new registers need to be introduced. However, as we have discussed in [15, 19] that introducing new registers will not affect the correctness of circuit description if they are not referenced in the datapaths to those original registers.

Let  $G$  a set of combinational signals an architecture can generate and  $H$  be the set of datapath of an architecture. Let  $c \in G$  represent some combinational signal occurring either in some datapaths in  $H$  or in some test part of controls. Let  $r \notin R$  be a new register we newly introduce to the architecture to store  $c$  and replace occurrences of  $c$  in datapaths, let  $H'$  be the new set of datapaths extended from the original set of datapaths such that

1.  $|H| = |H'|$ , and
2. For every  $h' \in H'$ , there exists one and only one  $h \in H$  such that for  $r \in R \cup \{r\}$ ,

$$h'(r) = \begin{cases} h(r) & r \in R \\ c[r' \leftarrow h(r'), \forall r' \in V(c)] & r = r' \end{cases}$$

where  $V(c)$  is the set of registers occurring in  $c$  and  $c[r' \leftarrow h(r'), \forall r' \in V(c)]$  means replacing occurrences of  $r'$  in  $c$  by  $h(r')$  for every  $r' \in V(c)$ .

Let  $r \notin R$  be a new register,  $H$  be a set of datapaths on  $R$  and  $R_r = R \cup \{r\}$ . An natural extension of  $h \in H$  to  $R_r$  is  $h'_r(r) = r$  and  $h'(r') = h(r')$  if  $r' \in R$ . Also, let  $h_c \in [R \rightarrow T_\Sigma]$  such

that  $h^c(r) = c$  and  $h^c(r') = r'$  for every  $r' \in R$ . Then the new set of datapaths  $H'$  defined above equals:

$$H' = \{ h^c \circ h_r \mid h \in H \}$$

Therefore, the relation between  $B_{R,H}$  and  $B_{R',H'}$  can be characterized as  $B_{R,H} \cong h^c \circ B_{R',H'}$ .

Based on the above discussion, if

$$F(x) = \text{case} \begin{array}{l} p_1(x) \rightarrow F(h_1(x)) \\ \vdots \\ p_n(x) \rightarrow F(h_n(x)) \end{array}$$

is circuit description on an architecture  $A_R$ . Let  $h'_i = h^c \circ h_i$  and  $p''_i, h''_i$  are obtained from  $p, h'$  by substituting occurrences of  $c$  with  $r$  for every  $i = 1, \dots, k$  and

$$F'(x) = \text{case} \begin{array}{l} p''_1(x) \rightarrow F(h''_1(x)) \\ \vdots \\ p''_n(x) \rightarrow F(h''_n(x)) \end{array}$$

by a circuit description on architecture  $A_{R'}$  then

$$F(x) \cong F' \circ h_c(x) = F'(h_c(x))$$

In the case of our example,  $F'(h_c(x))$  reminds us that  $I$  register needs to be properly initialized before  $F'$  can be started.

## 6.4 Serialization

Specification 5 presents a new challenge to design transformation. After a series of transformations, computations represented by some terms in the system description have become too complicated to implement directly if we assume that every term in the description takes one cycle to realize. For example, the following memory operation

```
wr(arith(rd(M a-i) rd(M a-s) rd(M dcr2(a-s))) M dcr2(a-s))
```

in the *else* part of the description requires four memory operations (three *reads* and one *write*) in one cycle. It would be very expensive to do this. In real design, such a computation is realized by a sequence of feasible computations (of the architecture) each of which takes one cycle. For example, assume an architecture allows the following:

1. Memory can be either read or written not more than once in any clock cycle;
2. Results of memory read can be saved in registers  $r_1, r_2$  and  $r_3$ ;
3. There are datapaths to route content of  $r_1, r_2$  to data inputs of the arithmetic unit and  $r_3$  to its instruction input.

Then the computation could possibly be broken into the following 4 steps:

1. rd(M a-s) and saved to some register  $r_1$ ;
2. rd(M dcr2(a-s)) and saved to some register  $r_2$ ;

3. rd(M a-i) and saved to some register  $r_3$ ;
4. wr(arith( $r_3 r_1 r_2$ ) M dcr2(a-s)).

This transformation, called *serialization*, is analogous to the problems of *allocation and scheduling* in high level synthesis (e.g. [16, 17]), *register allocation and code generation* in compiler design (e.g. [20]) and *microcode generation* (e.g. [21]). [9] gives a formalization of the serialization problem in the current algebraic framework.

Unfortunately, it is readily proved that the serialization problem is unsolvable in general [9]. Nevertheless, heuristic algorithms have been found to cope with similar problems (such as register allocation or microcode generation). Although research effort has been made to solve this problem [22], finding suitable heuristics to solve the serialization problem remains a topic in our research.

## 7 Conclusion

This paper presents a formalized approach to hardware design derivation. The basic idea is unifying verification and derivation in a single formal framework. The approach proposes iterative transformation which verifies architecture implementation and synthesis control transformation. An example of digital design derivation is presented to demonstrate how the approach works. The example exposes a few transformations that are used very often in regular designs. Formal treatment of those transformations are given in the proposed framework.

The goal of this research can be summarized as follows:

1. Providing a framework in which hardware architecture can be specified algebraically;
2. Providing separated description for hardware architecture and hardware control;
3. Providing a practically feasible method to verify correctness of architecture implementation;
4. Providing a meaning preserving transformation method to synthesize hardware control according to control specification and architecture implementation.

## References

- [1] CAMURATI, P., AND PRINETTO, P. Formal verification of hardware correctness: Introduction and survey of current research. *Computer* 21, 7 (1988).
- [2] GOGUEN, J. A., THATCHER, J. W., AND WAGNER, E. G. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology*, R. Yeh, Ed. Prentice-Hall, Englewood Cliffs, N.J. 07632, 1978, ch. 5, pp. 80–149.
- [3] EHRIG, H., AND MAHR, B. *Fundamentals of Algebraic Specification 1; Equations and Initial Semantics*, vol. 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
- [4] MESEGUER, J., AND GOGUEN, J. Initiality, induction, and computability. In *Algebraic Methods in Semantics*, M. Nivat and J. C. Reynolds, Eds. Cambridge University Press, 1985, pp. 459–541.

- 
- [5] WAND, M. Final algebra semantics and data type extensions. *Journal of Computing System Science* 19 (1979), 27–44.
- [6] KAMIN, S. Final data types and their specification. *ACM Transaction of Programming Languages and Systems* 5, 1 (January 1983), 97–123.
- [7] BERGSTRA, J. A., AND TUCKER, J. V. Initial and final algebra semantics for data type specification: Two characterization theorems. *SIAM Journal of Computing* Vol 12, No. 2 (May 1983), 366–387.
- [8] ZHU, Z., AND JOHNSON, S. D. An algebraic framework for data abstraction in hardware description. In *Proceedings of The Workshop of Design Correct Circuits, Oxford, England* (1990), G. Jones, Ed., Springer-Verlag.
- [9] ZHU, Z., AND JOHNSON, S. D. An algebraic characterization of structural synthesis for hardware. In *Proceedings of The international Workshop on The Applied Formal Methods for Correct VLSI Designs* (1989), L. Claesen, Ed., North Holland.
- [10] JOHNSON, S. D. *Synthesis of Digital Designs from Recursion Equations*. The MIT Press, Cambridge, 1984.
- [11] JOHNSON, S. D., BOSE, B., AND BOYER, C. D. A tactical framework for digital design. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P. Subrahmanyam, Eds. Kluwer Academic Publishers, Boston, 1988, pp. 349–383.
- [12] JOHNSON, S. D., AND BOSE, B. A system for digital design derivation. Tech. Rep. TR-289, Department of Computer Science, Indiana University, Bloomington IN 47405-4101, August 1989.
- [13] GOGUEN, J. A. OBJ as a theorem prover with application to hardware verification. Tech. Rep. SRI-CS1-88-4R2, SRI International, 1988.
- [14] GERHART, S. L., MUSSER, D. R., AND THOMPSON, D. H. An overview of AFFIRM: A specification and verification system. In *Proceedings IFIP 80* (1980), S. H. Lavington, Ed., North-Holland, pp. 343–348.
- [15] ZHU, Z., AND JOHNSON, S. D. A product oriented hardware algebra. In progress, May 1990.
- [16] MCFARLAND, M. C., PARKER, A. C., AND CAMPOSANO, R. Tutorial on high-level synthesis. In *Proceedings of the 25th ACM/IEEE Design Automation Conference* (Anaheim, CA, 1988), ACM/SIGDA, pp. 330–336.
- [17] CAMPOSANO, R. Behavior-preserving transformations for high-level synthesis. In *VLSI Specification, Verification and Synthesis: Mathematical Aspects* (New York, July 1989), M. Leeser and G. Brown, Eds., Proceedings of Mathematical Sciences Institute Workshop, Cornell University, Springer-Verlag. Lecture Notes in Computer Science Vol-408.
- [18] GRÄTZER, G. *Universal Algebra*. Springer-Verlag, 1979.
- [19] JOHNSON, S. D. Manipulating logical organization with system factorizations. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects, Mathematical Sciences Institute Workshop* (July, 1989), G. B. M. Leeser, Ed., Cornell University, Ithaca, NY, USA, Springer Verlag, pp. 260–281. Lecture Notes in Computer Science Vol 408.

- 
- [20] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compiler: Principle, Techniques and Tools*. Eddison Wesley Publishing Company, 1988.
- [21] MAHMOOD, M., MAVADDAT, F., ELMASRY, M. I., AND CHENG, M. H. M. A formal language model of local microcode synthesis. In *Proceedings of The International Workshop on The Applied Formal Method for Correct VLSI Designs* (Leuven, Belgium, 1989), L. Claesen, Ed., Elsevier Science Publishers B.V.
- [22] BOYER, D. C., AND ZHU, Z. Current research of serialization problem in digital design derivation. Tech. rep., Computer Science Department, Indiana University, 1990.