INDIANA UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

TECHNICAL REPORT NO. 372

Integrating Boolean Verification with Formal Derivation

Bhaskar Bose, Steven D. Johnson, and Shyamsundar Pullela

FEBRUARY 1993

To appear in the proceedings of the *1993 IFIP Conference on Hardware Description Languages and their Applications* (*CHDL '93*), Ottawa, Canada, April, 1993.

Integrating Boolean Verification with Formal Derivation [†]

Bhaskar Bose, Steven D. Johnson, and Shyamsundar Pullela
Indiana University
Computer Science Department
Bloomington, Indiana

February 1993

Abstract

This summary describes results in integrating formal derivational reasoning with low level verification. The reported work is part of a project to construct an FPGA realization of Hunt's FM9001 Microprocessor description by applying the DDD (Digital Design Derivation) System in conjunction with low level verification systems. The purpose is to study the interaction between derivation and verification in hardware design. The result of this work is a derived FM9001 implemented in FPGAs defined by a rigorous path to hardware which integrates both derivation and verification.

1 Introduction

Philosophically, *derivation* and *verification* represent contrasting approaches to design. Derivation aims at deriving a "correct by construction" design. Verification aims at constructing a "proof of correctness" for a post factum design. However, as researchers and engineers gain design experience in a formal framework, both approaches are emerging as interdependent facets of design [12]. Experience shows that derivation systems impose restrictions on the design, and verification systems result in design descriptions which are impractical for implementation [7], or proofs too difficult to construct for an arbitrary specification [1, 4].

[†] Research reported herein was supported, in part, by NSF: The National Science Foundation, under grants numbered DCR 85-21497, MIP 87-07067 and MIP 89-21842, and by NASA: The National Aeronautics and Space Administration under grant number NGT-50861.

From the perspective of derivation, a derivation system based on a fixed set of rules for building hardware restricts the design space. There are too many aspects of design which are unaccounted for in any practical derivation system. From the perspective of verification, provability must be considered throughout the design process. In practice, it is difficult to engineer a proof, so one must design for verification.

Our research sets out to address deficiences in both derivation and verification by integrating them in a unified framework. In our framework, we define a formalism for hardware derivation based on the algebraic manipulation of purely functional forms [9]. The derivation system, DDD, [11, 8] is used to decompose and restructure a design in such a manner as to derive a significant portion of the design and isolate the verification problems to small building blocks. Verification is then applied to the isolated building blocks. The intuition is that for a given design, the relationship between its specification and implementation has two aspects. One aspect of the implementation can be systematically derived from the specification. This portion, though critical, represents the uninteresting part of the design. The other aspect of the design represents the interesting engineering in the design and is better addressed by verification.

The DDD-FM9001 project was an experiment to construct an implementation of the FM9001 [3] using this approach. We applied the DDD system to the Nqthm [2] specification of the FM9001. A set of transformations were applied to decompose and reorganize the design. Complex components, such as the memory, register file, ALU, incrementor, and decrementor, were isolated using DDD's abstraction mechanisms. Technology dependent, highly optimized implementations of the ALU, incrementor, and decrementor were engineered and verified against their respective isolated components. The derived and verified components were then assembled into a single ACTEL FPGA. The memory and register file were implemented externally with SRAM components.

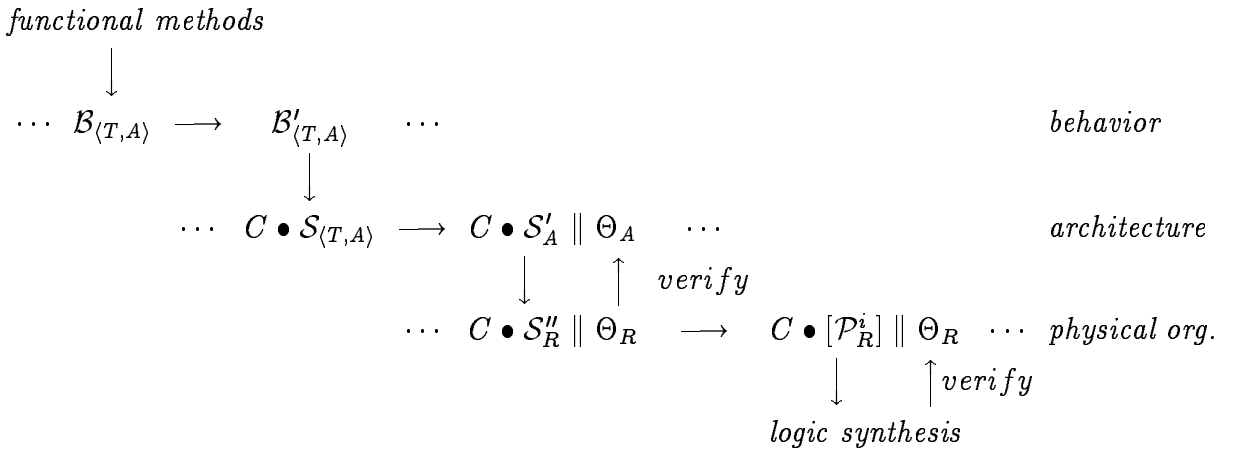
Section 2 describes the DDD system and the methodology surrounding design by algebraic transformation. A characterization of the derivation process is outlined defining a path from behavior to structure and then to physical organization. Section 3 gives an introduction to the DDD-FM9001 project highlighting some of its goals. A brief sketch of the derivation of the DDD-FM9001 is given. Section 4 describes the Boolean verification methodology which was integrated into the derivation. The steps involved in verifying the DDD-FM9001 ALU are described. In section 5, we reexamine our goals for DDD in light of the DDD-FM9001 project.

2 DDD - Digital Design Derivation System

DDD (Digital Design Derivation System) is a transformation system which implements a basic design algebra for synthesizing digital circuit descriptions from high level functional specifications. DDD is much like a proof checker in the sense that it automates the transformations

needed for circuit synthesis, but requires substantial guidance to perform a derivation. The system is implemented in the Lisp dialect Scheme [6, 13] as a collection of transformations that operate on s-expressions. DDD was developed to help explore and demonstrate a formalization of digital design based on a functional algebra. The system is used interactively to transform higher level behavioral specifications into hierarchical boolean systems to which logic synthesis tools are then applied.

In DDD, a sequence of transformations are applied to an initial specification defining a *derivation path* towards an implementation satisfying an intended set of design constraints. Design tactics and constraints imposed by the designer sketch a complex design space with many possible paths between specification and implementation. However, in practice the derivation has distinct phases. The diagram below is a characterization of the derivation. In the derivation path below, $\mathcal{X}_B$ stands for a DDD expression, $\mathcal{X}$, written in terms of a ground type, B.



The initial behavior description $\mathcal{B}_{\langle T,A\rangle}$ is in a restricted class of *iterative* function-definition schemes. Behavior descriptions are folded and unfolded ($\mathcal{B}_{\langle T,A\rangle} \rightarrow \mathcal{B}'_{\langle T,A\rangle}$) to achieve a proper scheduling of operations.

Typically, the ground type $\langle T, A\rangle$ contains complex (T) and simple (A) components, and might be parameterized. Examples of parameterized types are memories, parameterized by address and content, queues, parameterized by items and length, and so forth. From a suitable behavior description, DDD automatically builds an abstract *system* description composed of a decision combinator, $\mathcal{C}$, representing control, and a structural component, $\mathcal{S}_{\langle T,A\rangle}$, representing architecture. The system is expressed over the same ground type ($\mathcal{B}'_{\langle T,A\rangle} \rightarrow C \bullet \mathcal{S}_{\langle T,A\rangle}$). The $\bullet$ operator denotes composition. $\mathcal{S}_{\langle T,A\rangle}$ is abstract because it may include signals ranging over complex entities of the ground type. A sequence of factorization steps ($C \bullet \mathcal{S}_{\langle T,A\rangle} \rightarrow C \bullet \mathcal{S}'_A \parallel \Theta_A$) decomposes $\mathcal{S}$ into a system of modules, encapsulating complex signals as coprocesses, (e.g. Θ_A). The $\parallel$ operator denotes communicating subsystems.

This notion of factorization is exploited in isolating modules for verification. As complex objects are factored, the resulting expression is reduced to terms of the simpler type, A.

A third class of transformations introduces a lower-level representation, R, in place of A in the system $\mathcal{S}''$. Ultimately, this decomposition produces a hierarchy of boolean subsystems. Then, $\mathcal{S}''_R$ is partitioned into synthesizable subsystems, $[\mathcal{P}^i_R]$. Logic synthesis is used to assemble the subsystems to the appropriate technology.

3 The DDD-FM9001 Project

The DDD-FM9001 Project was developed to study the interaction between derivational reasoning systems and verification systems in the context of a substantial example. One of the benchmarks in the study is the construction of an implementation of the FM9001 Microprocessor description using derivation and verification. The derived FM9001 is referred to as the DDD-FM9001. This work is a continuation of the work done in [12] in which the DDD system was applied to the FM8501 [7], and FM8502 Nqthm specifications.

The FM9001 is a 32-bit microprocessor representing the third generation processor description defined by Hunt and mechanically verified using the Nqthm theorem prover [3]. The proof establishes an equivalence relationship between four levels of specifications ranging from an *abstract programmers model interpreter* to a *netlist*. The highest level of specification is a collection of six recursive functions (including auxiliary functions) defining an instruction level interpreter. This level is referred to as the programmers model. The composition of these six functions define an abstract behavioral description and was used as input to the DDD system.

3.1 The DDD-FM9001 Derivation

The DDD-FM9001 derivation was done interactively with the DDD system. Several derivation paths were explored and the derivation refined. The final derivation script includes 30 coarsely grained commands to the derivation system. In actuality there are 1000 lines in the script with a total of 38k characters. However, much of explosion is due to repetition at the boolean level and can be reduced with more coarsely grained transformations. A brief sketch of the derivation follows.

First, a set of foldings and unfoldings were applied to the initial behavioral specification in order to achieve a proper scheduling of operations [14]. Abstract operations on memory and the register file were serialized to restrict memory access to at most one per state. The operations were also serialized to insure that accesses to memory and the register file could be multiplexed. Next, an initial system description consisting of a decision combinator, denoting control and a structural component, denoting architecture were derived. The next

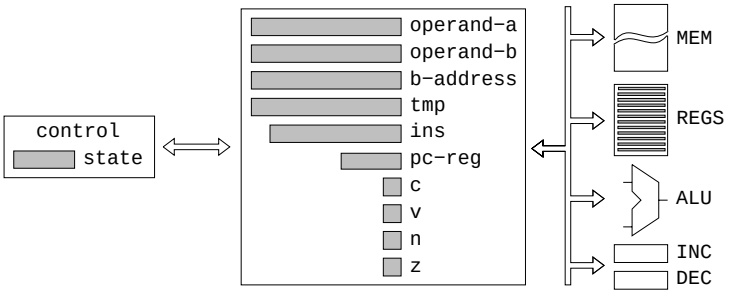


Figure 1: DDD-FM9001 Architecture

step was to transform the design into a reasonable description of functional components for implementation. Here a logical organization was imposed on the design. Operations were encapsulated in modules and complex components such as the memory, register file, ALU, incrementor, and decrementor, were factored from the description [10]. In the next phase of the derivation, concrete representations were introduced for the constants and operations of the ground type [8]. A collection of boolean subsystems were then generated. Technology dependent, highly optimized implementations of the ALU, incrementor, and decrementor were engineered and verified against the specifications of the respective factored components.

The result of the derivation yielded a design decomposition consisting of a controller, datapath, and abstract modules for memory, the register file, ALU, incrementor, and decrementor. Figure 1 sketches this decomposition. The controller, datapath, verified ALU, and verified incrementor and decrementor were assembled onto the FPGA. The memory and register file were implemented with an external SRAM.

4 Verification of the DDD-FM9001 ALU

The DDD-FM9001 design was targeted to the ACTEL FPGA architecture. The architecture is a matrix of *logic modules*, each implementing a four-input multiplexor. The verification of the arithmetic modules in the DDD-FM9001 reduced to verifying a boolean term, derived from the FM9001 ALU specification, against a hand designed multiplexor implementation. It was necessary to apply algebraic techniques to construct a boolean term from the FM9001 specification.

First, a functional abstraction of a four-input multiplexor, *mux* (shown below), was defined. The function was used as the basis for implementing the DDD-FM9001 ALU. The DDD-FM9001 ALU implementation is based on a carry lookahead adder optimized to minimize the use of ACTEL logic modules and gate delays. The design has been implemented with the ACTEL FPGAs architecture in mind. The circuit is well engineered and represents a significant improvement over the ALU available from the ACTEL library. The implementation consists of 367 logic modules.

```
(defn v-alu (c a b op)
  (cond ((equal op [bvec "0000"]) (cvzbv f f (v-buf a)))
        ((equal op [bvec "0001"]) (cvzbv-inc a))
        ((equal op [bvec "0010"]) (cvzbv-v-adder c a b))
        ((equal op [bvec "0011"]) (cvzbv-v-adder f a b))
        ((equal op [bvec "0100"]) (cvzbv-neg a))
        ((equal op [bvec "0101"]) (cvzbv-dec a))
        ((equal op [bvec "0110"]) (cvzbv-v-subtracter c a b))
        ((equal op [bvec "0111"]) (cvzbv-v-subtracter f a b))
        ((equal op [bvec "1000"]) (cvzbv-v-ror c a))
        ((equal op [bvec "1001"]) (cvzbv-v-asr a))
        ((equal op [bvec "1010"]) (cvzbv-v-lsr a))
        ((equal op [bvec "1011"]) (cvzbv f f (v-xor a b)))
        ((equal op [bvec "1100"]) (cvzbv f f (v-or  a b)))
        ((equal op [bvec "1101"]) (cvzbv f f (v-and a b)))
        ((equal op [bvec "1110"]) (cvzbv-v-not a))
        (t                 (cvzbv f f (v-buf a)))))
```

Figure 2: FM9001 ALU Specification

```
(defn cvzbv-v-adder (c a b)
  (cvzbv (v-adder-carry-out c a b)
         (v-adder-overflowp c a b)
         (v-adder-output    c a b)))
```

Figure 3: FM9001 ALU `cvzbv-v-adder` Definition

```
(defn v-adder (c a b)
  (if (nlistp a)
      (cons (boolfix c) nil)
      (cons (b-xor3 c (car a) (car b))
            (v-adder (b-or (b-and (car a) (car b))
                           (b-or (b-and (car a) c)
                                 (b-and (car b) c)))
                     (cdr a)
                     (cdr b)))))
```

Figure 4: FM9001 ALU Carry Ripple Adder Definition

$$
\begin{aligned}
mux \;=\; & \lambda(in_0\ in_1\ in_2\ in_3\ c_1\ c_0). \\
& or(and(not(c_1)and(not(c_0)\ in_0)) \\
& \quad or(and(not(c_1)\ and(c_0\ in_1)) \\
& \quad\quad or(and(c_1\ and(not(c_0)\ in_2)) \\
& \quad\quad\quad and(c_1\ and(c_0\ in_3)))))
\end{aligned}
$$

Next, symbolic evaluation, as discussed by Darringer in [5], in which the base operators are extended to return symbolic values and symbols are introduced as input values in place of real data objects, was applied to both the Nqthm specification and the multiplexor implementation of the ALU. For example, suppose the symbolic inputs a and b are supplied as actual data to a procedure with formal parameters A and B. Then the symbolic execution of the assignment statement $C = A + 2 * B$ would assign the symbolic formula $(a + 2 * b)$ to C. The symbolic execution of the arithmetic modules resulted in the construction of boolean expressions denoting their function.

As a simple example we look at the definition of the FM9001 ALU, `v-alu` (Figure 2). The FM9001 ALU takes a 4-bit op-code, two 32-bit a and b registers and a carry bit, expressed as bit vectors. More specifically, we can look at the `Add` operation, `(cvzbv-v-adder f a b)`, which is the fourth branch of the conditional statement in the definition of `v-alu` and denoted by the op-code `0011`. The definition of `cvzbv-v-adder` is shown in Figure 3. The abstract programmers model of `Add` is defined by a recursive definition of a ripple carry adder, `v-adder` (Figure 4). This is not apparent from the definition of `cvzbv-v-adder`, however, the auxiliary functions expand to either `v-adder` or a boolean term. Symbolically evaluating `(cvzbv-v-adder f a b)` yields a boolean term. The result are then passed through a boolean verifier to establish equivalence.

5 Concluding Remarks

The FM9001 provided the perfect experimental test bed to explore aspects of integrating derivation and verification. The DDD system and the Boyer-Moore theorem prover operate on the same concrete syntax. This allows DDD to read the FM9001 specification directly. The implementation is derived by applying DDD transformations to the specification. The ALU, incrementor, decrementor, register file and the memory were factored out. At this stage of derivation the datapath and control were assembled directly into hardware. Memory and register file were implemented with a RAM. The ACTEL implementation of the ALU, incrementor and decrementor were verified against the FM9001 specification.

The main goal of this work is to show how both derivation and verification can be integrated in digital design. A derivation system could be used to derive significant portions

of a design and then to isolate the parts to be designed into small building blocks. Verification could then be applied to these isolated building blocks. This experiment showed us that formal derivation can be integrated with low level verification to get the advantages of both approaches.

References

[1] Dominique D. Borrione, Laurence V. Pierre, and Ashraf M. Salem. Formal verification of VHDL descriptions in the Prevail environment. *IEEE Design & Test of Computers*, pages 42–56, June 1992.

[2] Robert S. Boyer and J. Struther Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[3] Bishop Brock and Warren A. Hunt. The FM9001 proof script. Technical report, Computational Logic Incorporated, 1993.

[4] Avra Cohn. A proof of correctness of the VIPER microprocessor: The first level. Technical Report 104, University of Cambridge, Computer Laboratory, January 1987.

[5] John A. Darringer. The application of program verification techniques to hardware verification. In *Proceedings of the* 16^{th} *Design Automation Conference*, June 1979.

[6] R. Kent Dybvig. *The SCHEME Programming Language*. Prentice-Hall, 1987.

[7] Jr. Hunt, Warren A. *FM8501: A Verified Microprocessor*. PhD thesis, The University of Texas at Austin, 1985. Also published as Technical Report 47 (December, 1985).

[8] S. D. Johnson and Bhaskar Bose. A system for mechanized digital design derivation. In Subramanyam, editor, *Proceedings of ACM International Workshop on Formal Methods in VLSI Design*. To be published, January 1991.

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

[10] Steven D. Johnson. Manipulating logical organization with system factorizations. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 260–281. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, Springer-Verlag, July 1989.

[11] Steven D. Johnson, Bhaskar Bose, and C. David Boyer. A tactical framework for digital design. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer Academic Publishers, Boston, 1988.

[12] Steven D. Johnson, Robert M. Wehrmeister, and Bhaskar Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Luc Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 385–404, Amsterdam, Netherlands, 1989. IMEC, Elsevier.

[13] Jonathan Rees and William Clinger. The revised^3 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.

[14] Zheng Zhu and Steven D. Johnson. An algebraic framework for data abstraction in hardware description. In Jones and Sheeran, editors, *Proceedings of The Oxford Workshop on Designing Correct Circuits*. Springer, 1990.