Research Prospectus

Algebra for Digital Design Derivation

August, 1989

[*This is the project description for NSF grant MIP89-21842, funded for the period from August, 1989 through July, 1991—$\mathcal{SDJ}$*]

1. Introduction

This research investigates aspects of digital design in a functional algebra. The engineering paradigm is to obtain correct implementations through a sequence of algebraic transformations on a specification. This is synthesis in a formal framework; the term *derivation* is used to emphasize that source and target descriptions are dialects of a single modeling notation. A higher order functional notation is chosen for its simplicity, its power, and its affinity to digital system description.

A primary goal is to characterize methods. By casting design in a formal framework we can relate engineering tactics and look for unifying generalizations. Applications of this research lie at the frontiers of design automation, where there is the greatest need to support conceptual processes. With progress in silicon compilation and logic synthesis, these frontiers are rapidly expanding. Higher abstractions in digital engineering must be exposed in order to better integrate design techniques, reasoning methods, and tools.

The challenge is to distill valid methodology from ad hoc engineering tactics. A central topic is algebra for maintaining *orthogonal hierarchies* in a design. All aspects of digital description are subject to a compositional (i.e., structured or hierarchical) treatment. However, a given design *instance* decomposes in different ways for different reasons. Sustaining the conceptual structure of a design is difficult, and is often sacrificed to the greater need for a secure path to physcial realization. Engineers must work at levels of description from which correct hardware synthesis can be assured. Currently, these levels are relatively low, or "flat;" hence, much of the burden of interpreting real and simulated design behavior is borne by the engineer. As design costs grow in proportion to other engineering tasks, the need for abstraction increases.

This research is concerned with the mechanics of how conceptual structures are maintained in the process of implementation. At this stage, we are focused on three central aspects of design organization. *Logical organization* is the structure expressed, for example, in a functional block diagram. It is sometimes called the "structural aspect" of description. *Physical organization* is the structure of a realization, as seen, for example, in a floor plan. It is sometimes called the "geometric aspect" of description. *Temporal organization* is the way an implementation is conceived to operate in time, as specified by an algorithm. It is sometimes called the "behavioral aspect" of description.

Section 2 of this proposal surveys our previous research and outlines current formal topics. Our approach differs with contemporary work in behavioral synthesis, although it has the same objective: codifying useful engineering abstractions. Conventional synthesis evolves *from* practice, codifying useful design abstractions and techniques. A formal treatment evolves *toward* practice, specializing a comprehensive theory to the hardware implementation task. By building from established implementation paths, conventional synthesis guarantees realizability, but is often limited in its scope. Formal methods promise greater generality and rigor, but must first forge complete paths to hardware.

Thus, despite a theoretical orientation, this research is experimental in character. We devote a significant portion of effort to carrying our approach all the way to hardware, and exercising it on large problems. A mechanized algebra has been developed and used (in conjunction with logic synthesis tools) to derive several working circuits. These experiences have been crucial in exposing important problems, and should clarify this research to the practicing community. Section 3 summarizes recent experimentation and future development plans.

The proposed research advances on three frontiers. We continue to develop tools and investigate methods for algebraic synthesis. Currently, these tools serve the research purpose of exploring design at a realistic scale. As the research becomes better understood, they will develop into a useful engineering system. Second, we will further explore formal aspects of design description, along the established in previous research. Finally, we expect to make progress in the integration of this research with other treatments; in particular, mechanized logics for verificaiton. These topics are discussed in Section 4.

2. Formal Aspects of the Research

The foundations of this work lie in the areas of denotational semantics, functional programming, and comparative schemata. An early topic was to extend Wand's approach to language synthesis, which involves extraction of virtual machine descriptions from denotational language definitions [40, 8]. Our aim was to further transform these machines into specialized hardware for language execution.

The powerful methods of denotational semantics are not restricted to the implementation of programming languages. *Functional programming* applies the same methods to ordinary programming problems [17, 18]. It is a discipline in which programs are algebraically manipulated with some purpose in mind, such as performance improvement, program verification, or compilation. In other words, goal directed transformation is fundamental to the programming paraigm as well as its semantic theory. This research draws from a large body of techniques for software synthesis [10], many of which adapt directly to hardware construction.

Early research in functional programming compared its expressive power with that of imperative languages. The central result is that the class of *iterative recursion equations* (i.e. fully tail-recursive systems of function definitions) characterize finite-state control models [33]. Since finite-state models are used to specify hardware control, it follows that iterative systems can do the same. In fact, there is a direct syntactic relationship between iterative control descriptions and another class of linear recursion schemata called *sequential system descriptions,* which describe network structure and model discrete-time behavior. Our treatement of digital synthesis follows from this interative characterization [22, 23, 24].

In this approach, synthesis is really translation between dialects of a *single* modeling language of functional expressions. A *design derivation* is a sequence of expressions.

$$\mathcal{E}_0 \quad \xrightarrow{T_0} \quad \mathcal{E}_1 \quad \xrightarrow{T_1} \quad \cdots \quad \xrightarrow{T_{k-1}} \quad \mathcal{E}_k$$

Following current terminology [6], we sometimes call source expression $\mathcal{E}_0$ a 'specification' and target expression $\mathcal{E}_k$ an 'implementation'*. However, it is more accurate to say that a design is specified by the sequence of transformations, $\langle T_0, \ldots, T_{k-1} \rangle$,

* The word 'realization' is used in [22], [23], and [24], but this term is now reserved for the resulting physical circuit.

which is applied to $\mathcal{E}_0$. This sequence expresses the design intent and can, therefore, be regarded as the synthesis program.

Derivation diagrams, like the one above, are an informal notation for discussing topics of research and synthesis exercises. The engineering task is to build a sequence to satisfy a set of implementation constraints. Assuming the transformations preserve some notion of behavioral equivalence; derived implementations are correct by construction.

Much of the algebra we study is valid at any level of description—an important benefit of the approach. A specification may be expressed in terms of an arbitrary *basis type,* or vocabulary of constants, operations and tests. Let $\mathcal{C}_B$ be an interative control specification, $\mathcal{C}$ over the basis B. The characterization discussed earlier gives a mechanical translation into a system description, $\mathcal{S}_B$, expressed in terms of the same basis:

$$\cdots \quad \mathcal{C}_B \quad \longrightarrow \quad \mathcal{S}_B \quad \cdots$$

This basic transformation is called *system synthesis.* When B is relatively simple, $\mathcal{S}_B$ can be then be automatically reduced to a boolean implementation and realized by logic synthesis. When B is complex, it must be decomposed into simpler parts.

Silicon compilers typically support a fixed basis, always including binary fields, often providing for common abstractions such as *integer,* and sometimes supporting simple *array* declarations [11]. One of our goals is to provide the kind of type abstraction found in higher level programming languages. The following partial sketch of a complex basis gives a vocabulary for later discussions; it reflects the actual basis for the of the derivations reviewed in Section 3.

- A Lisp processor operates on a *heap* of *lists,* subject to access operations, *car* and *cdr*, and an allocator, *cons.*
- Atomic values in the heap are *numbers* and *strings*, each with its own set of information processing operations.
- The heap is implemented by a *memory* with *read* and *write* operations.
- A memory *address* is subject to simple arithmetic operations and tests.
- A memory *content* further decomposes into *tag* and *data* fields. One kind of datum is an *address.*
- A memory is represented by a $2^n \times m$ *RAM* device; an address by an n-bit vector; a content by an m-bit vector.
- *Bit* entites are subject to the usual boolean operations.

It is a task of the design algebra to cope with such hierarchies. The general issues are not unique to hardware, but this work is addressed to the specific problems of that domain.

2.1. Logical Organization and Factorization.

Functional specifications often treat complex data types as values. For example, a memory is often modeled as a function from addresses to contents. *Abstract component factorizations* were proposed in [22] to encapsulate such abstractions in system descripitons. Object oriented description is not excluded, but this kind of algebra permits modularity to be introduced later. In [27] this algebra is generalized to *system factorizations,* which are also used to allocate operations and manipulate communication ports.

Given a control specification, $\mathcal{C}_{T(A)}$, over an aggregate basis, $T(A)$, system synthesis produces a system description $\mathcal{S}_{T(A)}$. Intuitively, factorization decomposes $\mathcal{S}_{T(A)}$ into two subsystems:

$$\cdots \quad \mathcal{C}_{T(A)} \quad \longrightarrow \quad \boxed{\mathcal{S}_{T(A)} \quad \longrightarrow \quad \mathcal{S}^1_A \circ \mathcal{T}^1_A} \quad \cdots$$

where the '$\circ$' denotes process composition. For example, instances of the parameterized base type *memory*(*address*, *content*), factor as a MEMORY process with *address* and *content* ports.

Factorizations maintain correctness while a logical organization is imposed on a design. A byproduct of factorization is a synthesized specification of how $\mathcal{T}^1$ must behave in order to preserve the original behavior of $\mathcal{S}$. Since $\mathcal{S}^1 \circ \mathcal{T}^1$ is now expressed in terms of the simpler type A, there has been a reduction in the level of description. The *internal* description of $\mathcal{T}^1$ may still be abstract.

In [25], system factorization is distilled to six elementary laws. Thus, we are approaching a primitive of algebra for manipulating the conceptual architecture of a design. We are not so far along in formalizing the second aspect design organization, discussed next.

2.2. Physical Organization and Representation.

Logical organization is conceptual: "this register holds an address." Sometimes, a design's physical organization reflects an entirely different structure: "this block of 'logic' is a bit slice of the principal data path." Where logical organization follows the hierarchy of type parameterization, physical organization appears to follow a hierarchy of type *representation.* Its algebra has to do with replacing types rather than encapsulating them; For example, memory addresses and contents are replaced by binary vectors.

Suppose the abstract type A is represented by a concrete type R. A derivation must correctly *incorporate* R in the system description $\mathcal{S}^1_A$:

$$\cdots \mathcal{C}_{T(A)} \quad \longrightarrow \quad \mathcal{S}_{T(A)} \quad \longrightarrow \quad \boxed{\mathcal{S}^1_A \circ \mathcal{T}^1_A \quad \longrightarrow \quad \mathcal{S}^2_R \circ \mathcal{T}^2_R} \quad \cdots$$

Several issues make this an interesting problem for hardware synthesis research. Abstract operations are sometimes implemented by concrete combinations, inducing refinements to control flow. Little is known about how representations compose under the structural constraints of hardware design.

Finally, the notion varies as to what constitutes a representation. In the representation diagram below, suppose α and ρ give abstraction and representation mappings between A and R. Let F_A and F_R stand for the functions described by $\mathcal{S}^1 \circ \mathcal{T}^1$ and $\mathcal{S}^2 \circ \mathcal{T}^2$ in the derivation diagram.

$$\begin{array}{ccc} A & \xrightarrow{F_A} & A \\ {\scriptstyle\alpha}\big\uparrow\big\downarrow{\scriptstyle\rho} & & {\scriptstyle\rho}\big\uparrow\big\downarrow{\scriptstyle\alpha} \\ R & \xrightarrow{F_R} & R \end{array}$$

Hunt's ALU proof [20] has the form, $\alpha F_R = F_A \alpha$, which asserts, "Device F_R works for any *representable* value." In contrast, Boute proposes assertions of the form $F_A = \alpha F_R \rho$ which might be read, "F_R implements F_A for all abstract values." These distinctions are significant because α and ρ are not always inverses—they may not even be functions.

Furthermore, $\mathcal{S}^2$ is subject to orthogonal decomposition. This is the main topic in [27], where boolean representations are projected into bit slices. The paper

[3], touches on hierarchical aspects of restructuring in a multiple chip design (See Section 3.1). This is preliminary work at the lowest levels of representation. Future research addresses representation in more generality.

The most advanced work on this topic is by Sheeran, who defines second order functions for restructuring and retiming [37]. She has successfully applied her algebra to systolic designs and regular arrays of combinational circuitry. We plan to adapt these results to data path manipulations. The entailed algebra is exacting and crucially important, for it sustains correctness as geometric qualities are imposed on implementations. Formal verification methods have run into difficulty at this stage of design. As discussed in Section 3.3, our experimentation shows that algebraic derivation can closing the remaining gap which physical realization.

2.3. Temporal Organization and Coordination

Control is the least developed aspect of our approach, although we have some results for systems governed by single controllers. *Serializing transformations* discussed in [3, 42] manipulate control in order to satisfy architectural constraints. The relationship between 'scheduling' and 'allocation' in high level synthesis [5] is characterized in our work as an interplay between serialization and factorization (Section 2.1).

One issue is illustrated by the derviation diagram below:

$$\begin{tikzcd}
\cdots \arrow[r] & \mathcal{C}^1_{T(A)} \arrow[r, "(1)"] \arrow[d, "(2)"'] & S^2_A \circ \mathcal{T}^2_A \circ \mathcal{T}^{2'}_A \arrow[d, "?"] & \\
& \mathcal{C}^3_{T(A)} \arrow[r, "(3)"] & S^4_A \circ \mathcal{T}^4_A \arrow[r] & \cdots
\end{tikzcd}$$

A factorization (1) may require two (or more) devices, $\mathcal{T}^2 \circ \mathcal{T}^{2'}$, to perform the all the encapsulated operations. Serialization (2) produces a control description with less parallelism, so that the same factorization (3) generates just one module. In [27] for example, a factorization of arithmetic operations resulted in three ALUs, while. in [3] the same specification was first serialized to obtain a single ALU. However, we would like a more direct derivation path for such manipulations of architecture.

It is a research topic to develop more direct algebra for transforming $\mathcal{S}^2 \circ \mathcal{T}_A^2 \circ \mathcal{T}_A^{2\prime}$ to $\mathcal{S}^4 \circ \mathcal{T}_A^4$.

A related goal is to find suitable abstractions of 'process.' Most existing formalisms employ a primitive notion of *event* for this purpose. However, the indeterminacy of events is a fundamental problem for functional modeling because of difficulties in capturing infinite qualities, such as fairness and liveness. System factorization (Section 2.1) is a form of process decomposition, but it is not yet a general treatment for *sequential* processes. Again, there are alternative design paths to consider:

$$
\begin{array}{ccccccc}
\cdots & \longrightarrow & \mathcal{C}^1_{T(A)} & \xrightarrow{(1)} & S^2_{T(A)} & & \\
 & & ?\big\downarrow & & \big\downarrow ? & & \\
 & & \mathcal{C}^3_A \circ \mathcal{CT}_A & \xrightarrow{(3)} & S^4_A \circ \mathcal{T}^4_A & \longrightarrow \cdots &
\end{array}
$$

A derivation might follow the path of system synthesis (1) followed by factorization. However, when $\mathcal{T}^4$ is sequential, a protocol is involved, and our factorization algebra does not yet synthesize protocols. Control specification $\mathcal{C}^1$ might first be decomposed into communicating sequential processes, $\mathcal{C}^3$ and $\mathcal{CT}$, the latter implementing $T(A)$. From that point, deriving $\mathcal{S}^4 \circ \mathcal{T}^4$ is a a standard system synthesis (3). However, we do not yet have algebra for decomposing control specifications in this way.

Derivations applied to a verified microprocesser, summarized in Section 3.3, are concrete examples of these problems. Proposed research will explore methods for process decomposition and adapt the algbra for them. Process calculi, such as Milne's CIRCAL [34] and Gopalakrishnan's HOP [14] are attractive because of their algebraic flavor. Approaches based on temporal logics and formal-automaton models, such as Clarke's [4], Dill's [9], and Lynch's [32], have already shown promise but are much farther removed from our semantic foundations. Algebraic theories recently developed by Harman and Tucker [16] are a promising lead, as are some of Sheeran's functionals for retiming [37]. Possible problems in adopting any of these approaches include problems with higher order expression, difficulties in sustaining the relationship between control and information flow, and the need to associate verification conditions with the transformations.

2.4. Directions for Formal Research

Engineering entails balancing design concerns. Our research investigates how temporal, logical, and physical concerns interact. It leads to techniques for managing the interaction, and generalizations to other aspects of digital description. Our algebra for logical organization is the most broadly developed. Our treatment of physical organization has progressed only to the point of supporting specific representation tactics, such as bit slicing. Control manipulations are the least aspect of the algebra.

The derivation diagrams sketch a complex design space with many possible paths between specification and implementation. We have begun to identify landmarks for successful synthesis. The work continues along two fronts. We shall expand the formal topics entailed in the previous discussions, and at the same time, investigate design management through a closer integration of factorization, representation, and serialization. Mechanized support is essential, due to the exacting nature of the algebra on large designs. The next section reviews progress in automation and experimentation with mechanized derivation.

3. Experimental Aspects of the Research.

The formal constructions of [22, 23, 24] are implemented in an interactive system for digital design derivation, called DDD [26]. This research tool establishes a path to physical realization, linking the theoretical approach to demonstrable practice. The DDD system is a specialized editor, a loosely organized collection of transformations integrated with back-end logic synthesis tools. We expect DDD to evolve into a practical design tool—in fact, it has become practical in local research. However, its primary function at this time is the illumination of research issues.

In the projects described below, DDD implements all of the controlling circuitry and most of the data path. Standard components are used for memories ALUs, and so forth. Wire-wrap prototyping is done on the Logic Engine development system [36]. The target technologies have been PLD (programmable logic device) components and PLA (programmable logic array) layouts. We are now exploring tactics for other technology targets, including standard cells, path programmable logic, and generic architectures of our own design.

DDD manipulates purely functional Lisp s-expressions, which can be executed to explore design behavior. This is especially useful at higher levels of description, where the engineer is dealing with symbolic abstractions. The designs in Sections 3.1 and 3.2 were thoroughly exercised in this fashion. Such *design animation* is very much a part of the practical paradigm, but automated reasoning is by no means precluded. The derivations in Section 3.3 were done to explore the relationship between algebraic synthesis and direct verification.

3.1. A Derived Garbage Collector

The first large derivation experiment was a prototype stop-and-copy garbage collector, first realized in PLDs and then retargeted to VLSI. The upper derivation path in the diagram below is detailed in [27]. Control specification $\mathcal{C}^0$, based on a functional memory model, was transformed (1) to system description $\mathcal{S}^1$ and then factored (2) into subsystems, which included the principal data path ($\mathcal{S}^2$), two memory modules ($\mathcal{T}$ and $\mathcal{T}'$) and an arithmetic subsystem ($\mathcal{A}$). A binary representation, $A \to R$, was incorporated (3) and the system was then partitioned into PLD programs. The PLDs were programmed using available logic synthesis tools. The result was a highly parallel prototype which collected heaps at about sixty times the estimated rate of a M68000 benchmark.

$$\begin{array}{ccccccccc}
\mathcal{C}^0_{T(A)} & \xrightarrow{(1)} & \mathcal{S}^1_{T(A)} & \xrightarrow{(2)} & \begin{array}{c}\mathcal{S}^2_A \circ \mathcal{A}_A \\ \circ\mathcal{T}_A \circ {\mathcal{T}_A}'\end{array} & \xrightarrow{(3)} & \begin{array}{c}\mathcal{S}^2_R \circ \mathcal{A}_A \\ \circ\mathcal{T}_R \circ {\mathcal{T}_R}'\end{array} & \Longrightarrow & PLD \\
{\scriptstyle (4)}\big\downarrow & & & & & & & & \\
\mathcal{C}^3_{T(A)} & \xrightarrow{(5)} & \mathcal{S}^4_{T(A)} & \xrightarrow{(6)} & \mathcal{S}^5_A \circ \mathcal{A}_A \circ \mathcal{T}_A & \xrightarrow{(7)} & \mathcal{S}^5_R \circ \mathcal{A}_R \circ \mathcal{T}_R & \Longrightarrow & PLA
\end{array}$$

Essentially the same derivation (5–7) was used to derive a VLSI implementation using PLAs [3]. However, less parallelism was available, due to technology constraints and external memory requirements. A system of *serializing transformations* (4, see Section 2.3) was applied to $\mathcal{C}^0$, resulting in a refined control description, $\mathcal{C}^3$, from which PLA programs were derived. The bit-slice decomposition was further partitioned into a three-chip set. This is our best example of hierarchy in physical organization.

The PLA version of the garbage collector is the first chip for this research project, and it suffers electrical problems. However, the chip was thoroughly validated in switch level simulation, and the method of validation is of interest. Test patterns were literally computed by the specification: $\mathcal{C}^3$, executing as a Lisp program, was linked to a production heap processing system. Execution traces were fed directly to a `COSMOS` simulation of the derived circuit. Thus, the correlation of source and target behaviors was accomplished without manual intervention and without implementing an interpreter to extract the traces. This is a good demonstration of how animation with an executable modeling language can be used in a practical design environment.

3.2. A Derived Lisp Computer

A second exercise done in 1988 was the derivation of a simple lisp processor, using Henderson's specification of the *SECD machine* [17]. New topics exposed in this project have to do with integrating designs at the system level. The design, realized in PLD and MSI components, consists of four subsystems: a CPU, yet another garbage collector, a serial I/O interface, and heap initializer [41]. All but the interface were derived in DDD. The next diagram sketches the derivation.

An SECD CPU ($\mathcal{C}^1 \to \mathcal{S}^3 \to \mathcal{S}^5 \to \mathcal{S}^7$) and storage manager ($\mathcal{C}^2 \to \mathcal{S}^4 \to \mathcal{S}^6 \to \mathcal{S}^8$) were done collaterally. Both derivations followed the usual path of system synthesis, factorization, and representation. The two subsystems shared a single data path to memory, but their specifications employed different abstractions for it, designated by $T(A)$ and $U(A)$ in the diagram. Only after the representation R was incorporated could the distinct views be resolved to the more primitive $\mathcal{V}_R$.

We would like to move to higher levels of description in which the interaction of subsystems (e.g. $\mathcal{C}^1 \circ \mathcal{C}^2$) is formally specified (See Section 3.3). Resolving distinct data abstractions in a single representation is a matter for theoretical study.

Graham and Birtwistle have done a HOL verification of an SECD chip [15], which we would like to compare to the derived version. The next project explores possible integration with another verification system.

$$
\begin{tikzcd}
\mathcal{C}^1_{T(A)} \arrow[d, "(1)"'] & ? \arrow[l] \arrow[r] & \mathcal{C}^2_{U(A)} \arrow[d, "(2)"'] \\
\mathcal{S}^3_{T(A)} \arrow[d, "(3)"'] & & \mathcal{S}^4_{U(A)} \arrow[d, "(4)"'] \\
\mathcal{S}^5_A \circ \mathcal{T}_A \arrow[d, "(5)"'] & & \mathcal{S}^6_A \circ \mathcal{U}_A \arrow[d, "(6)"'] \\
\mathcal{S}^7_R \circ \mathcal{T}_R \arrow[r, "?"] & \mathcal{S}^7_R \circ \mathcal{V}_R \circ \mathcal{S}^8_R \arrow[d, Rightarrow] & \mathcal{S}^8_R \circ \mathcal{U}_R \arrow[l, "?"'] \\
& PLD &
\end{tikzcd}
$$

3.3. Derivation of the FM8501 Microprocessor

The FM8501 is a general register machine whose gate level description is mechanically verified in Boyer-Moore logic by Hunt [20]. DDD was applied to both the source and target descriptions [29]. This was easy to do because the concrete syntax of Boyer-Moore logic is exactly the language manipulated by DDD. The project was undertaken to develop a thesis for the interdependence of verification and synthesis. As witnessed by Hunt, Joyce [30], Cohn [7], and others, sequential hardware verification is greatly complicated by the need to reverify lower levels of description. Evidently, a significant part of the problem stems from structural differences between architectural and physical descriptions. Translation between these two organizations is one of our research topics (Section 2.2).

Hunt proved that an instruction level specification, called `SOFT`, is implemented by a micro-program interpreter, called `BIG`. We first attempted to derive `BIG` from `SOFT`. The derived architecture, $\mathcal{S}^3 \circ \mathcal{M} \circ \mathcal{A}$ below, was close to Hunt's, but there were significant differences. All of them were due to a change in the model of memory between the two levels of description. The derived memory $\mathcal{M}$, encapsulated `SOFT`'s functional abstraction. In `BIG`, memory is described a concurrent process, $\mathcal{P}$, and

$\mathcal{S}^4$ includes state and control for a synchronization protocol.

$$\begin{array}{ccccccc}
\mathtt{SOFT}_{M(A)} & \xrightarrow{(1)} & \mathcal{S}^1_{M(A)} & \xrightarrow{(2)} & \mathcal{S}^2_A \circ \mathcal{M}_A \circ \mathcal{A}_A & \xrightarrow{(3)} & \mathcal{S}^3_R \circ \mathcal{M}_R \circ \mathcal{A}_R \\
\{\text{Hunt}\} \Uparrow & & & & & & \\
\mathtt{BIG}_A \circ \mathcal{P}_A & \xrightarrow{(4)} & \mathcal{S}^4_A \circ \mathcal{P}_A \circ \mathcal{A}_A & \xrightarrow{(5)} & \mathcal{S}^5_R \circ \mathcal{P}_R \circ \mathcal{A}_R & \xrightarrow{(6)} & \begin{array}{c} PLA \\ \textit{programs} \end{array}
\end{array}$$

The exercise highlights the inventive aspect of Hunt's proof, which related two preconceived memory models. The next question for research is whether this aspect could be isolated and separately verified. If reasoning can be focused on the ingenious qualities of a design then the prospects for mechanical verification are improved.

We next applied DDD to BIG (BIG $\rightarrow \mathcal{S}^4 \rightarrow \mathcal{S}^5 \rightarrow PLA$), in order to explore design management as physical organization is developed. Hunt demonstrated that a gate level description could be compiled directly from BIG. However, a bottom-up compilation is explosive: BIG expands to 11 million gates before reducing to 1,800 [20]. We successfully employed DDD to guide a top-down expansion, in which the largest intermediate description was about 150,000 characters. No new tactics were involved, although the exercise did require enhancement to DDD.

Our research is evidence that algebraic manipulation is a better way of dealing with translation at lower levels of description. However, this exercise also highlights the necessity of verification. The correctness of a derived implementation follows not only from the validity of the algebra, but also from the validity of representations. In this case, Hunt's proof of a binary representation of arithmetic is an antecedent to any claim for mechanical correctness. In addition, synthesis may generate new verification conditions for the derived circuit's environment. Incorporating the process $\mathcal{P}$ for memory abstraction $\mathcal{M}$ changes the behavior of FM8501 in a way that must be reflected in its specification.

3.4. Directions for Experimental Research

We will continue to develop the DDD system as a vehicle for experimentation. The system will be applied to more and more varied designs. As is noted in Section 2.4, there will be near-term emphasis on managing alternative derivation paths. In practical terms, this requires us to develop top level facilities for strategic design management.

The experimentation will broaden in several ways. First, we will integrate more technologies, developing tactics for a broader collection of logic synthesis tools. Second, we will also apply our research at system levels, along lines established in Section 3.2, and add program transformation techniques from functional programming research. Third, we will address new kinds of problems, particularly those with a significant degree of event coordination. In particular, we will address forms of pipelining and network interface applications.

4. Review of Topics and Related Work

This is an investigation of conceptual structures used in VLSI and digital system design. Various decompositions of design are considered, with the object of distilling general principles for the integration of engineering methods and tools. The research uses a theory of functions to model design processes, which are characterized by algebraic laws. This theory gives a coherent mathematical framework for exploring the higher level abstractions of digital description. The synthesis algebra is automated, providing a tool for interactive digital design derivation. Experimentation with this sytem demonstrates the practical prospects of the approach, and helps focus the research on issues of scope and scale.

Our research emphasizes rigor and generality; we continue to apply a fully abstract modeling theory to digital engineering. Our approach emphasizes experimentation and demonstration; we develop algebra from experience with concrete examples. A central topic is managing the often conflicting decompositions of logical, physical and temporal organization. In general, we seek unified ways to maintain abstraction in design.

The primary deliverables of this research are foundational. We are defining a kernel of basic laws from which synthesis tactics are composed. However, the DDD system (Section 3) will remain available as a demonstration vehicle and over time will evolve into a viable engineering tool.

A narrow path has been established from reasonably abstract specifications to synthesizable implementations. Present research aims at securing the gaps in this path by implementing the tactics now used to carry designs to realization.

We expect to broaden the range of possible target technologies and the spectrum of designs classes considered (Section 2.4). We expect the most progress in the treatment of representation and physical organization. It appears that a subset of Sheeran's Ruby algebra [37] can be used for the task of type incorporation (Section 2.2). Abstractions of temporal organization and process decomposition are certainly needed; it is likely that our research will adopt prior results in this area (Section 2.3). Applicative process oriented languages, such as SBL [38] and HOP [14], are attractive, although temporal logic formalisms, such as SML [4], appear to support better reasoning capability.

The most exciting recent development has been in the relationship between algebraic synthesis and formal verification. The practical prospects for both approaches are improved as ways are found to integrate them. The FM8501 exercise (Section 3.3) demonstrates that a mechanized functional algebra can contribute to mechanically assured correctness. In particular, it was established that DDD could manage the abstractions used by Hunt in his machine descriptions. This particular relationship will be explored further.

References

[1] Bickford, Mark and Mandayam Srivas. Hardware Verification using Clio. In M. Leeser and G. Brown (eds.), *VLSI Specification, Verification and Synthesis: Mathematical Aspects* (Proceedings of Mathematical Sciences Institute work shop, Cornell University, July, 1989), Springer, New York, in preparation.

[2] Boute, Raymond. Systems Semantics: Principles, Applications and Implementations. *ACM Trans. Prog. Lang. and Systems,* **10**(1):118–155 (1988).

[3] Boyer, C. David, and Steven D. Johnson. Using the Digital Design Derivation System: Case Study of a VLSI Garbage Collector. In J. Darringer and F. Ramming (eds.) *Proceedings of the IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages* (CHDL-89), Elsevier, Amsterdam, in preperation.

[4] Browne, Michael C., Edmund M. Clarke, and David L. Dill. Automatic Circuit Verification using Temporal Logic: Two New Examples. In G. Milne and P. Subrahmanyam(eds.) *Formal Aspects of VLSI Design*, North-Holland, Amsterdam, 1986, 113–124.

[5] Camposano, Raul. Behavior-preserving Transformations for High-Level Synthesis. in M. Leeser and G. Brown (eds.), *VLSI Specification, Verification*

and Synthesis: Mathematical Aspects (Proceedings of Mathematical Sciences Institute work shop, Cornell University, July, 1989), Springer, New York, in preparation.

[6] Camurati, Paolo and Paolo Prinetto. Formal Verification of Hardware Correctness: Introduction and Survey of Current Research. *Computer*, vol. 21, No. 7, 1988.

[8] Clinger, William D.. The Scheme 311 Compiler: An Exercise in Denotational Semantics. *Conf. Record of the 1984 ACM Symp. on LISP and Functional Programming,* Austin, August, 1984, 356–364.

[7] Cohn, Avra. Correctness Properties of the Viper Block Model: The Second Level. Preliminary papers for the Banff Hardware Verification Workshop, June 1988, proceedings to appear.

[13] Borriello, G., and E. Detjens. High-Level Synthesis: Current Status and Future Directions. *Proc. 25th ACM/IEEE Design Automation Conference*, Anaheim, June, 1988.

[9] Dill, David L.. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* Ph.D. dissertation, Carnegie-Mellon University, 1988.

[10] Friedman, Daniel P., *et. al.*. *Programming Languages: Abstraction, Representation, and Implementation.* In progress.

[11] Gajski, Daniel D. (ed). *Silicon Compilation.* Addison-Wesley, Reading, 1988.

[12] German Steven M., and Yu Wang. Formal verification of parameterized hardware designs. *Proc. IEEE International Conference on Computer Design: VLSI* in *Computer,* 1985.

[15] Graham, Brian and Graham Birtwistle. Formalising the Design of an SECD Chip. In M. Leeser and G. Brown (eds.), *VLSI Specification, Verification and Synthesis: Mathematical Aspects* (Proceedings of Mathematical Sciences Institute work shop, Cornell University, July, 1989), Springer, New York, in preparation.

[14] Gopalakrishnan, Ganesh. Specification and Verification of Pipelined Hardware in HOP. In J. Darringer and F. Ramming (eds.) *Proceedings of the IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages* (CHDL-89), Elsevier, Amsterdam, in preperation.

[16] Harman, N.A. and J.V. Tucker. Clocks, Retimings, and the Formal Specification of a UART. In G.J. Milne(ed.) *The Fusion of Hardware Design and Verification*, North- Holland, Amsterdam, July, 1988, 375–396.

[17] Henderson, Peter. *Functional Programming, Application and Implementation.* Prentice-Hall, Englewood Cliffs, NJ, 1980.

[19] Hill, Fredrick J., and Gerald R. Peterson. *Introduction to Switching Theory and Logical Design.* John Wiley and Sons, New York, 1981, Third edition.

[20] Hunt, Warren A., Jr.. *FM8501: A Veified Microprocessor.* PhD. dissertation, The University of Texas at Austin. Also published as Technical Report 47 (December, 1985) Institute of Computing Science, The University of Texas at Austin, 1985.

[21] Johnson, Steven D.. Circuits and Systems: Implementing Communication with Streams. *Proc. 10th IMACS World Congress on Systems Simulation and Scientific Computation*, vol. 5, *eds.* W.F. Ames and R. Vichnevetsky, Motreal, August, 1982.

[22] Johnson, Steven D.. *Synthesis of Digital Designs from Recursion Equations.* The MIT Press, Cambridge, 1984.

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

[24] 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.

[25] Johnson, Steven D.. Manipulating logical organization with system factorizations. In M. Leeser and G. Brown (eds.), *VLSI Specification, Verification and Synthesis: Mathematical Aspects* (Proceedings of Mathematical Sciences Institute work shop, Cornell University, July, 1989), Springer, New York, in preparation.

[26] Johnson, Steven D. and Bhaskar Bose. A system for digital design derivation. Submitted abstract.

[27] Johnson, Steven D., Bhaskar Bose, and C. David Boyer. A Tactical Framework for Digital Design. In G. Birtwistle and P.A. Subrahmanyam (eds.) *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers, Boston, 1988, 349–383.

[28] Johnson, Steven D. and C. David Boyer. Modelling Transistors Applicatively. In G.J. Milne(ed.) *The Fusion of Hardware Design and Verification*, North-Holland, Amsterdam, July, 1988, 77–98.

[29] Johnson, Steven D., Robert M. Wehrmeister and Bhaskar Bose. On the Interplay of Synthesis and Verification: Experiments with the FM8501 Processor Description. Submitted for publication.

[30] Joyce, Jeffrey J.. Multi-Level Verification of a Simple Microprocessor. Progress Report, Computer Laboratory, University of Cambridge.

[31] Keutzer, Kurt and Wayne Wolf. Anatomy of a Hardware Compiler. ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta Georgia, 1988.

[33] Manna, Zohar. *Mathematical Theory of Computation.* McGraw- Hill, New York, 1974.

[34] Milne, G. J.. CIRCAL and the representation of Communication Concurrency and Time. *ACM Transactions on Programming Languages and Systems,* **7**(2), (1985).

[35] O'Donnell, John T.. Hardware Description with Recursion Equations. *Proc. 8th International Symposium on Computer Hardware Description Languages and their Applications*, North-Holland, Amsterdam, April, 1987, 363–382.

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

[37] Sheeran, Mary. Retiming and Slowdown in Ruby. In G.J. Milne(ed.) *The Fusion of Hardware Design and Verification*, North-Holland, Amsterdam, July, 1988, 289–308.

[38] D.R. Smith and Mandayam Srivas. Hardware Specification and Testing using SBL. Preliminary papers for the Banff Hardware Verification Workshop, June 1988, proceedings to appear.

[39] Verkest, D., *et. al.*. Formal Techniques for Proving Correctness of Parameterised Hardware using Correctness Preserving Transformations. In G.J. Milne(ed.) *The Fusion of Hardware Design and Verification*, North- Holland, Amsterdam, July, 1988, 77–98.

[40] Wand, Mitchell. Deriving Target Code as a Representation of Continuation Semantics. *ACM Trans. Programming Languages and Systems* 4(3), 496–517.

[41] Wehrmeister, Robert M. The Derivation of an SECD Machine: Experiences with a Transformation System (working title). Indiana University Computer Science Department Technical Report, in progress.

[42] Zhu, Zheng X. and Steven D. Johnson. An Algebraic Characterization of Architectural Constraints in Hardware Descriptions (working title). Submitted.