DDD-FM9001: Derivation of a Verified Microprocessor [†]

by

Bhaskar Bose

Submitted to the faculty of the Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

December 1994

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

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee	Prof. Steven D. Johnson, Ph.D (Principal Advisor)
November 11, 1994	Prof. David E. Winkel, Ph.D
	Prof. David S. Wise, Ph.D
	Prof. J. Michael Dunn, Ph.D

To my parents, Asim and Tripti.

Abstract

Derivation and *verification* represent alternate approaches to design. Derivation aims at deriving a "correct by construction" design while verification aims at constructing a post factum "proof of correctness" for a design. However, as researchers and engineers gain design experience in a formal framework, both approaches are emerging as interdependent facets of design. The thesis of this work is that alternate forms of formal reasoning must be integrated if formal methods are to support the natural analytical and generative reasoning that takes place in engineering practice.

As a vehicle for this research, the DDD digital design derivation system was implemented to study formal hardware design in an algebraic framework. DDD is a first-order transformation system which mechanizes a basic design algebra for synthesizing digital circuit descriptions from high-level functional specifications. The system is a collection of correctness preserving transformations that promote a top-down design methodology where the discipline of applicative programming is adapted to hardware verification.

As a non-trivial illustration of these ideas, the derivation of the DDD-FM9001 is presented. The DDD-FM9001 is a 32-bit general purpose microprocessor mechanically derived directly from Hunt's Boyer-Moore Logic FM9001 microprocessor specification. The derivation involved the use of three mechanical verification tools: the DDD digital design derivation system, the Nqthm theorem prover, and the COSMOS boolean tautology checker. The DDD digital design derivation system was used to derive a significant portion of the design leaving relatively small portions to be verified by the other verification tools. The result of this experiment was a derived FM9001 defined by a rigorous path to hardware.

Acknowledgments

I would like to thank Steven D. Johnson and David E. Winkel for their support, assistance, and friendship. Their breadth of knowledge and sagacity has provided me with the guidance to pursue this investigation. For the knowledge and wisdom that they have shared with me, I will always be indebted.

The relationship between teacher and student is something I hold very dear to me. Steve's insight into the research issues were instrumental in the success of this work. I am grateful to have such a teacher.

I would like to thank the other members of my committee, David S. Wise and J. Michael Dunn. Their encouragement and belief in this work has been a source of strength. I would also like to thank Warren A. Hunt, Jr. for providing the FM9001 specification and hardware implementation, and for providing a body of work that serves as a context for this research. I would like to thank the Formal Methods Branch at NASA Langley Research Center, who through the NASA Graduate Student Researchers Program, provided financial support for this work.

The derivation system presented herein was first implemented by myself in September 1986, while C. David Boyer applied it to a non-trivial design example. It is from this synergy that this work derives its spirit. I am particularly indebted to my dear friend and colleague M. Esen Tuna. His contributions to this work has added a dimension that would not have otherwise existed. Others who have contributed to this work include Ignacio Celis, Kathy Fisler, William A. Hunt, Paul S. Miner, Kamlesh Rath, Shyamsundar Pullela, Robert M. Wehrmeister, and Zheng Zhu.

And finally to my wife Ansuya Bose, like Steve, she too is an educator and has given me guidance and support throughout this writing. Her contributions to this work are manifest in its very existence. To Ansuya, with love, I thank you.

–bb–

Contents

List of Figures

Chapter 1

Introduction

Derivation and *verification* represent alternate approaches to design.

Derivation aims at deriving a
"correct by construction" design.

Verification aims at constructing a
post factum "proof of correctness" for a design.

However, as researchers and engineers gain design experience in a formal framework, both approaches are emerging as interdependent facets of design [44, 8, 54]. The thesis of this work is that alternate forms of formal reasoning must be integrated if formal methods are to support the natural analytical and generative reasoning that takes place in engineering practice.

The DDD-FM9001 is a 32-bit general purpose microprocessor formally derived from Hunt's Boyer-Moore Logic FM9001 microprocessor specification [35, 36]. The derivation involved the use of three mechanical verification tools: the DDD digital design derivation system [42, 40, 6, 41] , the Nqthm theorem prover [11], and the COSMOS boolean tautology checker [14, 15]. The project was undertaken to explore the nature of integration between derivation and verification.

The DDD digital design derivation system is a first-order transformation system that implements a basic design algebra for synthesizing digital circuit descriptions from high-level functional specifications. The system promotes a top-down design methodology, based on the concept of executable and verifiable specifications. Nqthm is a quantifier free, first order logic theorem prover based on the Boyer-Moore Logic. The logic is mechanized by a collection of LISP programs that allow the user to axiomatize inductively constructed data types, define recursive functions, and prove theorems about them. The COSMOS boolean tautology checker, is a boolean formula manipulation tool derived from the COSMOS symbolic simulator that constructs Binary Decision Diagrams from boolean terms. The system is used to establish equivalence between two boolean expressions.

The DDD-FM9001 project is an experiment to construct an implementation of the FM9001 using a design strategy that exploits the strengths of different formal systems to establish a rigorous path from an abstract high-level specification to a hardware realization. The DDD system is used to derive a significant portion of the design leaving relatively small portions to be verified by the other verification tools. In this derivation exercise, a set of transformations is applied to decompose and reorganize the design. Complex components, such as the memory, register-file, ALU, incrementor, and decrementor, are isolated using DDD's abstraction mechanisms. Technology dependent, highly optimized implementations of the ALU, incrementor, and decrementor are engineered and verified against their respective isolated components using COSMOS. Binary representations, verified by Hunt in Nqthm for the FM9001, are used to project the design to boolean equations. The memory and register-file are implemented with standard RAM components. The result of this experiment is a derived FM9001 implemented in FPGAs (Field Programmable Gate Arrays) defined by a rigorous path to hardware.

The FM9001 and DDD-FM9001 share a unique property - each microprocessor has a mechanically verified, formal relation to the same top-level mathematical specification. Figure 1.1 illustrates the relationship between Hunt's FM9001 and the

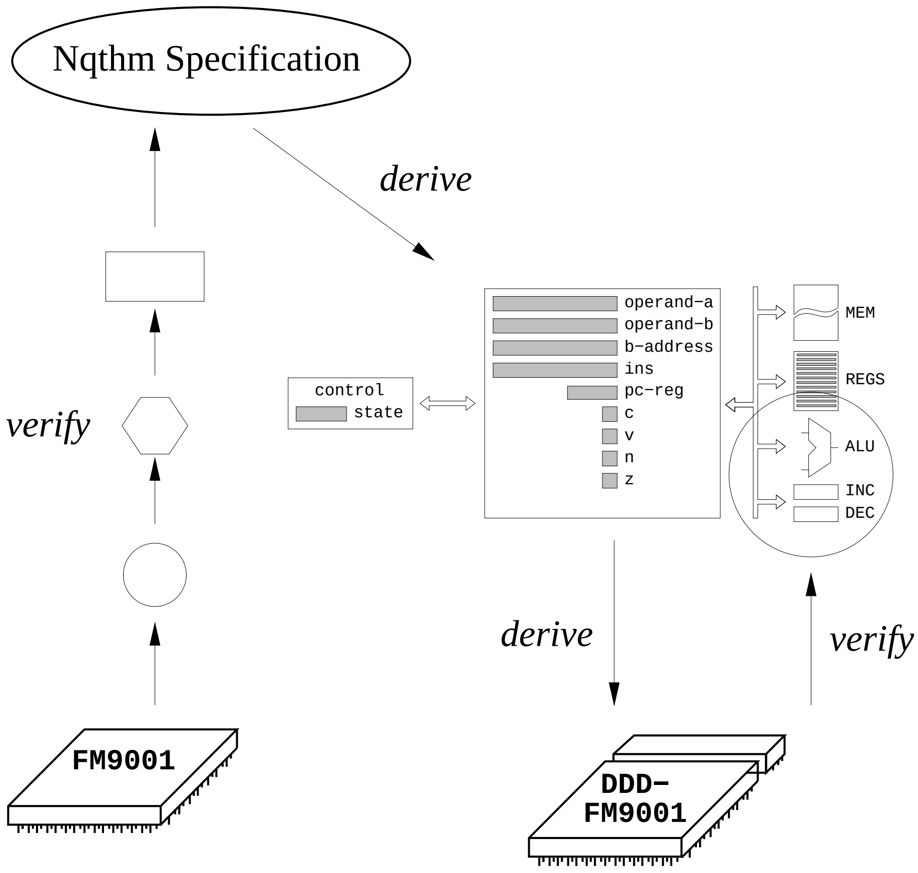


Figure 1.1: FM9001 Verification vs. DDD-FM9001 Derivation

DDD-FM9001. On the left is Hunt's verification. Hunt verified four levels of specification in Nqthm - the abstract programmer's model, the two valued logic level, the three valued logic level, and a gate level model. The FM9001 is realized on a single custom gate-array by LSI Logic. On the right is the derivation of the DDD-FM9001. In this case, derivation is used to decompose and restructure the design. Equivalence checking is used to verify various arithmetic components. The design is realized in an ACTEL FPGA and an external register-file.

Experience shows that derivation systems impose restrictions on the design, and verification systems result in design descriptions that are impractical for implementation [34] or proofs too difficult to construct [23]. This research sets out to address these deficiencies by integrating derivation and verification in a unified framework. It seems appropriate that algebraic transformations would provide a reasonable approach to this problem since the massive restructuring and decomposition necessary in reorganizing the design represent purely syntactical manipulations. A derivation system such as DDD can decompose and restructure the design in such a manner as to derive a significant portion of the design and isolate the verification problems to small building blocks. Verification can then be applied to the isolated building blocks. The intuition is that for a given design, the relationship between its specification and implementation has two facets. One facet of the implementation can be systematically derived from the specification. This portion, though critical, represents the uninteresting part of the design. The other facet of the design represents the interesting engineering in the design and is better addressed by verification.

1.1 Dissertation Contributions

The primary goal of this research has been the study of formal hardware design in an algebraic framework. To this end, the DDD digital design derivation system has been developed as a research vehicle for engineers and scientists.

This work extends the experimentation on the interplay between derivation and verification reported in [44]. Previous work applied the DDD system to Hunt's FM8501 description [34]. The FM8501 is a 16-bit microprocessor verified by Hunt in the Nqthm theorem prover. Results of the DDD/FM8501 experiment exposed the need to take a broader view of formal reasoning in design. The experiment illustrated how alternative modes of reasoning could be applied to a single design. The work showed how the massive restructuring involved at lower levels of abstraction could be implemented more easily by derivation, and how the inventive aspects of a design could be isolated for verification.

The DDD-FM9001 exercise extends the previous work on the FM8501 in three ways. First, the derivation was upgraded in conjunction with Hunt's refinements to representation. Second, much more of the algebra was mechanized; in fact, the entire gate-level hardware description was generated either by mechanical derivation or verified using boolean equivalence methods [3, 14]. Finally, the DDD-FM9001 was realized in hardware and subjected to side-by-side functional tests with Hunt's chip. Figure 1.2 is a photograph of the hardware prototype in which Hunt's FM9001 and the DDD-FM9001 are fully operational. Details are given in Chapter 5.

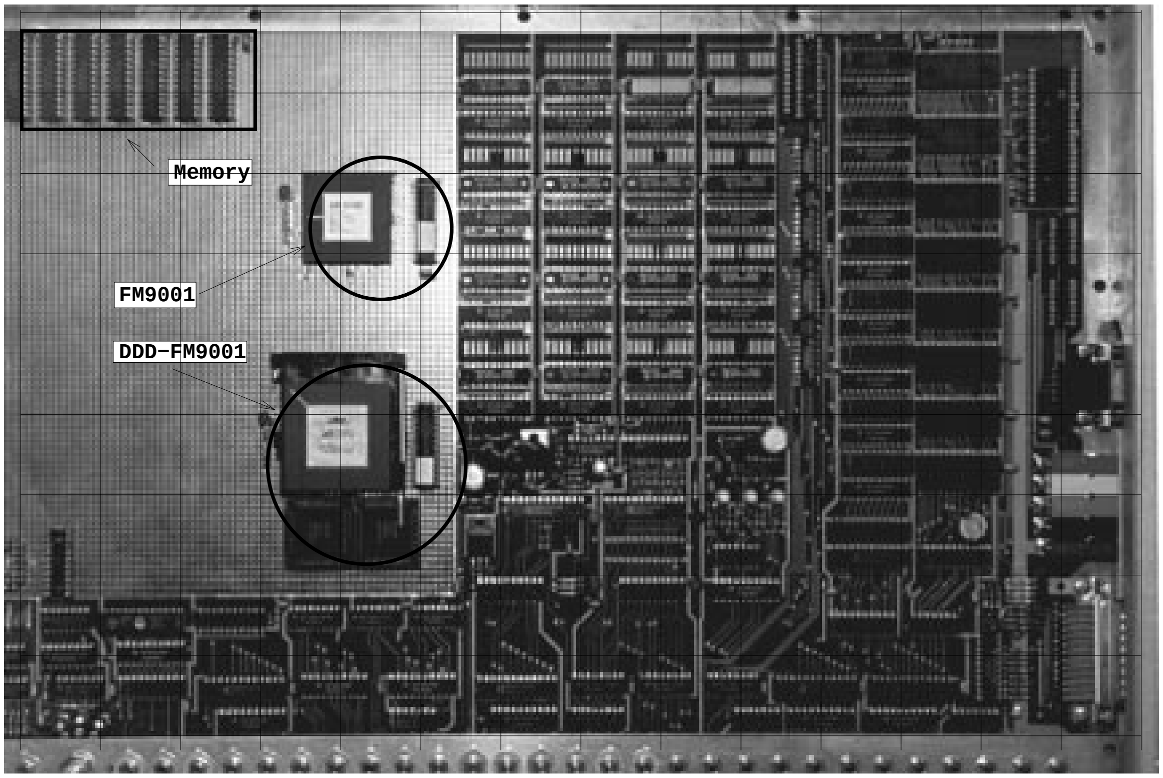


Figure 1.2: DDD-FM9001/FM9001 Logic Engine Prototype Environment

1.2 Outline

This dissertation sets out to present in a meaningful way the details of the derivation of the DDD-FM9001. Chapter 2 explores related research, including brief descriptions of derivation exercises using the DDD system. Chapter 3 describes the DDD system. A characterization of the derivation process is outlined defining a path from behavior to structure to physical organization. An example illustrates the path from behavior to architecture. Chapter 4 contains a documentary of the significant aspects of the DDD-FM9001 derivation, in which the reader is carried through key points of the derivation in an attempt to mirror the intellectual exercise imposed by the derivational methodology. Chapter 5 describes the DDD-FM9001 hardware realization. Chapter 6 concludes this thesis with remarks and future directions. Appendix A contains the top-level Boyer-Moore Logic specification of the FM9001 as well as details of the instruction set, condition codes, addressing modes, and register addresses for the FM9001. Appendix B contains the complete DDD derivation script. Appendix C contains the initial and final structural specifications derived from the behavioral specification.

Chapter 2

Related Research

With the growing complexity in VLSI technology, the need for powerful design tools has become essential to the future of computer design. Formal methods in VLSI are playing a fundamental role in design automation as mathematical techniques provide a rigorous framework for reasoning about complex designs and guaranteeing integrity in the design process.

2.1 Formal Verification Systems

Formal verification systems mechanize the mathematical reasoning necessary in design. They provide proof management support for the overwhelming detail associated with the description of hardware systems, and span a range of expressive power from fully automatic model checkers to manually guided theorem provers.

2.1.1 Model Checking

Model checking is a fully automatic technique that involves extracting models from implementations and verifying properties about the model in a formal framework. Binary decision diagrams (BDDs) [3] have emerged as an effective data representation for digital systems. One variation of BDDs, called Ordered BDDs (OBDDs) [14] has the special property that for a given ordering of the variables the representations are canonical. Thus OBDDs are used frequently to solve problems in the areas of boolean verification and finite-state machine equivalence [16, 49].

The "bdd" program [15], used to verify the arithmetic components of the DDD-FM9001, is a boolean formula manipulation tool integrated with the COSMOS symbolic simulator which constructs OBDDs from boolean terms. The system is used to establish equivalence of boolean expressions. In "bdd", boolean equations are represented as acyclic graphs, with variables as the internal nodes and TRUE and FALSE (1 and 0) as the terminal nodes. Because of the canonical nature of OBDDs, the test for equality of two boolean expressions and other operations on boolean expressions reduce to simple graph algorithms operating on OBDDs. For example, verifying the equality of two boolean equations reduces to the graph equality test on OBDDs representing the two equations. Although in the worst case the size of the OBDDs can be exponential in the number of variables, there is substantial empirical evidence that OBDDs are of reasonable space and time complexity for most of the boolean expressions in digital design [14]. Although, the techniques in model checking are completely automatic and have been used to verify some very large systems [21], they are limited in their ability to verify many properties of a given specification.

2.1.2 Theorem Proving and Proof Checking

A *mechanical theorem prover* for a given formal logic is a computer program that, when given a formula of that logic, attempts to determine whether there is a proof of the formula. A *mechanical proof checker* for a given formal logic is a computer program that, when given a proof of a theorem, checks that it is a valid proof. Theorem provers and proof checkers provide a powerful framework for verification and have been used to verify significant designs.

HOL is a general theorem-proving system developed at the University of Cambridge [31] based on higher-order logic [18]. Higher-order logic is predicate logic that allows quantification over predicates and functions. HOL is based on five primitive axioms and eight primitive inference rules. All proofs reduce to the primitive axioms and rules. The system has been used to verify several microprocessor designs.

Cohn [23, 24] verified parts of the VIPER microprocessor in the HOL theorem prover. VIPER was designed by Britain's Royal Signals and Radar Establishment (RSRE) at Malvern to provide a high-integrity, formally verified microprocessor for use in safety-critical systems. More recently, Levitt et al. [48], applying Windley's generic interpreter theory [74], used HOL to verify the VIPER instruction set. Brian Graham et al. at the University of Calgary verified an implementation of an SECD machine [32] in HOL. The SECD is a specialized microprocessor for LISP, first specified by Peter Landin [46] in the 1960s. Joyce verified the TAMARACK [45], a 16-bit micro-coded microprocessor, in HOL. Joyce verified TAMARACK at the transistor level and fabricated an 8-bit version in CMOS. The design is based on early hardware verification efforts by Gordon on the design and verification of a simple computer [30] using a precursor to HOL, the LCF-LSM theorem prover [29].

Nqthm is a quantifier free, first order logic theorem prover based on the Boyer-Moore Logic [11]. The logic is mechanized by a collection of LISP programs that permit the user to axiomatize inductively constructed data types, define recursive functions, and prove theorems about them.

In 1985, Hunt used the Nqthm theorem prover to verify a 16-bit general purpose microprocessor called the FM8501 [34]. The architecture has eight 16-bit registers, a 16-bit address space, 26 instructions, and four memory addressing modes. In the FM8501 proof, Hunt established an equivalence relation between specifications of an abstract programmer's model, called `soft`, and an implementation, a hardware interpreter model, called `big-machine`. Subsequent work by Hunt scaled this proof effort to a 32-bit version called FM8502.

Recent work by Hunt is the verification of the FM9001 [35], a 32-bit general purpose microprocessor. The FM9001 is defined in the Boyer-Moore logic and is mechanically verified using the Nqthm theorem prover. The proof establishes an equivalence relation between four levels of specification ranging from an instruction-level programmer's model interpreter to an optimized gate-level description. Unlike Hunt's previous verification efforts of the FM8501 and FM8502, the FM9001 is realized in hardware. The FM9001 is packaged in a 121-pin gate-array fabricated by LSI Logic, Inc. The chip has a 32-bit ALU, a 16×32-bit register-file, a 32-bit data I/O bus, a 32-bit address bus, a programmable program counter, a memory interface, clock, reset, and a scan path.

PVS (Prototype Verification System) [57] is a theorem prover for specification and verification for simply typed higher-order logic. The system incorporates a fairly rich set of built-in types and type constructors, and integrates a powerful proof checker with the type system. PVS provides a combination of direct control by the user for the higher levels of proof development, and powerful automation for the lower levels. This combination, makes it a powerful and effective tool. Miner has used PVS in a series of hardware verification experiments [54].

2.1.3 Design by Algebraic Transformation

In"Synthesis of Digital Designs from Recursion Equations" [37], Johnson defines a formal approach to hardware design based on the algebraic manipulation of purely functional forms. The design methodology supports fundamental aspects of design in a unified framework. In this framework, the discipline of applicative programming is adapted to hardware verification.

This approach to design, called *derivation*, is a branch of formal verification where design is viewed as a translation of notation, starting with an abstract *specification* ranging over abstract data types and deriving an intended target *implementation* by the application of correctness preserving transformations and constructions. The implementation is said to be "correct-by-construction" eliminating the need for post-factum verification. This process is a translation between dialects of recursive expressions

$$\mathcal{E}_0 \xrightarrow{\tau_0} \mathcal{E}_1 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_{k-1}} \mathcal{E}_k.$$

$\mathcal{E}_0$ represents a source expression and $\mathcal{E}_k$ an implementation. The arcs τ_0 to τ_{k-1} represent applications of transformations. A design is determined by an initial specification $\mathcal{E}_0$, and a sequence of transformations $< \tau_0, \cdots, \tau_{k-1} >$. The DDD digital design derivation system is a mechanization of this theory.

Other approaches that follow closely to this methodology, include Sheeran's Ruby relational algebra [66, 67, 68] that supports a transformational approach to verification. In this method, a circuit is described by a binary relation, and the language permits simple relations to be composed into more complex ones by using a variety of combining forms. The algebra has been implemented in a rewriting tool called T-Ruby [65]. Vemuri defines a set of transformations [71, 72] based on a free algebra for register transfer level designs. Gaboury and M.I. Elmasry's PLUSH (Predicate Logic Used for Synthesis Hardware) system [27] is a transformational approach based on the Prolog language.

Johnson's work provides the theoretical foundation for the body of work surrounding the DDD system and serves as a context for several non-trivial design derivation exercises. These derivation exercises provide a rich test bed for the development of the DDD system as researchers investigate an algebraic approach to digital design.

Boyer used the DDD system to derive a stop-and-copy garbage collector for Scheme [42]. The implementation was realized in MSI level EPAL (Erasable Programmable Array Logics) technology and demonstrated a derivation path from an abstract behavior specification to a hardware realization. Boyer subsequently re-derived this design in a multi-chip VLSI implementation [10]. Wehrmeister used DDD to derive the control and architecture of a computer based on Landin's SECD machine [73]. The initial specification was derived from Henderson's Lisp specification [33]. Another derivation exercise was to derive an implementation of Winkel and Prosser's state machine for playing the dealers hand in a game of Black Jack [41]. The experiences gained from these derivation exercises have been invaluable to the understanding of hardware design in a formal framework and begin to demonstrate the need to integrate formal systems.

Recent designs have included other formal systems in the derivation process. Early experiments with Hunt's FM850x [34] series of verified microprocessor designs were initiated to study the interplay of derivation and verification [44]. Details of this experimentation is discussed in the next section. Miner derived a fault-tolerant clock synchronization circuit for life-critical systems [54] integrating the DDD system, the PVS theorem prover [57], and the COSMOS boolean verifier. Burger derived a hardware prototype for the execution of compiled Scheme using DDD and boolean verification [70].

2.2 Integrating Formal Systems

The integration of formal systems has been an emerging thesis of this work. The investigation grew out of recognition that although powerful verification systems, such as Nqthm and HOL, have been used to successfully verify non-trivial designs, hardware verification efforts such as Cohn's VIPER proof [23], Joyce's TAMARACK proof, and Hunt's FM8501 proof [34], were facing the same class of design problem. A design description structured for the purpose of mechanical proof may have to be restructured for the purpose of physical implementation. The re-verification effort is a non-trivial corollary to the correctness proof, and it is evident that this kind of restructuring should be an algebraic process. On the other hand, a transformation system depends on the correctness of the specifications, representations, and cannot fully account for every aspect of design. In principle, a design methodology that integrates various formal systems would allow the designer to use the most effective method for a given task while maintaining a rigorous integrity of the design.

For example, the implementation of the FM8501 was compiled by Hunt by expanding the internal register expressions. The design expanded to over 11 million gates. The identification of common sub-expressions reduced the description to 1,789 gates. Although the 1,789 gates is tractable by today's logic synthesis systems, it represents a "sea of gates", rather than a design decomposed into logical and physical structures appropriate for a practical design. In addition, the scaling of this method to more complex machines results in an exponential growth in the size of the expansion. For example, going from the 16-bit FM8501 to the 32-bit FM8502 or 32-bit FM9001 could not have been synthesized this way. As a solution to this problem, In the FM9001 verification, Hunt embedded a formal hardware description language (HDL) within the Boyer-Moore logic to describe the structure of the target specification [36]. This gave Hunt a real implementation language that he could manipulate formally.

The VIPER project involved an attempt to verify the specification at three levels of abstraction (Top Level, Major State, and Block), using Gordon's HOL theorem prover [31], an attempt to verify a corresponding set of specifications written in LCF_LCM, a translation of the Block level LCF_LSM specification into the simulation language ELLA [55], and a translation of the gate-level ELLA specification into the logic synthesis tools HILO and FDL for fabrication. The complete HOL proof of the VIPER processor was not completed at the time the processor was fabricated [12]. The only formal proof in the VIPER design effort was Cohn's HOL proof of the Top

Level and Major State specifications. Cohn gives a detailed account of the difficulties in completing the proof in [24].

Early experiments with Hunt's FM850x series of verified processors [44] began to explore the interdependence of both approaches. In the FM8501 proof [34], Hunt established an equivalence relation between specifications of an abstract programmer's model, called `soft`, and an implementation, a hardware interpreter model, called `big-machine`. DDD was applied to both `soft` and `big-machine`. The derived architecture for `soft` was quite close to Hunt's implementation, however, it did not contain certain key registers such as those associated with the memory protocol. These registers were not expected to arise in the derivation since they did not exist in the original specification of `soft`. In fact, this difference highlighted an essential aspect of Hunt's proof establishing an equivalence relation between a functional model of memory with that of a process model of memory.

In the second derivation, DDD was applied to the hardware interpreter model, `big-machine`, to guide a top-down expansion of the design. Unlike Hunt's approach, in which a bottom-up expansion of `big-machine` resulted in over 11 million gates (reduced to 1,789 gates with the identification of like terms), algebraic manipulations were used to unfold, decompose, and restructure `big-machine` while containing the size of the expansion. The derived architecture was identical to Hunt's block diagram.

From the FM8501 experiment two central ideas emerged. The derivation of `soft` exposed elements of an implementation that could not be derived from a specification. These elements reflected isolated components of a design that must be proved. The `big-machine` derivation illustrated the need for transformational algebra to restructure and decompose a design in order to manage the logical and physical organization necessary to construct a realization targeted to a particular technology. The experiences with the FM8501 gave insight into the DDD-FM9001 derivation.

Other works to integrate various formal systems include Joyce's work in two-level verification methodology, where the HOL theorem prover is used for the ingenious aspects of system verification and the COSMOS automatic BDD based methods are used for other verification tasks [64]. Schneider et al. [63] have also integrated boolean verification. Some preliminary work in this area, defining possible directions is Giunchiglia's work on Reasoning Structures [28]. In this work Giunchiglia et al. propose a graph based reasoning structure that provide a basis for interaction and sharing of information among inference procedures and with other reasoning systems.

Chapter 3

DDD - Design Derivation System

3.1 Introduction

DDD (Digital Design Derivation System) is a transformation system that implements a basic design algebra for synthesizing digital circuit descriptions from high-level functional specifications [42, 40, 6]. The system is a formalization of digital design based on a functional algebra. 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 [26, 22] as a collection of transformations that operate on s-expressions. The system is intended to provide a mechanized algebraic tool set for design derivation. DDD is used interactively to transform higher level behavioral specifications into hierarchical boolean systems [76, 75] to which logic synthesis tools are then applied.

The DDD system has been integrated with the Berkeley OCT Tools [69] to generate VLSI layouts, the Altera APLUS software [4] to generate MSI-level components, and the ACTEL Action Logic System [1] to generate FPGA (Field Programmable Gate Array) implementations. For the purpose of developing prototypes, the DDD system has been integrated with the Logic Engine Hardware Development Platform [78]. This hardware prototyping environment incorporates a transparent hardware/software interface written in SCHEME which allows for the execution of DDD hardware descriptions interacting with the implemented hardware components.

3.2 The Derivation Path

In DDD, a sequence of transformations is 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. In practice, however, the derivation path has distinct phases. The diagram below expands on the characterization of derivation introduced earlier in Chapter 2, Section 2.1.3. In the derivation path below, $\mathcal{X}_B$ denotes a DDD expression, $\mathcal{X}$, written in terms of a ground type, B.

$$\begin{tikzcd}
\mathcal{B}_{\langle T,A\rangle} \arrow[r, phantom, "\Leftrightarrow\!\!\rightarrow"] & \mathcal{B}'_{\langle T,A\rangle} \arrow[d] & \cdots & & \textit{behavior} \\
\cdots & C \bullet \mathcal{S}_{\langle T,A\rangle} \arrow[r, phantom, "\Leftrightarrow\!\!\rightarrow"] & C \bullet \mathcal{S}'_A \parallel \Theta_A \arrow[d, shift right] & \cdots & \textit{structure} \\
& \cdots & C \bullet \mathcal{S}''_R \parallel \Theta_R \arrow[u, shift right, "verify"'] \arrow[r, phantom, "\Leftrightarrow\!\!\rightarrow"] & C \bullet [\mathcal{P}^i_R] \parallel \Theta_R \arrow[d] & \textit{physical org.} \\
& & & \textit{logic synthesis} &
\end{tikzcd}$$

A sequence of transformations is applied to an initial behavioral specification $\mathcal{B}_{\langle T,A\rangle}$ in order to derive an intended target realization. The initial behavior description $\mathcal{B}_{\langle T,A\rangle}$ is in a restricted class of *iterative* function definition schemes expressed in terms of a complex basis consisting of abstract operations and predicates, as well as concrete operations and objects. 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 arithmetic operations parameterized by integers.

Design derivation in the DDD system has three major phases in which a typical design may undergo many iterations. A class of transformations, called *behavioral transformations*, manipulates the behavioral specification. These transformations usually involve manipulating control and architecture in a tightly integrated relation. Some examples include folding and unfolding transformations to achieve a proper scheduling of operations and transformations to move operations between control and architecture [81].

From a suitable behavior description, DDD automatically builds an abstract *sequential system* description ($\mathcal{B}'_{\langle T,A\rangle} \rightarrow C \bullet \mathcal{S}_{\langle T,A\rangle}$) composed of a decision combinator, $\mathcal{C}$, representing control, and a structural component, $\mathcal{S}_{\langle T,A\rangle}$, representing an initial estimation of architecture. The system is expressed over the same ground type $\langle T, A\rangle$. The $\bullet$ operator denotes simple composition. $\mathcal{S}_{\langle T,A\rangle}$ is abstract because it may include signals ranging over complex entities in the ground type.

A second class of transformations, called *structural transformations*, manipulates the sequential system description. These transformations are intended to refine the structural specification to an architecture. 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 co-processes, (e.g. Θ_A), isolating components of the specification for verification, mapping to existing hardware components, or further algebraic refinement. The $\parallel$ operator denotes the composition of communicating subsystems. The resulting expression is reduced to terms of the simpler type, A.

A third class of transformations, called *projection transformations*, introduces a lower-level representation, R, in place of A for $C \bullet \mathcal{S}'_A$. *Verification* is necessary to establish correctness of the representations.

Ultimately, this decomposition produces a hierarchy of boolean subsystems, which are then partitioned into synthesizable subsystems ($C \bullet \mathcal{S}''_R \parallel \Theta_R \rightarrow C \bullet [\mathcal{P}^i_R] \parallel \Theta_R$). These boolean subsystems are then passed to logic synthesis tools to generate hardware realizations.

3.3 An Example: Fibonacci

This section introduces the phases of derivation in the DDD system in the context of an example. The example is derived from Johnson's construction of the recursive Fibonacci specification [37]

$$fib(x) = lt?(x,2) \rightarrow add(fib(dec(dec(x))), fib(dec(x)))$$

to an iterative form. The "$\rightarrow$" denotes the conditional operator.

Beginning with an iterative definition of the Fibonacci function

$$g(x,y,z) = lt?(x,2) \rightarrow y, g(dec(x), z, add(y,z))$$

where

$$fib(x) = g(x,1,1).$$

The first phase in the derivation is to apply a series of transformations at the behavioral level. In this phase the *dec* and *add* operations are serialized so that they may be combined into a single logic unit later in the derivation. The first step is to introduce a function h, such that the call to g will fold within the definition of h. Introducing a definition h yields

$$\begin{aligned} g(x,y,z) &= lt?(x,2) \rightarrow y, g(dec(x), z, add(y,z)) \\ h(x,y,z) &= g(x, z, add(y,z)). \end{aligned}$$

The next step is to fold the call, $g(dec(x), z, add(y, z))$, in g into the definition of h resulting in

$$\begin{aligned} g(x, y, z) &= lt?(x, 2) \rightarrow y, h(dec(x), y, z) \\ h(x, y, z) &= g(x, z, add(y, z)). \end{aligned}$$

The transformation rewrites the original call to g with a call to h with the appropriate unification of parameters. This has the effect of splitting the *dec* and *add* operations between two separate definitions.

The next step in the derivation constructs a structural description from the behavioral specification. This is performed automatically by DDD and represents a pivotal transformation in the derivation. The construction from behavior to structure is done in two steps. The first step introduces a new parameter, w to encode which function is in control and rewrites g and h as a single recursion equation

$$\begin{array}{l} f(w, x, y, z) = \\ \quad case\ w \\ \quad\quad \mathtt{g} : lt?(x, 2) \rightarrow y, f(w, dec(x), y, z) \\ \quad\quad \mathtt{h} : f(g, x, z, add(y, z)). \end{array}$$

The parameter w is defined to range over the set of control tokens $\{\mathtt{g}, \mathtt{h}\}$ corresponding to the definitions g and h respectively. The function f is constructed by adding w to the formal parameter list, changing each recursive call to g and h to the corresponding call to f, and constructing a case statement encoding which function is in control.

The second step is to decompose f into decision combinator representing control and a system of equations representing an initial architecture. The key step is to derive a selection combination from the conditional structure of the specification.

The derived selector is

$$\begin{array}{l} \mathit{select}([s, p0], v0, v1, v2) = \\ \quad \mathit{case}\ s \\ \qquad \mathtt{g} : p0 \rightarrow v0, v1 \\ \qquad \mathtt{h} : v2 \end{array}$$

By distributing *select* over calls to equations, traces (see the expression below) of the individual parameters are obtained. The derived system of equations is

$$\begin{array}{rcl} w &=& \mathtt{g}\ !\ \mathit{select}(\mathit{status}, \mathtt{g}, \mathtt{h}, \mathtt{g}) \\ x &=& n\ !\ \mathit{select}(\mathit{status}, x, \mathit{dec}(x), x) \\ y &=& 1\ !\ \mathit{select}(\mathit{status}, y, y, z) \\ z &=& 1\ !\ \mathit{select}(\mathit{status}, z, z, \mathit{add}(y, z)) \\ \mathit{status} &=& \mathit{list}(w, \mathit{lt?}(x, 2)) \\ \mathit{rdy} &=& \mathit{and}(\mathit{equal?}(w, \mathtt{g}), \mathit{lt?}(x, 2)) \\ \mathit{ans} &=& y. \end{array}$$

Each equation is of the form

$$X = S$$

which denotes an infinite sequence of uniformly typed value traces, $< S^0, S^1, \ldots >$. The equation

$$X = v\ !\ S$$

denotes the sequence of value traces $< v, S^0, S^1, \ldots >$ where the "!" introduces a delay and is interpreted as a register. The construction guarantees that given a particular sequence of input events, the structural description produces the same output-event sequence as does the original behavior specification and ans will contain $fib(n)$ the first time rdy is true. For example, given $n = 4$, tracing the sequence of values for each of the equations yields the following value traces

$$\begin{aligned}
w &= <\mathtt{g},\mathtt{h},\mathtt{g},\mathtt{h},\mathtt{g},\mathtt{h},\mathtt{g},\ldots> \\
x &= <4,3,3,2,2,1,1,\ldots> \\
y &= <1,1,1,1,2,2,3,\ldots> \\
z &= <1,1,2,2,3,3,5,\ldots> \\
status &= <(\mathtt{g},\mathtt{f}),(\mathtt{h},\mathtt{f}),(\mathtt{g},\mathtt{f}),(\mathtt{h},\mathtt{f}),(\mathtt{g},\mathtt{f}),(\mathtt{h},\mathtt{t}),(\mathtt{g},\mathtt{t}),\ldots> \\
rdy &= <\mathtt{f},\mathtt{f},\mathtt{f},\mathtt{f},\mathtt{f},\mathtt{f},\mathtt{t},\ldots> \\
ans &= <1,1,1,1,2,2,3,\ldots>\ .
\end{aligned}$$

The second phase in the derivation is to refine the structural specification to an architecture. A sequence of factorization steps decomposes the system of equations into a system of communicating modules. In this example, the architecture is refined by subsuming the *dec* and *add* operations by a single component. The transformation is possible since these two operations do not occur simultaneously, as imposed by the earlier serialization of these two operations.

The results of factoring *dec* and *add* are the synthesis of an abstract component, *alu*, and the derivation of four equations, *alu_out*, *inst*, *op_a*, and *op_b* to communicate with the factored component. The original occurrences of *add* and *dec* are replaced with the output of the factored component. The resulting architecture is

$$\begin{aligned}
w &= \mathtt{g}\,!\,select(status,\mathtt{g},\mathtt{h},\mathtt{g}) \\
x &= n\,!\,select(status,x,alu\_out,x) \\
y &= 1\,!\,select(status,y,y,z) \\
z &= 1\,!\,select(status,z,z,alu\_out) \\
status &= list(w,lt?(x,2)) \\
rdy &= and(equal?(w,\mathtt{g}),lt?(x,2)) \\
ans &= y \\
inst &= select(status,\mathtt{nop},\mathtt{dec},\mathtt{add}) \\
op\_a &= select(status,?,x,y) \\
op\_b &= select(status,?,?,z) \\
alu\_out &= alu(inst,op\_a,op\_b)
\end{aligned}$$

where

$$\begin{array}{l} alu(inst, op\_a, op\_b) = \\ \quad case\ inst \\ \qquad \texttt{nop} :? \\ \qquad \texttt{dec} : dec(op\_a) \\ \qquad \texttt{add} : add(op\_a, op\_b). \end{array}$$

The third phase of the derivation is to introduce a lower-level representation. The architecture is still abstract in the sense that signals represent integer values. To obtain a concrete binary description, these signals are instantiated with bit-vectors of appropriate width. Type declarations are used to project each variable, constant, and operator to a binary representation. For instance, the projection of

$$y = 1\,!\,select(status, y, y, z)$$

to a binary representation of three bits is declared with

$$\begin{array}{rcl} y & \Rightarrow & list(y_0, y_1, y_2) \\ z & \Rightarrow & list(z_0, z_1, z_2) \\ 1 & \Rightarrow & list(t, f, f) \end{array}$$

This type information is used to rewrite equation y to a system of three equations, y_0, y_1, and y_2, one for each bit. The constant 1 and signal z are also projected to their respective bit equivalence. The new system of equations is

$$\begin{array}{rcl} y_0 & = & \texttt{t}\,!\,select(status, y_0, y_0, z_0) \\ y_1 & = & \texttt{f}\,!\,select(status, y_1, y_1 z_1) \\ y_2 & = & \texttt{f}\,!\,select(status, y_2, y_2 z_2). \end{array}$$

In a similar manner, the control specification, *select*, and the rest of the architecture is projected to a bit representation. At this stage the entire design is at the binary level. Subsequent transformations manipulate the specification for mapping to a particular target technology.

3.4 The Specification Language

Specifications are written in SCHEME. The specifications are executable and verifiable, and written in a purely functional style where there are no side-effects. Descriptions are built from *applicative terms*, *constants*, *identifiers*, *conditional expressions*, and *function definitions*, and express globally synchronized systems. The input description does not require fixed representations therefore allowing greater freedom in exploring alternative architectures. Both control and architecture are derived from such specifications. The control logic is automatically synthesized while the architecture is derived by structural refinement.

In DDD, there are two classes of specifications, behavioral and structural. A construction from behavior to structure establishes the equivalence between the two classes of specification. In this section an informal introduction to a subset of SCHEME syntax, used in DDD specifications, is presented. A complete language definition and formal semantics for SCHEME can be found in [22]. This introduction is followed by a description of the DDD behavioral and structural specifications.

3.4.1 Scheme Syntax

SCHEME, a dialect of LISP, is a statically scoped, applicative order, functional language that is well suited for symbolic manipulation. The language definition is a small core of syntactic forms from which all other forms are built. These core forms, a set of extended syntactic forms derived from them, and a library of primitive procedures make up the full SCHEME

SCHEME supports operations on structured data such as *strings*, denoted with double quotes, *"abcd"*; *lists*, denoted by parenthesized sequences, *(l1 l2 ... ln)*; and *vectors*. SCHEME also supports operations on more traditional data such as numbers and symbols. Programs are made up of forms (lists), identifiers (symbols), and constant data (strings, numbers, vectors, quoted lists, quoted symbols, etc.).

Procedures are defined with *function expressions.* A function expression has the form

`(lambda (`*id* `...)` *exp1 exp2 ...).*

The identifiers *id ...* are the formal parameters of the procedure and the sequence of expressions *exp1 exp2* ... is its body. Unlike SCHEME, DDD is a first-order system and does not allow functions as values, so lambda expressions are seen only in defining equations. In addition, DDD expressions only allow a single expression in the body of a function expression.

Objects can be associated with a name at top level by a *top-level definition.* A top-level definition has the form

`(define` *id exp*`).`

The identifier *id* is bound at top-level to the value of the expression *exp.*

Two forms of conditional expressions are used in DDD. An *if-then-else expression*

`(if` *test consequent alternative*`)`

returns the *consequent* if *test* is true and the *alternative* otherwise. A *case-expression*

`(case` *val* `(`*key exp* `...) ...)`

returns the value of the last *exp* if the corresponding label *key* equals *val.*

Local definitions are made with *let* and *letrec* expressions. A *let expression*

`(let ([`*id val*`] ...)` *exp1 exp2* `...)`

creates a local binding in which each identifier *id* is bound to the value of the corresponding expression *val.* These bindings are valid in the body of the *let expression.*

`letrec` is a form similar to *let*, but allows mutually recursive bindings. A *letrec* *expression* has the form

`(letrec ([`*id val*`] ...)` *exp1 exp2* `...).`

Local stream bindings are defined by the language extension `system-letrec`. This form is a derived from `letrec`, but defines mutually recursive stream bindings. This is further discussed in Section 3.4.3. A *system-letrec expression* has the form

`(system-letrec ([`*id val*`] ...)` *exp1 exp2* `...).`

For each of the forms, `let`, `letrec`, and `system-letrec`, DDD expressions impose the added restriction to the general SCHEME form that only a single expression is allowed in the body.

3.4.2 Behavior Specification

A DDD behavior specification is an abstract algorithmic description that defines the functionality of the circuit by specifying a sequence of operations and control decisions. The specification describes what operations must occur, but not how they are implemented in hardware. The specification is abstract in the sense that the underlying representations of the signals and operators are not specified and may exist at arbitrary levels of abstraction. A general form of behavioral specification in DDD is

```
(define ⟨name⟩
  (lambda ⟨inputs⟩
    (letrec ([⟨state⟩ (lambda ⟨registers⟩ ⟨exp⟩)] ...)
      ⟨initial-state⟩))))
```

where

```
       ⟨name⟩  ::=  ⟨identifier⟩
     ⟨inputs⟩  ::=  (⟨var⟩ ...)
⟨initial-state⟩ ::=  ⟨state-call⟩
      ⟨state⟩  ::=  ⟨identifier⟩
  ⟨registers⟩  ::=  (⟨var⟩ ...)
        ⟨exp⟩  ::=  (let ((⟨var⟩ ⟨val⟩) ...) ⟨exp⟩)
                |   (if ⟨bool⟩ ⟨exp⟩ ⟨exp⟩)
                |   (case ⟨val⟩ ((⟨key⟩ ⟨exp⟩) ...)
                |   ⟨state-call⟩
 ⟨state-call⟩  ::=  (⟨state⟩ ⟨val⟩ ...)
        ⟨var⟩  ::=  ⟨identifer⟩
        ⟨val⟩  ::=  ⟨?⟩ | ⟨var⟩ | ⟨const⟩ | (⟨var⟩ ⟨val⟩ ...)
        ⟨key⟩  ::=  ⟨identifier⟩
```

A hardware description is defined by a set of *inputs*, an *initial state*, and a set of mutually recursive procedure definitions representing the internal states of the machine, called *state definitions*, denoted by

```
(letrec
    ([⟨state⟩ (lambda ⟨registers⟩ ⟨exp⟩)] ...)
  ...)
```

The specification must be *iterative*: each state definition is a conditional expression in which the alternatives are tail-recursive and is equivalent to the class of schemata associated with finite-state machines. Each state definition represents a machine state or computation in the algorithm defined by the expression. However, the notion of state is abstract. A state at one level of specification may correspond to several micro-states at lower levels of abstraction.

Within each state definition, a uniform parameter list, ⟨registers⟩, denotes an abstract set of registers and I/O ports. The notion of registers is abstract and may contain arbitrary objects such as registers, memories, stacks, and communication channels.

A state definition expression ⟨exp⟩ defines the body of the state definition. Each expression is defined in terms of the binding construct, `let`, conditional statements, `if` and `case`, or a call to another state, ⟨state-call⟩. The `let` expression provides a means of defining combinational signals, and associating a name with that expression. The `if` and `case` statements, implement a conditional control construct. A ⟨state-call⟩ is a function invocation that denotes a parallel assignment to a set of registers and a transfer of control from one state to another.

A ⟨val⟩ denotes a value of a particular ground type at some intended level of abstraction. Valid terms for ⟨val⟩ are ? (a special symbol that denotes a "don't care" value), variables, constants, booleans, arithmetic operations, boolean operations, routing primitives, and operations on abstract data types.

As an example, the serialized Fibonacci definition presented in Section 3.3 is written as a DDD behavioral specification.

```
(define fib
  (lambda (n)
    (letrec
        ((g (lambda (x y z)
              (if (lt? x 2)
                  y
                  (h (dec x) y z))))
         (h (lambda (x y z)
              (g x z (add y z)))))
      (g n 1 1)))).
```

The specification `fib` takes as input `n`, and computes the `n`-th Fibonacci number. The circuit is defined as a two-state machine: `g` and `h`, with registers `x`, `y` and `z`. The initial invocation of the circuit `(g n 1 1)` assigns values `n`, `1` and `1` to each of the registers, `x`, `y`, and `z` respectively, and transfers control to state `g`. In state `g` the machine, while the condition `(lt? x 2)` is false, transfers control to state `h` updating the register `x` with `(dec x)`. Registers `y` and `z` are unchanged. In state `h` the machine transfers control to state `g` simultaneously updating register `y` with `z` and `z` with `(add y z)`. Once the condition `(lt? x 2)` is satisfied, the algorithm terminates with the value from register `y`.

3.4.3 Structural Specification

A DDD structural specification defines the components in a circuit and their connectivity. The specification expresses logical behavior and physical organization, but does not address electrical characteristics of the circuit. Timing is coordinated by storage elements called *registers* whose behavior in turn is governed by an external synchronizing *clock*. The structural specification is built from *stream equations*, which denote infinite sequences of values over time and a selection combinator defining the control logic of the circuit.

A general form of structural specification in DDD is

```
(define ⟨name⟩
  (lambda ⟨inputs⟩
    (system-letrec
      (⟨streqns⟩ ...)
    ⟨outputs⟩)))
```

where

```
   ⟨name⟩   :=  ⟨identifier⟩
 ⟨inputs⟩   :=  (⟨var⟩ ...)
⟨outputs⟩   :=  ⟨var⟩
             |  (list ⟨var⟩ ...)
 ⟨streqns⟩  :=  [⟨var⟩ (! ⟨init⟩ ⟨val⟩)]
             |  [⟨var⟩ ⟨val⟩]
    ⟨var⟩   :=  ⟨identifier⟩
   ⟨init⟩   :=  ⟨val⟩
    ⟨val⟩   ::= ⟨?⟩ | ⟨var⟩ | ⟨const⟩ | (⟨var⟩ ⟨val⟩ ...)
```

A structural description is defined by a set of *inputs*, *outputs*, and a set of mutually recursive stream equations, denoted by

```
(system-letrec
    (⟨streqns⟩ ...)
 ...)
```

where ⟨strqns⟩ are of the form

(⟨var⟩ (! ⟨init⟩ ⟨val⟩))
or
(⟨var⟩ ⟨val⟩)

A *stream equation*

$$X = S$$

denotes an infinite sequence of uniformly typed values, $< S^0, S^1, \ldots >$. The equation

$$X = v \,!\, S$$

denotes the sequence of values $< v, S^0, S^1, \ldots >$. The ! introduces a delay to the sequence of values and is interpreted as a "register". A constant, c, denotes a constant sequence of values, $< c, c, \ldots >$. The expression $f(S)$ applies the function f element by element to the stream of values from S, denoting the sequence of values, $< f(S^0), f(S^1), \ldots >$.

As an example, consider a counter modeled as a stream.

$$X = 1 \,!\, inc(X)$$

X denotes the stream of values $< 1, 2, 3, 4, 5, ... >$ defined over integers. X is a signal with state, and represents the output of the equation. The equation is initialized with the value 1. $inc(X)$ denotes a combinational incrementor, whose input is X and output is the result of applying the function inc to the value of X.

Another example is a memory modeled as a stream.

$$\mathit{MEM} = \mathit{MEM}^0 \;!\; \mathit{if(WRITE,\ MemWrite(MEM,\ Addr,\ X),\ MEM)}$$

MEM denotes a stream of memories $< MEM^0, MEM^1, MEM^2, MEM^3, ... >$. *MEM*^{0} is the initial memory. If the *WRITE* signal is asserted, *MemWrite* will return a new memory object, with address, *Addr*, updated with value *X*. Otherwise, the memory is returned unchanged. It is important to note that the level of abstraction is arbitrary. In the first example values ranged over integers. In the second example values ranged over memories.

A system of stream equations is *well-formed* when it has the "no combinational feedback" property. In order to establish this property, it is sufficient to guarantee that every feedback cycle must contain a delay element. This corresponds to the Hoffman model for sequential circuits. For example, in the system consisting of a single equation, $X = f(X)$, the sequence is nowhere defined, whereas $X = v \,!\, f(X)$, X_i is, by induction, defined for all i. For example,

$$\begin{aligned} X^1 &= v \\ X^2 &= f(X^1) \\ &\vdots \\ X^n &= f(X^{n-1}). \end{aligned}$$

Continuing with the Fibonacci example introduced in section 3.4.2, the corresponding structural specification is

```
(define fib
  (lambda (n)
    (system-letrec
        ([w (! g (select status g h g))]
         [x (! n (select status x (dec x) x))]
         [y (! 1 (select status y y z))]
         [z (! 1 (select status z z (add y z)))]
         [status (list w (lt? x 2))]
         [rdy (and (equal? w g) (lt? x 2))]
         [ans y])
      (list w x y z status rdy ans))))
```

where

```
          (define select
            (lambda ([s, p0], v0, v1, v2)
              (case s
                (g (if p0 v0 v1))
                (h v2)))).
```

The signals `w`, `x`, `y`, and `z` are registers with initial values `g`, `n`, `1`, and `1`, respectively. There next value is computed as a result of the selection combinator `select`. `select` computes its values based on the `status` signal. The components in the system are a decrement operator, `dec`, and an adder operator, `add`. The status signal is a composite signal of the `w` signal representing the current machine state and the `(lt? x 2)` test. The `rdy` signal is a composite signal of when the state is `g` and the `(lt? x 2)` test. The `ans` equation has the value of `y`. The interpretation is that `ans` will contain the value of `(fib n)` the first time `rdy` is true.

3.5 Description of Transformations

This section presents the transformations in the DDD system. The first part presents a set of definitions and vocabulary for the presentation of the transformations. Following sections describe *behavioral* and *structural* transformations, the construction from *behavior to structure*, and *projection* which incorporates a representation. Small examples are used to convey intuition about the transformations.

Definition 3.5.1: Let E and E' be expressions.

$$E \Rightarrow E'$$

denotes the application of a transformation from E to obtain E'. Similarly,

$$E \Leftrightarrow E'$$

denotes the application of an invertible transformation, from E to obtain E' or E' to obtain E.

Definition 3.5.2: A *term* is either a constant, a (typed) variable, or an application $f(T_1, \ldots, T_n)$ of an operation f to the terms T_i, where $i \in \{1, \ldots, n\}$.

Definition 3.5.3: An *identifier* is either a variable or a list of identifiers $(X_1, \ldots, X_n)$.

Definition 3.5.4: Let E be an expression over identifier X and let T be an arbitrary term.

$$E\,[T_1/X_1, \ldots, T_n/X_n]$$

denotes the expression obtained by simultaneous substitutions of T_i for X_i in E, $i \in \{1, \ldots, n\}$. The notion of substitution will later be extended to the notion of expansion where the T_i may be arbitrary closed expressions.

Definition 3.5.5: A *decision combinator*, C, is a selector that selects between n alternatives based on a set of predicates, p. The binary selection operation is defined as

$$sel(p, v_1, v_2) = \begin{cases} v_1 \text{ if } p \text{ is true} \\ v_2 \text{ if } p \text{ is false} \\ ?_\tau \text{ if } p \text{ is } ?_{bool} \end{cases}$$

This binary selector is generalized to n-way selectors built from combinations of *sel*.

Some laws on *sel* are:

$$\begin{array}{rcl}
sel(p,?,a) & \Rightarrow & a \\
sel(p,a,?) & \Rightarrow & sel(p,a,b) \\
sel(p,?,b) & \Rightarrow & sel(p,a,b) \\
sel(p,a,a) & \Leftrightarrow & a \\
sel(true,a,b) & \Leftrightarrow & a \\
sel(false,a,b) & \Leftrightarrow & b \\
sel(p,sel(q,a,b),sel(q,c,d)) & \Leftrightarrow & sel(q,sel(p,a,c),sel(p,b,d)).
\end{array}$$

As an abbreviation of notation, the expression

$$selp\{T_1, \ldots, T_n\}$$

denotes the application of the selector, *sel*, based on predicate, p, and the set of terms $\{T_1, \ldots, T_n\}$. The terms within the braces does not specify an ordering in the selector alternatives. The notation

$$selp_i\{T_1, \ldots, T_n\}$$

denotes the term T_n^i, which represents the $i'th$ alternative in the selector term.

Definition 3.5.6: s_1 and $s_2 \in S$, where S is the set of stream equations, are *compatible*, denoted by, $s_1 \cong s_2$, if they have the same (unifiable) types, τ, and their right hand sides are syntactically equivalent. Terms are syntactically equivalent if they are the same term, or match on don't cares.

Two terms, v and v' are compatible if they are provably equal. For example, if v can be rewritten to v' under the algebraic laws of the type. The notion of compatibility extends inductively with the structure of signal expressions. Specifically, if $u \cong u'$ and $v \cong v'$, then *(sel p u v)* $\cong$ *(sel p u' v')* in the case of a selector.

3.5.1 Behavioral Transformations

The class of behavioral transformations manipulates the behavioral description.

Transformation 3.5.1: *Distribute Conditional*

Distribute conditional takes an `if` expression within a function application and distributes the function application across the conditional. Given an arbitrary function application

$$f(T_1, \ldots, T_i, \ldots, T_n)$$

and T_i is a conditional expression of the form $p \rightarrow T_j, T_k$, where $j, k \in \{1, \ldots, n\}$, then the conditional is moved outside the call to f giving

$$p \rightarrow f(T_1, \ldots, T_j, \ldots, T_n), f(T_1, \ldots, T_k, \ldots, T_n)$$

where T_i is replaced with T_j in the consequent and T_k in the alternative.

For instance, distributing the conditional in

$$f(p \rightarrow x, y) \Leftrightarrow p \rightarrow f(x), f(y).$$

Transformation 3.5.2: *Unfolding/Folding*

Unfolding replaces an application of a function with its definition. *Folding* is the opposite of unfolding. Given an arbitrary function application

$$f(T_1, \ldots, T_n)$$

and its definition

$$f(X_1, \ldots, X_n) = E$$

then by unfolding, $f(T_1, \ldots, T_n)$ is replaced with $E\,[T_1/X_1, \ldots, T_n/X_n]$, instantiating the formal parameters with the actuals.

For instance, given the definition $f(x) = g(x, 1)$ we can rewrite

$$f(a) \Leftrightarrow g(a, 1)$$

Transformation 3.5.3: *State Introduction/Elimination*

State Introduction introduces a new state definition. Given a set of state definitions

$$\begin{aligned} f_1(X_1, \ldots, X_n) &= E_1 \\ &\vdots \\ f_m(X_1, \ldots, Xn) &= E_m \end{aligned}$$

a new state definition

$$f_{m+1}(X_1, \ldots, X_n) = E_{m+1}$$

can be introduced where

$$f_{m+1} \notin \{f_1, \ldots, f_m\}.$$

For instance, in the Fibonacci example, given state g

$$g(x, y, z) = lt?(x, 2) \rightarrow y, g(dec(x), z, add(y, z))$$

introducing state h yields

$$\begin{aligned} g(x, y, z) &= lt?(x, 2) \rightarrow y, g(dec(x), z, add(y, z)) \\ h(x, y, z) &= g(x, z, add(y, z)). \end{aligned}$$

In DDD, state introduction, and folding is used to *serialize* the behavioral specification. The transformations are used to replace a function call with a series of function calls, whose composition is equivalent to the original term.

For example, the *inc* and *dec* operations are serialized by introducing a state g, and folding the call $f(inc(x), dec(y))$ into g:

$$\begin{array}{rcl}
f(x,y) & = & p \to r, f(inc(x), dec(y)) \\
& \Downarrow & \text{introduce } g \\
f(x,y) & = & p \to r, f(inc(x), dec(y)) \\
g(x,y) & = & f(x, dec(y)) \\
& \Downarrow & \text{fold } f(inc(x), dec(y)) \text{ in } g \\
f(x,y) & = & p \to r, g(inc(x), y) \\
g(x,y) & = & f(x, dec(y)).
\end{array}$$

Constraints on the design may impose restrictions on what can be accomplished in a single state definition; hence, it is often necessary to introduce new state definitions, and fold these into the existing specification in order to reduce a complex state definition to a series of simpler ones.

3.5.2 Behavior to Structure Construction

Transformation 3.5.4: *ItrSys→SeqSys*

ItrSys→SeqSys constructs a structural sequential system from an iterative behavioral specification. The construction provides a formal bridge between behavior and structure.

The iterative recursion scheme

$$f(x_1, \ldots, x_n) = p \to r, f(t_1, \ldots, t_n)$$

is realized by the simultaneous system of equations

$$\begin{array}{rcl}
X_1 & = & x_1 \,!\, t_1 \\
X_2 & = & x_2 \,!\, t_2 \\
& \vdots & \\
X_n & = & x_n \,!\, t_n \\
RDY & = & p \\
ANS & = & r
\end{array}$$

In DDD, a sequential system is composed of the decision combinator, *sel*, and a system description of a set of equations. The decision combinator is derived from the conditional structure implicit in the behavioral specification. The set of equations is derived from the formal parameters defined in the behavioral specification. Each equation is of the form:

$$X_i = x_0 \,!\, sel(p, t_1, \ldots, t_n)$$

where ! denotes the binary delay operator for registered values, x_0 denotes an initial value and $sel(p, t_1, \ldots, t_n)$ denotes the selection of possible values, $t_1, \ldots, t_n$ based on predicate p.

3.5.3 Structural Transformations

The class of structural transformations manipulate the structural description.

Transformation 3.5.5: *Identification/Expansion*

Identification is giving a name to an expression by adding an equation for it. Identification of like terms by an equation has the effect of eliminating redundant circuitry. *Expansion* is the inverse of identification.

Suppose $X_i = S_i$, $i \in \{1, \ldots, n\}$ is an arbitrary system of stream equations, and T is an arbitrary term in S_i, then an equation

$$X_{n+1} = T$$

can be introduced, replacing all occurrences of T in $S_1, \ldots, S_n$ with X_{n+1}.

For instance, identifying $add(a, b)$ with z in

$$\begin{array}{rcl} x & = & sel(p, add(a,b), r) \end{array} \Leftrightarrow \begin{array}{rcl} x & = & sel(p, z, r) \\ z & = & add(a,b). \end{array}$$

Transformation 3.5.6: *Merge*

Merge yields a merging of signal equations by instantiating don't cares, which are denoted by `?`, and like terms. A physical interpretation is that several signals share a common bus when there are no value conflicts.

Suppose $X_i = S_i$, $i \in \{1, \ldots, n\}$ is an arbitrary system of stream equations, and the $S_i^\prime s$ are compatible (see Definition 3.5.6), then these n equations can be replaced with

$$\begin{array}{rcl} X_1 & = & merge(S_i) \\ X_2 & = & X_1 \\ \cdots & & \\ X_n & = & X_1. \end{array}$$

Of course, normally all remaining occurrences of $X_2 \ldots X_n$ in the original system would be eliminated by replacing them with X_1.

For instance, merging x and y in

$$\begin{array}{rcl} x & = & n\,!\,sel(p, y, ?, x) \\ y & = & n\,!\,sel(p, ?, w, x) \end{array}$$

returns

$$\begin{array}{rcl} x & = & z \\ y & = & z \\ z & = & n\,!\,sel(p, z, w, z). \end{array}$$

Transformation 3.5.7: *Generalization*

Generalization yields the introduction of don't care arguments to normalize function calls across several functions. Suppose $f(X_1, \ldots, X_n)$ and $g(X_1, \ldots, X_m)$ are functions with airity m and n respectively, and $m > n$, then f can be extended to f' such that the

$$f'(X_1, \ldots, X_n, \ldots, X_m) = f(X_1, \ldots, X_n)$$

where the arguments to f' are padded with $m \Leftrightarrow n$ don't care arguments.

In DDD, this transformation normalizes function calls across selectors. In particular, this is later used in a process called factorization in Transformation 3.5.9.

For instance, generalizing $f(x, y)$ in

$$z = sel(p, f(x, y), g(u, v, w))$$

returns

$$z = sel(p, f'(x, y, ?), g(u, v, w))$$

with f extended to

$$f'(a, b, c) = f(a, b).$$

Transformation 3.5.8: *Distribution*

Distribution is the distribution of the decision combinator, *sel*, over a set of normalized function applications. Given an arbitrary selector with normalized functions

$$sel(p, f(X_1), \ldots, f(X_n))$$

the selector can be distributed over f as in

$$f(sel(p, X_1, \ldots, X_m)).$$

For instance, distributing *sel* in

$$z = sel(p, f(x, y), f(u, v)$$

returns

$$z = f(sel(p, x, y), sel(p, u, v)).$$

Transformation 3.5.9: *Factorization*

Factorization encapsulates a set of operations or signals into a subsystem in order to isolate them from the description. The encapsulated subsystem is called an *abstract component*. The transformation maintains the correct connectivity between the system description and the abstract component.

In DDD there are two ways of doing factorizations. The first, called a *general factorization*, is to state the set of operations that are to be encapsulated. The subject terms are those in which members of the set are applied. The second, called a *signal factorization*, encapsulates a signal; in this case the subject terms are those in which that signal's name occurs as an argument.

Given $X_i = S_i$, $i \in \{1..n\}$, is a system of stream equations, and $f^1(T_1^1, \ldots, T_n^1)$, $f^2(T_1^2, \ldots, T_n^2)$, ..., $f^m(T_1^m, \ldots, T_n^m)$ are function applications occuring in one or more of the S_is, then factorization synthesizes an abstract component

$$\begin{array}{l} abs(i, v_1, \ldots, v_n) = \\ \quad case\ i \\ \qquad \mathtt{f_1} : f_1(v_1, \ldots, v_n) \\ \qquad \mathtt{f_2} : f_2(v_1, \ldots, v_n) \\ \qquad \quad \vdots \\ \qquad \mathtt{f_m} : f_m(v_1, \ldots, v_n) \end{array}$$

and introduces the equation

$$X_{abs} = abs(selp\{\mathtt{f}^1, \ldots, \mathtt{f}^m\}, selp\{T_1^1, \ldots, T_1^m\}, \ldots, selp\{T_n^1, \ldots, T_n^m\})$$

to communicate with the abstract component.

Transformation 3.5.10: *Simplification*

Simplification takes the partial knowledge of the input in order to simplify an expression. Given an arbitrary function application, $f(T_1, \ldots, T_n)$ and its definition

$$f(X_1, \ldots, X_n) = E$$

then, f can be simplified with respect to the known inputs $T_1, \ldots, T_n$ in E and replaced with the specialized function f' where

$$f'() = E\,[T_1/X_1, \ldots, T_n/X_n]$$

replacing $f(T_1, \ldots, T_n)$ with $f'()$.

In DDD, simplification of selectors is used to derive the state generator, and the various instruction generators found in a typical design. For instance, the state equation from the Fibonacci example

$$w \;=\; \mathtt{g}\,!\,\mathit{select}(\mathit{status}, \mathtt{g}, \mathtt{h}, \mathtt{g})$$

where

$$\begin{array}{l} \mathit{select}([s, p0], v0, v1, v2) = \\ \quad \mathit{case}\ s \\ \quad\quad \mathtt{g} : p0 \rightarrow v0, v1 \\ \quad\quad \mathtt{h} : v2 \end{array}$$

is rewritten as

$$w \;=\; \mathtt{g}\,!\,\mathit{next}_w(\mathit{status})$$

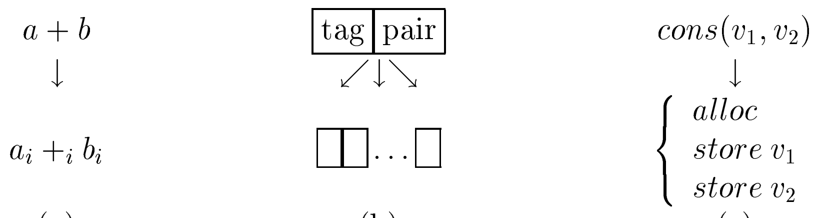
where

$$\begin{array}{l} \mathit{next}_w([s, p0]) = \\ \quad \mathit{case}\ s \\ \quad\quad \mathtt{g} : p0 \rightarrow \mathtt{g}, \mathtt{h} \\ \quad\quad \mathtt{h} : \mathtt{g}. \end{array}$$

The definition of next_w denotes the next-state generator for the Fibonacci specification.

3.5.4 Projection Transformations

Projection is a mechanism of incorporating representations of a more abstract type by another more concrete type. The notion of incorporating representations spans a continuum:



(a) (b) (c)

Bit projection (a) is the simplest case of incorporating representations, but is a necessary step that must be taken in any derivation of hardware. In this case, variables and operators are projected to their equivalent bit entities. A slightly more complex form of projection (b) is where complex data types are projected to simple records. This can be a hierarchical process. The notion of projection is also extended to the implementation of functions (c), where a single function is projected to a series of function calls or a protocol in the target architecture.

DDD implements those forms of projection that allow the derivation of real hardware from concrete behavioral specifications. This is at least as well as any existing synthesis system does.

Transformation 3.5.11: *Projection*

Projection takes a set of binary type declarations and a structural description, and returns a structural description at the target level of representation.

For instance, the projection of

$$x = 0\,!\,sel(p, x, ptr(y))$$

to a binary representation of three bits is declared with

$$\begin{array}{rcl} x & \Rightarrow & list(x_0, x_1, x_2) \\ y & \Rightarrow & list(y_0, y_1, y_2, y_3, y_4) \\ 0 & \Rightarrow & list(f, f, f) \\ ptr & \Rightarrow & ptr(a) = firstn(a, 3). \end{array}$$

The type information is used to rewrite equation x to a system of three equations, x_0, x_1, and x_2, one for each bit. The new system of equations is

$$\begin{aligned} x_0 &= f \,!\, sel(p, x_0, y_0) \\ x_1 &= f \,!\, sel(p, x_1, y_1) \\ x_2 &= f \,!\, sel(p, x_2, y_2). \end{aligned}$$

The constant 0 and signal y are projected to their respective bit equivalences. The operation *ptr* coerces a 4-bit signal to the first three bits of its argument.

Chapter 4

Derivation of the DDD-FM9001

The DDD-FM9001 derivation was done interactively with the DDD system. The transformations were applied to the specification, developing a derivation path through the design space satisfying an intended set of design constraints. An initial script was a sequence of commands to the DDD system. Several derivation paths were explored and the derivation refined. The final derivation script includes thirty coarsely grained commands to the derivation system. The complete script, consisting of approximately 1,000 lines, can be found in Appendix A.

Figure 4.1 illustrates the derivation path from Hunt's *FM9001 Specification* to the *DDD-FM9001 Realization.* Transformations on the descriptions are shown as labeled arcs, $< \tau_1 ... \tau_6, \upsilon_1 >$, where τ_n denotes the application of a transformation and υ_r denotes verification. The diagram is intended to characterize the distinct phases of the derivation. A class of transformations called, *behavioral transformations*, are applied to the initial specification in order to achieve a proper scheduling of operations. Once a suitable behavioral description is derived, DDD constructs an abstract system description, composed of a decision combinator, *Select*, representing control, and a structural component, *System*, representing an initial estimation of architecture. A second class of transformations, called *structural transformations*, imposed a logical organization on the design. In this phase of the derivation, the structural description was refined to an architecture. A sequence of factorization steps was applied to isolate the memory, register-file, and arithmetic components from the description. Implementations for the arithmetic components were engineered by hand mechanically verified with respect to the factored components. A third class of transformations introduced a lower-level representation producing a hierarchy of boolean subsystems. A final gate-level description was input to the ACTEL logic synthesis tool.

$$\begin{tikzcd}
\mathit{fm9001\ specification} \arrow[d] \\
Behsys_0 \arrow[r, leftrightarrow, "unfold\,let"] & Behsys_1 \arrow[r, leftrightarrow, "expand\,if*"] & Behsys_2 \arrow[r, leftrightarrow, "serialize"] & Behsys_3 \arrow[r, leftrightarrow, "distribute\,if"] & Behsys_4 \arrow[dllll, bend left, "behavior\ to\ structure"'] \\
Select \bullet System \arrow[r, leftrightarrow, "factor"] & Select \bullet System' \left\| \begin{matrix} mem \\ regs \\ alu \\ inc \\ dec \end{matrix} \right. \arrow[r, leftrightarrow, "partial\ eval"] \arrow[dl, bend left, "project"'] & Select \bullet System'' \left\| \begin{matrix} mem \\ regs \\ alu \\ inc \\ dec \\ next{\Leftrightarrow}state \end{matrix} \right. \\
Select_R \bullet System''_R \left\| \, next{\Leftrightarrow}state \right._R , \arrow[d] & < alu, inc, dec >_R \arrow[u, "verify"'] \\
\mathit{logic\ synthesis}
\end{tikzcd}$$

Figure 4.1: DDD-FM9001 Derivation Path

The block diagram for the DDD-FM9001 is shown in Figure 4.2. This diagram illustrates the DDD-FM9001 architecture, showing the registers and the interconnectivity between them. Shaded areas denote components that are verified, while the rest of the design is derived. The register-file (R), is treated as a primitive as is the

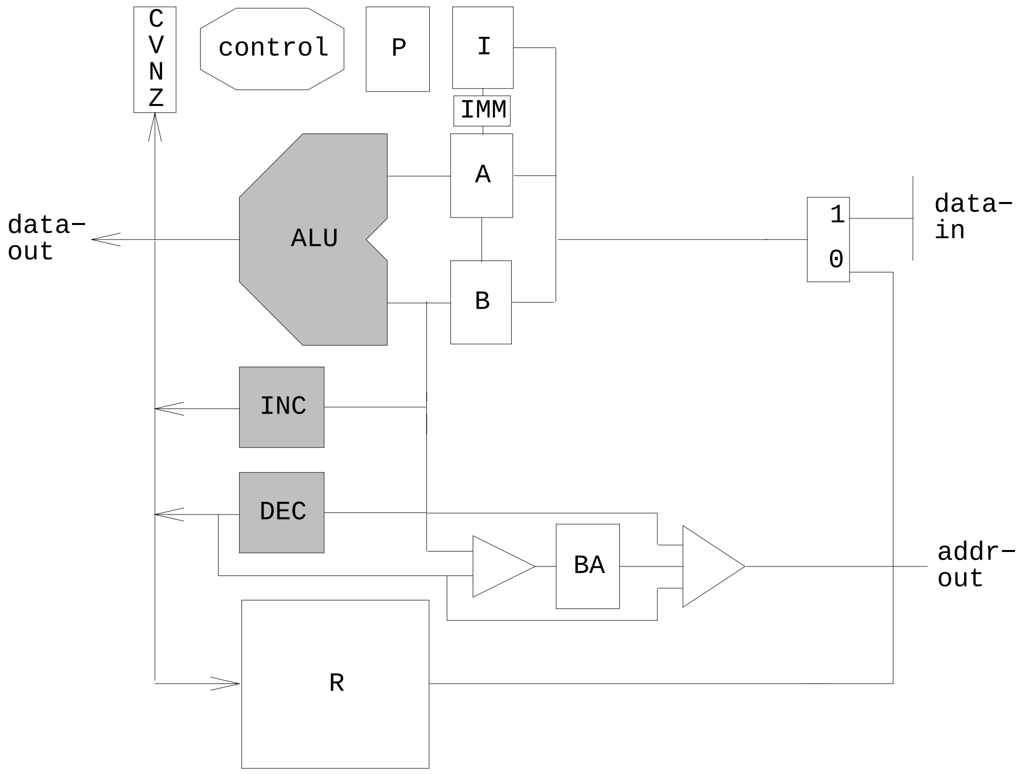


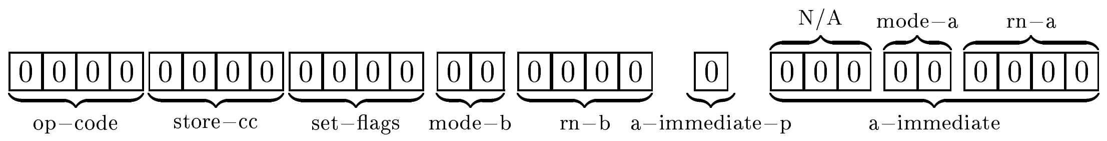
Figure 4.2: DDD-FM9001 Block Diagram

memory (not shown). Data enters the processor from the data input port (data-in) and is either latched in the operand-a (A), operand-b (B), or instruction register (I). Internally, data flows from the operand-b to the operand-a register, and from the instruction register to the operand-a register. Input data may flow into the register-file or into latches for processing by the ALU. Output from the ALU can be stored into the register-file or memory. The data bus is bi-directional but is denoted here as data-in and data-out. The register-file and the memory are on the data bus. A detailed account of the derivation follows.

4.1 FM9001 Specification

The FM9001 specification defines a 32-bit general purpose microprocessor architecture with sixteen 32-bit registers, a programmable program counter, sixteen instructions, and five addressing modes. The highest level of specification is the abstract *programmer's model*, in which a collection of six recursive functions defines an instruction level interpreter. The state of the machine consists of a memory, `mem`, a register-file, `regs`, four flags, `flags`, a register containing the address of the program counter, `pc-reg`, an instruction register, `i-reg`, the A and B operand registers, `operand-a` and `operand-b`, and an address calculation register, `b-address`.

The FM9001 has a 32-bit instruction word with the high order 4-bits unspecified.



The instruction set is shown in Figure 4.3. A 4-bit op-code selects among sixteen instructions. A 4-bit store-cc field assigns the result conditionally according to `flags`. A 4-bit set-flags field sets the flags conditionally. The B operand is a 6-bit mode/register (mode-b/rn-a) pair determining the addressing mode, and register address. The A operand is a 10-bit field. If the high order bit a-immediate-p is set, the low order nine bits are treated as a signed immediate. Otherwise, the low order six bits of the A operand are a mode/register pair (mode-a/rn-a) identical to the B operand. The addressing modes are immediate, register direct, register indirect, register indirect with pre-decrement, and register indirect with post-increment. Details of the instruction set, condition codes, addressing modes, and register addresses are given in Appendix A.

0000	$b \leftarrow a$	Move
0001	$b \leftarrow a + 1$	Increment
0010	$b \leftarrow a + b + c$	Add with carry
0011	$b \leftarrow a + b$	Add
0100	$b \leftarrow 0 \Leftrightarrow a$	Negation
0101	$b \leftarrow a \Leftrightarrow 1$	Decrement
0110	$b \leftarrow b \Leftrightarrow a \Leftrightarrow c$	Subtract with borrow
0111	$b \leftarrow b \Leftrightarrow a$	Subtract
1000	$b \leftarrow a \gg 1$	Rotate right, shifted through carry
1001	$b \leftarrow a \gg 1$	Arithmetic shift right, top bit copied
1010	$b \leftarrow a \gg 1$	Logical shift right, top bit zero
1011	$b \leftarrow a \oplus b$	Exclusive or
1100	$b \leftarrow a \vee a$	Or
1101	$b \leftarrow a \wedge b$	And
1110	$b \leftarrow \neg a$	Not
1111	$b \leftarrow a$	Move

Figure 4.3: FM9001 Instruction Set

Each of Hunt's six recursive functions are discussed in detail. The top-level definition which iterates the machine through instruction cycles is

```
(defn fm9001-intr (state oracle)
  (if (nlistp oracle) state
      (fm9001-intr (fm9001-step state (car oracle))
                   (cdr oracle))))
```

The `oracle` is used to determine which register is the program counter and a termination condition for the machine. If the `(nlistp oracle)` condition is satisfied, the algorithm terminates with the current `state`, otherwise the machine cycles executing a single instruction and updating the `oracle`. The thread of control follows the state sequence `fm9001-step`, `fm9001-fetch`, `fm9001-operand-a`, `fm9001-operand-b`, and `fm9001-alu-operation`. Each of these functions is discussed in order to trace the fetch-execute instruction cycle of the FM9001.

```
(defn fm9001-step (state pc-reg)
  (let ((p-state (car state)) (mem (cadr state)))
    (fm9001-fetch (regs p-state) (flags p-state) mem pc-reg)))
```

The `fm9001-step` function unpacks the `state` into the register-file, `regs`, the flags, `flags`, and the memory, `mem`. The function then calls the next state `fm9001-fetch`.

```
(defn fm9001-fetch (regs flags mem pc-reg)
  (let ((pc (read-mem pc-reg regs)))
    (let ((ins (read-mem pc mem)))
      (let ((pc+1 (v-inc pc)))
        (let ((new-regs (write-mem pc-reg regs pc+1)))
          (fm9001-operand-a new-regs flags mem ins))))))
```

The `fm9001-fetch` function fetches the contents of the program counter, fetches the instruction from memory, and increments and updates the program counter. Control is then transferred to state `fm9001-operand-a`.

```
(defn fm9001-operand-a (regs flags mem ins)
  (let ((a-immediate-p (a-immediate-p ins))
        (a-immediate (sign-extend (a-immediate ins) 32))
        (mode-a (mode-a ins))
        (rn-a (rn-a ins)))
    (let ((reg (read-mem rn-a regs)))
      (let ((reg- (v-dec reg))
            (reg+ (v-inc reg)))
        (let ((operand-a (if* a-immediate-p a-immediate
                             (if* (reg-direct-p mode-a) reg
                                  (if* (pre-dec-p mode-a)
                                       (read-mem reg- mem)
                                       (read-mem reg mem))))))
          (let ((new-regs (if* a-immediate-p regs
                              (if* (pre-dec-p mode-a)
                                   (write-mem rn-a regs reg-)
                                   (if* (post-inc-p mode-a)
                                        (write-mem rn-a regs reg+)
                                        regs)))))
            (fm9001-operand-b new-regs flags mem ins operand-a)))))))
```

The `fm9001-operand-a` function decodes the instruction and readies the A operand, `operand-a`. First, the register index is loaded from the register-file and stored in the variable `reg`. Then, depending on the addressing mode, simultaneously, the register-file is updated and the A operand is loaded with the appropriate value. The indirect addresses for pre-decrement and post-increment are computed and stored in variables `reg-` and `reg+`, respectively. The following table shows the effect on the A operand of the various addressing modes.

Operation	Address Mode
`operand-a` ← `a-immediate`	Immediate
`operand-a` ← `@rn-a`	Register Direct
`operand-a` ← `@(rn-a)`	Register Indirect
`operand-a` ← `@-(rn-a)`	Register Indirect with Pre-decrement
`operand-a` ← `@(rn-a)+`	Register Indirect with Post-increment

Control is then transferred to state `fm9001-operand-b`.

```
(defn fm9001-operand-b (regs flags mem ins operand-a)
  (let ((mode-b (mode-b ins))
        (rn-b (rn-b ins)))
    (let ((reg (read-mem rn-b regs)))
      (let ((reg- (v-dec reg))
            (reg+ (v-inc reg)))
        (let ((b-address (if* (pre-dec-p mode-b) reg- reg)))
          (let ((operand-b (if* (reg-direct-p mode-b)
                                reg
                                (read-mem b-address mem)))
                (new-regs (if* (pre-dec-p mode-b)
                               (write-mem rn-b regs reg-)
                               (if* (post-inc-p mode-b)
                                    (write-mem rn-b regs reg+)
                                    regs))))
            (fm9001-alu-operation new-regs flags mem ins operand-a
                                  operand-b b-address)))))))
```

The `fm9001-operand-b` function decodes the instruction and readies the B operand, `operand-b`. The instruction decode is similar to `fm9001-operand-a`, except there is no immediate value computation, and the `b-address` is held for the final state, `fm9001-alu-operation`.

```
(defn fm9001-alu-operation (regs flags mem ins operand-a operand-b b-address)
  (let ((op-code (op-code ins))
        (store-cc (store-cc ins))
        (set-flags (set-flags ins))
        (mode-b (mode-b ins))
        (rn-b (rn-b ins)))
    (let ((cvzbv (v-alu (c-flag flags) operand-a operand-b op-code))
          (storep (store-resultp store-cc flags)))
      (let ((bv (bv cvzbv)))
        (let ((new-regs (if* (and* storep (reg-direct-p mode-b))
                             (write-mem rn-b regs bv)
                             regs))
              (new-flags (update-flags flags set-flags cvzbv))
              (new-mem   (if* (and* storep (not* (reg-direct-p mode-b)))
                              (write-mem b-address mem bv)
                              mem)))
          (list (list new-regs new-flags) new-mem))))))
```

The `fm9001-alu-operation` decodes the instruction, executes the instruction and conditionally stores the result.

4.2 Transformations on Behavior

In the first phase of the derivation, a set of transformations is applied to the FM9001 behavioral specification. These transformations preserve the meaning of the specification while impacting the resulting structural model of the derived circuit. This relationship between behavior and structure is based on how DDD interprets the behavior constructs of the specification. Therefore, many of the transformations at this stage prepare the specification for the construction from behavior to structure. Other transformations at this stage impose an ordering on operations. The transformations applied are *let unfolding*, *function expansion*, *serialization*, and *if distribution*. These steps are described in the following sections in the order in which they are applied.

4.2.1 Unfolding: `let`

In the FM9001, the initial specification contains *let expressions.* In DDD, *let expressions* denote combinational signals. These bindings refer to intermediate values that are ultimately stored in registers.

For example, in the function definition `fm9001-fetch`

```
...
(fm9001-fetch
  (lambda (regs flags mem pc-reg ins ...)
    (let ((pc (read-mem pc-reg regs)))
      (let ((ins (read-mem pc mem)))
        (let ((pc+1 (v-inc pc)))
          (let ((new-regs (write-mem pc-reg regs pc+1)))
            (fm9001-operand-a new-regs flags mem pc-reg ins ...)))))))
...
```

let expressions are used to name intermediate values for computing the `fm9001-fetch` state of the processor. Once these intermediate values are computed, the processor stores the necessary results in its register set, and transfers control to the `fm9001-operand-a` state. The true effect on the state of the processor in this step is to update the register-file, `regs`, with an incremented program counter and load the instruction register, `ins`, from the memory location specified with the value from the program counter.

In DDD, *let expressions* provide a mechanism for naming a particular combinational computation. When DDD derives a structural specification, DDD maintains the names of such signals. In the FM9001 specification, maintaining signals such as `pc`, `pc+1`, or `new-regs`, simply clutter the specification. Although naming could be done later, at this point these definitions are expanded to eliminate the intermediate signal names.

For example, to eliminate the intermediate signal `pc`, a simple unfolding of the *let expression* is done. This has the effect of replacing each occurrence of `pc` in the body of the *let expression* with its binding.

```
...
(fm9001-fetch
  (lambda (regs flags mem pc-reg ins ...)
    (let ((ins (read-mem (read-mem pc-reg regs) mem)))
      (let ((pc+1 (v-inc (read-mem pc-reg regs))))
        (let ((new-regs (write-mem pc-reg regs pc+1)))
          (fm9001-operand-a new-regs flags mem pc-reg ins ...))))))
...
```

Successive *unfold let* transformations are applied to the specification until all the *let expressions* and the bindings they denote are eliminated. The final result can be seen below.

```
...
(fm9001-fetch
 (lambda (regs flags mem pc-reg ins ...)
   (fm9001-operand-a (write-mem pc-reg regs (v-inc (read-mem pc-reg regs)))
                     flags mem pc-reg (read-mem (read-mem pc-reg regs) mem)
                     ...))))))
...
```

4.2.2 Expanding: `if*`

The conditional structure, denoted by *if statements* and *case statements* in specifications is key to DDD's fundamental transformation of decomposing control and architecture. In the FM9001 top-level specification the *if statement* is not visible because it is buried within the definition of the function `if*`. The specification fragment below illustrates the problem.

```
(fm9001-operand-a
 (lambda (... operand-a ...)
   (fm9001-operand-b
    ...
    (if* (pre-dec-p (mode-a ins))
             (read-mem (v-dec (read-mem (rn-a ins) regs)) mem)
             (read-mem (read-mem (rn-a ins) regs) mem))
    ...)))
```

The definition of `if*`

```
(defn if* (a b c) (if a b c))
```

hides the `if` operation from the top-level specification. To uncover the `if` in `if*`, *function expansion* is applied to replace all applications of `if*` with its definition and instantiated arguments. The resulting specification fragment shows how the `if*` operation is replaced with `if`.

```
(fm9001-operand-a
 (lambda (... operand-a ...)
   (fm9001-operand-b
    ...
    (if (pre-dec-p (mode-a ins))
            (read-mem (v-dec (read-mem (rn-a ins) regs)) mem)
            (read-mem (read-mem (rn-a ins) regs) mem))
    ...)))
```

This example may seem trivial in terms of expand function. However, the transformation is general and is used to replace an occurrence of a function application with its body. In this example, the term `if*` is not simply being replaced with `if`, but the formal parameters are being instantiated with the actual parameters in the application of `if*` with the body of `if*`.

4.2.3 Scheduling `mem-write` and `mem-read`

In a process analogous to *scheduling* in high-level synthesis [50], the DDD system is guided through a series of *folding* and *unfolding* transformations in order to im-

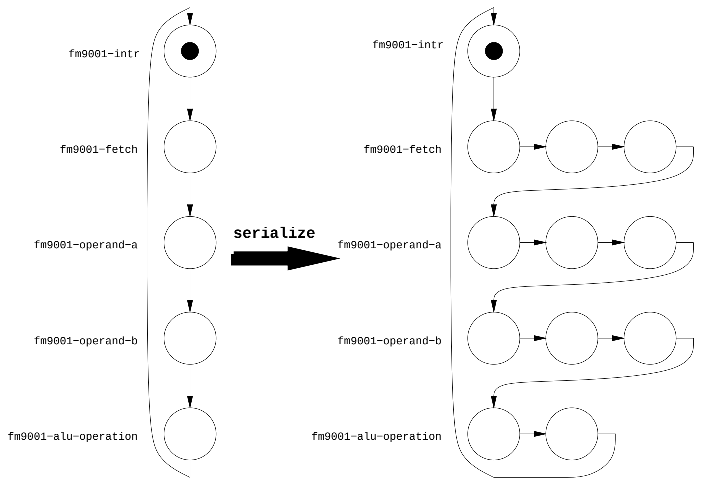


Figure 4.4: DDD-FM9001 Serialization

pose a desired scheduling of the memory and register-file operations, `mem-read` and `mem-write`. The goal is to reduce the parallelism in the original specification. In this phase, called *serialization*, a function call is replaced by a sequence of function calls, whose composition is equivalent to the original term.

Seven serialization steps, adding seven new states to the design (Figure 4.4), are necessary to produce a design in which abstract operations on the memory and the register-file are restricted to at most one memory access per state. The operations are also serialized to insure that accesses to the memory and the register-file can be multiplexed. This is necessary because the target FPGA technology will not support the entire logic of the FM9001 and the register-file, thus forcing a design constraint that the register-file be implemented outside the chip. It will be necessary to have the memory and register-file share the bus in order to minimize the pins used.

Each of the four states, `fm9001-fetch`, `fm9001-operand-a`, `fm9001-operand-b`, and `fm9001-alu-operation` are serialized. Each serialization transformation involves the introduction of a new state and the folding of the new state into the state to be serialized.

First, the `fm9001-fetch` state is serialized.

```
(fm9001-fetch
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-operand-a (write-mem pc-reg regs (v-inc (read-mem pc-reg regs)))
                     flags mem pc-reg (read-mem (read-mem pc-reg regs) mem)
                     operand-a operand-b b-address oracle)))
```

The original state is decomposed into three states:

```
(fm9001-fetch
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-fetch_1 regs flags mem pc-reg ins operand-a
                   (read-mem pc-reg regs) b-address oracle)))
(fm9001-fetch_1
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-fetch_2 regs flags mem pc-reg (read-mem operand-b mem)
                   operand-a operand-b b-address oracle)))
(fm9001-fetch_2
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-operand-a (write-mem pc-reg regs (v-inc operand-b))
                     flags mem pc-reg ins operand-a operand-b
                     b-address oracle)))
```

In state `fm9001-fetch`, the program counter is read from the register-file and loaded into the `operand-b` register. This register is used as a temporary hold for the program counter. In state `fm9001-fetch_1`, the instruction is loaded from memory into the instruction register. Finally, in state `fm9001-fetch_2`, the program counter value is incremented and written to the register-file.

Second, the `fm9001-operand-a` state is serialized.

```
[fm9001-operand-a
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-operand-b
    (if (a-immediate-p ins) regs
        (if (pre-dec-p (mode-a ins))
            (write-mem (rn-a ins) regs
              (v-dec (read-mem (rn-a ins) regs)))
            (if (post-inc-p (mode-a ins))
                (write-mem (rn-a ins) regs
                  (v-inc (read-mem (rn-a ins) regs)))       regs)))
    flags mem pc-reg ins
    (if (a-immediate-p ins) (sign-extend (a-immediate ins) thirty-two)
        (if (reg-direct-p (mode-a ins)) (read-mem (rn-a ins) regs)
            (if (pre-dec-p (mode-a ins))
                (read-mem (v-dec (read-mem (rn-a ins) regs)) mem)
                (read-mem (read-mem (rn-a ins) regs) mem))))
    operand-b b-address oracle))]
```

The original state is decomposed into three states:

```
[fm9001-operand-a
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-operand-a_1 regs flags mem pc-reg ins operand-a
                       (read-mem (rn-a ins) regs) b-address oracle))]
[fm9001-operand-a_1
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-operand-a_2
    (if (a-immediate-p ins) regs
        (if (pre-dec-p (mode-a ins))
            (write-mem (rn-a ins) regs (v-dec operand-b))
            (if (post-inc-p (mode-a ins))
                (write-mem (rn-a ins) regs (v-inc operand-b)) regs)))
    flags mem pc-reg ins operand-a operand-b b-address oracle))]
[fm9001-operand-a_2
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-operand-b
    regs flags mem pc-reg ins
    (if (a-immediate-p ins) (sign-extend (a-immediate ins) thirty-two)
        (if (reg-direct-p (mode-a ins)) operand-b
            (if (pre-dec-p (mode-a ins)) (read-mem (v-dec operand-b) mem)
                (read-mem operand-b mem)))) operand-b b-address oracle))]
```

In the `fm9001-operand-a` state, the contents of the register index is loaded into the `operand-b` register. As in the previous serialization step, the `operand-b` register is used as a temporary. In the `fm9001-operand-a_1` state, the register-file is updated depending on the addressing mode. Finally, in the `fm9001-operand-a_2` state, the `operand-a` register is loaded with either, the immediate field of the instruction, the `operand-b` register, or a value from memory, depending on the addressing mode.

Third, the `fm9001-operand-b` state is serialized.

```
[fm9001-operand-b
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-alu-operation
    (if (pre-dec-p (mode-b ins))
        (write-mem (rn-b ins) regs
          (v-dec (read-mem (rn-b ins) regs)))
        (if (post-inc-p (mode-b ins))
            (write-mem (rn-b ins) regs
              (v-inc (read-mem (rn-b ins) regs)))
            regs))
    flags mem pc-reg ins operand-a
    (if (reg-direct-p (mode-b ins))
        (read-mem (rn-b ins) regs)
        (read-mem
          (if (pre-dec-p (mode-b ins))
              (v-dec (read-mem (rn-b ins) regs))
                     (read-mem (rn-b ins) regs)) mem)      )
    (if (pre-dec-p (mode-b ins)) (v-dec (read-mem (rn-b ins) regs))
        (read-mem (rn-b ins) regs)) oracle))]
```

The original state is decomposed into three new states. As before, `operand-b` register is used as a temporary to hold the value of the register index. In state `fm9001-operand-b_1` the register-file is updated depending on the addressing mode. Finally, in state `fm9001-operand-b_2`, the `operand-b` register is loaded with either, the `operand-b` register or a value from memory depending on the addressing mode. In addition, the `b-address` register is updated.

```
[fm9001-operand-b
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-operand-b_1 regs flags mem pc-reg ins operand-a
                        (read-mem (rn-b ins) regs)  b-address oracle))]
[fm9001-operand-b_1
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-operand-b_2
    (if (pre-dec-p (mode-b ins))
        (write-mem (rn-b ins) regs (v-dec operand-b))
        (if (post-inc-p (mode-b ins))
            (write-mem (rn-b ins) regs (v-inc operand-b))  regs))
    flags mem pc-reg ins operand-a operand-b b-address oracle))]
[fm9001-operand-b_2
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-alu-operation
    regs flags mem pc-reg ins operand-a
    (if (reg-direct-p (mode-b ins)) operand-b
        (read-mem
          (if (pre-dec-p (mode-b ins))
              (v-dec operand-b) operand-b) mem)                  )
    (if (pre-dec-p (mode-b ins))
        (v-dec operand-b) operand-b) oracle))]
```

Finally, the `fm9001-alu-operation` state is serialized.

```
[fm9001-alu-operation
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-intr
    (if (and* (store-resultp (store-cc ins) flags)
              (reg-direct-p (mode-b ins)))
        (write-mem (rn-b ins)
          regs
          (bv (v-alu (c-flag flags) operand-a
                     operand-b (op-code ins))))   regs)
    (update-flags flags (set-flags ins)
                  (v-alu (c-flag flags)
                         operand-a operand-b (op-code ins)))
    (if (and* (store-resultp (store-cc ins) flags)
              (not* (reg-direct-p (mode-b ins))))
        (write-mem b-address mem
          (bv (v-alu (c-flag flags)
                     operand-a operand-b (op-code ins))))     mem)
    pc-reg ins operand-a operand-b b-address oracle))]
```

The original state is decomposed into two new states:

```
[fm9001-alu-operation
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-alu-operation_1
    (if (and* (store-resultp (store-cc ins) flags)
              (reg-direct-p (mode-b ins)))
        (write-mem (rn-b ins) regs
          (bv (v-alu (c-flag flags)
          operand-a operand-b (op-code ins))))     regs)
    flags mem pc-reg ins operand-a operand-b b-address oracle))]
[fm9001-alu-operation_1
 (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
   (fm9001-intr
    regs
    (update-flags flags (set-flags ins)
                  (v-alu (c-flag flags)
                         operand-a operand-b (op-code ins)))
    (if (and* (store-resultp (store-cc ins) flags)
              (not* (reg-direct-p (mode-b ins))))
        (write-mem b-address mem
          (bv (v-alu (c-flag flags)
          operand-a operand-b (op-code ins))))     mem)
    pc-reg ins operand-a operand-b b-address oracle))]
```

The `fm9001-alu-operation` state executes the ALU operation and updates the register-file. The second state, `fm9001-alu-operation_1` executes the ALU operation and updates the memory. The redundancy of the ALU operations is a consequence of unfolding the let bindings earlier in the derivation process. This has no consequence to the resulting implementation since the operations are merged into a single instance during structural refinement.

4.2.4 Distributing: `if`

Next, the `if` statements within each state call are distributed outside the call in order to move the decision point from architecture to control. Conditional expressions in DDD define the control abstraction of the circuit. As a result of how DDD constructs the control specification from the behavioral description, *if expressions* located outside the state call are captured in the control specification, and *if expressions* inside a state call are implemented in the architecture. The *distribute if* transformation allows for the distribution of the *if expression* from a state call to outside the call, and thus has the effect of moving the decision into the control specification.

For example, in

```
...
[fm9001-operand-a_1
 (lambda (regs ...)
   (fm9001-operand-a_2
       (if (a-immediate-p ins)
           regs
           (if (pre-dec-p (mode-a ins))
               (write-mem (rn-a ins) regs (v-dec operand-b))
               (if (post-inc-p (mode-a ins))
                   (write-mem (rn-a ins) regs (v-inc operand-b))
                   regs)))
       flags mem pc-reg ins operand-a operand-b b-address oracle)
   )] ...
```

three conditionals are nested within the call to `fm9001-operand-a_2`. The nested `if` structure determines the next value for `regs` register. In this case, the other registers within the system, such as `flags`, `mem`, `pc-reg`, etc., do not change.

Distributing the conditionals outside the function call results in four separate functional calls to `fm9001-operand-a_2` within the conditional branch structure, each updating `regs` with the appropriate value.

```
...
[fm9001-operand-a_1
 (lambda (regs ...)
   (if (a-immediate-p ins)
       (fm9001-operand-a_2
          regs
          flags mem pc-reg ins operand-a operand-b b-address oracle)
       (if (pre-dec-p (mode-a ins))
           (fm9001-operand-a_2
              (write-mem (rn-a ins) regs (v-dec operand-b))
              flags mem pc-reg ins operand-a operand-b b-address oracle)
           (if (post-inc-p (mode-a ins))
               (fm9001-operand-a_2
                  (write-mem (rn-a ins) regs (v-inc operand-b))
                  flags mem pc-reg ins operand-a operand-b b-address oracle)
               (fm9001-operand-a_2
                  regs
                  flags mem pc-reg ins operand-a operand-b b-address oracle)
               ))))] ...
```

Distributing `if` yields a textually larger specification, with the decision points moved into the control. This transformation adds approximately 1400 characters to the specification. This growth is essentially from the duplication that occurs when distributing a conditional and copying the redundant information. Simplification at later stages of the design may negate the growth encountered at this stage.

The final behavioral form is fully serialized, all `let` bindings are unfolded, all `if`'s are distributed outside recursive function calls, and the `if` function is exposed. This final behavioral form is sufficient to construct an initial structural description.

4.3 Behavior to Structure

The next phase in the derivation involves the application of the *behavior to structure* construction in order to build a structural description from the behavioral specification. The construction decomposes the behavioral specification into a selection combinator denoting control and a structural component denoting the initial architecture. The transformation is completely automatic and requires no user guidance. The resulting description is still abstract in the sense that the description does not detail the functional modules, nor reflect any constraints on the datapath.

The *behavior to structure* construction is applied to the appropriate behavioral form. This results in the selection combinator, `select` and an initial system description. An illustration of the sequential system is shown in Figure 4.5.

```
(define select
  (lambda* ((s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
            v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18
            v19 v20 v21 v22 v23 v24)
    (case s
      [fm9001-intr (if p0 v0 v1)]
      [fm9001-fetch v2]
      [fm9001-fetch_1 v3]
      [fm9001-fetch_2 v4]
      [fm9001-operand-a v5]
      [fm9001-operand-a_1 (if p1 v6 (if p2 v7 (if p3 v8 v6)))]
      [fm9001-operand-a_2 (if p1 v9 (if p4 v10 (if p2 v11 v12)))]
      [fm9001-operand-b v13]
      [fm9001-operand-b_1 (if p5 v14 (if p6 v15 v16))]
      [fm9001-operand-b_2 (if p7 (if p5 v17 v18) (if p5 v19 v20))]
      [fm9001-alu-operation (if p8 v21 v22)]
      [fm9001-alu-operation_1 (if p9 v23 v24)])))
```

The decision combinator, `select`, is a function of eleven predicates, `s`, and `p0 ... p9`, and returns one of 25 alternatives, `v0...v24`. These parameters reduce as further transformations are applied. However, this is not necessarily the goal that drives the transformation process. It is in fact the refinement of this structural description to an architecture that drives this process.

The derived structural specification consists of a system of 11 equations

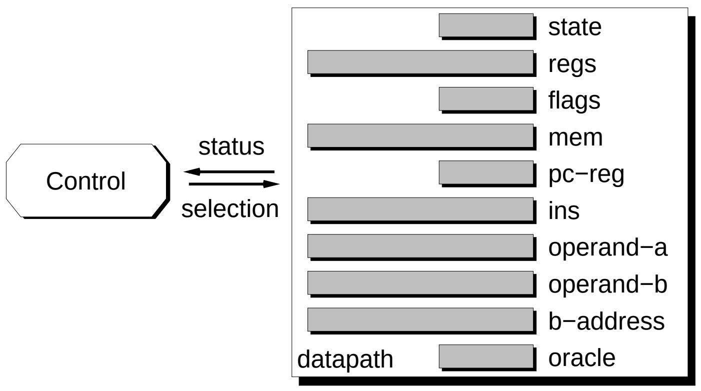


Figure 4.5: DDD-FM9001 Initial Sequential System

```
(system-letrec
 ([status    (xps state ...)]
  [state     (! fm9001-intr (select status ...))]
  [regs      (! init-regs (select status ...))]
  [flags     (! init-flags (select status ...))]
  [mem       (! init-mem (select status ...))]
  [pc-reg    (! init-pc-reg (select status ...))]
  [ins       (! ? (select status ...))]
  [operand-a (! ? (select status ...))]
  [operand-b (! ? (select status ...))]
  [b-address (! ? (select status ...))]
  [oracle    (! init-oracle (select status ...))])
 ...)
```

The first equation, `status`, encapsulates the predicates used by `select`. The remaining ten equations denote registered values and are defined by an initial value and the decision combinator, `select`, to compute the next value. The equation `state` encodes the states of the machine. The other equations correspond to the formal parameters in the behavioral specification.

The key step in the construction from behavior to structure is the abstraction of the decision combinator. This isolates the notion of control from the description while expressing the components and their connectivity. An overview of each equation is detailed below.

```
[status
 (xps state
      (nlistp oracle)
      (a-immediate-p ins)
      (pre-dec-p (mode-a ins))
      (post-inc-p (mode-a ins))
      (reg-direct-p (mode-a ins))
      (pre-dec-p (mode-b ins))
      (post-inc-p (mode-b ins))
      (reg-direct-p (mode-b ins))
      (and* (store-resultp (store-cc ins) flags)
            (reg-direct-p (mode-b ins)))
      (and* (store-resultp (store-cc ins) flags)
            (not* (reg-direct-p (mode-b ins)))))]
```

The `status` equation names the predicates in the system used by the selector in determining the next value for the system of equations. The `status` equation contains the present `state` value, corresponding to the `s` formal parameter in `select`, and the ten predicates corresponding to `p0 ... p9` in `select`.

```
[state
 (! fm9001-intr
    (select status fm9001-intr fm9001-fetch fm9001-fetch_1
            fm9001-fetch_2 fm9001-operand-a fm9001-operand-a_1
            fm9001-operand-a_2 fm9001-operand-a_2 fm9001-operand-a_2
            fm9001-operand-b fm9001-operand-b fm9001-operand-b
            fm9001-operand-b fm9001-operand-b_1 fm9001-operand-b_2
            fm9001-operand-b_2 fm9001-operand-b_2 fm9001-alu-operation
            fm9001-alu-operation fm9001-alu-operation
            fm9001-alu-operation fm9001-alu-operation_1
            fm9001-alu-operation_1 fm9001-intr fm9001-intr))]
```

The `state` equation encodes the transfer of control from each of the twelve original state definitions, `fm9001-intr`, ..., `fm9001-alu-operation_1`. The `state` equation represents the "next-state" function, where the `state` register holds the present control state and the result of the selector evaluation determines the next state.

```
[regs
 (! init-regs
    (select status . . . . (write-mem pc-reg regs (v-inc operand-b)) . .
            (write-mem (rn-a ins) regs (v-dec operand-b))
            (write-mem (rn-a ins) regs (v-inc operand-b)) . . . . .
            (write-mem (rn-b ins) regs (v-dec operand-b))
            (write-mem (rn-b ins) regs (v-inc operand-b)) . . . . .
            (write-mem (rn-b ins) regs
                       (bv (v-alu (c-flag flags)
                                  operand-a operand-b (op-code ins)))) . . .))]
```

The `regs` equation denotes the 16-word register-file. The register-file either holds its value, or is updated by a write operation. (The "." is used as an abbreviation of when the signal value remains unchanged).

```
[flags
 (! init-flags
    (select status . . . . . . . . . . . . . . . . . . . . . . .
            (update-flags flags (set-flags ins)
                          (v-alu (c-flag flags)
                                 operand-a operand-b (op-code ins)))
            (update-flags flags (set-flags ins)
                          (v-alu (c-flag flags)
                                 operand-a operand-b (op-code ins)))))]
```

The `flags` equation denotes the four flags, carry (`c`), overflow (`v`), negative (`n`), and zero (`z`). The flags either hold their value or are updated from the result of `update-flags`.

```
[mem
 (! init-mem
    (select status . . . . . . . . . . . . . . . . . . . . . . .
            (write-mem b-address mem
                       (bv (v-alu (c-flag flags)
                                  operand-a operand-b (op-code ins)))) .))]
```

The `mem` equation denotes the memory. The memory either holds its value, or is updated with the write operation. The memory address is determined by the `b-address` register, and the data is the result of the `v-alu` operation.

```
[pc-reg
 (! init-pc-reg
    (select status . (car oracle) . . . . . . . . . . . . . . . . . . . . . . .))]
```

The `pc-reg` equation determines which register in the register-file is the current program counter. It updates its value from the `oracle`.

```
[ins
 (! ins-?
    (select status . . . (read-mem operand-b mem) . . . . . . . . . . . . . . .
            . . . . . . .))]
```

The `ins` equation denotes the instruction register. The instruction equation either holds its value or is updated by a read operation on the memory.

```
[operand-a
 (! operand-a-?
    (select status . . . . . . . . . (sign-extend (a-immediate ins) thirty-two)
            operand-b (read-mem (v-dec operand-b) mem) (read-mem operand-b mem)
            . . . . . . . . . . . .))]
[operand-b
 (! operand-b-?
    (select status . . (read-mem pc-reg regs) . . (read-mem (rn-a ins) regs) .
            . . . . . . (read-mem (rn-b ins) regs) . . . . .
            (read-mem (v-dec operand-b) mem) (read-mem operand-b mem) . . . .))]
```

The `operand-a` and `operand-b` equations hold the A and B operands, respectively. In addition, the `operand-b` register is used to hold intermediate values from the program counter, register-file, and memory. This is a consequence of the design decision made during the serialization phase of the derivation (see Section 4.2.3).

```
[b-address
 (! b-address-?
    (select status . . . . . . . . . . . . . . . . . (v-dec operand-b)
            operand-b (v-dec operand-b) operand-b . . . .))]
```

The `b-address` equation denotes the address to memory for the write operation. The register either holds its value, or is updated with a value from either the `operand-b` register or the result of the `v-dec` operation.

```
[oracle
 (! init-oracle
    (select status . (cdr oracle) . . . . . . . . . . . . . . . . . . . . . .
.))])
```

The `oracle` equation polls values from the outside determining which register is going to be the program counter and if the external termination condition has been satisfied.

4.4 Factoring `mem`, `regs`, and `v-alu`

The initial system description describes a structural specification but is still abstract in terms of an implementation. Further refinement of the architecture is necessary to move towards a more physically meaningful description in which complex data types are isolated.

In this phase of the derivation, DDD's abstraction mechanisms, *signal factorization* and *general factorization*, are used to transform the design into a reasonable description of communicating functional components. The functional components are the memory, `mem`, the register-file, `regs`, and the arithmetic operations `v-alu`, `v-inc`, and `v-dec`.

Factorization serves two purposes in this derivation exercise. First, *signal factorization* is applied to the `mem` and `regs` signals in order to isolate them into a subsystem for assembly into hardware. Second, *general factorization* is applied to the `v-alu`, `v-inc`, and `v-dec` operations in order to isolate these operations for verification.

4.4.1 Factoring `mem`

Signal factorization is applied to the `mem` equation in order to isolate the memory from the description. The result of the memory factorization is the synthesis of an abstract component specification that encapsulates the memory signal as a communicating process, and the derivation of four equations to communicate with the factored component.

The expression below isolates the key elements of the memory factorization before *signal factorization* is applied.

```
(system-letrec
 (...
  [mem
    (! init-mem
      (select status . . . . . . . . . . . . . . . . . . . . . . .
              (write-mem b-address mem
                         (bv (v-alu (c-flag flags)
                                    operand-a operand-b
                                    (op-code ins)))) .))]
  [ins
   (! ?
      (select status . . . (read-mem operand-b mem) . . . . . . . . . .
              . . . . . . . . . . .))]
  [operand-a
   (! ?
      (select status . . . . . . . . .
              (sign-extend (a-immediate ins) thirty-two) operand-b
              (read-mem (v-dec operand-b) mem) (read-mem operand-b mem)
              . . . . . . . . . . . . .))]
  [operand-b
   (! ?
      (select status . . (read-mem pc-reg regs) . .
              (read-mem (rn-a ins) regs) . . . . . . .
              (read-mem (rn-b ins) regs) . . . . .
              (read-mem (v-dec operand-b) mem)
              (read-mem operand-b mem) . . . .))]
  ...) ...)
```

Highlighted is the equation for memory, `mem`, which includes the write operation, `(write-mem ... mem)`. Also highlighted are the five read operations on the memory, `(read-mem ... mem)`. The goal is to encapsulate the memory operations into a single abstract component.

The result of applying *signal factorization* on the `mem` signal yields the system

```
(system-letrec
 (...
  [ins
   (! ?
      (select status . . . mem-out . . . . . . . . . . . . . . . . .
              . . . .))]
  [operand-a
   (! ?
      (select status . . . . . . . . .
              (sign-extend (a-immediate ins) thirty-two) operand-b
              mem-out mem-out . . . . . . . . . . . .))]
  [operand-b
   (! ?
      (select status . . (read-mem pc-reg regs) . .
              (read-mem (rn-a ins) regs) . . . . . . .
              (read-mem (rn-b ins) regs) . . . . .
              mem-out mem-out . . . .))]
  [mem-out (abs-mem init-mem mem-inst mem-addr mem-data)]
  [mem-inst
   (select status # # # mem-read-mem # # # # # # # mem-read-mem
           mem-read-mem # # # # # # mem-read-mem mem-read-mem # #
           mem-write-mem #)]
  [mem-addr
   (select status ? ? ? operand-b ? ? ? ? ? ? ? ? (v-dec operand-b)
           operand-b ? ? ? ? ? ? (v-dec operand-b) operand-b ? ?
           b-address ?)]
  [mem-data
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           (bv (v-alu (c-flag flags)
                      operand-a operand-b (op-code ins))) ?)]
  ...) ...)
```

The four equations derived are memory instruction, `mem-inst`, memory address, `mem-addr`, memory data, `mem-data`, and memory output, `mem-out`. Each occurrence of the read operation, `(read-mem ... mem)`, is replaced with the output of the abstract component. DDD generates a "no operation" signal, `#`, in the memory instruction, and appropriate "don't care" signals, `?`, in the data and address equations, corresponding to when the memory equation remains unchanged.

The synthesized abstract component specification is

```
(define abs-mem
  (lambda (*mem* inst v0 v1)
    (let ([constructor
           (lambda (inst object v0 v1)
             (case inst
               [# object]
               [mem-read-mem object]
               [mem-write-mem (write-mem v0 object v1)]))]
          [probe
           (lambda (inst object v0)
             (case inst
               [#             (read-mem v0 object)]
               [mem-write-mem (read-mem v0 object)]
               [mem-read-mem  (read-mem v0 object)]))])
      (system-letrec
       ([object (! *mem* ((stream constructor) inst object v0 v1))]
        [probe ((stream probe) inst object v0)])
       probe))))
```

The abstract component, abs-mem, takes an initial memory, an instruction, an address, and data, and returns a value depending on the current address. The component maintains the state of the memory internally.

4.4.2 Factoring regs

Signal factorization is applied to the regs equation in order to factor the register-file from the description. The transformation is identical to the memory factorization except that it generates signals to communicate with the register-file. This is expected since mem and regs are defined by the same functional model of memory.

The result of the register-file factorization is the synthesis of the abstract component specification that encapsulates the register-file as a communicating process, and the derivation of four equations to communicate with the factored component.

The expression below isolates the key elements of the register-file factorization before *signal factorization* is applied. Highlighted is the defining equation for the register-file, regs, which includes the write operation, (write-mem ... regs). Also highlighted, are the read operations on the register-file, (read-mem ... regs).

```
(system-letrec
 (...
  [regs
     (! init-regs
        (select status . . . .
                (write-mem pc-reg regs (v-inc operand-b)) . .
                (write-mem (rn-a ins) regs (v-dec operand-b))
                (write-mem (rn-a ins) regs (v-inc operand-b)) . . .
                . . (write-mem (rn-b ins) regs (v-dec operand-b))
                (write-mem (rn-b ins) regs (v-inc operand-b)) . . .
                . . (write-mem (rn-b ins) regs
                                     (bv (v-alu (c-flag flags)
                                                operand-a operand-b
                                                (op-code ins)))) . . .))]
  [operand-b
   (! ?
      (select status . . (read-mem pc-reg regs) . .
              (read-mem (rn-a ins) regs) . . . . . . .
              (read-mem (rn-b ins) regs) . . . . . mem-out
              mem-out . . . .))]
      ...) ...)
```

The result of applying *signal factorization* on the regs signal yields the system

```
(system-letrec
 (...
  [operand-b
   (! ?
      (select status . . regs-out . . regs-out . . . . . . .
              regs-out . . . . . mem-out mem-out . . . .))]
  [regs-out (abs-regs init-regs regs-inst regs-addr regs-data)]
  [regs-inst
   (select status # # regs-read-mem # regs-write-mem
           regs-read-mem # regs-write-mem regs-write-mem #
           # # # regs-read-mem regs-write-mem
           regs-write-mem # # # # #
           regs-write-mem # # #)]
  [regs-addr
   (select status ? ? pc-reg ? pc-reg (rn-a ins) ? (rn-a ins) (rn-a ins)
           ? ? ? ? (rn-b ins) (rn-b ins) (rn-b ins) ? ? ? ? ? (rn-b ins)
           ? ? ?)]
  [regs-data
   (select status ? ? ? ? (v-inc operand-b) ? ? (v-dec operand-b)
           (v-inc operand-b) ? ? ? ? ? (v-dec operand-b)
           (v-inc operand-b) ? ? ? ? ?
           (bv (v-alu (c-flag flags)
                      operand-a operand-b (op-code ins))) ? ? ?)]
      ...) ...)
```

The four equations derived are the register-file instruction, `regs-inst`, the register-file address, `regs-addr`, the register-file data, `regs-data`, and the register-file output, `regs-out`. Each occurrence of the read operation, `(read-mem ... regs)`, is replaced with the output of the abstract component.

The synthesized abstract component specification is

```
(define abs-regs
   (lambda (*regs* inst v0 v1)
      (let ([constructor
              (lambda (inst object v0 v1)
                (case inst
                    [# object]
                    [regs-read-mem object]
                    [regs-write-mem (write-mem v0 object v1)]))]
             [probe
              (lambda (inst object v0)
                (case inst
                    [#             (read-mem v0 object)]
                    [regs-write-mem (read-mem v0 object)]
                    [regs-read-mem  (read-mem v0 object)]))])
         (system-letrec
            ([object (! *regs* ((stream constructor) inst object v0 v1))]
             [probe ((stream probe) inst object v0)])
          probe))))
```

The abstract component, `abs-regs`, takes an initial register-file, an instruction, an address, and data, and returns a value depending on the current address. The component maintains the state of the register-file internally.

4.4.3 Factoring `v-alu`

General factorization is applied to the `v-alu` operation in order to isolate the ALU operation. The factorization of components into abstract modules provides a mechanism by which multiple occurrences of operations are merged into a single component and individual components can be verified. Once factored, the `v-alu` operation is replaced by an efficient verified implementation. Details of the verification are found in Section 4.5.

The result of the ALU factorization is the synthesis of an abstract component, `abs-v-alu`. In addition, six equations are derived to communicate with the factored component. The expression below isolates the key elements of the ALU factorization before *general factorization* is applied. Highlighted are the `v-alu` operations. There are two calls to `v-alu` in the `mem-regs-data` equation, and two calls in the `update-flags-out` equation.

```
(system-letrec
 (...
  [mem-regs-data
   (select status ? ? data-in data-in (v-inc operand-b) data-in ?
           (v-dec operand-b) (v-inc operand-b) ? ? data-in data-in data-in
           (v-dec operand-b) (v-inc operand-b) ? ? ? data-in data-in
           (bv (v-alu (c-flag flags) operand-a operand-b (op-code ins)) )
           ? (bv (v-alu (c-flag flags) operand-a operand-b (op-code ins)) )
           ?)]
  [update-flags-out
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           (update-flags flags (set-flags ins)
                         (v-alu (c-flag flags) operand-a operand-b
                            (op-code ins))                            )
           (update-flags flags (set-flags ins)
                         (v-alu (c-flag flags) operand-a operand-b
                            (op-code ins))                            ))]
  ...) ...)
```

The result of applying *general factorization* on the `v-alu` signal yields the abstract component:

```
(define abs-v-alu
  (lambda (inst v0 v1 v2 v3)
    (let ([constructor
           (lambda (inst v0 v1 v2 v3)
             (case inst
               [# ?]
               [v-alu-out-v-alu (v-alu v0 v1 v2 v3)]))])
      (system-letrec
       ([object ((stream constructor) inst v0 v1 v2 v3)])
       object))))
```

The abstract component, `abs-v-alu`, takes an instruction, a carry-in flag, two operands, a and b, and an opcode, and returns the result of applying the command `(v-alu ...)` to the arguments. Unlike signal factorization, general factorization encapsulates a set of combinational operations where there is no state to maintain.

The transformed system description is

```
(system-letrec
 (...
  [mem-regs-data
   (select status ? ? data-in data-in (v-inc operand-b) data-in ?
           (v-dec operand-b) (v-inc operand-b) ? ? data-in data-in data-in
           (v-dec operand-b) (v-inc operand-b) ? ? ? data-in data-in
           (bv v-alu-out) ? (bv v-alu-out) ?)]
  [update-flags-out
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           (update-flags flags (set-flags ins) v-alu-out)
           (update-flags flags (set-flags ins) v-alu-out))]
  [v-alu-out
     (abs-v-alu v-alu-out-inst v-alu-out-carryin
                v-alu-out-opa v-alu-out-opb v-alu-out-opcode)]
  [v-alu-out-inst
   (select status # # # # # # # # # # # # # # # # # # # # #
           v-alu-out-v-alu # v-alu-out-v-alu v-alu-out-v-alu)]
  [v-alu-out-carryin
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           (c-flag flags) ? (c-flag flags) (c-flag flags))]
  [v-alu-out-opa
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           operand-a ? operand-a operand-a)]
  [v-alu-out-opb
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           operand-b ? operand-b operand-b)]
  [v-alu-out-opcode
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           (op-code ins) ? (op-code ins) (op-code ins))]
...) ...)
```

The six equations derived are the instruction equation, `v-alu-out-inst`, the carry-in equation, `v-alu-out-carryin`, the operand a equation, `v-alu-out-opa`, the operand b equation, `v-alu-out-opb`, the opcode equation, `v-alu-out-opcode`, and the output equation, `v-alu-out`. Each occurrence of the ALU operation, `(v-alu ...)`, is replaced with the output of the abstract component.

As a further refinement to the system description, *expand stream* is applied to the `v-alu-out-opa` and `v-alu-out-opa` equations in order to eliminate them from the description. The selectors are simplified since each of the selector alternatives result in the singleton `operand-a` and `operand-b`, respectively. The effect is the elimination of the named equation and the replacement of each occurrence of that equation with its definition. This works out nicely since the two equations were merely renaming the existing `operand-a` and `operand-b` equations.

```
(system-letrec
 (...
  [mem-regs-data
   (select status ? ? data-in data-in (v-inc operand-b) data-in ?
           (v-dec operand-b) (v-inc operand-b) ? ? data-in data-in data-in
           (v-dec operand-b) (v-inc operand-b) ? ? ? data-in data-in
           (bv v-alu-out) ? (bv v-alu-out) ?)]
  [update-flags-out
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           (update-flags flags (set-flags ins) v-alu-out)
           (update-flags flags (set-flags ins) v-alu-out))]
  [v-alu-out
   (abs-v-alu v-alu-out-inst v-alu-out-carryin
              operand-a operand-b v-alu-out-opcode)]
  [v-alu-out-inst
   (select status # # # # # # # # # # # # # # # # # # # # #
           v-alu-out-v-alu # v-alu-out-v-alu v-alu-out-v-alu)]
  [v-alu-out-carryin
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           (c-flag flags) ? (c-flag flags) (c-flag flags))]
  [v-alu-out-opcode
   (select status ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
           (op-code ins) ? (op-code ins) (op-code ins))]
  ...) ...)
```

4.4.4 Factoring v-inc and v-dec

General factorization is applied to the incrementor, `v-inc`, and decrementor,`v-dec`, operations in order to isolate them for verification. The result of factoring the `v-inc` and `v-dec` operations is the synthesis of two abstract components for each operation, `abs-v-inc` and `abs-v-dec`, which encapsulates the `v-inc` and `v-dec` operations, respectively. In addition, three equations for each operation are derived within the system description to communicate with the factored components.

The expression below isolates the key elements of the incrementor and decrementor factorization. Highlighted are the `v-inc` and `v-dec` operations. The factorization is done sequentially, applying it to the `v-inc` operation first and then the `v-dec` operation, although the expression below highlights both operations.

```
(system-letrec
 (...
  [b-address
   (! ?
      (select status . . . . . . . . . . . . . . . . .
              (v-dec operand-b) operand-b (v-dec operand-b)
              operand-b . . . .))]
  [mem-addr
   (select status ? ? ? operand-b ? ? ? ? ? ? ? (v-dec operand-b)
           operand-b ? ? ? ? ? ? (v-dec operand-b) operand-b ? ?
           b-address ?)]
  [mem-regs-data
   (select status ? ? data-in data-in (v-inc operand-b) data-in ?
           (v-dec operand-b) (v-inc operand-b) ? ? data-in data-in data-in
           (v-dec operand-b) (v-inc operand-b) ? ? ? data-in data-in
           (bv v-alu-out)
           ? (bv v-alu-out)
           ?)]
  ...) ...)
```

The result of applying *general factorization* on the `v-inc` and `v-dec` signal yields the abstract components:

```
(define abs-v-inc
  (lambda (inst v0)
    (let ([constructor
           (lambda (inst v0)
             (case inst
               [# ?]
               [v-inc-out-v-inc (v-inc v0)]))])
      (system-letrec
       ((object ((stream constructor) inst v0)))
       object))))
```

and

```
(define abs-v-dec
  (lambda (inst v0)
    (let ([constructor
           (lambda (inst v0)
             (case inst
               [# ?]
               [v-dec-out-v-dec (v-dec v0)]))])
      (system-letrec
       ((object ((stream constructor) inst v0)))
       object)))).
```

The abstract components `abs-v-inc` and `abs-v-dec` each takes an instruction and data signal, and return the result of applying `(v-inc ...)` and `(v-dec ...)` to the appropriate arguments, respectively.

The transformed system description is

```
(system-letrec
 (...
  [b-address
   (! b-address-?
      (select status . . . . . . . . . . . . . . . . . [v-dec-out]
              operand-b [v-dec-out] operand-b . . . .))]
  [mem-addr
   (select status ? ? ? operand-b ? ? ? ? ? ? ? [v-dec-out] operand-b
           ? ? ? ? ? ? [v-dec-out] operand-b ? ? b-address ?)]
  [mem-regs-data
   (select status ? ? data-in data-in [v-inc-out] data-in ? [v-dec-out]
           [v-inc-out] ? ? data-in data-in data-in [v-dec-out] [v-inc-out]
           ? ? ? data-in data-in (bv v-alu-out) ? (bv v-alu-out) ?)]
  [v-inc-out (abs-v-inc v-inc-out-inst v-inc-out-v0)]
  [v-inc-out-inst
   (select status # # # # v-inc-out-v-inc # # # v-inc-out-v-inc
           # # # # # # v-inc-out-v-inc # # # # # # # # #)]
  [v-inc-out-v0
   (select status ? ? ? ? operand-b ? ? ? operand-b ? ? ? ? ? ?
           operand-b ? ? ? ? ? ? ? ? ?)]
  [v-dec-out (abs-v-dec v-dec-out-inst v-dec-out-v0)]
  [v-dec-out-inst
   (select status # # # # # # # v-dec-out-v-dec # # #
           v-dec-out-v-dec # # v-dec-out-v-dec # # v-dec-out-v-dec
           # v-dec-out-v-dec # # # # #)]
  [v-dec-out-v0
   (select status ? ? ? ? ? ? ? operand-b ? ? ? operand-b ? ?
           operand-b ? ? operand-b ? operand-b ? ? ? ? ?)]
  ...) ...)
```

For the v-inc factorization, the derived equations are the instruction equation, v-inc-out-inst, the data equation, v-inc-out-v0, and the incrementor output equation, v-inc-out. For the v-dec factorization, the derived equations are the instruction equation, v-dec-out-inst, the data equation, v-dec-out-v0, and the decrementor output equation, v-dec-out. Each occurrence of (v-inc ...) and (v-dec ...) are replaced with the output of their respective abstract components.

As is the case in the derived equations in the v-inc factorization, the incrementor and decrementor factorization generate redundant signals. The v-inc-out-v0 and v-dec-out-v0 equations in the expression above reduce to the operand-b equation.

Expand stream is applied to the redundant equations in order to eliminate them. The effect on the system description is shown below. Both equations are eliminated and their occurrence is replaced with the operand-b signal.

```
(system-letrec
 (...
  [b-address
   (! b-address-?
      (select status . . . . . . . . . . . . . . . . . . v-dec-out
              operand-b v-dec-out operand-b . . . .))]
  [mem-addr
   (select status ? ? ? operand-b ? ? ? ? ? ? ? v-dec-out operand-b
           ? ? ? ? ? ? v-dec-out operand-b ? ? b-address ?)]
  [mem-regs-data
   (select status ? ? data-in data-in v-inc-out data-in ? v-dec-out
           v-inc-out ? ? data-in data-in data-in v-dec-out v-inc-out
           ? ? ? data-in data-in (bv v-alu-out) ? (bv v-alu-out) ?)]
  [v-inc-out (abs-v-inc v-inc-out-inst operand-b)]
  [v-inc-out-inst
   (select status # # # # v-inc-out-v-inc # # # v-inc-out-v-inc
           # # # # # # v-inc-out-v-inc # # # # # # # # #)]
  [v-dec-out (abs-v-dec v-dec-out-inst operand-b)]
  [v-dec-out-inst
   (select status # # # # # # # v-dec-out-v-dec # # #
           v-dec-out-v-dec # # v-dec-out-v-dec # # v-dec-out-v-dec
           # v-dec-out-v-dec # # # # #)]
...) ...)
```

4.5 Verifying v-alu, v-inc and v-dec .

The DDD-FM9001 design is targeted to the ACTEL FPGA architecture. This architecture is a matrix of *logic modules*, each implementing a four-input multiplexor. Verification of the arithmetic modules is necessary to replace the ripple-carry adder in Hunt's specification with an efficient implementation designed specifically for the target technology. The verification of the arithmetic modules in the DDD-FM9001 reduces to verifying a boolean term, derived from the FM9001 specification, against a hand-designed multiplexor implementation. It is 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), is defined. This function is used as the basis for implementing the DDD-FM9001 ALU. The DDD-FM9001 ALU implementation is based on a carry-look-ahead adder that optimizes the use of ACTEL logic modules and gate delays. 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.

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

Next, symbolic evaluation is applied to both the Nqthm specification and the multiplexor implementation of the ALU. This method is similar to symbolic evaluation that is discussed by Darringer [25], in which the base operators are extended to return symbolic values and symbols are introduced as input values in place of real data objects, 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 results in the construction of boolean expressions denoting their function. Ordered binary decision diagrams (OBDDs) are then constructed from the resulting boolean expressions and verified for equivalence. The method is illustrated in the following example.

4.5.1 Boolean Verification: An Example

Consider the example of a 1-bit full adder. The boolean equations defining the full-adder are given below where a and b are the bits to be added and c is the carry input. Figure 4.6 illustrates the OBDD for sum and $carry$.

$$\begin{aligned} sum &= a \oplus b \oplus c & (4.1) \\ carry &= ab + c(a+b) & (4.2) \end{aligned}$$

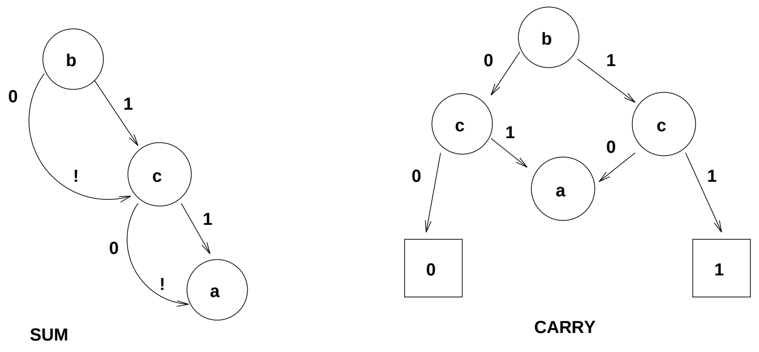


Figure 4.6: OBDDs of sum and $carry$

The full-adder equations may be defined by the following function definitions, *sum_spec* and *carry_spec*. The functions *sum_spec* and *carry_spec* are specifications that implement equations (4.1) and (4.2) respectively.

$$\begin{aligned} sum\_spec(c,a,b) &= xor(a, xor(b,c)) \\ carry\_spec(c,a,b) &= or(and(a,b), and(c, or(a,b))) \end{aligned}$$

Consider an implementation of a full-adder consisting of four 4-input multiplexors. Both a schematic, (Figure 4.7), and a function definition of the implementation is shown below.

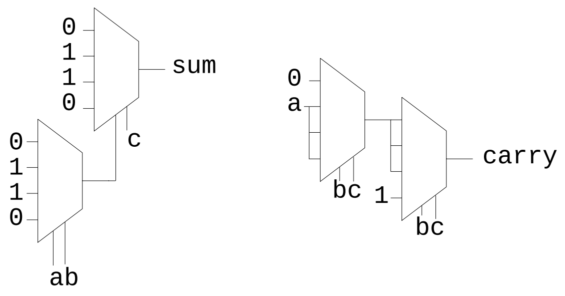


Figure 4.7: Multiplexor Implementation: 1-Bit Full-Adder

$$\begin{aligned}
sum\_imp(c,a,b) &= mux(0,1,1,0,mux(0,1,1,0,a,b),c)\\
carry\_imp(c,a,b) &= mux(mux(0,a,a,a,b,c),\\
&\qquad mux(0,a,a,a,b,c),\\
&\qquad mux(0,a,a,a,b,c),1,b,c)
\end{aligned}$$

The goal is to verify that the 4-input multiplexor implementation, *sum_imp* and *carry_imp*, implements the equations specified by the functions *sum_spec* and *carry_spec*. The next step is to symbolically evaluate *sum_spec*, *carry_spec*, *sum_imp*, and *carry_imp* to construct boolean expressions denoting their respective definitions.

```
sum_spec = (xor a (xor b c))
carry_spec = (or (and a b) (and c (or a b)))
```

```
sum_imp = (or (and (not (or (and (not a) (and (not b) 0)) (or (and (not a)
          (and b 1)) (or (and a (and (not b) 1)) (and a (and b 0))))))
          (and (not c) 0)) (or (and (not (or (and (not a) (and (not b) 0))
          (or (and (not a) (and b 1)) (or (and a (and (not b) 1))
          (and a (and b 0)))))) (and c 1)) (or (and (or (and (not a)
          (and (not b) 0)) (or (and (not a) (and b 1)) (or (and a (and
          (not b) 1)) (and a (and b 0))))) (and (not c) 1)) (and (or (and
          (not a) (and (not b) 0)) (or (and (not a) (and b 1)) (or (and a
          (and (not b) 1)) (and a (and b 0))))) (and c 0)))))
```

```
carry_imp = (or (and (not b) (and (not c) (or (and (not b) (and (not c) 0))
            (or (and (not b) (and c a)) (or (and b (and (not c) a))
            (and b (and c a))))))) (or (and (not b) (and c (or (and
            (not b) (and (not c) 0)) (or (and (not b) (and c a)) (or
            (and b (and (not c) a)) (and b (and c a))))))) (or (and b
            (and (not c) (or (and (not b) (and (not c) 0)) (or (and
            (not b) (and c a)) (or (and b (and (not c) a)) (and b
            (and c a))))))) (and b (and c 1)))))
```

Finally, the task simplifies to the verification of two boolean systems. The results of symbolically evaluating the specification and implementation are then passed through a boolean verifier.

$$\begin{aligned} \mathit{sum\_spec} &= \mathit{sum\_imp} \\ \mathit{carry\_spec} &= \mathit{carry\_imp} \end{aligned}$$

4.5.2 Verifying `v-alu`

The top-level specification of the FM9001 ALU is

```
(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))))).
```

The ALU specification, `v-alu`, returns a bit-vector, `(cvzbv)`, consisting of four components: the carry-out `c`, overflow bit `v`, zero bit `z`, and the result bit-vector `bv`. The function takes a 4-bit op-code `op`, two 32-bit A and B registers `a` and `b`, and a carry-in bit `c`. The ACTEL mux implementation is verified with respect to boolean terms constructed from each branch of `v-alu`.

Consider the addition operation corresponding to the `v-alu` opcode, `0011`, highlighted in the definition above. Given the definition of `cvzbv-v-adder`,

```
(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)))
```

where

```
(defn v-adder-output (c a b)
  (firstn (length a) (v-adder c a b)))
```

is defined by a recursive definition of a ripple-carry adder, `v-adder`

```
(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))))).
```

Symbolically evaluating `(v-adder f a b)` yields a bit-vector of boolean terms for the ALU output, `v-alu-out_0..31`

```
v-alu-out_0  = (b-xor (b-xor f a0) b0)
v-alu-out_1  = (b-xor (b-xor c0 a1) b1)
   ...
v-alu-out_31 = (b-xor (b-xor c30 a31) b31)
```

and a bit-vector for carry, `c0..c31`

```
c0 = (b-or (b-and a0 b0)
           (b-or (b-and a0 f) (b-and b0 f)))
c1 = (b-or (b-and a1 b1)
           (b-or (b-and a1 c0) (b-and b1 c0)))
...
c31 =  (b-or (b-and a31 b31)
             (b-or (b-and a31 c30) (b-and b31 c30))).
```

The result is then passed through a boolean verifier to establish equivalence with the ACTEL mux implementation.

4.5.3 Verifying `v-inc` and `v-dec`

The incrementor and decrementor were verified using the same method.

```
(defn v-inc (x)
  (v-adder-output t x (nat-to-v 0 (length x))))
(defn v-dec (x)
  (v-subtracter-output t (nat-to-v 0 (length x)) x))
(defn v-subtracter-output (c a b)
  (v-adder-output (b-not c) (v-not a) b))
```

4.6 Deriving the Next-State and Command Generator

The next stage of the derivation is to derive the command and next-state generators, that represent the control for the DDD-FM9001. The command generator issues command codes to the datapath to signal which register transfer to occur, while the state generator computes the next-state function. The command generator's computation depends on the status predicates that includes the current state.

4.6.1 The Next-State Generator

The state generator computes the next-state function for the DDD-FM9001. *Simplification* is applied to the selector, `select`, with respect to `state`.

Highlighting the `state` equation in

```
(system-letrec
 (...
  [state
    (! fm9001-intr
       (select status fm9001-intr fm9001-fetch fm9001-fetch_1 fm9001-fetch_2
               fm9001-operand-a fm9001-operand-a_1 fm9001-operand-a_2
               fm9001-operand-a_2 fm9001-operand-a_2 fm9001-operand-b
               fm9001-operand-b fm9001-operand-b fm9001-operand-b
               fm9001-operand-b_1 fm9001-operand-b_2 fm9001-operand-b_2
               fm9001-operand-b_2 fm9001-alu-operation fm9001-alu-operation
               fm9001-alu-operation fm9001-alu-operation fm9001-alu-operation_1
               fm9001-alu-operation_1 fm9001-intr fm9001-intr))]
      ...) ...)
```

```
(define select
  (lambda* ((s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9) v0 v1 v2 v3 v4 v5 v6 v7 v8 v9
           v10 v11 v12 v13 v14 v15 v16 v17 v18 v19 v20 v21 v22 v23 v24)
    (case s
      [fm9001-intr (if p0 v0 v1)]
      [fm9001-fetch v2]
      [fm9001-fetch_1 v3]
      [fm9001-fetch_2 v4]
      [fm9001-operand-a v5]
      [fm9001-operand-a_1 (if p1 v6 (if p2 v7 (if p3 v8 v6)))]
      [fm9001-operand-a_2 (if p1 v9 (if p4 v10 (if p2 v11 v12)))]
      [fm9001-operand-b v13]
      [fm9001-operand-b_1 (if p5 v14 (if p6 v15 v16))]
      [fm9001-operand-b_2 (if p7 (if p5 v17 v18) (if p5 v19 v20))]
      [fm9001-alu-operation (if p8 v21 v22)]
      [fm9001-alu-operation_1 (if p9 v23 v24)])))
```

Simplification of `select` with respect to `state` returns the `state` equation with the updated expression. What is returned is the modified system expression

```
(system-letrec
 (...
  [state (! fm9001-intr (next-state state p0))]
   ...) ...)
```

and the definition for the `next-state` function

```
(define next-state
  (lambda (s p0)
    (case s
      [fm9001-intr (if p0 fm9001-intr fm9001-fetch)]
      [fm9001-fetch fm9001-fetch_1]
      [fm9001-fetch_1 fm9001-fetch_2]
      [fm9001-fetch_2 fm9001-operand-a]
      [fm9001-operand-a fm9001-operand-a_1]
      [fm9001-operand-a_1 fm9001-operand-a_2]
      [fm9001-operand-a_2 fm9001-operand-b]
      [fm9001-operand-b fm9001-operand-b_1]
      [fm9001-operand-b_1 fm9001-operand-b_2]
      [fm9001-operand-b_2 fm9001-alu-operation]
      [fm9001-alu-operation fm9001-alu-operation_1]
      [fm9001-alu-operation_1 fm9001-intr])))
```

In `next-state`, each of the original parameter v's in `select`, are instantiated with the symbolic constants `fm9001-intr`, etc... The function `next-state` now computes the same function as `select` applied to the `state` equation, however, it is defined in a specialized selector.

4.6.2 The Command Generator

The command generator is derived from the function `select`. The command generator is a function from status predicates to commands. *Simplification* is applied to the `select` function with respect to a set of assignments to the command codes. These assignments are automatically generated by DDD. This is a non-trivial state assignment problem that can be manually guided. The bindings are simple sequential assignments of values to command codes. For example, the command code, `v0`, is interpreted as command code 0. These assignments are automatically generated and map symbolic values, `v0,...,v22` to binary values.

Given the selector, `select`, and a set of assignments for each of the command codes (Figure 4.8)

```
(define select
  (lambda* ((s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9) v0 v1 v2 v3 v4 v5 v6 v7 v8 v9
           v10 v11 v12 v13 v14 v15 v16 v17 v18 v19 v20 v21 v22)
    (case s
      [fm9001-intr (if p0 v0 v1)]
      [fm9001-fetch v2]
      [fm9001-fetch_1 v3]
      [fm9001-fetch_2 v4]
      [fm9001-operand-a v5]
      [fm9001-operand-a_1 (if p1 v6 (if p2 v7 (if p3 v8 v6)))]
      [fm9001-operand-a_2 (if p1 v9 (if p4 v10 (if p2 v11 v12)))]
      [fm9001-operand-b v13]
      [fm9001-operand-b_1 (if p5 v14 (if p6 v15 v6))]
      [fm9001-operand-b_2 (if p7 (if p5 v16 v17) (if p5 v18 v19))]
      [fm9001-alu-operation (if p8 v20 v6)]
      [fm9001-alu-operation_1 (if p9 v21 v22)])))
```

cmd	value	cmd	value	cmd	value	cmd	value
v0	0	v6	6	v12	12	v18	18
v1	1	v7	7	v13	13	v19	19
v2	2	v8	8	v14	14	v20	20
v3	3	v9	9	v15	15	v21	21
v4	4	v10	10	v16	16	v22	22
v5	5	v11	11	v17	17		

Figure 4.8: Command Code Assignments

The result of applying simplification is the command generator:

```
(define cmd
  (lambda (s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
    (case s
      [fm9001-intr (if p0 v0 v1)]
      [fm9001-fetch v2]
      [fm9001-fetch_1 v3]
      [fm9001-fetch_2 v4]
      [fm9001-operand-a v5]
      [fm9001-operand-a_1 (if p1 v6 (if p2 v7 (if p3 v8 v6)))]
      [fm9001-operand-a_2 (if p1 v9 (if p4 v10 (if p2 v11 v12)))]
      [fm9001-operand-b v13]
      [fm9001-operand-b_1 (if p5 v14 (if p6 v15 v6))]
      [fm9001-operand-b_2 (if p7 (if p5 v16 v17) (if p5 v18 v19))]
      [fm9001-alu-operation (if p8 v20 v6)]
      [fm9001-alu-operation_1 (if p9 v21 v22)])))
```

In `cmd`, values `v0,...,v22` are now interpreted as command encodings rather than parameters.

4.7 Projection

At this stage in the derivation, the structural specification has been refined to an architecture. Complex objects such as the memory and register-file have been factored and the arithmetic components have been verified with respect to an implementation. However, the architecture is still defined over an abstract basis.

In the next phase of the derivation, DDD's rewriting mechanism, *projection*, is used to map the architecture to a binary basis. An essential component of *projection* is a set of mappings from symbolic entries defined over the abstract basis to the target basis. The binary representations used in the derivation of the DDD-FM9001 are either generated automatically by DDD or derived from the FM9001 specification. *Projection* is applied to the command generator, next-state generator and the datapath.

4.7.1 Projecting the Command Generator

Projecting `cmd` with the representations in figure 4.9 derives five separate command generators, one for each command bit. Definitions for each bit are returned. Projecting:

```
(define cmd
  (lambda (s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
    (case s
      [fm9001-intr (if p0 v0 v1)]
      [fm9001-fetch v2]
      [fm9001-fetch_1 v3]
      [fm9001-fetch_2 v4]
      [fm9001-operand-a v5]
      [fm9001-operand-a_1 (if p1 v6 (if p2 v7 (if p3 v8 v6)))]
      [fm9001-operand-a_2 (if p1 v9 (if p4 v10 (if p2 v11 v12)))]
      [fm9001-operand-b v13]
      [fm9001-operand-b_1 (if p5 v14 (if p6 v15 v6))]
      [fm9001-operand-b_2 (if p7 (if p5 v16 v17) (if p5 v18 v19))]
      [fm9001-alu-operation (if p8 v20 v6)]
      [fm9001-alu-operation_1 (if p9 v21 v22)])))
```

returns

```
(define cmd_0
  (lambda (s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
    (case s
      [fm9001-intr (if p0 0 1)]
      [fm9001-fetch 0]
      [fm9001-fetch_1 1]
      [fm9001-fetch_2 0]
      [fm9001-operand-a 1]
      [fm9001-operand-a_1 (if p1 0 (if p2 1 (if p3 0 0)))]
      [fm9001-operand-a_2 (if p1 1 (if p4 0 (if p2 1 0)))]
      [fm9001-operand-b 1]
      [fm9001-operand-b_1 (if p5 0 (if p6 1 0))]
      [fm9001-operand-b_2 (if p7 (if p5 0 1) (if p5 0 1))]
      [fm9001-alu-operation (if p8 0 0)]
      [fm9001-alu-operation_1 (if p9 1 0)])))
```

```
(define cmd_1
  (lambda (s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
    (case s
      [fm9001-intr (if p0 0 0)]
      [fm9001-fetch 1]
      [fm9001-fetch_1 1]
      [fm9001-fetch_2 0]
      [fm9001-operand-a 0]
      [fm9001-operand-a_1 (if p1 1 (if p2 1 (if p3 0 1)))]
      [fm9001-operand-a_2 (if p1 0 (if p4 1 (if p2 1 0)))]
      [fm9001-operand-b 0]
      [fm9001-operand-b_1 (if p5 1 (if p6 1 1))]
      [fm9001-operand-b_2 (if p7 (if p5 0 0) (if p5 1 1))]
      [fm9001-alu-operation (if p8 0 1)]
      [fm9001-alu-operation_1 (if p9 0 1)])))
```

```
(define cmd_2
  (lambda (s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
    (case s
      [fm9001-intr (if p0 0 0)]
      [fm9001-fetch 0]
      [fm9001-fetch_1 0]
      [fm9001-fetch_2 1]
      [fm9001-operand-a 1]
      [fm9001-operand-a_1 (if p1 1 (if p2 1 (if p3 0 1)))]
      [fm9001-operand-a_2 (if p1 0 (if p4 0 (if p2 0 1)))]
      [fm9001-operand-b 1]
      [fm9001-operand-b_1 (if p5 1 (if p6 1 1))]
      [fm9001-operand-b_2 (if p7 (if p5 0 0) (if p5 0 0))]
      [fm9001-alu-operation (if p8 1 1)]
      [fm9001-alu-operation_1 (if p9 1 1)])))
```

```
(define cmd_3
  (lambda (s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
    (case s
      [fm9001-intr (if p0 0 0)]
      [fm9001-fetch 0]
      [fm9001-fetch_1 0]
      [fm9001-fetch_2 0]
      [fm9001-operand-a 0]
      [fm9001-operand-a_1 (if p1 0 (if p2 0 (if p3 1 0)))]
      [fm9001-operand-a_2 (if p1 1 (if p4 1 (if p2 1 1)))]
      [fm9001-operand-b 1]
      [fm9001-operand-b_1 (if p5 1 (if p6 1 0))]
      [fm9001-operand-b_2 (if p7 (if p5 0 0) (if p5 0 0))]
      [fm9001-alu-operation (if p8 0 0)]
      [fm9001-alu-operation_1 (if p9 0 0)])))
```

```
(define cmd_4
  (lambda (s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
    (case s
      [fm9001-intr (if p0 0 0)]
      [fm9001-fetch 0]
      [fm9001-fetch_1 0]
      [fm9001-fetch_2 0]
      [fm9001-operand-a 0]
      [fm9001-operand-a_1 (if p1 0 (if p2 0 (if p3 0 0)))]
      [fm9001-operand-a_2 (if p1 0 (if p4 0 (if p2 0 0)))]
      [fm9001-operand-b 0]
      [fm9001-operand-b_1 (if p5 0 (if p6 0 0))]
      [fm9001-operand-b_2 (if p7 (if p5 1 1) (if p5 1 1))]
      [fm9001-alu-operation (if p8 1 0)]
      [fm9001-alu-operation_1 (if p9 1 1)]))))
```

4.7.2 Projecting the Next-State Generator

Projecting `next-state` with the representations in figure 4.9 derives four separate next-state generators, one for each state bit. Definitions for each bit are returned. Projecting:

cmd	bits_0..4
v0	00000
v1	10000
v2	01000
v3	11000
v4	00100
v5	10100
v6	01100
v7	11100
v8	00010
v9	10010
v10	01010
v11	11010

cmd	bits_0..4
v12	00110
v13	10110
v14	01110
v15	11110
v16	00001
v17	10001
v18	01001
v19	11001
v20	00101
v21	10101
v22	01101

state	bits_0..3
fm9001-intr	0000
fm9001-fetch	1000
fm9001-fetch_1	0100
fm9001-fetch_2	1100
fm9001-operand-a	0010
fm9001-operand-a_1	1010
fm9001-operand-a_2	0110
fm9001-operand-b	1110
fm9001-operand-b_1	0001
fm9001-operand-b_2	1001
fm9001-alu-operation	0101
fm9001-alu-operation_1	1101

Figure 4.9: Representations

```
(define next-state
  (lambda (s p0)
    (case s
      [fm9001-intr (if p0 fm9001-intr fm9001-fetch)]
      [fm9001-fetch fm9001-fetch_1]
      [fm9001-fetch_1 fm9001-fetch_2]
      [fm9001-fetch_2 fm9001-operand-a]
      [fm9001-operand-a fm9001-operand-a_1]
      [fm9001-operand-a_1 fm9001-operand-a_2]
      [fm9001-operand-a_2 fm9001-operand-b]
      [fm9001-operand-b fm9001-operand-b_1]
      [fm9001-operand-b_1 fm9001-operand-b_2]
      [fm9001-operand-b_2 fm9001-alu-operation]
      [fm9001-alu-operation fm9001-alu-operation_1]
      [fm9001-alu-operation_1 fm9001-intr])))
```

returns

```
(define next-state_0
  (lambda (s p0)
    (case s
      [fm9001-intr (if p0 0 1)]
      [fm9001-fetch 0]
      [fm9001-fetch_1 1]
      [fm9001-fetch_2 0]
      [fm9001-operand-a 1]
      [fm9001-operand-a_1 0]
      [fm9001-operand-a_2 1]
      [fm9001-operand-b 0]
      [fm9001-operand-b_1 1]
      [fm9001-operand-b_2 0]
      [fm9001-alu-operation 1]
      [fm9001-alu-operation_1 0])))
```

```
(define next-state_1
  (lambda (s p0)
    (case s
      [fm9001-intr (if p0 0 0)]
      [fm9001-fetch 1]
      [fm9001-fetch_1 1]
      [fm9001-fetch_2 0]
      [fm9001-operand-a 0]
      [fm9001-operand-a_1 1]
      [fm9001-operand-a_2 1]
      [fm9001-operand-b 0]
      [fm9001-operand-b_1 0]
      [fm9001-operand-b_2 1]
      [fm9001-alu-operation 1]
      [fm9001-alu-operation_1 0])))
```

```
(define next-state_2
  (lambda (s p0)
    (case s
      [fm9001-intr (if p0 0 0)]
      [fm9001-fetch 0]
      [fm9001-fetch_1 0]
      [fm9001-fetch_2 1]
      [fm9001-operand-a 1]
      [fm9001-operand-a_1 1]
      [fm9001-operand-a_2 1]
      [fm9001-operand-b 0]
      [fm9001-operand-b_1 0]
      [fm9001-operand-b_2 0]
      [fm9001-alu-operation 0]
      [fm9001-alu-operation_1 0])))

(define next-state_3
  (lambda (s p0)
    (case s
      [fm9001-intr (if p0 0 0)]
      [fm9001-fetch 0]
      [fm9001-fetch_1 0]
      [fm9001-fetch_2 0]
      [fm9001-operand-a 0]
      [fm9001-operand-a_1 0]
      [fm9001-operand-a_2 0]
      [fm9001-operand-b 1]
      [fm9001-operand-b_1 1]
      [fm9001-operand-b_2 1]
      [fm9001-alu-operation 1]
      [fm9001-alu-operation_1 0]))))
```

4.7.3 Projecting the Datapath

The next step is to project the datapath. This process represents a massive transformation in DDD. Each variable, constant, and operator is projected to a binary representation. For example, consider the `regs-addr` equation that denotes the address of the register-file,

```
(system-letrec
 (...
  [regs-addr
   (select status pc-reg pc-reg pc-reg pc-reg pc-reg (rn-a ins)
           pc-reg (rn-a ins) (rn-a ins) pc-reg pc-reg pc-reg pc-reg
           (rn-b ins) (rn-b ins) (rn-b ins) pc-reg pc-reg pc-reg
           pc-reg (rn-b ins) pc-reg pc-reg)]
  ...) ...).
```

The `regs-addr` equation is projected to a set of four equations. The result is a system of four equations, each defined over a single bit.

```
(system-letrec
 (...
  [regs-addr_0
   (select status pc-reg_0 pc-reg_0 pc-reg_0 pc-reg_0 pc-reg_0 ins_0
           pc-reg_0 ins_0 ins_0 pc-reg_0 pc-reg_0 pc-reg_0 pc-reg_0
           ins_10 ins_10 ins_10 pc-reg_0 pc-reg_0 pc-reg_0
           pc-reg_0 ins_10 pc-reg_0 pc-reg_0)]
  [regs-addr_1
   (select status pc-reg_1 pc-reg_1 pc-reg_1 pc-reg_1 pc-reg_1 ins_1
           pc-reg_1 ins_1 ins_1 pc-reg_1 pc-reg_1 pc-reg_1 pc-reg_1
           ins_11 ins_11 ins_11 pc-reg_1 pc-reg_1 pc-reg_1
           pc-reg_1 ins_11 pc-reg_1 pc-reg_1)]
  [regs-addr_2
   (select status pc-reg_2 pc-reg_2 pc-reg_2 pc-reg_2 pc-reg_2 ins_2
           pc-reg_2 ins_2 ins_2 pc-reg_2 pc-reg_2 pc-reg_2 pc-reg_2
           ins_12 ins_12 ins_12 pc-reg_2 pc-reg_2 pc-reg_2
           pc-reg_2 ins_12 pc-reg_2 pc-reg_2)]
  [regs-addr_3
   (select status pc-reg_3 pc-reg_3 pc-reg_3 pc-reg_3 pc-reg_3 ins_3
           pc-reg_3 ins_3 ins_3 pc-reg_3 pc-reg_3 pc-reg_3 pc-reg_3
           ins_13 ins_13 ins_13 pc-reg_3 pc-reg_3 pc-reg_3
           pc-reg_3 ins_13 pc-reg_3 pc-reg_3)]
  ...) ...)
```

Definitions for each of the operations `rn-a` and `rn-b` are used from the FM9001 base definitions. These act as routing primitives that return the binary projection of the expression `(rn-a ins)` or `(rn-b ins)`.

Chapter 5

DDD-FM9001 Realization

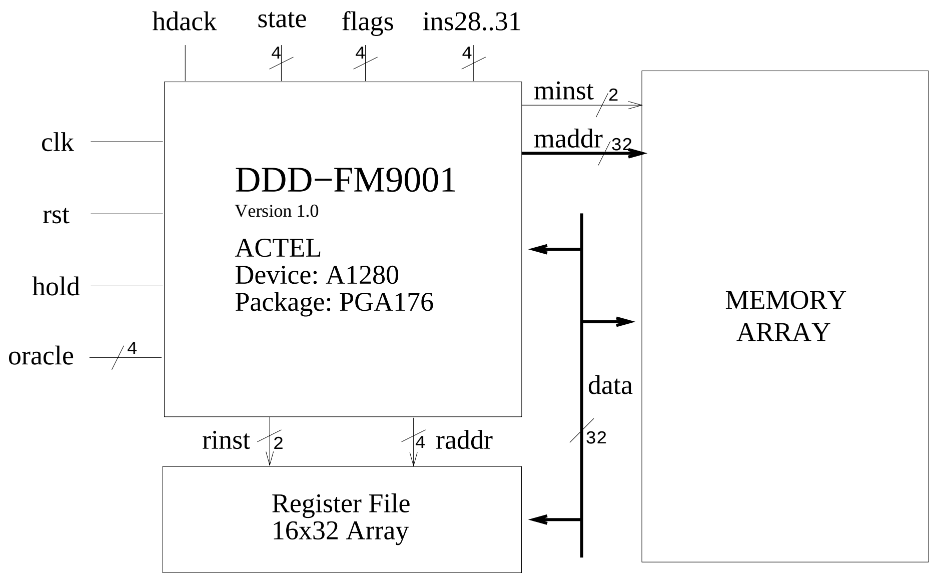


Figure 5.1: DDD-FM9001 Realization Schematic

The DDD-FM9001 is realized in a 176-pin Actel FPGA (Field Programmable Gate Array) with an external register-file. Figure 5.1 is a schematic of the processor and its register-file and memory subsystems. An external high speed cache SRAM module implements the 16×32-bit register-file. The FPGA module transfers data to and from the register-file via the 32-bit bi-directional data bus. The device pin assignments are shown in Figure 5.2.

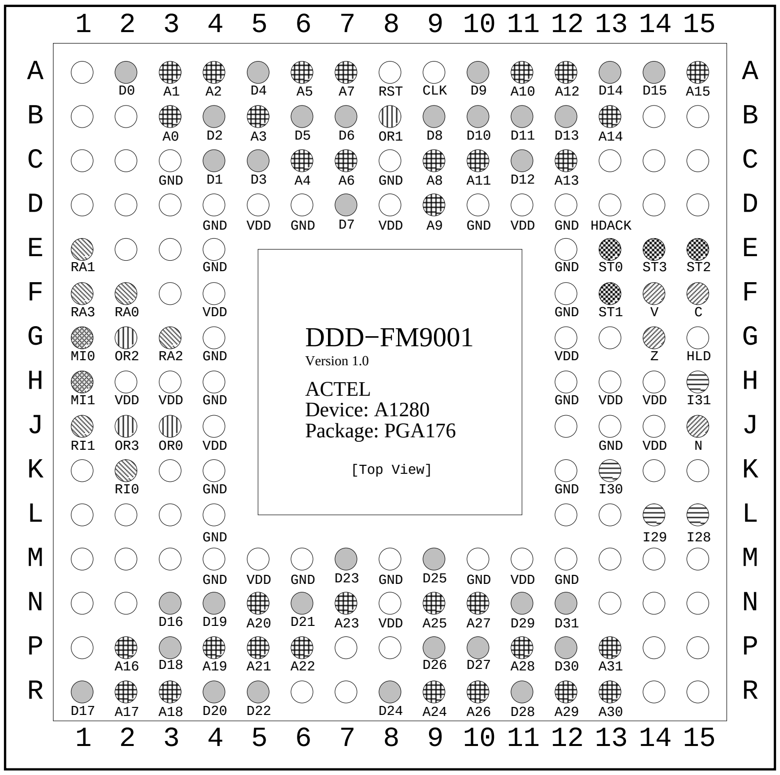

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A D0 A1 A2 D4 A5 A7 RST CLK D9 A10 A12 D14 D15 A15 A
B A0 D2 A3 D5 D6 OR1 D8 D10 D11 D13 A14 B
C GND D1 D3 A4 A6 GND A8 A11 D12 A13 C
D GND VDD GND D7 VDD A9 GND VDD GND HDACK D
E RA1 GND GND ST0 ST3 ST2 E
F RA3 RA0 VDD GND ST1 V C F
G MI0 OR2 RA2 GND VDD Z HLD G
H MI1 VDD VDD GND GND VDD VDD I31 H
J RI1 OR3 OR0 VDD GND VDD N J
K RI0 GND GND I30 K
L GND I29 I28 L
M GND VDD GND D23 GND D25 GND VDD GND M
N D16 D19 A20 D21 A23 VDD A25 A27 D29 D31 N
P A16 D18 A19 A21 A22 D26 D27 A28 D30 A31 P
R D17 A17 A18 D20 D22 D24 A24 A26 D28 A29 A30 R
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
DDD−FM9001
Version 1.0
ACTEL
Device: A1280
Package: PGA176
[Top View]

Figure 5.2: DDD-FM9001 Pin Assignments

The internal logic consists of 92 I/O modules including clock and reset, 1136 logic modules, and 172 sequential modules. The FPGA contains three 4-bit registers for the control state, status flags, and program counter address, four 32-bit registers for the instruction, A and B operands, and b-address, a 32-bit ALU, a 32-bit bi-directional data bus, a 32-bit address bus, four bits to select the program counter, a 2-bit memory instruction signal, a hold and hold-acknowledge signal, and a set of observability pins for the state, flags, and high four bits of the instruction register. In addition, there is a 2-bit instruction and 4-bit address to communicate with the external register-file. The external I/O pins are defined in Table 5.1.

The DDD-FM9001 implements a simple bus protocol with the hold and hold-acknowledge signals. Asserting the hold signal, the processor completes the current

External I/O Pins	Description
Inputs:	
CLK	System clock.
RST	Reset.
HLD	Hold.
OR[0..3]	Sets pc-reg input.
Bi-directionals:	
D[0..31]	Bidirectional data bus (tristate).
Outputs:	
HDACK	Hold acknowledge.
Z,N,V,C	Flags (z, n, v, c).
ST[0..3]	Control state.
I[28..31]	High 4 bits of instruction register.
Memory Signals:	
MI0	Memory read/write (tristate, active low).
MI1	Memory select (tristate, active low).
A[0..31]	Memory address bus (tristate).
Register-File Signals:	
RI0	Register-file read/write (tristate, active low).
RI1	Register-file select (tristate, active low).
RA[0..3]	Register-file address bus (tristate).

Table 5.1: DDD-FM9001 I/O Pin Descriptions

instruction and cycles in the hold state. While in the hold state, the memory, register-file and data signals are tri-stated.

5.1 Design Validation

The DDD-FM9001 was validated against:

- The Boyer-Moore Logic specification of the FM9001.
- The LSI Logic FM9001 Chip.

Validation was necessary to bridge the gap between the formal derivation and the physical hardware realization. The executable property of the Boyer-Moore Logic was exploited during the validation phase of the design. A set of programs was executed on the software model and validated against both the DDD-FM9001 and

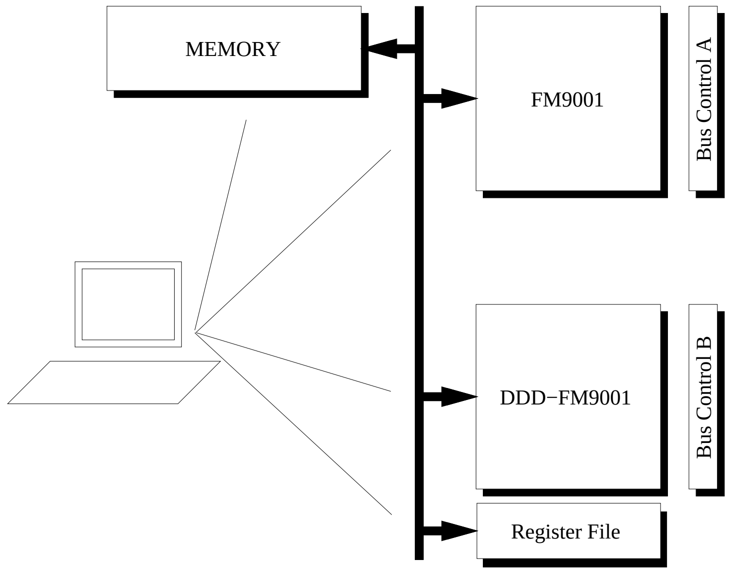


Figure 5.3: Hardware Test Environment

FM9001 processors. The DDD-FM9001 was also subjected to extensive side-by-side comparisons with the FM9001 chip. All modes of validation yielded consistent results.

5.1.1 Hardware Test Environment

The Logic Engine hardware development platform is used to build a test environment for the DDD-FM9001. The test environment consists of the DDD-FM9001 processor, the FM9001 processor, Scheme software running on a workstation, and control logic to interface the workstation, FM9001, DDD-FM9001, and the memory subsystem. Figure 5.3 illustrates the main systems on the test environment.

5.1.2 Software Interface

The software interface, written in Scheme, provides seamless integration between the hardware objects on the Logic Engine and hardware descriptions in software. The interface controls all aspects of the hardware implementations for executing programs on either processor. Software functions provide access to the FM9001, DDD-FM9001, memory, and external register-file. Table 5.2 lists the set of functions.

Functions	Description
Memory/Register-File Control Functions	
mem := `mem` | `regs`	
`(read-mem` *addr mem*`)`	Read from *mem*.
`(write-mem` *addr mem data*`)`	Write to *mem*.
`(clear-mem` *n m mem*`)`	Clear *mem* range from *n* to *m*.
`(list-mem` *n m mem*`)`	Return *mem* range from *n* to *m*.
`(test-mem` *n m mem*`)`	Test *mem* range from *n* to *m*.
`(load-mem` *n pgm mem*`)`	Loads *pgm* into *mem* starting from location *n*.
FM9001/DDD-FM9001 Control Functions	
`(dfm)/(fm)`	Set current processor to DDD-FM9001/FM9001.
`(init-fm9001)`	Initialize.
`(step-fm9001` *n*`)`	Execute *n* instruction cycles.
`(reset-fm9001)`	Reset.
`(run-fm9001)`	Run.
`(stop-fm9001)`	Stop.
`(execute-fm9001` *pgm* `[`*n*`])`	Executes *pgm* *n* instruction cycles.

Table 5.2: DDD-FM9001 Logic Engine Interface

Programs are downloaded to the memory, and either processor or software CPU model is run against the memory. The memory primitives, `mem-read` and `mem-write`, access either the memory or register-file. These functions are compatible with the Boyer-Moore Logic functions and provide the ability to execute the software specification against the hardware memory and/or register-file. A set of functions provides control over the DDD-FM9001 and FM9001 processors. The functions `(dfm)` and `(fm)` toggle which processor is active. Either processor may be initialized, stepped a set number of instruction cycles, reset, run, or stopped. A high-level function, `(execute-fm9001` *pgm* `[`*n*`])`, resets the active processor, loads the `pgm` into memory, and executes the program.

	FM9001 Verification	DDD-FM9001 Derivation
Script entries	2957 entries	1000 lines
Execution time	4hrs (Sparc 2)	30min (Sparc 2)
Netlist	91K characters 2215 lines	69K characters 1178 lines
I/O Pins	95 32 bi-directional	92 32 bi-directional

Figure 5.4: Quantitative Comparison

5.2 Comparing the FM9001 and DDD-FM9001

In terms of hardware realizations, the FM9001 and DDD-FM9001 execute the same instruction set and exhibit the same state to state behavior for each instruction cycle. At the end of each instruction, the contents of the register-file, flags, and memory are equivalent. However, the implementations differ in several key respects.

Some of the quantitative differences between the FM9001 verification and DDD-FM9001 derivation are given in Figure 5.4. It is perhaps worth noting that a quantitative comparison between both methodologies is naive. The statistics on Hunt's verification does not reflect the time involved in designing an implementation, whereas, in the derivational approach, the implementation is a byproduct of the derivation. The table provides some basis for analysis, however the interesting comparisons relate to the differences in architecture.

The block diagrams for the FM9001 and DDD-FM9001 (Figure 5.6 and 5.5) show similar yet distinct architectures. This is not a surprise since the goal of the derivation was not to achieve the identical implementation as the FM9001, but to derive an implementation that preserved the behavior of the initial specification.

The DDD-FM9001 implements both an incrementor and decrementor. The FM9001

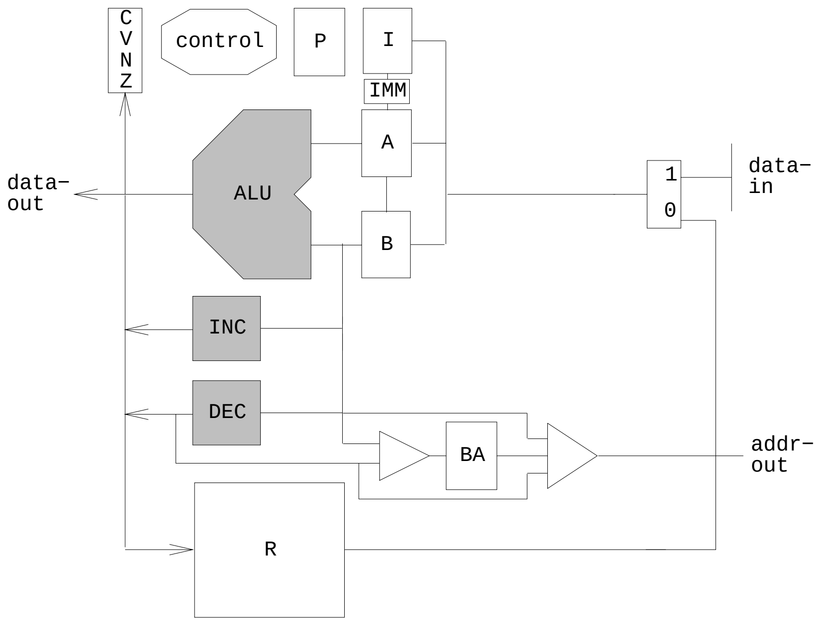


Figure 5.5: DDD-FM9001 Block Diagram

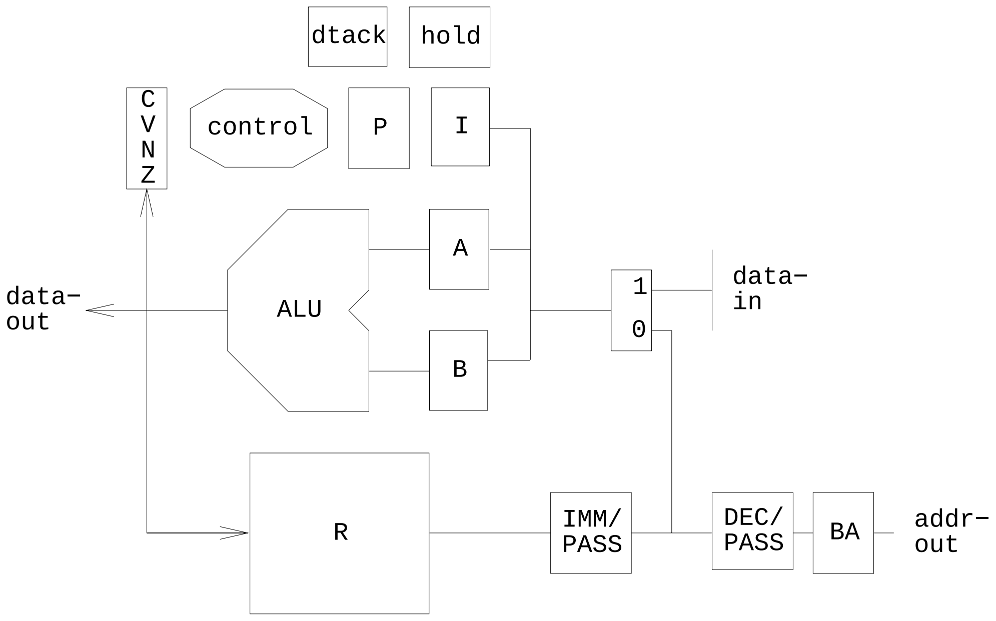


Figure 5.6: FM9001 Block Diagram

implements only a decrementor. The FM9001 ALU is used to implement the increment operation. In the DDD-FM9001 derivation, separate incrementors and decrementors reflect a design decision. An equally valid derivation path would have been to serialize the design such that the ALU, incrementor and decrementor operations were scheduled without any resource conflicts. This would enable the factorization of the arithmetic operations as a single component and verify it against an implementation.

The significant difference between the FM9001 and the DDD-FM9001 is the absence of the `dtack` register and the scan path, in the derived design. The `dtack` signal relates to the change in the model of memory from a functional abstraction to a process abstraction. This difference isolates an aspect of the verification of the FM9001 that could not be derived since it did not exist in the original specification. A formalism for process decomposition to address this issue has been developed by Rath[60], however, at the time of this derivation exercise, the function was not integrated with DDD.

Chapter 6

Conclusion

As researchers begin to understand the interplay of multiple formal systems and how they interact in design, formal methods can move towards a design framework that supports alternate forms of reasoning while maintaining integrity in the design process. In this dissertation, I have investigated the interplay of derivation and verification in the context of a mechanically verified design where the derivation of a verified microprocessor, the DDD-FM9001, is presented. The derivation involved the use of three mechanical verification tools: the DDD digital design derivation system, the Nqthm theorem prover, and the COSMOS boolean tautology checker. However, this work points toward the broader issues relating to how multiple reasoning systems in a peer relationship interact in design. These subtle issues must be understood if we are to formalize the design process.

6.1 The Interplay of Derivation and Verification

The interplay of derivation and verification is a finely grained process with many interdependent aspects. A derivation system depends on verification to establish the correctness of the specifications and representations, and provides a formal justification for *ad hoc* transformations. On the other hand, derivation can be used to isolate verification problems to small building blocks. At lower levels, a design description structured for the purpose of mechanical proof may have to be restructured for the purpose of physical implementation. The re-verification effort is a non-trivial corollary to the correctness proof, and it is evident that this kind of restructuring should be an algebraic process.

One area of future work is the need for secure interaction between formal systems. The notion of interacting systems spans a broad range of possibilities ranging from transformation systems, theorem provers, automatic tautology checkers, type systems, and even the human engineer. Formalization of the interfaces and interaction of formal systems remains an open problem.

6.2 The DDD System

The DDD system was conceived as a research vehicle for engineers and scientists to study the formalization of hardware in an algebraic framework. Toward this goal, DDD has provided an illuminating context for relating theory to practice. The system and the algebra it implements is continually evolving as researchers attempt to formalize the design process.

6.3 Design Derivation

Design derivation allows the designer the flexibility to sketch a complex design space with many possible paths between specification and implementation. It is fairly easy to explore a number of design possibilities in a formal manner, without committing to a particular design decision. For example, various derivation paths were explored during the factorization phase of the DDD-FM9001 derivation. Several alternatives to the factorization of `v-alu`, `v-inc` and `v-dec`, were explored. One path explored was to factor `v-inc` and `v-dec` into a single component. There was no gain in terms of the target hardware. Another path explored was to factor all the arithmetic operations into a single component. This strategy provided the greatest gain, however, it was not employed during the first version of the DDD-FM9001. This would be a reasonable tactic in a new derivation.

Although the ability to safely explore the design space is secured by the DDD algebra, development of tools in the areas of analysis, visualization, and automation would provide practical support for design derivation. Analysis tools to estimate the cost/benefits for various design decisions would provide an effective means of navigation through the design space. Alternative forms of visual representation would facilitate reasoning about specific facets of the design. Automation in the form of tacticals would automate segments of the derivation process. These tacticals would operate on top of the derivation system thus maintaining the integrity of the design.

6.4 The Derivation Script

The derivation script reflects the designer's interaction with the DDD system. In a formal sense, the derivation script is a correctness proof in which the implementation is constructed as a byproduct. Although the final script is a top-down construction from a specification to an implementation, the intellectual effort might be characterized as top-down, bottom-up, and even inside-out. It is common to apply a transformation, analyze the result and gain some insight into the design, backtrack, and follow an alternate path.

For example, in the DDD-FM9001, serialization of the memory and register-file operations was not done on the first iteration of the derivation. During the later stages of structural refinement of the architecture, the algebra would not allow for the memory and register-file to share a common bus because operations on the memory and register-file were occurring simultaneously. The solution was to return to the original behavioral specification, serialize the memory and register-file operations, and continue with the rest of the derivation. Development of tools to aid in the management of derivation scripts would provide another perspective understanding the design process.

6.5 Closing Remarks

The need for heterogeneous reasoning stems from the observation that design is a reasoning process that involves analysis, deduction, and generation. A design tool should not define a methodology, but rather it should reflect the nature of design. The idealized environment consists of multiple formal systems with secure interaction between them. Each system is independent with a formal interface by which they interact. Proofs in one system are interpreted as valid in another, eliminating the need to re-validate proofs across system boundaries. The designer is able to employ the most effective tool for any given design context. Through experimental research, this dissertation takes a step toward this ideal but much more remains to be done. It is my sincere hope that others will draw insight from this experience.

Bibliography

[1] Actel Corporation, Sunnyvale, CA. *ACT Family Field Programmable Gate Array Databook*, 1991.

[2] D. Agnew, L. Claesen, and R. Camposano, editors. *Computer Hardware Description Languages*. North-Holland, Amsterdam, 1993.

[3] S. B. Akers. Binary decision diagrams. *IEEE Trans. on Comput.*, C-27:pages 509–516, June 1978.

[4] Altera. *Altera Programmable Logic User System User Guide*. Santa Clara, CA, 1985.

[5] Graham Birtwistle and P.A. Subrahmanyan, editors. *VLSI Specification, Verification and Synthesis*. Kluwer, Boston, 1988.

[6] Bhaskar Bose. DDD - a transformation system for digital design derivation - reference manual. Technical Report 331, Computer Science Dept., Indiana University, May 1991.

[7] Bhaskar Bose and Steven D. Johnson. DDD-FM9001: Derivation of a verified microprocessor. an exercise in integrating verification with formal derivation. In George J. Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science*, volume 683, pages 191–202. Springer, Berlin, 1993.

[8] Bhaskar Bose, Steven D. Johnson, and Shyamsundar Pullela. Integrating boolean verification with formal derivation. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 139–146. North-Holland, Amsterdam, 1993.

[9] Bhaskar Bose, M. Esen Tuna, and Steven D. Johnson. System factorization in codesign: A case study of the use of formal techniques to achieve hardware-software decomposition. In *1993 IEEE International Conference on Computer Design*, pages 458–461. IEEE, October 1993.

[10] C. David Boyer and Steven D. Johnson. Using the digital design derivation system: Case study of a VLSI garbage collector. In John A. Darringer and Franz J. Rammig, editors, *Computer Hardware Description Languages and their Applications*, pages 235–246. North-Holland, Amsterdam, 1990.

[11] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook.* Academic Press, Boston, 1988.

[12] Bishop Brock and Warren A. Hunt, Jr. Report on the formal specification and partial verification of the VIPER microprocessor. Technical Report 46, Computational Logic Inc., January 1990.

[13] Bishop Brock, Warren A. Hunt, Jr., Matt Kaufmann, and William D. Young. The formal specification and verification of the FM9001 microprocessor. Technical Report 86, Computational Logic Inc., October 1994.

[14] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Comput.*, C-35:pages 677–691, August 1986.

[15] Randal E. Bryant. COSMOS: A Compiled Simulator for MOS Circuits. In *24th ACM/IEEE Design Automation Conference*, 1987.

[16] J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. In *International Workshop on Formal methods in VLSI Design*, January 1991.

[17] R.M. Burstall and John Darlington. A transformation system for developing recursive programs. *Comm. ACM*, 24(1):pages 44–67, January 1977.

[18] A. Church. A formulation of the simple theory of types. *Symbolic Logic*, 5(1), 1940.

[19] Luc J.M. Claesen, editor. *Formal VLSI Correctness Verification.* North-Holland, Amsterdam, 1990.

[20] Luc J.M. Claesen, editor. *Formal VLSI Specification and Synthesis.* North-Holland, Amsterdam, 1990.

[21] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 15–30. North-Holland, Amsterdam, 1993.

[22] William Clinger and Jonathan Rees. The revised^4 report on the algorithmic language SCHEME. *ACM Lisp Pointers*, 4(3):pages 1–55, 1991.

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

[24] Avra Cohn. Correctness properties of the VIPER block model: The second level. In *preliminary papers for the Banff Hardware Verification Workshop*, June 1988.

[25] John A. Darringer. The application of program verification techniques to hardware verification. In *16th ACM/IEEE Design Automation Conference*, 1979.

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

[27] P. Gaboury and M.I. Elmasry. Using program transformation for VLSI design automation. In Luc J.M. Claesen, editor, *Formal VLSI Specification and Synthesis*, pages 43–59. North-Holland, Amsterdam, 1990.

[28] Fausto Giunchiglia, Paolo Pecchiari, and Carolyn Talcott. Reasoning structures - an architecture for open mechanized reasoning systems. Working Draft.

[29] Michael J.C. Gordon. LCF-LSM, a system for specifying and verifying hardware. Technical Report 41, Computer Laboratory, The University of Cambridge, September 1983.

[30] Michael J.C. Gordon. Proving a computer correct using the LCF-LSM hardware verification system. Technical Report 42, Computer Laboratory, The University of Cambridge, September 1983.

[31] Michael J.C. Gordon. HOL: a proof generating system for higher-order logic. In Graham Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer, Boston, 1988.

[32] Brian T. Graham. *The SECD Microprocessor.* Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, 1992.

[33] Peter Henderson. *Functional Programming, Application and Implementation.* Prentice-Hall, Englewood Cliffs, NJ, 1980. C. A. R. Hoare, Series Editor (international series in computer science).

[34] Warren A. Hunt, Jr. *FM8501: A Verified Microprocessor.* PhD thesis, University of Texas at Austin, December 1985.

[35] Warren A. Hunt, Jr. Theorem provers in circuit design. In V. Stavridou, T.F. Melham, and R.T. Boute, editors, *International Conference on Theorem Provers in Circuit Design – Theory, Practice, and Experience.* North-Holland, June 1992.

[36] Warren A. Hunt, Jr. and Bishop Brock. A formal HDL and its use in the FM9001 verification. In C.A.R. Hoare and M.J.C. Gordon, editors, *Mechanized Reasoning in Hardware Design*, pages 35–48. Prentice-Hall, Englewood Cliffs, N.J., 1992.

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

[38] Steven D. Johnson. Digital design in a functional calculus. In G.J. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 45–58. North-Holland, Amsterdam, 1986.

[39] Steven D. Johnson. Manipulating logical organization with system factorizations. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Sythesis: Mathematical Aspects, Lecture Notes in Computer Science*, volume 408, pages 260–281. Springer, Berlin, 1989.

[40] Steven D. Johnson and Bhaskar Bose. A system for digital design derivation. Technical Report 289, Computer Science Dept., Indiana University, August 1989.

[41] Steven 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*, January 1991.

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

[43] Steven D. Johnson and C. David Boyer. Modelling transistors applicatively. In George J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 397–420. North-Holland, Amsterdam, 1988.

[44] Steven D. Johnson, Robert M. Wehrmeister, and Bhaskar Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Luc J.M. Claesen, editor, *Formal VLSI Specification and Synthesis*, pages 117–136. North-Holland, Amsterdam, 1990.

[45] Jeffrey J. Joyce. Multi-level verification of a simple microprocessor. Technical report, Computer Laboratory, University of Cambridge, 1987. Progress Report.

[46] P.J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):pages 308–320, 1964.

[47] M. Leeser and G. Brown, editors. *Hardware Specification, Verification and Sythesis: Mathematical Aspects, Lecture Notes in Computer Science*, volume 408. Springer, Berlin, 1989.

[48] Karl Levitt, Tejkumar Arora, Tony Leung, Sara Kalvala, E. Thomas Schubert, Philip Windley, Mark Heckman, and Gerald C. Cohen. *Formal Verification of a Microcoded VIPER Microprocessor using HOL.* NASA Contractor Report 4489, 1993.

[49] J.C. Madre and J.P. Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *25th ACM/IEEE Design Automation Conference*, June 1988.

[50] Michael C. McFarland, Alice C. Parker, and Raul Camposano. Tutorial on high-level synthesis. In *25th ACM/IEEE Design Automation Conference*, pages 330–336, 1988.

[51] George J. Milne, editor. *The Fusion of Hardware Design and Verification.* North-Holland, Amsterdam, 1988.

[52] George J. Milne and Laurence Pierre, editors. *Correct Hardware Design and Verification Methods*, volume 683. Springer, Berlin, 1993.

[53] George J. Milne and P.A. Subrahmanyam, editors. *Formal Aspects of VLSI Design.* North-Holland, Amsterdam, 1986.

[54] Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of formal design systems in the development of a fault-tolerant clock synchronization circuit. In *13th Symp. on Reliable Distributed Systems*, pages 128–137, October 1994.

[55] J.D. Morison, N.E. Peeling, and T.L. Thorp. The design rationale of ELLA, a hardware design and description language. In *CHDL'85*, 1985.

[56] John T. O'Donnell. Hydra: Hardware description in a functional language using recursion equations and high order combining forms. In George J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 309–328. North-Holland, Amsterdam, 1988.

[57] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE),* volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[58] Kamlesh Rath, Bhaskar Bose, and Steven D. Johnson. Derivation of a DRAM memory interface by sequential decomposition. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 438–441. IEEE, October 1993.

[59] Kamlesh Rath, Ignacio Celis, and Robert M. Wehrmeister. RTBA: A generic bit-sliced bus architecture for datapath synthesis. Technical Report 321, Computer Science Dept., Indiana University, December 1990.

[60] Kamlesh Rath and Steven D. Johnson. Toward a basis for protocol specification and process decomposition. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 169–186. North-Holland, Amsterdam, 1993.

[61] Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson. Behavior tables: A basis for system representation and transformational system synthesis. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 736–740. IEEE, November 1993.

[62] Kamlesh Rath, M. Esen Tuna, and Steven D. Johnson. Specification and synthesis of bounded indirection. Technical Report 398, Computer Science Dept., Indiana University, February 1994.

[63] Klaus Schneider, Ramayya Kumar, and Thomas Kropf. Hardware verification using first order BDDs. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 45–62. North-Holland, Amsterdam, 1993.

[64] Carl-Johan Seger and Jeffrey J. Joyce. A two-level formal verification methodology using HOL and COSMOS. In K.G. Larsen and A. Skou, editors, *Computer*

Aided Verification, Lecture Notes in Computer Science, volume 575, pages 299–309. Springer, 1991.

[65] Robin Sharp and Ole Rasmussen. Rewriting and constraints in T-Ruby. In George J. Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification Methods, Lecture Notes in Computer Science*, volume 683, pages 226–241. Springer, Berlin, 1993.

[66] Mary Sheeran. *uFP, a Language for VLSI Design.* PhD thesis, Programming Research Group, Oxford University, 1983.

[67] Mary Sheeran. muFP, an algebraic VLSI design language. In *Proceedings of the ACM Symp. on LISP and Functional Programming*, 1984.

[68] Mary Sheeran. Retiming and slowdown in Ruby. In George J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 289–308. North-Holland, Amsterdam, 1988.

[69] Rick L. Spickelmier. *Release Notes for Oct Tools Distribution 5.1.* Electronics Research Laboratory, University of California, Berkeley, August 1991.

[70] M. Esen Tuna, Steven D. Johnson, and Bob Burger. Continuations in hardware-software codesign. In *1994 IEEE International Conference on Computer Design*, pages 264–269. IEEE, October 1994.

[71] Ranganadha R. Vemuri. *A Transformational Approach to Register-Transfer-Level Design-Space Exploration.* PhD thesis, Case Western Reserve University, January 1989.

[72] Ranganadha R. Vemuri and Christos A. Papachristou. On control-step assignment in a transformational synthesis system: $\rfloor$-expressions and their algebra.

In Gabriele Saucier and Paul Michael McLellan, editors, *Logic and Architecture Synthesis for Silicon Compilers*, pages 177–199. North-Holland, 1989.

[73] Robert M. Wehrmeister. Derivation of an SECD machine: Experience with a transformational approach to synthesis. Technical Report 290, Computer Science Dept., Indiana University, September 1989.

[74] P.J. Windley. *The Formal Verification of Generic Interpreters.* PhD thesis, University of California, Davis, 1990.

[75] David Winkel. The use of PALs in CPU design. Technical Report 204, Computer Science Dept., Indiana University, October 1986.

[76] David Winkel. What next for PAL-DEVICES - the second generation challenge. Technical Report 188, Computer Science Dept., Indiana University, May 1986.

[77] David Winkel and Frank Prosser. *The Art of Digital Design.* Prentice-Hall, Englewood Cliffs, N.J., 1980.

[78] David Winkel, Franklin Prosser, Robert Wehrmeister, William C. Hunt, and Caleb Hess. A student VLSI hardware tester. In *Proceedings of the Microelectronic Systems Education Conference and Exposition*, pages 15–24, 1990.

[79] Zheng Zhu. *Structured Hardware Design Transformations.* PhD thesis, Computer Science Dept., Indiana University, 1992.

[80] Zheng Zhu and Steven D. Johnson. An algebraic characterization of structural synthesis for hardware. In Luc J.M. Claesen, editor, *Formal VLSI Specification and Synthesis*, pages 261–270. North-Holland, Amsterdam, 1990.

[81] Zheng Zhu and Steven D. Johnson. An algebraic framework for data abstraction in hardware description. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits*, pages 50–67. Springer, Berlin, 1990.

[82] Zheng Zhu and Steven D. Johnson. An example of interactive hardware transformation. In Subramanyam, editor, *Proceedings of ACM International Workshop on Formal Methods in VLSI Design*, January 1991.

[83] Zheng Zhu and Steven D. Johnson. Capturing synchronization specifications for sequential compositions. In *1994 IEEE International Conference on Computer Design*, pages 117–121. IEEE, October 1994.

Appendix A

The FM9001 Specification

```
(defn fm9001-intr (state oracle)
  (if (nlistp oracle)
      state
      (fm9001-intr (fm9001-step state (car oracle))
                   (cdr oracle))))

(defn fm9001-step (state pc-reg)
  (let ((p-state (car state))
        (mem     (cadr state)))
    (fm9001-fetch (regs p-state) (flags p-state) mem pc-reg)))

(defn fm9001-fetch (regs flags mem pc-reg)
  (let ((pc (read-mem pc-reg regs)))
    (let ((ins (read-mem pc mem)))
      (let ((pc+1 (v-inc pc)))
        (let ((new-regs (write-mem pc-reg regs pc+1)))
          (fm9001-operand-a new-regs flags mem ins))))))

(defn fm9001-operand-a (regs flags mem ins)
  (let ((a-immediate-p (a-immediate-p ins))
        (a-immediate   (sign-extend (a-immediate ins) 32))
        (mode-a        (mode-a ins))
        (rn-a          (rn-a  ins)))
    (let ((reg (read-mem rn-a regs)))
      (let ((reg- (v-dec reg))
            (reg+ (v-inc reg)))
        (let ((operand-a (if* a-immediate-p
                              a-immediate
                              (if* (reg-direct-p mode-a)
                                   reg
                                   (if* (pre-dec-p mode-a)
                                        (read-mem reg- mem)
                                        (read-mem reg mem))))))
          (let ((new-regs (if* a-immediate-p
                               regs
                               (if* (pre-dec-p mode-a)
                                    (write-mem rn-a regs reg-)
                                    (if* (post-inc-p mode-a)
                                         (write-mem rn-a regs reg+)
                                         regs)))))

            (fm9001-operand-b new-regs flags mem ins operand-a)))))))

(defn fm9001-operand-b (regs flags mem ins operand-a)
  (let ((mode-b (mode-b ins))
        (rn-b   (rn-b  ins)))
    (let ((reg (read-mem rn-b regs)))
      (let ((reg- (v-dec reg))
            (reg+ (v-inc reg)))
        (let ((b-address (if* (pre-dec-p mode-b)
                              reg-
                              reg)))
          (let ((operand-b (if* (reg-direct-p mode-b)
                                reg
                                (read-mem b-address mem)))
                (new-regs (if* (pre-dec-p mode-b)
                               (write-mem rn-b regs reg-)
                               (if* (post-inc-p mode-b)
                                    (write-mem rn-b regs reg+)
                                    regs))))

            (fm9001-alu-operation new-regs flags mem ins operand-a operand-b
                                  b-address)))))))

(defn fm9001-alu-operation (regs flags mem ins operand-a operand-b b-address)
  (let ((op-code   (op-code ins))
        (store-cc  (store-cc ins))
        (set-flags (set-flags ins))
        (mode-b    (mode-b   ins))
        (rn-b      (rn-b    ins)))
    (let ((cvzbv  (v-alu (c-flag flags) operand-a operand-b op-code))
          (storep (store-resultp store-cc flags)))
      (let ((bv (bv cvzbv)))
        (let ((new-regs   (if* (and* storep (reg-direct-p mode-b))
                     (write-mem rn-b regs bv)
                     regs))
    (new-flags  (update-flags flags set-flags cvzbv))
    (new-mem    (if* (and* storep (not* (reg-direct-p mode-b)))
                     (write-mem b-address mem bv)
                     mem)))
(list (list new-regs new-flags) new-mem))))))
```

FM9001 32-bit Instruction (high 4-bits unspecified):

$$\underbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{0}}_{\text{op−code}}\ \underbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{0}}_{\text{store−cc}}\ \underbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{0}}_{\text{set−flags}}\ \underbrace{\boxed{0}\boxed{0}}_{\text{mode−b}}\ \underbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{0}}_{\text{rn−b}}\ \underbrace{\boxed{0}}_{\text{a−immediate−p}}\ \underbrace{\overbrace{\boxed{0}\boxed{0}\boxed{0}}^{\text{N/A}}\ \overbrace{\boxed{0}\boxed{0}}^{\text{mode−a}}\ \overbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{0}}^{\text{rn−a}}}_{\text{a−immediate}}$$

Interpretation of the op-code:

0000	$b \leftarrow a$	Move
0001	$b \leftarrow a + 1$	Increment
0010	$b \leftarrow a + b + c$	Add with carry
0011	$b \leftarrow a + b$	Add
0100	$b \leftarrow 0 - a$	Negation
0101	$b \leftarrow a - 1$	Decrement
0110	$b \leftarrow b - a - c$	Subtract with borrow
0111	$b \leftarrow b - a$	Subtract
1000	$b \leftarrow a \gg 1$	Rotate right, shifted through carry
1001	$b \leftarrow a \gg 1$	Arithmetic shift right, top bit copied
1010	$b \leftarrow a \gg 1$	Logical shift right, top bit zero
1011	$b \leftarrow a \oplus b$	Exclusive or
1100	$b \leftarrow a \vee a$	Or
1101	$b \leftarrow a \wedge b$	And
1110	$b \leftarrow \neg a$	Not
1111	$b \leftarrow a$	Move

Condition codes for store-cc:

0000	$(\neg c)$	Carry clear
0001	(c)	Carry set
0010	$(\neg v)$	Overflow clear
0011	(v)	Overflow set
0100	$(\neg n)$	Plus
0101	(n)	Negative
0110	$(\neg z)$	Not equal
0111	(z)	Equal
1000	$(\neg c \wedge \neg z)$	High
1001	$(c \vee z)$	Low or same
1010	$(n \wedge v \vee \neg n \wedge \neg v)$	Greater or equal
1011	$(n \wedge \neg v \vee \neg n \wedge v)$	Less than
1100	$(n \wedge v \wedge \neg z \vee \neg n \wedge \neg v \wedge \neg z)$	Greater than
1101	$(z \vee n \wedge \neg v \vee \neg n \wedge v)$	Less or equal
1110	(t)	True
1111	(nil)	False

Condition codes for set-flags:

0000	- - - -
0001	- - - z
0010	- - n -
0011	- - n z
0100	- v - -
0101	- v - z
0110	- v n -
0111	- v n z
1000	c - - -
1001	c - - z
1010	c - n -
1011	c - n z
1100	c v - -
1101	c v - z
1110	c v n -
1111	c v n z

The A operand is a 10 bit field. If the high order bit is set, the low order 9 bits are treated as a signed immediate. Otherwise, the low order six bits of the A operand are a mode/register pair identical to the B operand.

Addressing Modes for A and B operands:

00	Register Direct
01	Register Indirect
10	Register Indirect with Pre-decrement
11	Register Indirect with Post-increment

Register numbers: (register 15 is usually used as the program counter)

0000	Register 0
0001	Register 1
0010	Register 2
0011	Register 3
0100	Register 4
0101	Register 5
0110	Register 6
0111	Register 7
1000	Register 8
1001	Register 9
1010	Register 10
1011	Register 11
1100	Register 12
1101	Register 13
1110	Register 14
1111	Register 15

Appendix B

The DDD-FM9001 Derivation Script

```
;; ***********************************************************************
;; *                                                                     *
;; * DDD-FM9001: Derivation Script                                       *
;; * Author: Bhaskar Bose                                                *
;; *   File: script.ss                                                   *
;; *                                                                     *
;; ***********************************************************************

;; Load iterative specification
;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define ITRSYS_0 (read-file "DFM9001/ITRSYS_0"))


;; ***********************************************************************
;; *                Transformations on Behavior                          *
;; *                ---------------------------                          *
;; *                                                                     *
;; * Transformations applied:                                            *
;; *              unfold-let: Remove all let bindings                    *
;; *         expand-function: Expand if*                                 *
;; *                add-spec: Serialize                                  *
;; *        order-state-defs: Order state definitions                    *
;; *    distribute-iterative: Distribute if conditional                  *
;; ***********************************************************************

;; Remove all let bindings
;; ~~~~~~~~~~~~~~~~~~~~~~~
(define ITRSYS_0.1
  (unfold-let '(pc ins pc+1 new-regs) 'fm9001-fetch ITRSYS_0))
```

```
(define ITRSYS_0.2
  (unfold-let '(a-immediate-p a-immediate mode-a
                              rn-a reg reg- reg+ operand-a new-regs)
              'fm9001-operand-a ITRSYS_0.1))

(define ITRSYS_0.3
  (unfold-let '(mode-b rn-b reg reg- reg+ b-address operand-b new-regs)
              'fm9001-operand-b ITRSYS_0.2))

(define ITRSYS_1
  (unfold-let '(op-code store-cc set-flags mode-b rn-b cvzbv storep bv
                        new-regs new-flags new-mem)
              'fm9001-alu-operation ITRSYS_0.3))

;; Unfold applications of if*
;; ~~~~~~~~~~~~~~~~~~~~~~~~~~
(define ITRSYS_2 (expand-function '(defn if* (a b c) (if a b c)) ITRSYS_1))

;; Serialize fm9001-fetch -> fetch,fetch_1,fetch_2
;;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define ITRSYS_2.1
  (add-spec
   '[fm9001-fetch_1
     (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
       (fm9001-operand-a
        (write-mem pc-reg regs (v-inc operand-b))
        flags
        mem
        pc-reg
        (read-mem operand-b mem)
        operand-a
        operand-b
        b-address
        oracle
        ))] ITRSYS_2))

(define ITRSYS_2.2
  (add-spec
   '[fm9001-fetch_2
     (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
       (fm9001-operand-a
        (write-mem pc-reg regs (v-inc operand-b))
```

```
        flags
        mem
        pc-reg
        ins
        operand-a
        operand-b
        b-address
        oracle
        ))] ITRSYS_2.1))

;; Serialize fm9001-operand-a -> operand-a,operand-a_1,operand-a_2
;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define ITRSYS_2.3
  (add-spec
   '[fm9001-operand-a_1
     (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
       (fm9001-operand-b
        (if (a-immediate-p ins)
            regs
            (if (pre-dec-p (mode-a ins))
                (write-mem (rn-a ins) regs (v-dec operand-b))
                (if (post-inc-p (mode-a ins))
                    (write-mem (rn-a ins) regs (v-inc operand-b))
                    regs)))
        flags
        mem
        pc-reg
        ins
        (if (a-immediate-p ins)
            (sign-extend (a-immediate ins) thirty-two)
            (if (reg-direct-p (mode-a ins))
                operand-b
                (if (pre-dec-p (mode-a ins))
                    (read-mem (v-dec operand-b) mem)
                    (read-mem operand-b mem))))
        operand-b
        b-address
        oracle))] ITRSYS_2.2))

(define ITRSYS_2.4
  (add-spec
   '[fm9001-operand-a_2
```

```
      (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
        (fm9001-operand-b
         regs
         flags
         mem
         pc-reg
         ins
         (if (a-immediate-p ins)
             (sign-extend (a-immediate ins) thirty-two)
             (if (reg-direct-p (mode-a ins))
                 operand-b
                 (if (pre-dec-p (mode-a ins))
                     (read-mem (v-dec operand-b) mem)
                     (read-mem operand-b mem))))
         operand-b
         b-address
         oracle))] ITRSYS_2.3))

;; Serialize fm9001-operand-b -> operand-b,operand-b_1,operand-b_2
;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define ITRSYS_2.5
  (add-spec
   '[fm9001-operand-b_1
      (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
        (fm9001-alu-operation
         (if (pre-dec-p (mode-b ins))
             (write-mem (rn-b ins) regs (v-dec operand-b))
             (if (post-inc-p (mode-b ins))
                 (write-mem (rn-b ins) regs (v-inc operand-b))
                 regs))
         flags
         mem
         pc-reg
         ins
         operand-a
         (if (reg-direct-p (mode-b ins))
             operand-b
             (read-mem (if (pre-dec-p (mode-b ins))
                           (v-dec operand-b)
                           operand-b)
                       mem))
         (if (pre-dec-p (mode-b ins))
```

```
            (v-dec operand-b)
            operand-b)
        oracle))] ITRSYS_2.4))

(define ITRSYS_2.6
  (add-spec
   '[fm9001-operand-b_2
     (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
       (fm9001-alu-operation
        regs
        flags
        mem
        pc-reg
        ins
        operand-a
        (if (reg-direct-p (mode-b ins))
            operand-b
            (read-mem (if (pre-dec-p (mode-b ins))
                          (v-dec operand-b)
                          operand-b)
                      mem))
        (if (pre-dec-p (mode-b ins))
            (v-dec operand-b)
            operand-b)
        oracle))] ITRSYS_2.5))

;; Serialize fm9001-alu-operation -> alu-operation,alu-operation_1
;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define ITRSYS_3
  (add-spec
   '[fm9001-alu-operation_1
     (lambda (regs flags mem pc-reg ins operand-a operand-b b-address oracle)
       (fm9001-intr
        regs
        (update-flags flags (set-flags ins)
                      (v-alu (c-flag flags) operand-a operand-b (op-code ins)))
        (if (and* (store-resultp (store-cc ins) flags)
                  (not* (reg-direct-p (mode-b ins))))
            (write-mem b-address mem
                       (bv (v-alu (c-flag flags)
                                  operand-a operand-b (op-code ins))))
            mem)
```

```
pc-reg
ins
operand-a
operand-b
b-address
oracle))] ITRSYS_2.6))
```

```
;; Order state definitions
;; ~~~~~~~~~~~~~~~~~~~~~~~
(define ITRSYS_3.1
  (order-state-defs ITRSYS_3
                    '(fm9001-intr
                      fm9001-fetch
                      fm9001-fetch_1
                      fm9001-fetch_2
                      fm9001-operand-a
                      fm9001-operand-a_1
                      fm9001-operand-a_2
                      fm9001-operand-b
                      fm9001-operand-b_1
                      fm9001-operand-b_2
                      fm9001-alu-operation
                      fm9001-alu-operation_1)))

;; Distribute if conditional
;; ~~~~~~~~~~~~~~~~~~~~~~~~~
(define ITRSYS_4 (distribute-iterative ITRSYS_3.1))
```

```
;; ***************************************************************************
;; *                   Behavior to Structure                                 *
;; *                   ---------------------                                 *
;; *                                                                         *
;; * Transformations applied:                                                *
;; *          itrsys->seqsys: behavior to structure construction             *
;; *                                                                         *
;; ***************************************************************************

;; Behavior to Structure
;; ~~~~~~~~~~~~~~~~~~~~~
(define SEQSYS_1   (itrsys->seqsys ITRSYS_4))

(define SELECT_1   (seqsys.select  SEQSYS_1))
(define SYSTEM_1   (seqsys.system  SEQSYS_1))
(define BASE_1     (seqsys.base    SEQSYS_1))
```

```
;; ***********************************************************************
;; *                 Transformations on Structure                         *
;; *                 ---------------------------                          *
;; *                                                                      *
;; * Transformations applied:                                             *
;; *    factor-streqn: Factor MEM equation                                *
;; *    factor-streqn: Factor REGS equation                               *
;; *   factor-general: Factor V-ALU operations                            *
;; *   factor-general: Factor V-INC operations                            *
;; *   factor-general: Factor V-DEC operations                            *
;; *                                                                      *
;; ***********************************************************************

;; factor MEM
;; ~~~~~~~~~~
(define SYSTEM_2.1+abs+base
  (factor-streqn 'MEM-OUT 'ABS-MEM 'MEM SYSTEM_1 '(READ-MEM * MEM)))

(define SYSTEM_2.1 (factor.system SYSTEM_2.1+abs+base))
(define ABS_2.1    (factor.abs    SYSTEM_2.1+abs+base))
(define BASE_2.1   (factor.base   SYSTEM_2.1+abs+base))

(define SYSTEM_2.2
  (let* ([SYSTEM SYSTEM_2.1]
         [SYSTEM (merge-streams 'MEM-INST 'MEM-PROBE-0-INST 'MEM-INST SYSTEM)]
         [SYSTEM (merge-streams 'MEM-V0   'MEM-PROBE-0-V0 'MEM-V0     SYSTEM)]
         [SYSTEM (merge-streams 'MEM-V1   'MEM-PROBE-0-V1 'MEM-V1     SYSTEM)]
         [SYSTEM (remove-stream 'MEM-V1                              SYSTEM)]
         [SYSTEM (rename-stream 'MEM-V2   'MEM-V1                    SYSTEM)]
         [SYSTEM (rename-stream 'MEM-V0   'MEM-ADDR                  SYSTEM)]
         [SYSTEM (rename-stream 'MEM-V1   'MEM-DATA                  SYSTEM)]
         [SYSTEM (rename-stream 'MEM-V0-? 'MEM-ADDR-?                SYSTEM)]
         [SYSTEM (rename-stream 'MEM-V1-? 'MEM-DATA-?                SYSTEM)]
         )
    SYSTEM))

;; Factor REGS
;; ~~~~~~~~~~~
(define SYSTEM_3.1+abs+base
  (factor-streqn 'REGS-OUT 'ABS-REGS 'REGS SYSTEM_2.2 '(READ-MEM * REGS)))

(define SYSTEM_3.1 (factor.system SYSTEM_3.1+abs+base))
```

```
(define ABS_3.1    (factor.abs    SYSTEM_3.1+abs+base))
(define BASE_3.1   (factor.base   SYSTEM_3.1+abs+base))

(define SYSTEM_3.2
  (let* ([SYSTEM SYSTEM_3.1]
         [SYSTEM (merge-streams 'REGS-INST 'REGS-PROBE-0-INST 'REGS-INST SYSTEM)]
         [SYSTEM (merge-streams 'REGS-V0   'REGS-PROBE-0-V0 'REGS-V0     SYSTEM)]
         [SYSTEM (merge-streams 'REGS-V1   'REGS-PROBE-0-V1 'REGS-V1     SYSTEM)]
         [SYSTEM (remove-stream 'REGS-V1                                  SYSTEM)]
         [SYSTEM (rename-stream 'REGS-V2    'REGS-V1                      SYSTEM)]
         [SYSTEM (rename-stream 'REGS-V0    'REGS-ADDR                    SYSTEM)]
         [SYSTEM (rename-stream 'REGS-V1    'REGS-DATA                    SYSTEM)]
         [SYSTEM (rename-stream 'REGS-V0-? 'REGS-ADDR-?                   SYSTEM)]
         [SYSTEM (rename-stream 'REGS-V1-? 'REGS-DATA-?                   SYSTEM)]
         )
    SYSTEM))

;; Further optimizations
;; Merge MEM-DATA,REGS-DATA,make bidirec
;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define SYSTEM_4.1
  (let* ([SYSTEM SYSTEM_3.2]
         [SYSTEM (merge-streams 'MEM-DATA 'REGS-DATA 'MEM-REGS-DATA SYSTEM)])
    SYSTEM))

(define SYSTEM_4.2
  (let* ([SYSTEM SYSTEM_4.1]
         ;make v1 bidirectional
         [SYSTEM (identify-stream 'A 'MEM-OUT  SYSTEM)]
         [SYSTEM (identify-stream 'B 'REGS-OUT SYSTEM)]
         [SYSTEM (merge-streams   'A 'B 'C SYSTEM)] ;; creates data-in
         ;; merge data-in/out
         [SYSTEM (merge-streams   'C 'MEM-REGS-DATA 'MEM-REGS-DATA SYSTEM)]
         ;; make mem-regs-data have data-in
         [SYSTEM (identify-stream 'A 'MEM-OUT  SYSTEM)]
         [SYSTEM (identify-stream 'B 'REGS-OUT SYSTEM)]
         [SYSTEM (merge-streams   'A 'B 'DATA-IN SYSTEM)]

         [SYSTEM2 (identify-stream 'A 'MEM-WRITE-MEM SYSTEM)]
         [SYSTEM2 (identify-stream 'B 'REGS-WRITE-MEM SYSTEM2)]
         [SYSTEM2 (merge-streams   'A 'B 'C SYSTEM2)]
```

```
          [SYSTEM2 (generate-load
                     '(MEM-WRITE-MEM REGS-WRITE-MEM) 'C 'DATA-OUT-ENABLE SYSTEM2)]
          [SYSTEM  (add-stream (extract-stream 'DATA-OUT-ENABLE SYSTEM2) SYSTEM)]
          ;generate tri-state signal for memory address and instruction
          [SYSTEM  (generate-load-cmd '(0) 'MEM-TRISTATE SYSTEM)]

          ;identify update-flags
          [SYSTEM (identify-stream 'UPDATE-FLAGS-OUT '(UPDATE-FLAGS * * *) SYSTEM)])
    SYSTEM))

;; Factor V-ALU
;; ~~~~~~~~~~~~

(define SYSTEM_5.1+abs+base
  (factor-general 'V-ALU-OUT 'ABS-V-ALU '(V-ALU) SYSTEM_4.2))

(define SYSTEM_5.1 (factor.system SYSTEM_5.1+abs+base))
(define ABS_5.1    (factor.abs    SYSTEM_5.1+abs+base))
(define BASE_5.1   (factor.base   SYSTEM_5.1+abs+base))

(define SYSTEM_5.2
  (let* ([SYSTEM SYSTEM_5.1]
          [SYSTEM (rename-stream 'V-ALU-OUT-V0 'V-ALU-OUT-CARRYIN SYSTEM)]
          [SYSTEM (rename-stream 'V-ALU-OUT-V1 'V-ALU-OUT-OPA SYSTEM)]
          [SYSTEM (rename-stream 'V-ALU-OUT-V2 'V-ALU-OUT-OPB SYSTEM)]
          [SYSTEM (rename-stream 'V-ALU-OUT-V3 'V-ALU-OUT-OPCODE SYSTEM)]
          ;; Expand ALU port
          [SYSTEM (expand-stream 'V-ALU-OUT-OPA SYSTEM)]
          [SYSTEM (expand-stream 'V-ALU-OUT-OPB SYSTEM)]
          ;; nops should be better named
          [SYSTEM (rename-stream 'V-ALU-OUT-INST-NOP 'V-ALU-OUT-NOP SYSTEM)])
    SYSTEM))

;; Factor V-INC,V-DEC
;; ~~~~~~~~~~~~~~~~~~

(define SYSTEM_6.1+abs+base
  (factor-general 'V-INC-OUT 'ABS-V-INC '(V-INC) SYSTEM_5.2))

(define SYSTEM_6.1 (factor.system SYSTEM_6.1+abs+base))
(define ABS_6.1    (factor.abs    SYSTEM_6.1+abs+base))
(define BASE_6.1   (factor.base   SYSTEM_6.1+abs+base))
```

```
(define SYSTEM_6.2+abs+base
  (factor-general 'V-DEC-OUT 'ABS-V-DEC '(V-DEC) SYSTEM_6.1))

(define SYSTEM_6.2 (factor.system SYSTEM_6.2+abs+base))
(define ABS_6.2    (factor.abs    SYSTEM_6.2+abs+base))
(define BASE_6.2   (factor.base   SYSTEM_6.2+abs+base))

(define SYSTEM_6.3
  (let* ([SYSTEM SYSTEM_6.2]
         ;; Expand V-INC, V-DEC port
         [SYSTEM (expand-stream 'V-INC-OUT-V0 SYSTEM)]
         [SYSTEM (expand-stream 'V-DEC-OUT-V0 SYSTEM)]
         ;; nops should be better named
         [SYSTEM (rename-stream 'V-INC-OUT-INST-NOP 'V-INC-OUT-NOP SYSTEM)]
         [SYSTEM (rename-stream 'V-DEC-OUT-INST-NOP 'V-DEC-OUT-NOP SYSTEM)]
         )
    SYSTEM))

(define BASE_6 (list BASE_1
                     BASE_2.1 ABS_2.1
                     BASE_3.1 ABS_3.1
                     BASE_5.1 ABS_5.1
                     BASE_6.1 ABS_6.1
                     BASE_6.2 ABS_6.2
                     ))

(define SEQSYS_6  (build-seqsys SELECT_1 SYSTEM_6.3 BASE_6))

;; Instantiate don't care values
;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define SYSTEM_7
  (let* ([SYSTEM SYSTEM_6.3]
         [SYSTEM (rename-stream 'MEM-ADDR-?          'OPERAND-B          SYSTEM)]
         [SYSTEM (rename-stream 'REGS-ADDR-?         'PC-REG             SYSTEM)]
         [SYSTEM (rename-stream 'MEM-REGS-DATA-?     'V-INC-OUT          SYSTEM)]
         [SYSTEM (rename-stream 'V-ALU-OUT-CARRYIN-? '(C-FLAG FLAGS)     SYSTEM)]
         [SYSTEM (rename-stream 'V-ALU-OUT-OPCODE-?  '(OP-CODE INS)      SYSTEM)]
         [SYSTEM (rename-stream 'UPDATE-FLAGS-OUT-?
                                '(UPDATE-FLAGS FLAGS (SET-FLAGS INS)
                                               V-ALU-OUT)           SYSTEM)])
    SYSTEM))
```

```
(define SELECT_7 (seqsys.select SEQSYS_6))
(define BASE_7   (seqsys.base   SEQSYS_6))

;; Define next state objects
;; ~~~~~~~~~~~~~~~~~~~~~~~~~

; make next-state & instruction generators

(define STATE         (extract-stream 'STATE        SYSTEM_7))
(define ORACLE        (extract-stream 'ORACLE       SYSTEM_7))

(define NEXT-STATE       (optimize-sel (partial-eval STATE        SELECT_7)))
(define NEXT-ORACLE      (optimize-sel (partial-eval ORACLE       SELECT_7)))

(define V-INC-OUT (extract-stream 'V-INC-OUT SYSTEM_7))
(define V-DEC-OUT (extract-stream 'V-DEC-OUT SYSTEM_7))
(define UPDATE-FLAGS-OUT (extract-stream 'UPDATE-FLAGS-OUT SYSTEM_7))

(define SYSTEM_7 (rename-stream 'DATA-IN-? 'MEM-OUT SYSTEM_7))

(define SYSTEM_7.2
  (remove-streams '(STATE
                    V-ALU-OUT-INST
                    V-ALU-OUT
                    MEM-OUT
                    REGS-OUT
                    V-INC-OUT-INST
                    V-INC-OUT
                    V-DEC-OUT-INST
                    V-DEC-OUT
                    ORACLE
                    )
                  SYSTEM_7))

(define SEQSYS_8 (optimize-seqsys SELECT_7 SYSTEM_7.2))
(define SYSTEM_8 (seqsys.system SEQSYS_8))
(define SELECT_8 (seqsys.select SEQSYS_8))
```

```
;; ***************************************************************************
;; *                           Projection                                    *
;; *                           ----------                                    *
;; *                                                                         *
;; * Transformations applied:                                                *
;; *        project: Project DATAPATH                                        *
;; *    project-sel: Project SELECT                                          *
;; *    project-sel: Project NEXT-STATE                                      *
;; *                                                                         *
;; ***************************************************************************
;; Project DATAPATH
;; ~~~~~~~~~~~~~~~~

(define DATAPATH.BIN
  (sort-streqns
   (project-streqns ddd-fm9001.rep (system.streams DATAPATH) 32)))

(define REGISTERS.BIN
  (append (eval-slice 'STATE     ddd-fm9001.rep)
          (eval-slice 'B-ADDRESS ddd-fm9001.rep)
          (eval-slice 'FLAGS     ddd-fm9001.rep)
          (eval-slice 'INS       ddd-fm9001.rep)
          (eval-slice 'OPERAND-A ddd-fm9001.rep)
          (eval-slice 'OPERAND-B ddd-fm9001.rep)
          (eval-slice 'PC-REG    ddd-fm9001.rep) ))


;; Project NEXT-STATE and SELECT
;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(define SELECT_8.BIN   (project-sel cmd.rep   SELECT_8))
(define NEXT-STATE.BIN (project-sel state.rep NEXT-STATE))
```

Appendix C

The DDD-FM9001 Structural Specification

```
;; ***********************************************************************
;; *                                                                     *
;; * DDD-FM9001: Initial Structural Description                          *
;; *     Author: Bhaskar Bose                                            *
;; *       File: seqsys_1.ss                                             *
;; *                                                                     *
;; ***********************************************************************
(define select
  (lambda* ((s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
           v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18
           v19 v20 v21 v22 v23 v24)
   (case s
     [fm9001-intr (if p0 v0 v1)]
     [fm9001-fetch v2]
     [fm9001-fetch_1 v3]
     [fm9001-fetch_2 v4]
     [fm9001-operand-a v5]
     [fm9001-operand-a_1 (if p1 v6 (if p2 v7 (if p3 v8 v6)))]
     [fm9001-operand-a_2 (if p1 v9 (if p4 v10 (if p2 v11 v12)))]
     [fm9001-operand-b v13]
     [fm9001-operand-b_1 (if p5 v14 (if p6 v15 v16))]
     [fm9001-operand-b_2 (if p7 (if p5 v17 v18) (if p5 v19 v20))]
     [fm9001-alu-operation (if p8 v21 v22)]
     [fm9001-alu-operation_1 (if p9 v23 v24)])))

(define DDD-FM9001
  (lambda (init-regs init-flags init-mem init-pc-reg init-oracle)
    (system-letrec
     ([STATUS
       (XPS STATE (NLISTP ORACLE) (A-IMMEDIATE-P INS) (PRE-DEC-P (MODE-A INS))
      (POST-INC-P (MODE-A INS)) (REG-DIRECT-P (MODE-A INS))
      (PRE-DEC-P (MODE-B INS)) (POST-INC-P (MODE-B INS))
      (REG-DIRECT-P (MODE-B INS))
      (AND* (STORE-RESULTP (STORE-CC INS) FLAGS)
            (REG-DIRECT-P (MODE-B INS)))
      (AND* (STORE-RESULTP (STORE-CC INS) FLAGS)
            (NOT* (REG-DIRECT-P (MODE-B INS)))))]
[STATE
 (! fm9001-intr
    (SELECT STATUS FM9001-INTR FM9001-FETCH FM9001-FETCH_1
            FM9001-FETCH_2 FM9001-OPERAND-A FM9001-OPERAND-A_1
            FM9001-OPERAND-A_2 FM9001-OPERAND-A_2 FM9001-OPERAND-A_2
            FM9001-OPERAND-B FM9001-OPERAND-B FM9001-OPERAND-B
            FM9001-OPERAND-B FM9001-OPERAND-B_1 FM9001-OPERAND-B_2
            FM9001-OPERAND-B_2 FM9001-OPERAND-B_2 FM9001-ALU-OPERATION
            FM9001-ALU-OPERATION FM9001-ALU-OPERATION
            FM9001-ALU-OPERATION FM9001-ALU-OPERATION_1
            FM9001-ALU-OPERATION_1 FM9001-INTR FM9001-INTR))]
[REGS
 (! init-regs
    (SELECT STATUS REGS REGS REGS REGS
            (WRITE-MEM PC-REG REGS (V-INC OPERAND-B)) REGS REGS
            (WRITE-MEM (RN-A INS) REGS (V-DEC OPERAND-B))
            (WRITE-MEM (RN-A INS) REGS (V-INC OPERAND-B)) REGS REGS REGS
            REGS REGS (WRITE-MEM (RN-B INS) REGS (V-DEC OPERAND-B))
            (WRITE-MEM (RN-B INS) REGS (V-INC OPERAND-B)) REGS REGS REGS
            REGS REGS (WRITE-MEM (RN-B INS) REGS
                                 (BV (V-ALU (C-FLAG FLAGS)
                                            OPERAND-A OPERAND-B
                                            (OP-CODE INS))))
            REGS REGS REGS))]
[FLAGS
 (! init-flags
    (SELECT STATUS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS
            FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS
            FLAGS FLAGS FLAGS FLAGS
            (UPDATE-FLAGS FLAGS (SET-FLAGS INS)
             (V-ALU (C-FLAG FLAGS) OPERAND-A OPERAND-B (OP-CODE INS)))
            (UPDATE-FLAGS FLAGS (SET-FLAGS INS)
             (V-ALU (C-FLAG FLAGS) OPERAND-A OPERAND-B (OP-CODE INS)))))]
[MEM
 (! init-mem
    (SELECT STATUS MEM MEM MEM MEM MEM MEM MEM MEM MEM MEM MEM MEM MEM
            MEM MEM MEM MEM MEM MEM MEM MEM MEM MEM
            (WRITE-MEM B-ADDRESS MEM
                       (BV (V-ALU (C-FLAG FLAGS)
                                  OPERAND-A OPERAND-B
                                  (OP-CODE INS)))) MEM))]
[PC-REG
 (! init-pc-reg
    (SELECT STATUS PC-REG (CAR ORACLE) PC-REG PC-REG PC-REG PC-REG
            PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG
            PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG
            PC-REG PC-REG PC-REG))]
[INS
 (! ins-?
    (SELECT STATUS INS INS INS (READ-MEM OPERAND-B MEM) INS INS
            INS INS INS INS INS INS INS INS INS INS INS INS INS
            INS INS INS INS INS INS))]
[OPERAND-A
 (! operand-a-?
    (SELECT STATUS OPERAND-A OPERAND-A OPERAND-A OPERAND-A OPERAND-A
            OPERAND-A OPERAND-A OPERAND-A OPERAND-A
            (SIGN-EXTEND (A-IMMEDIATE INS) THIRTY-TWO) OPERAND-B
            (READ-MEM (V-DEC OPERAND-B) MEM) (READ-MEM OPERAND-B MEM)
            OPERAND-A OPERAND-A OPERAND-A OPERAND-A OPERAND-A OPERAND-A
            OPERAND-A OPERAND-A OPERAND-A OPERAND-A OPERAND-A
            OPERAND-A))]
[OPERAND-B
 (! operand-b-?
    (SELECT STATUS OPERAND-B OPERAND-B (READ-MEM PC-REG REGS) OPERAND-B
            OPERAND-B (READ-MEM (RN-A INS) REGS) OPERAND-B OPERAND-B
            OPERAND-B OPERAND-B OPERAND-B OPERAND-B OPERAND-B
            (READ-MEM (RN-B INS) REGS) OPERAND-B OPERAND-B OPERAND-B
            OPERAND-B OPERAND-B (READ-MEM (V-DEC OPERAND-B) MEM)
            (READ-MEM OPERAND-B MEM) OPERAND-B OPERAND-B OPERAND-B
            OPERAND-B))]
[B-ADDRESS
 (! b-address-?
    (SELECT STATUS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS
            B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS
            B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS
            (V-DEC OPERAND-B) OPERAND-B (V-DEC OPERAND-B) OPERAND-B
            B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS))]
 [ORACLE
  (! init-oracle
     (SELECT STATUS ORACLE (CDR ORACLE) ORACLE ORACLE ORACLE ORACLE
             ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE
             ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE
             ORACLE ORACLE ORACLE))]
 )
(XPS STATUS STATE REGS FLAGS MEM PC-REG INS OPERAND-A OPERAND-B
     B-ADDRESS ORACLE))))
```

```
;; *************************************************************************
;; *                                                                       *
;; * DDD-FM9001: Refined Structural Description                            *
;; *     Author: Bhaskar Bose                                              *
;; *       File: seqsys_6.3.ss                                             *
;; *                                                                       *
;; *************************************************************************
(define select
  (lambda* ((s p0 p1 p2 p3 p4 p5 p6 p7 p8 p9)
           v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18
           v19 v20 v21 v22 v23 v24)
   (case s
     [fm9001-intr (if p0 v0 v1)]
     [fm9001-fetch v2]
     [fm9001-fetch_1 v3]
     [fm9001-fetch_2 v4]
     [fm9001-operand-a v5]
     [fm9001-operand-a_1 (if p1 v6 (if p2 v7 (if p3 v8 v6)))]
     [fm9001-operand-a_2 (if p1 v9 (if p4 v10 (if p2 v11 v12)))]
     [fm9001-operand-b v13]
     [fm9001-operand-b_1 (if p5 v14 (if p6 v15 v16))]
     [fm9001-operand-b_2 (if p7 (if p5 v17 v18) (if p5 v19 v20))]
     [fm9001-alu-operation (if p8 v21 v22)]
     [fm9001-alu-operation_1 (if p9 v23 v24)])))


(define DDD-FM9001
  (lambda (init-regs init-flags init-mem init-pc-reg init-oracle)
    (system-letrec
     ([STATUS
       (XPS STATE (NLISTP ORACLE) (A-IMMEDIATE-P INS) (PRE-DEC-P (MODE-A INS))
            (POST-INC-P (MODE-A INS)) (REG-DIRECT-P (MODE-A INS))
            (PRE-DEC-P (MODE-B INS)) (POST-INC-P (MODE-B INS))
            (REG-DIRECT-P (MODE-B INS))
            (AND* (STORE-RESULTP (STORE-CC INS) FLAGS)
                  (REG-DIRECT-P (MODE-B INS)))
            (AND* (STORE-RESULTP (STORE-CC INS) FLAGS)
                  (NOT* (REG-DIRECT-P (MODE-B INS)))))]
      [STATE
       (! fm9001-intr
          (SELECT STATUS FM9001-INTR FM9001-FETCH FM9001-FETCH_1
                  FM9001-FETCH_2 FM9001-OPERAND-A FM9001-OPERAND-A_1
            FM9001-OPERAND-A_2 FM9001-OPERAND-A_2 FM9001-OPERAND-A_2
            FM9001-OPERAND-B FM9001-OPERAND-B FM9001-OPERAND-B
            FM9001-OPERAND-B FM9001-OPERAND-B_1 FM9001-OPERAND-B_2
            FM9001-OPERAND-B_2 FM9001-OPERAND-B_2 FM9001-ALU-OPERATION
            FM9001-ALU-OPERATION FM9001-ALU-OPERATION
            FM9001-ALU-OPERATION FM9001-ALU-OPERATION_1
            FM9001-ALU-OPERATION_1 FM9001-INTR FM9001-INTR))]
[FLAGS
 (! init-flags
    (SELECT STATUS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS
            FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS FLAGS
            FLAGS FLAGS FLAGS FLAGS UPDATE-FLAGS-OUT UPDATE-FLAGS-OUT))]
[PC-REG
 (! init-pc-reg
    (SELECT STATUS PC-REG (CAR ORACLE) PC-REG PC-REG PC-REG PC-REG
            PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG
            PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG PC-REG
            PC-REG PC-REG PC-REG))]
[INS
 (! ins-?
    (SELECT STATUS INS INS INS MEM-REGS-DATA INS INS INS INS INS INS INS
            INS INS INS INS INS INS INS INS INS INS INS INS INS INS))]
[OPERAND-A
 (! operand-a-?
    (SELECT STATUS OPERAND-A OPERAND-A OPERAND-A OPERAND-A OPERAND-A
            OPERAND-A OPERAND-A OPERAND-A OPERAND-A
            (SIGN-EXTEND (A-IMMEDIATE INS) THIRTY-TWO) OPERAND-B
            MEM-REGS-DATA MEM-REGS-DATA OPERAND-A OPERAND-A OPERAND-A
            OPERAND-A OPERAND-A OPERAND-A OPERAND-A OPERAND-A OPERAND-A
            OPERAND-A OPERAND-A OPERAND-A))]
[OPERAND-B
 (! operand-b-?
    (SELECT STATUS OPERAND-B OPERAND-B MEM-REGS-DATA OPERAND-B OPERAND-B
            MEM-REGS-DATA OPERAND-B OPERAND-B OPERAND-B OPERAND-B
            OPERAND-B OPERAND-B OPERAND-B MEM-REGS-DATA OPERAND-B
            OPERAND-B OPERAND-B OPERAND-B OPERAND-B MEM-REGS-DATA
            MEM-REGS-DATA OPERAND-B OPERAND-B OPERAND-B OPERAND-B))]
[B-ADDRESS
 (! b-address-?
    (SELECT STATUS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS
            B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS
            B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS B-ADDRESS
            V-DEC-OUT OPERAND-B V-DEC-OUT OPERAND-B B-ADDRESS B-ADDRESS
            B-ADDRESS B-ADDRESS))]
[ORACLE
 (! init-oracle
    (SELECT STATUS ORACLE (CDR ORACLE) ORACLE ORACLE ORACLE ORACLE
            ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE
            ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE ORACLE
            ORACLE ORACLE ORACLE))]
[[MEM MEM-OUT] (ABS-MEM init-mem MEM-INST MEM-ADDR MEM-REGS-DATA)]
[MEM-INST
 (SELECT STATUS MEM-NOP MEM-NOP MEM-NOP MEM-READ-MEM MEM-NOP MEM-NOP
         MEM-NOP MEM-NOP MEM-NOP MEM-NOP MEM-NOP MEM-READ-MEM
         MEM-READ-MEM MEM-NOP MEM-NOP MEM-NOP MEM-NOP MEM-NOP MEM-NOP
         MEM-READ-MEM MEM-READ-MEM MEM-NOP MEM-NOP MEM-WRITE-MEM
         MEM-NOP)]
[MEM-ADDR
 (SELECT STATUS MEM-ADDR-? MEM-ADDR-? MEM-ADDR-? OPERAND-B MEM-ADDR-?
         MEM-ADDR-? MEM-ADDR-? MEM-ADDR-? MEM-ADDR-? MEM-ADDR-?
         MEM-ADDR-? V-DEC-OUT OPERAND-B MEM-ADDR-? MEM-ADDR-?
         MEM-ADDR-? MEM-ADDR-? MEM-ADDR-? MEM-ADDR-? V-DEC-OUT
         OPERAND-B MEM-ADDR-? MEM-ADDR-? B-ADDRESS MEM-ADDR-?)]
[[REGS REGS-OUT] (ABS-REGS init-regs REGS-INST REGS-ADDR MEM-REGS-DATA)]
[REGS-INST
 (SELECT STATUS REGS-NOP REGS-NOP REGS-READ-MEM REGS-NOP REGS-WRITE-MEM
         REGS-READ-MEM REGS-NOP REGS-WRITE-MEM REGS-WRITE-MEM REGS-NOP
         REGS-NOP REGS-NOP REGS-NOP REGS-READ-MEM REGS-WRITE-MEM
         REGS-WRITE-MEM REGS-NOP REGS-NOP REGS-NOP REGS-NOP REGS-NOP
         REGS-WRITE-MEM REGS-NOP REGS-NOP REGS-NOP)]
[REGS-ADDR
 (SELECT STATUS REGS-ADDR-? REGS-ADDR-? PC-REG REGS-ADDR-? PC-REG
         (RN-A INS) REGS-ADDR-? (RN-A INS) (RN-A INS) REGS-ADDR-?
         REGS-ADDR-? REGS-ADDR-? REGS-ADDR-? (RN-B INS) (RN-B INS)
         (RN-B INS) REGS-ADDR-? REGS-ADDR-? REGS-ADDR-? REGS-ADDR-?
         REGS-ADDR-? (RN-B INS) REGS-ADDR-? REGS-ADDR-? REGS-ADDR-?)]
[MEM-REGS-DATA
 (SELECT STATUS MEM-REGS-DATA-? MEM-REGS-DATA-? DATA-IN DATA-IN
         V-INC-OUT DATA-IN MEM-REGS-DATA-? V-DEC-OUT V-INC-OUT
         MEM-REGS-DATA-? MEM-REGS-DATA-? DATA-IN DATA-IN DATA-IN
         V-DEC-OUT V-INC-OUT MEM-REGS-DATA-? MEM-REGS-DATA-?
         MEM-REGS-DATA-? DATA-IN DATA-IN (BV V-ALU-OUT) MEM-REGS-DATA-?
         (BV V-ALU-OUT) MEM-REGS-DATA-?)]
[DATA-IN
```

```
 (SELECT STATUS DATA-IN-? DATA-IN-? REGS-OUT MEM-OUT DATA-IN-? REGS-OUT
         DATA-IN-? DATA-IN-? DATA-IN-? DATA-IN-? DATA-IN-? MEM-OUT
         MEM-OUT REGS-OUT DATA-IN-? DATA-IN-? DATA-IN-? DATA-IN-?
         DATA-IN-? MEM-OUT MEM-OUT DATA-IN-? DATA-IN-? DATA-IN-?
         DATA-IN-?)]
[DATA-OUT-ENABLE
 (SELECT STATUS DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-FF
         DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-TT DATA-OUT-ENABLE-FF
         DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-TT DATA-OUT-ENABLE-TT
         DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-FF
         DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-TT
         DATA-OUT-ENABLE-TT DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-FF
         DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-FF
         DATA-OUT-ENABLE-TT DATA-OUT-ENABLE-FF DATA-OUT-ENABLE-TT
         DATA-OUT-ENABLE-FF)]
[MEM-TRISTATE
 (SELECT STATUS MEM-TRISTATE-TT MEM-TRISTATE-FF MEM-TRISTATE-FF
         MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF
         MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF
         MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF
         MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF
         MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF MEM-TRISTATE-FF
         MEM-TRISTATE-FF MEM-TRISTATE-FF)]
[UPDATE-FLAGS-OUT
 (SELECT STATUS UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-?
         UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-?
         UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-?
         UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-?
         UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-?
         UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-?
         UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-?
         UPDATE-FLAGS-OUT-? UPDATE-FLAGS-OUT-?
         (UPDATE-FLAGS FLAGS (SET-FLAGS INS) V-ALU-OUT)
         (UPDATE-FLAGS FLAGS (SET-FLAGS INS) V-ALU-OUT))]
[V-ALU-OUT-INST
 (SELECT STATUS V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP
         V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP
         V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP
         V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP
         V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP V-ALU-OUT-NOP
         V-ALU-OUT-NOP V-ALU-OUT-V-ALU V-ALU-OUT-NOP V-ALU-OUT-V-ALU
         V-ALU-OUT-V-ALU)]
 [V-ALU-OUT-CARRYIN
  (SELECT STATUS V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-?
          V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-?
          V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-?
          V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-?
          V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-?
          V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-?
          V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-? V-ALU-OUT-CARRYIN-?
          V-ALU-OUT-CARRYIN-? (C-FLAG FLAGS) V-ALU-OUT-CARRYIN-?
          (C-FLAG FLAGS) (C-FLAG FLAGS))]
 [V-ALU-OUT-OPCODE
  (SELECT STATUS V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-?
          V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-?
          V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-?
          V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-?
          V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-?
          V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-?
          V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-? V-ALU-OUT-OPCODE-?
          (OP-CODE INS) V-ALU-OUT-OPCODE-? (OP-CODE INS) (OP-CODE INS))]
 [V-ALU-OUT
  (ABS-V-ALU
   V-ALU-OUT-INST V-ALU-OUT-CARRYIN OPERAND-A OPERAND-B V-ALU-OUT-OPCODE)]
 [V-INC-OUT-INST
  (SELECT STATUS V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-NOP
          V-INC-OUT-V-INC V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-NOP
          V-INC-OUT-V-INC V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-NOP
          V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-V-INC
          V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-NOP
          V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-NOP V-INC-OUT-NOP
          V-INC-OUT-NOP)]
 [V-INC-OUT (ABS-V-INC V-INC-OUT-INST OPERAND-B)]
 [V-DEC-OUT-INST
  (SELECT STATUS V-DEC-OUT-NOP V-DEC-OUT-NOP V-DEC-OUT-NOP V-DEC-OUT-NOP
          V-DEC-OUT-NOP V-DEC-OUT-NOP V-DEC-OUT-NOP V-DEC-OUT-V-DEC
          V-DEC-OUT-NOP V-DEC-OUT-NOP V-DEC-OUT-NOP V-DEC-OUT-V-DEC
          V-DEC-OUT-NOP V-DEC-OUT-NOP V-DEC-OUT-V-DEC V-DEC-OUT-NOP
          V-DEC-OUT-NOP V-DEC-OUT-V-DEC V-DEC-OUT-NOP V-DEC-OUT-V-DEC
          V-DEC-OUT-NOP V-DEC-OUT-NOP V-DEC-OUT-NOP V-DEC-OUT-NOP
          V-DEC-OUT-NOP)]
 [V-DEC-OUT (ABS-V-DEC V-DEC-OUT-INST OPERAND-B)]
 )
(XPS STATUS STATE FLAGS PC-REG INS OPERAND-A OPERAND-B B-ADDRESS ORACLE
MEM-OUT MEM-INST MEM-ADDR REGS-OUT REGS-INST REGS-ADDR
MEM-REGS-DATA DATA-IN DATA-OUT-ENABLE MEM-TRISTATE
UPDATE-FLAGS-OUT V-ALU-OUT-INST V-ALU-OUT-CARRYIN V-ALU-OUT-OPCODE
V-ALU-OUT V-INC-OUT-INST V-INC-OUT V-DEC-OUT-INST V-DEC-OUT MEM REGS))))
```

```
;; ***********************************************************************
;; *                                                                     *
;; *                  Abstract Component Specifications                  *
;; *                  ---------------------------------                  *
;; *                                                                     *
;; *  Abstract components:                                               *
;; *     ABS-MEM: memory            ABS-V-ALU: alu                       *
;; *    ABS-REGS: register file     ABS-V-INC: incrementor               *
;; *                                ABS-V-DEC: decrementor               *
;; *                                                                     *
;; ***********************************************************************

;; MEM Abstract Component
;; ~~~~~~~~~~~~~~~~~~~~~~
(define ABS-MEM
  (lambda (*mem* INST V0 V1)
    (let ([constructor
           (lambda (inst object v0 v1)
             (case inst
               [mem-nop object]
               [mem-read-mem object]
               [mem-write-mem (write-mem v0 object v1)]))]
          [probe
           (lambda (inst object v0)
             (case inst
               [mem-nop       (read-mem v0 object)]
               [mem-write-mem (read-mem v0 object)]
               [mem-read-mem  (read-mem v0 object)]))])
      (system-letrec
       ([OBJECT (! *mem* ((stream constructor) INST OBJECT V0 V1))]
        [PROBE ((stream probe) INST OBJECT V0)])
       (XPS OBJECT PROBE)))))
```

```
;; REGS Abstract Component
;; ~~~~~~~~~~~~~~~~~~~~~~~
(define ABS-REGS
   (lambda (*regs* INST V0 V1)
      (let ([constructor
             (lambda (inst object v0 v1)
               (case inst
                   [regs-nop object]
                   [regs-read-mem object]
                   [regs-write-mem (write-mem v0 object v1)]))]
            [probe
             (lambda (inst object v0)
               (case inst
                   [regs-nop       (read-mem v0 object)]
                   [regs-write-mem (read-mem v0 object)]
                   [regs-read-mem  (read-mem v0 object)]))])
         (system-letrec
            ([OBJECT (! *regs* ((stream constructor) INST OBJECT V0 V1))]
             [PROBE ((stream probe) INST OBJECT V0)])
            (XPS OBJECT PROBE)))))
```

```
;; V-ALU Abstract Component
;; ~~~~~~~~~~~~~~~~~~~~~~~~
(define ABS-V-ALU
  (lambda (INST V0 V1 V2 V3)
    (let ([constructor
           (lambda (inst v0 v1 v2 v3)
             (case inst
               [v-alu-out-nop   (v-alu v0 v1 v2 v3)]
               [v-alu-out-v-alu (v-alu v0 v1 v2 v3)]))])
      (system-letrec
       ([OBJECT ((stream constructor) INST V0 V1 V2 V3)])
       OBJECT))))

;; V-INC Abstract Component
;; ~~~~~~~~~~~~~~~~~~~~~~~~
(define ABS-V-INC
  (lambda (INST V0)
    (let ([constructor
           (lambda (inst v0)
             (case inst
               [v-inc-out-nop (v-inc v0)]
               [v-inc-out-v-inc (v-inc v0)]))])
      (system-letrec
       ((OBJECT ((stream constructor) INST V0)))
       OBJECT))))

;; V-DEC Abstract Component
;; ~~~~~~~~~~~~~~~~~~~~~~~~
(define ABS-V-DEC
  (lambda (INST V0)
    (let ([constructor
           (lambda (inst v0)
             (case inst
               [v-dec-out-nop (v-dec v0)]
               [v-dec-out-v-dec (v-dec v0)]))])
      (system-letrec
       ((OBJECT ((stream constructor) INST V0)))
       OBJECT))))
```