Digital Design Derivation

1. Introduction

This research applies formal methods in logic, verification, and synthesis to digital design engineering. The work centers on the use of applicative notation for system description and functional algebras for design development. The general goal is to develop a comprehensive methodology for constructing correct hardware implementations. The research objectives are to establish a practicable algebraic framework for digital design, to clarify its relationship to conventional design methods, to explore its implications for design automation, and to incorporate contemporary formalisms for hardware verification.

We shall also explore the use of functional programming languages in hardware modeling and design-aids automation. This is is almost compulsory because the automation vehicle must readily represent the governing formalism. Automation of this work depends on the manipulation of functions, a capability not generally available. Higher-*order* constructs are of more immediate interest than higher-*level* specializations of syntax. Our view is that digital system description is one *dialect* of a computationally complete—and executable—modeling notation. We seek to reduce the hierarchy of representation and interpretation common in design aids systems, so that the integration of tools for modeling, simulation, analysis, and fabrication is more direct.

Past research in this area outlined an approach to "behavioral synthesis" in which sequential-system descriptions are derived from recursive specifications. This work is reported, in diminishing detail, in [9], [10], and [11], the last of which is appended. In the past year, some of these theoretic constructions have been implemented. Enough function has been achieved to generate hardware in PAL technology, and PLA generation in VLSI is at hand. The status of this work will be reported in [13], a draft of which is appended.

Though more mathematical rigor is needed to realize the prospects of present technology, it is equally important to see that rigor carried all the way to implementation. Our efforts in hardware generation have focused attention on the translation of abstract behavioral descriptions into concrete hardware descriptions. Though existing theory seems adequate for this task, at least for practical purposes, its potential is not realized. The dominant problem—maintaining hierarchies of structure—is often dismissed as theoretically superficial. Even so, any effort to attain descriptive abstraction must also provide secure paths to concrete levels, and this necessitates coping with structure. The underlying problems are formally challenging because designs are subject to orthogonal decompositions. Numerous structural hierarchies must be simultaneously maintained.

The use of electrical simulation to explore logical design is a typical symptom of the central research problem. Low-level simulation is often the only *secure* software connection to physical implementation. The engineer is willing to forgo the advantages of behavioral modeling unless correct fabrication can be assured. The use of "silicon compilers" still leads to low level simulation as compiled subsystems are put together. Research in digital description and verification has followed the same path as efforts in design automation, attacking individual aspects of design. Little work is addressed to organizing the global design effort. It can be expected that formal research will be confronted by the same integration problems faced in software design-aids. These problems are generally acknowledged as profound.

Digital engineering is *selective* refinement to lower levels of detail. Various interacting tactics are involved. Each aspect of refinement is subject to independent treatment, but in any design *instance,* these aspects become dependent facets. Since there are many degrees of freedom in design, more descriptive latitude is needed. The tendency of production systems to codify specific techniques leads inevitably to a morass of incompatible tools. Hardware modeling research has sought the necessary generality in higher-order calculi. To the degree that descriptive abstraction is attained, engineering can be initiated and executed at higher levels. However, abstraction increases the distance between specifications and implementations, further necessitating automation of the implementation task.

Though it is often held that good design is hierarchical, it is naive to conclude that physical organization follows logical decomposition. A hardware system represents many *disparate* logical hierarchies. Packaging, routing, path-width parameterization, performance constraints, testability requirements, functional decomposition, and so on, each projects design in a different dimension. The practicing engineer constantly juggles these projections, often narrowing to an individual path or element. The intent of proposed research is to exhibit one semantic framework that maintains arbitrary decompositions of description. This immediately raises the fundamental issue of meaningful interaction. Intelligent guidance must be assumed, but where automation subsumes engineering tactics there is a real danger of loosing the sense and direction(s) of design.

We are interested in developing notations and techniques that are conducive to creative expertise. Engineering participation is essential to the progress of this work. We enjoy a good relationship with qualified digital engineers, Franklin Prosser and David Winkel, who provide insights into design methods and strategies. We are now engaged in several design projects that together constitute an experimental laboratory for this research. These include an MSI implementation of a full computer system and a number of gate-level designs for storage architecture. The work leading to this proposal has already produced working implementations, one of which is discussed in Section 2.

Under the NSF CER program, this department is now building a research facility with a strong component of hardware design tools. Winkel and Prosser have developed, in-house, an excellent support vehicle for wire-wrap prototyping and microcode development. Full design-aids systems for VLSI have been acquired in the last year. The period of this grant will see several substantial designs derived as a result of the proposed research.

This work adapts results, techniques, and constructions from software-related research, exploiting results in denotational semantics, program transformation, compiler construction, and comparative schemata. The approach to hardware description correlates with software-oriented research done in this Department [2]. We expect to strengthen a connection between software engineering research and hardware engineering research, as they are conducted here.

2. Background

We describe digital systems in *applicative notation.* The fundamental expression is a *term,* such as $f(x, c)$. An *abstraction* is the expression of a function in lambda-notation; $\lambda X.E$ denotes the function with rule E and parameter X. When abstractions are named, it is conventional to omit the lambda, writing

$$AndNot(u, v) = and(u, not(v))$$

instead of

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

Often, abstractions serve the same purpose as symbols in schematics; the formal parameters giving internal names.

Functions may be freely expressed and applied. For example,

$$compose(f, g) = \lambda x.f(g(x))$$

applies its arguments in series. Functionals like *compose* are fundamental to any transformation algebra but also arise in descriptive contexts, often as a means to suppress representation details. An example can be seen later.

A *ground type*, is a basis of primitive operations, constants, and predicates, together with a collection of rules that relate them. It may be an aggregate of (more) *complex* and (more) *concrete* subtypes, such as "arrays of integers" or "functions on characters." Though it is assumed that types are developed as domains [23], it is often appropriate simply to address descriptions to some primitive vocabulary.

A *discrete interpretation* of applicative notation assigns individual values to variables; a term denotes the application of a function to argument-values. From this model is developed the language of recursive function definitions [16, 15] and the more abstract algebra of denotational semantics [23].

To describe systems, the discrete interpretation is extended to a *sequential interpretation.* If V is a set of values, then *behaviors of* V is the set of (computable) sequences $(v_0, v_1, \ldots)$ where each $v_i \in V$. The *signal* $\boxed{f}\,(X, \boxed{c}\,)$ denotes the sequence S in which $S_i = f(X_i, c)$. (The boxes are a convention to distinguish sequential expressions.) An operator '!' prefixes values to sequences; it often denotes storage/delay.

$$X = v\ !\ S \quad \text{means that} \quad X_0 = v \text{ and } X_{k+1} = S_k.$$

Sequential systems are described by simultaneous *signal*-defining equations, such as

$$SP(I) = O \quad \text{where} \quad \begin{cases} C = & \texttt{true}\ !\ I \\ O = & \boxed{\mathit{and}}\,(I, \boxed{\mathit{not}}\,(c)) \end{cases}$$

Recursions describe feed-back, although this example has none. *SP* is a logical description of a "single pulser" circuit [24], which produces a unit pulse for every pulse on its input, I.

It is intuitive to regard *SP* as a recurrence relation for sequences I, C, and O (as, for example, in [7]). However, signals may range over higher-order domains. In [13], for example *SP* is transformed to

$$SP(I) = O \text{ where } \left\{ \begin{array}{rl} C = & \boxed{\mathit{Abst}}\,(\texttt{true}\ !\ I) \\ O = & C(I) \end{array} \right.$$

where

$$\mathit{Abst}(v) \stackrel{\text{def}}{=} v \ \rightarrow\ [\lambda v.\texttt{false}],\ [\lambda v.v]$$

This was done to bridge levels of description; the two values of *Abst*, constant-`false` and the identity function, suggesting a pass transistor in NMOS. The symmetry in treatment of signals and components, in which both may be computed and applied, is of technical interest in our research. It is an extension of "proper abstraction," which has proven its worth in programming languages. Such treatments seem warranted, if not in the direct description of hardware, then certainly for systems (both formal and programmed) that describe how hardware is described.

In [9], a technique is developed to derive digital system descriptions from recursive-function specifications. The key fact is that the class of *iterative* recursion schemata characterizes synchronous digital systems. Synchronous systems are typically implemented with externally clocked hardware, but self-timed or buffered implementations are just as well described. Since the proof of the characterization is constructive, derivation is mechanizable. Iterative schemata have long been known to be equivalent to finite-state control descriptions; the derivation techninque recasts standard methods for control synthesis (e.g. [24]) in a functional algebra. However, unlike standard methods, the construction also develops the connectivity of *controlled* elements. That is, the extraction of the controlling subsystem establishes, as a by-product, a correct global description of architecture. Hence, the construction yields a correct starting point for subsequent refinement of architecture.

The theoretic construction in [9] has been implemented and used to develop working circuits. The research objective, now partly established, was to forge a path to actual hardware by restricting the class and level of specifications (discussed later), by addressing hospitable target technologies (e.g. programmable PALs and standardized gate arrays), and by exploiting available analytic tools (e.g. for boolean simplification, and fuse-map generation). A kind of editor has been developed for executing various correctness-preserving transformations on sequential descriptions. It employs the familiar algebra for terms with specializations for expressions of delay.

The "back end" of this prototype editor is a collection of syntax transducers that generate source input to available hardware-generation systems. One such interface has produced working circuitry in PAL technology. Partial VLSI layouts have been generated, but we are at least a year from attempting fabrication. For reasons that will become obvious, a path to printed-wire boards is of more immediate concern.

The physical implementation of behavioral descriptions entails descending to successively more concrete levels of description. There are two interacting aspects in this process: isolation of relevant subsystems and imposition of representations. The editor provides some basic algebra for the first, but, at this time, none for the second.

Given an initial behavioral specification, one task is to segregate that portion which is subject to detailed design. This problem has received little attention, but is a key to maintaining global correctness. We have implemented a general transformation, called *system factorization*, which encapsulates subsystems. *What* to factor is now a matter of judgement; and, in some cases, *how* to factor is determined interactively. From the exercises we have done, some typical reasons for factorization are the following (there are numerous examples in the appendices):

- *Combination.* This is simply the enclosure of a collection of terms by the formation of a combinator, a "macro." The first version of *SP,* above, might be expressed as

$$SP(I) = O \quad \text{where} \quad \begin{cases} C = \boxed{\mathit{Delay}_{\mathrm{t}}}\,(I) \\ O = \boxed{\mathit{AndNot}}\,(I, C) \end{cases}$$

  where

$$\mathit{AndNot}(u, v) \overset{\text{def}}{=} \mathit{and}(u, \mathit{not}(v))$$

  and

$$\mathit{Delay}_{\mathrm{t}}(S) \overset{\text{def}}{=} \texttt{true} \,!\, S$$

  SP itself is a combination and might have been factored from a larger system.

- *Information Hiding.* A more serious kind of factorization is done to encapsulate complex objects of the ground type. It is often a tactic to isolate internal data paths of a register architecture from the external structures (e.g. memories, stacks, co-processes) they operate against. In this context, it is called *abstract component factorization* in [9, 10, 11].
- *Targeting.* A typical example is the encapsulation of arithmetic operations as a multi-function arithmetic unit. This involves the synthesis of an instruction signal to govern the unit's function and the introduction of selector-terms to switch operands. There are many criteria for targeting, but we have found that in moderately large designs, reasonable factorizations can be found after a few trials.
- *Packaging.* The growth of programmable logic has increased the importance of partitioning descriptions into physical modules. For example, a "bit slice" partitioning can exploit the massive switching capabilities of array logic to eliminate the bussing of conventional register-transfer designs. Bit slicing is one example of how physical modularity can be completely orthogonal to functional modularity. For this reason, the translation of a functionally oriented behavioral description to a physically oriented circuit description is pervasive, although the transformations are based on simple algebra.

The second aspect of concrete design derivation is the introduction of representations for the ground type. This is different from the objective of system factorization, which is essentially to isolate a level of description. Representation is the *replacement* of one type by another. Briefly, given a type V one must present a set R and an interpretation, $\alpha\colon R \to V$, saying what value each element of R represents. An operation $f\colon V \to V$ is implemented by $F\colon R \to R$ if for all $r \in R, \alpha F(r) = f(\alpha r)$; that is, if F gets the right answer for every representable input value. (There are other versions of this statement, but this is the one used in most hardware verification research.)

In hardware, the most concrete type is usually binary logic. Hence, values in V are represented by tuples, $R = \mathit{Bit}^n$; and, to make matters worse, n may be left variable until the final stages of implementation. In the course of a design, "addition" might reasonably acquire a spectrum of signatures, such as

$$(\mathit{Integer} \times \mathit{Integer}) \;\to\; \mathit{Integer},$$

$$(\mathit{Bit}^n \times \mathit{Bit}^n) \;\to\; (\mathit{Bit}^n \times [\mathit{Carry}]\mathit{Bit}),$$

even

$$(\mathrm{Int} \times \mathit{Bit}^n) \;\to\; (\mathit{Bit}^{n+1}),$$

and ultimately

$$\mathit{Bit}^{2n} \;\to\; \mathit{Bit}^{n+1}.$$

Since we do not yet have automated means to address representation, our experiments have involved simple hierarchies, but even so, the experience is encouraging.

In the development of a garbage collector [13], factorizations were used to isolate the data paths for a register architecture. Communicative specifications for the factored modules (memories, arithmetic logic) were synthesized, while a correct global description was maintained. State generation and control encoding were automatically derived (following [9]), simplified, and assembled to programmable PALs after manual assignment of binary representations for symbolic values. What remained was a system of signal equations expressed in terms of record manipulation in the ground subtypes of "addresses" and "contents." These project to identity functions in the binary representation; hence, any group of data-path signals, reinterpreted in binary-logic, constituted a plausible bit slice. Packaging was a matter of selecting appropriate groups for automatic assembly into PAL. The result was eight distinct programs for thirty-four PALs, which comprised the registers and control for collecting a twenty-four bit address space.

The contribution of developmental work is its demonstration that the over-all organization of a design problem can be secured in an natural framework. Its purpose is to manage the exacting detail of design manipulation as various intellectual hierarchies are explored. We were able to project arbitrary subsystems into programmable technology, while preserving global coherence. This is crucial to the assurance of correct implementations. The whole of a digital design effort is a complex interleaving of tactical decompositions, whose general organization demands further study. While it is certainly true that the analysis and reasoning for specific tactics must continue to be researched, too little attention is paid to supporting the interplay of tactics.

Proposed Research

The experimental aspects of this work are driven by the need to address alternative technologies and more subtle design strategies. Much of this broadening comes through the piecemeal incorporation of classical techniques. In this, the primary concern is how these techniques might be unified in an abstract algebra. Such formalism taxes the patience of participating engineers, but has contributed to new insights into the use of current technology [25]. A purely functional perspective is also shedding light on modeling and simulation of hardware *and* software systems [12, 20].

Formal hardware description is approaching benefit to practice, but it remains to be demonstrated how more rigorous foundations can advance the engineering discipline. We hope to incorporate some good work by Sheeran [21, 22] in our more general framework. Our experimentation leaves little doubt about topics to be addressed in the continuation of this work.

We must first attack the representation of *simple* objects: getting from terms, say, of integers to terms of bits. Though the more general problems of higher-order representation are certainly relevant, they are not as urgent. This topic brings questions of type inference to the forefront. Type mismatches are the most common descriptive errors and the most difficult to correct. Type translations can be extremely confusing. Similar sentiments can be found in many related publications; we quote below from the conclusion of a survey by Boute [1]

> Additional streamlining is required for the second-order system semantics of sequential circuits. A promising approach appears to consist in formalizing and extending [an *FP* algebra] and introducing a formal type definition language into [system description languages].... Experience with our notation for describing complex computer architectures indicates the desirability of such a type language for other purposes as well.

In our experience, the last sentence is an understatement, but an understandable one because the dominant typing problems in hardware are structural: having to do mainly with product formation. The entailed inference is regarded as theoretically mundane, but there is a deeper issue at play. Temporal behavior, logical decompositions, physical decomposition, binary representation, all involve products. A VLSI description is virtually pure structure. It is difficult to maintain the intent of structurings, in part *because* they are formally indistinguishable.

One topic of proposed research is to develop and implement algebra for the imposition of representations in concrete descriptions. Mechanisms are needed to declare representations, to check for consistency, do trivial coercions, and to unify distinct objects through their representation.

Boute's work seems the closest to our own in notation and motivation. We are in near agreement with his notion of "open semantics," where a single descriptive syntax is subjected to a various interpretations (besides the discrete/behavioral models) for the purpose of analysis and implementation. Applied to automation, this means that a single circuit description is *executed* in distinct environments, to obtain logical, analytical, and geometrical information. Locally, O'Donnell has done the most work on this subject, showing how a description generates behavioral simulations, wiring lists, connectivity graphs, and layouts [20].

The thesis that circuit analyses and fabrication can be done in a compositional manner needs close examination. It can probably be made *true by adding more descriptive structure and semantic abstraction. This approach leads to more dimensions of of orthogonal decomposition, the central problem considered in this proposal.*

Our main difference with Boute is our use of reflexive domains in behavioral modeling; he proposes a more rigorous hierarchy of order, and makes a credible assertion that underlying types can be constructively built [1]. In our view, full abstraction should not be precluded from descriptive bases, but this perspective comes from a background in programming languages, not digital engineering. Gordon cogently argues for a freer use of higher-order constructs in digital verification [4]. Our motivation has more to do with specification:

It is a long-term goal to extend the frontier of what constitutes a hardware specification. We seek to develop a broader class of of functional descriptions from which digital implementations can be systematically obtained.

A more concrete goal is to incorporate a logic for reasoning about sequential systems. Gordon makes an attractive case for a pure predicate calculus [4], Mozskowski for a temporal logic [19], and Milne for a communicating calculus [17]. All provide a mechanized support specialized to hardware. A more immediate prospect is the verification work of Hunt [8].

Hunt's is one of the most exhaustive hardware verification exercises to appear in the literature. He employs the Boyer-Moore system to mechanically verify the correctness proof of a microprocessor description. German and Wang also use Boyer-Moore to prove *classes* of circuits [3]. Other microprocessor proofs have been done at Cambridge [5] and Calgary [14]. Hunt's approach is attractive for two reasons. First, it is based in applicative notation and can most readily be integrated in our development project. Second, Hunt (and also German and Wang) directly addresses the validity of binary representations.

There are two independent parts to Hunt's proof. First, he establishes gate-level implementations the machine's ground type (integer arithmetic). He then proves that a micro-coded interpreter implements the microprocessor's instruction semantics. This second proof is carried out in the more abstract type. Hunt's "verified microprocessor" is just a textual description, of course, given at mixed levels of detail. It is the kind of description, that we foresee transforming to physical hardware. The processor is specified in a form (it is an iterative recursion equation) that we have already shown be can be translatable to architecture. The subsequent incorporation of binary representations is the main topic of proposed research.

> *We see in Hunt's work a good opportunity to demonstrate the general relationship between direct verification and transformational development, applied to hardware [18]. The contribution of the proposed research is to develop a secure framework for the incorporation of verified subsystems in surrounding designs.*

In summary, the next two years of research focus on the latter stages of hardware derivation. The primary goal is to bridge lower levels of representation hierarchy by exploring the algebra needed to manipulate descriptive structure. Our experimental work will broaden the tactics and target media for implementation. Formal research centers on expanding the classification of functional specifications from which hardware may be obtained, and in particular, to study the use of recursive types in behavioral description. An immediate topic is to establish a connection to hardware verification research.

References

[1] Raymond T. Boute, "Current work on the semantics of digital systems," in: G. J. Milne and P. A. Subrahmanyam (eds.), *Formal Aspects of VLSI Design*, 1986, pp. 99-112.

[2] Daniel P. Friedman, Chris T. Haynes, Eugene Kohlbecker, and Mitchell Wand, *Programming Languages: Abstractions and their Implementations*, [working title] in progress.

[4] Mike Gordon, "LCF_LSM: A system for for specifying and verifying hardware," University of Cambridge Computer Laboratory Technical Report No. 41.

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

[6] Mike Gordon, "Why higher-order logic is a good formalism for specifying and verifying hardware," in: G. J. Milne and P. A. Subrahmanyam (eds.), *Formal Aspects of VLSI Design*, 1986, pp. 153–177.

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

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

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

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

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

[12] Steven D. Johnson, "Circuits and systems: Implementing communication with streams," *IMACS Transactions on Scientific Computation, Vol. II*, (ed.) M. Ruschitzka, (1983) 311–319.

[11] Steven D. Johnson, "Digital design in a functional calculus," in: G. J. Milne and P. A. Subrahmanyam (eds.), *Formal Aspects of VLSI Design*, 1986, pp. 45-58 [Proceedings of the 1985 Edinburgh Workship on VLSI].

[13] Steven D. Johnson, Bhaskar Bose, and C. David Boyer, "A tactical framework for digital design," to appear in the proceedings of the Hardware Verification Workshop, University of Calgary, January, 1987; *draft appended.*

[14] Jeffery Joyce, "The SECD Machine: A Study in Advanced Architecture," [unnumbered] Department of Computer Science, The University of Calgary [probably 1984].

[15] Jacques Loeckx and Kurt Sieber, *The Foundations of Program Verification,* translated by Ryan D. Stansifer, B.G. Teubner, Stuttgart, John Wiley & Sons, New York, 1984.

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

[17] George J. Milne, "CIRCAL: a calculus for circuit description," *INTEGRATION* **1** Nos. 2 and 3, 1983.

[18] George J. Milne, "Simulation and verification: related techniques for hardware analysis," *Proc. 7$^{\text{th}}$ Int. Symp. on Computer Hardware Description Languages and their Applications (CHDL '85), Tokyo,* North-Holland, 1985.

[19] Ben Moszkowski, "Executing temporal logic programs," University of Cambridge Computer Laboratory Technical Report No. 55 (1984).

[20] John T. O'Donnell, Hardware Description with Recursion Equations, *submitted for publication.*

[21] Mary Sheeran, "Designing regular array architectures using higher order functions," in J-P Jouannaud (ed.), *Functional Programming Languages and Computer Architecture,* Springer [*LNCS No. 201*], Berlin, 1985.

[22] Mary Sheeran, "muFP, a language for VLSI design," *Conf. Rec. 1984 Symp. on LISP and Functional Programming,* (August, 1984), 104–112.

[23] Joseph E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, 1977.

[24] David E. Winkel and Franklin P. Prosser. *The Art of Digital Design,* Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

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