Technical Report No. 323

DDD: A System for Mechanized Digital Design Derivation

by

Steven D. Johnson
Bhaskar Bose

December 1990
Revised March 1997

Computer Science Department
Indiana University
Bloomington, Indiana 47405-4101

DDD – A System for Mechanized Digital Design Derivation[1]

Steven D. Johnson and Bhaskar Bose
Indiana University Computer Science Department

December 1990
Revised March 1997[2]

[1]This research was supported, in part, by the National Science Foundation under grants numbered DCR85-21497, MIP87-07067, and MIP89-21842.

[2]This paper was presented at the *1991 IFIP and SIGDA Workshop on Formal Methods for VLSI Design*, held at Miami Florida in January 1991. A procedings of this workshop was never formally published, although a participants' paper set has been widely distributed. This revision was reformatted for Latex from the original's Tex source. The reformatting necessitated some superficial changes, but there are no intentional content changes.

Abstract

Our research group at Indiana University is investigating a formalization of digital system design that is based on functional algebra. We have developed a transformation system called *DDD* to facilitate this study. DDD stands for *digital design derivation*; the system is used interactively to translate higher level specifications into hierarchical boolean systems, to which logic synthesis tools are then applied. In this paper, we take a detailed look at how the system is used. In two examples, we examine the sequence of intermediate expressions produced as an implementation is derived. We discuss how these expressions are used at strategic levels of thinking. We illustrate how the choice of target technology influences the tactical course of derivation. Throughout, we try to give a sense of how functional abstractions are manipulated in the engineering process.

1 Introduction

The *DDD transformation system* [2] was developed to help us explore and demonstrate a formalization of digital design based on functional algebra. DDD stands for *digital design derivation*; the system is used interactively to transform higher level behavioral specifications into hierarchical boolean systems, to which logic synthesis tools are then applied. It operates on a dialect of functional modeling expressions, providing a uniform and visible representation of the design. Previous articles develop the theoretical underpinnings of the formalization [14, 13, 15, 16, 33, 34, 35]; and mention DDD in the context of large examples [17, 3, 18]. However, these papers give few details about the experience of using the system. Our goal in this paper is to give a more complete picture of the derivation process. With two small examples we examine the sequence of intermediate expressions produced as an implementation is derived. We discuss how these expressions are used at strategic levels of thinking and illustrate how the choice of target technology influences the tactical course of derivation. We try to give a sense of how abstractions—principally data abstractions—are manipulated in the course of design.

Although we consider algebraic derivation to be a candidate framework for high-level synthesis, we do not classify DDD as a synthesis system in the engineering sense. Currently, it performs little automatic optimization beyond that provided by the lower level CAD tools with which it is integrated. DDD does not yet "explore the design space." It automates the basic transformations needed to conduct synthesis. It has more in common with a basic theorem proving system, which automates the inference rules of a logic but requires substantial guidance to perform a proof. DDD assures correct application of the algebra, but requires substantial guidance to perform a design.

Neither do we regard the medium of DDD derivation—a simple functional modeling language—to be a hardware description language. While the expressions DDD manipulates are adequate for describing aspects of behavior and structure, more efficient notations are easy to devise. Furthermore these formulas do not explicitly contain design constraints, which instead are implied by the transformation commands of DDD. At best it is the combination of the derivation language and the manipulated expressions that describe the hardware. The true benefit of the notation, aside from its simplicity, is that it reflects a unified mathematical basis for the exposition of methods and the

rigorous description of design tools.

Section 2 is a survey of related research. Section 3 gives a summary of the syntax of functional Scheme expressions on which DDD operates (Scheme is a dialect of Lisp). Section 4 is an intuitive discussion of design derivation, sketching the typical phases of a DDD application in moving from behavior to structure and then to physical organization. The first example, in Section 5, details a derivation of the *Black-Jack dealer* described by Winkel and Prosser in their text book on digital design [25]. This simple example illustrates most aspects of the derivation but not its generality. In Section 6, we take a fragment of a larger specification through the early phases of derivation in order to illustrate how a more sophisticated data hierarchy is managed in DDD. In Section 7, we discuss future refinements and reexamine our goals for DDD in light of the examples.

2 Related Research

The use and manipulation of functional expressions is common in hardware applications. A number of hardware description languages (HDLs) are functional; *Silage* [11] and *ELLA* [23] are examples. The majority of structural HDLs are essentially functional. Functional calculus is also prominent in formal-methods research, where it vies for prominence with predicate, relational and process calculi.

Our approach is closely related to that of Sheeran, who has also developed functional algebra for manipulating hardware descriptions. However, there are differences both in notation and in emphasis. Sheeran began with an *FP* variant [28] which she later refined to a relational notation called *Ruby* [29]. This is a fitting choice for the regular structures she is most concerned with. The DDD algebra is specialized for manipulating data paths under sequential control. We regard Sheeran's work and ours to be compatible treatments of distinct intervals of the design spectrum.

In terms of abstract syntax, two related languages for hardware modeling are *SBL*, due to Gopalakrishnan, Srivas, and Smith [10], and *HOP*, developed later by Gopalakrishnan [9]. DDD operates on a more primitive concrete syntax of s-expressions (See Section 4), but a more important difference is the treatment of data abstraction. Data are encapsulated in SBL, and hardware modularity is strongly correlated to that encapsulation. DDD adopts a more open treatment of types, viewing them as logical structures

rather than physical modules.

One goal of our research is to explore interactions between synthesis and verification at higher levels of design specification. DDD manipulates the same language of first-order Lisp expressions that is used in "Boyer-Moore logic" [4]. In [18], we applied DDD directly to Hunt's mechanically verified FM8501 microprocessor descriptions [12]. The exercise demonstrated the interplay of two forms of automated reasoning; and in particular, showed that derivation (i.e. synthesis in a formal system) could obviate a significant portion of the correctness proof. Conversely, a synthesis system can maintain correctness only when given correct information to work with. There is a clear supporting role for verification in a synthesis oriented design.

A potential benefit of performing synthesis on expressions of a symbolic processing language is that these expressions can also be executed to model the design. Each of the intermediate expressions generated by DDD is a valid Scheme program, used to simulate and explore the implementation. O'Donnell's *HYDRA* system [24] is a related study of this aspect of methodology.

3 Terminology, Syntax, and Elementary Algebra

Our approach formalizes synthesis as a translation between dialects of a single modeling language of functional expressions. We use the term *derivation* to emphasize the character of this translation. A *design derivation* is a sequence of expressions.

$$\mathcal{E}_0 \xrightarrow{T_1} \mathcal{E}_1 \xrightarrow{T_2} \cdots \xrightarrow{T_k} \mathcal{E}_k$$

We sometimes call source expression $\mathcal{E}_0$ a *specification* and target expression $\mathcal{E}_k$ an *implementation*; and we reserve the word *realization* for the resulting physical circuit. It is most accurate to say that a design is specified by the sequence of equivalence preserving transformations $\langle T_0, \ldots, T_k \rangle$ applied to $\mathcal{E}_0$. Such a sequence expresses much of the design intent and should, therefore, be thought of as part of the specification. There is more about the general character of derivation in Section 4.

Since the structure of an expression is a useful interpretation of its meaning, no useful transformation can preserve correctness in all respects. To say

that a transformation is "equivalence preserving" means that it preserves a primary interpretation of an expression as a function.

The primitive vocabulary of a description, its *ground type* or *basis*, is a many sorted algebra [8]. This is a system of primitive constants, operations and tests, which in DDD always includes a boolean domain, a polymorphic selection operation, finite product formation, and a generic don't-care value, designated by '`?`'. Most of the DDD algebra is valid for uninterpreted bases; that is, it applies to any ground type and is peculiar to none.

With further development, we plan for type-specific knowledge (e.g. equational rewriting laws) to be programmable in DDD. We are in the process of integrating DDD with a type inference system in order to support this kind of extension. Although DDD does not support explicit type declarations, we claim that it preserves type consistency: if DDD is given a well-typed expression to begin with, then its transformations produce a well-typed expression.

3.1 Concrete Syntax: S-expressions

DDD manipulates a concrete syntax of functional *s-expressions* in the Lisp dialect *Scheme*. The s-expressions in this paper are slightly stylized in order to make them easier to read, but they are not far from actual representations seen in DDD. There is, however, one significant extension to legal Scheme syntax: nested formal parameters are allowed. We permit them because hardware description often involves multiple-valued functions. This extension is readily dealt with in an expansion package, such as Kohlbecker's [20].

Scheme is a statically scoped, applicative order dialect of Lisp, whose functional sublanguage is an implementation of a typed lambda calculus (i.e., a lambda calculus extended by primitive symbolic operations). A complete language definition can be found in [27]. It's equally important symbolic processing capability is thoroughly developed in several text books, such as [7], [30] and [1]. In the last of these a number of hardware modeling techniques are presented.

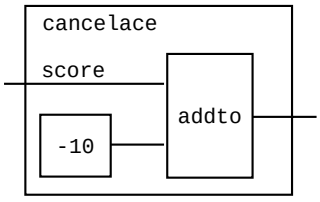
In the informal language summary that follows, upper case variables refer to expressions and lower case variables refer to values; the same letter associates the two: expression E has value e.

Scheme's symbolic processing subsystem is a space of *lists,* denoted by parenthesized sequences, $(\ell_1\ \ell_2\ \cdots\ \ell_n)$. Atomic objects include numbers and alphanumeric *names.* Unless F is a reserved symbol, the expression $(F\ E_1\ E_2\ \cdots\ E_n)$ denotes application of a primitive or defined function f

(i.e. F's value) to the actual arguments $e_1,\ e_2,\ \ldots, e_n$ (i.e., the respective values of the E_is).

A *function expression* has the form `(` $\lambda\, X\ E$`)`, where X is a formal parameter and E is an expression. As mentioned earlier, we allow X to be a nested list of identifiers where in standard Scheme, only a flat list is allowed. Lambda-expressions are used in the same way that higher level symbols are used in schematics: to suppress detail by enclosure in a "box." Since DDD does not at this time freely manipulate functional values, lambda-expressions are seen only in function definitions. The top-level command `(define` I `(` $\lambda\, X\ E$`))` assigns name I to the function-value of `(` $\lambda\, X\ E$`)`. Omitting some of the parentheses we write, for example,

```
define cancelace =
  λ(score) (addto score -10)
```



Two forms of conditional expression are used. An *if-then-else* expression has the form `(if` $T\ E_1\ E_2$`)`; it returns e_1 where test value t is true and e_2 where t is false. The `case` expression `(case` T `((`$v_1\ E_1$`) ... (`$v_n\ E_n$`)))` returns the value e_i whose corresponding label, v_i, equals t. The `case` labels are not evaluated.

Local definitions are made with `let`, `letrec`, and `system` expressions, which all have a similar form.

`let`	`letrec`	`system`
I_1 `=` E_1	F_1 `=` $\lambda\ X_1\ E_1$	X_1 `=` S_1
$\vdots$	$\vdots$	$\vdots$
I_n `=` E_n	F_n `=` $\lambda\ X_n\ E_n$	X_n `=` S_n
`in`	`in`	`in`
E	E	E

The `let` expression binds each name I_k to the value e_k and evaluates the *body* E. If `let` is replaced by `letrec`, the defining expressions are mutually recursive. As in the example above, `letrec` is used to make simultaneous systems of function definitions; forms like this are used for behavioral specification in DDD. Another form of recursion is used for structural description. In the primary interpretation, `system` expressions describe networks of nonfinite sequences, traditionally called *streams*. Streams are defined by a restricted

language of *signal expressions*, built from simple terms of the ground type and a *delay* function denoted by '!'. If signal expression S denotes the sequence $\langle s_1, s_2, \ldots \rangle$, then

$$\texttt{(!}\ E\ \ S\texttt{)}\ \text{denotes}\ \langle e, s_1, s_2, \ldots \rangle$$

and if `F` is a two-place operation symbol (for example)

$$\texttt{(F}\ \ S\ \ S'\texttt{)}\ \text{denotes}\ \langle f(s_1, s'_1), f(s_2, s'_2), \ldots \rangle.$$

Typically, some or all of the stream definitions in a `system` expression have a single, outermost delay operator with an undetermined initial value, X `= (! ?` S`)`. For brevity, we abbreviate these equations to

$$X \stackrel{\bullet}{=} S.$$

`System` expressions are also interpreted graphically as schematics, as for example in Figures 2 and 7.

3.2 Basic Algebra of Functional Expressions

One of the most basic laws of functional algebra asserts that a lambda expression can be expanded where it is applied; and conversely, that a term can be abstracted to the application of a lambda-expression. If $F \equiv$ $\lambda\,\texttt{(}X_1 \cdots X_n\texttt{)}\ \ E$ then

$$\texttt{(}F\ \ A_1 \cdots A_n\texttt{)} \Longleftrightarrow E[A_i/X_i],$$

where $E[A_i/X_i]$ is the expression that results if (free occurrences of) parameters X_i are simultaneously replaced by argument expressions A_i, $1 \leq i \leq n$. We sometimes say that the lambda-expression *encapsulates* the term E. In recursive systems, expansion and encapsulation are sometimes called *unfolding* and *folding*.

A *distributive law for conditionals* states that `if`-expressions and `case`-expressions can be distributed across function calls:

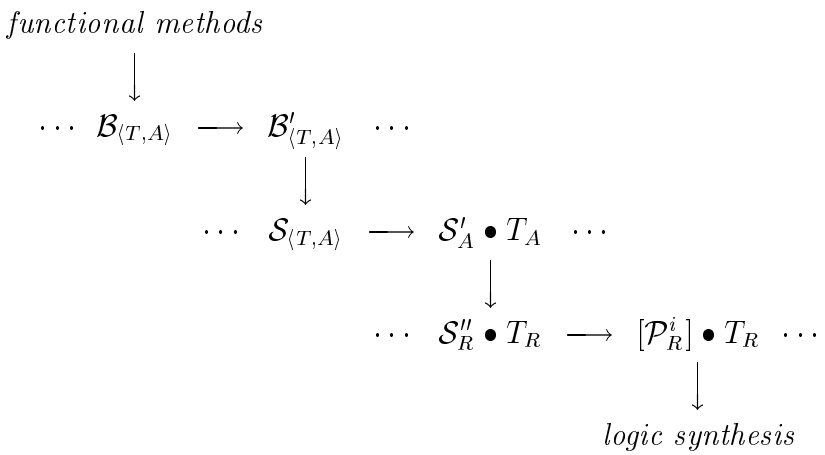
$$\texttt{(if}\ \ T\ \ \texttt{(}F\ \ A\texttt{)}\ \ \texttt{(}F\ \ B\texttt{))} \Longleftrightarrow \texttt{(}F\ \ \texttt{(if}\ \ T\ \ A\ \ B\texttt{))},$$

subject to the condition that either T be defined or F be strict (i.e., dependent on an argument).

Recursive `letrec` and `system` expressions are manipulated in the manner of other simultaneous systems of equations. In addition to fold/unfold transformations, these systems may be rearranged and reorganized hierarchically. A family of basic laws is presented in [16].

4 Sketch of the Derivation Process

The object is to find a *derivation*, or sequence of DDD transformations, resulting in an implementation that satisfies the intended design constraints. There may be many possible derivation paths, of course, but in practice the process has distinct phases. In this section, we expand on the characterization of derivation given at the beginning of Section 3. In the diagram below, $\mathcal{X}_B$ stands for a DDD expression $\mathcal{X}$ written in terms of a ground type B.



Ideally, an arbitrary (recursive, higher order, etc.) expression could serve as a starting point; but for the present, it is a functional-programming problem to obtain a form that DDD can handle. The initial behavior description $\mathcal{B}_{\langle T,A\rangle}$ is in a restricted class of *iterative* function-definition schemes. Typically, the ground type $\langle T, A\rangle$ contains complex (T) and simple (A) components. For example, a memory has address and content parameters, a stack may have memory and *address* components, and so forth.

Behavior descriptions are folded and unfolded ($\mathcal{B}_{\langle T,A\rangle} \rightarrow \mathcal{B}'_{\langle T,A\rangle}$) to achieve a proper scheduling of operations. Section 6 shows an example of this kind of transformation. From a suitable behavior description, DDD automatically builds an abstract `system` description expressed over the same ground type ($\mathcal{B}'_{\langle T,A\rangle} \rightarrow \mathcal{S}_{\langle T,A\rangle}$).

Considered as a structural description, $\mathcal{S}_{\langle T,A\rangle}$ is abstract because it may include signals ranging over complex components of the ground type. A sequence of *factorization* steps ($\mathcal{S}_{\langle T,A\rangle} \rightarrow \mathcal{S}'_A \bullet T_A$) decomposes $\mathcal{S}$ into a system of modules, encapsulating complex objects as coprocesses, (e.g. T_A). The resulting expression is reduced to terms of the simpler type, A.

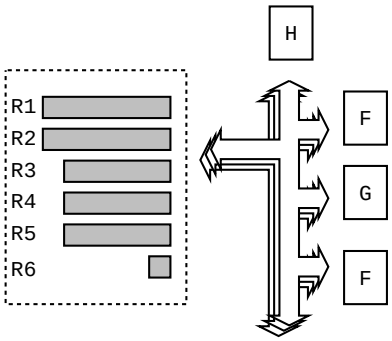
A third class of transformations incorporates a lower-level representation, R, in place of A ($\mathcal{S}'_A \rightarrow \mathcal{S}''_R$). Then, $\mathcal{S}''_R$ is partitioned into synthesizable subsystems, $\mathcal{P}^1_R, \mathcal{P}^2_R, \ldots$. Ultimately, this decomposition produces a hierarchy of boolean subsystems. Logic synthesis is used to assemble the subsystems to the appropriate technology.

4.1 Behavior to Structure

Derivation usually starts with a specification expressed as an *iterative* system of function definitions as depicted to the right. "Iterative" means that the system is entirely tail-recursive: DDD control specifications are essentially finite-state machine descriptions. The outer parameters $Y_1, \ldots, Y_k$ usually stand for communication ports, as explained in Section 5. The local parameters, $R_1, \ldots, R_n$ are usually interpreted as registers. Function call $(F_i\ T_1\ \cdots\ T_n)$ is interpreted as a simultaneous register update with values $T_1, \ldots, T_n$, together with a transfer of control to F_i. This interpretation corresponds quite closely to the state transition systems of Brown and Leeser [5]. As we shall see in Section 6, this form may be transformed in order to satisfy scheduling constraints.

```
define B = λ(Y1 ··· Yk)
   letrec
      F1 = λ(R1 ··· Rn) E1
         ⋮
      Fm = λ(R1 ··· Rn) Em
   in
      (Fstart ···)
```

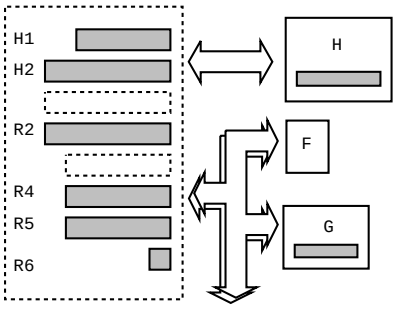
The initial DDD transformations decompose the specification into a `system` expression, depicting a controller operating against an architecture which is abstract in several respects. It may not detail the functional modules of a structural description, nor does it reflect constraints on the data path. The level of description may be far above that of an implementation. The diagram to the right illustrates aspects of its structure. Register entities (`R1`, `R2`, *etc.*) are connected to various ground operations (`H`, `F`, and `G` in the figure). A synthesized *decision combination*, not explicitly shown, governs data movement. Typically, there is a high degree of parallelism in data trans-



fer. The initial system, though correct, cannot be reduced to a realization immediately. It might contain, for example, abstract registers ranging over complex objects, such as queues or memories. Thus, it is primarily a formal structure upon which a true architecture still must be imposed.

4.2 Structure to Architecture

The next stage of derivation develops a logical organization analogous to a functional block diagram. Base operations are encapsulated in modules, and peripheral processes are factored out of the description.

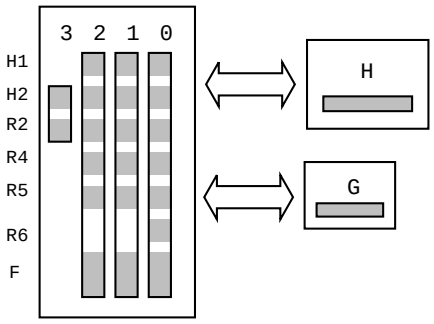


The diagram to the right illustrates the effects of these manipulations. Multiple instances of the common subexpression `F` are allocated to a single device. The register `R3` is incorporated with the function `G`; building a counter out of an incrementer and a register is an example. An important use of factorization corresponds to information hiding. A complex object may be encapsulated as a process entity with state. Such transformations may introduce new interface registers—in the diagram, `H1` and `H2` result when `R1` is incorporated with `H`. For example, a factored memory might leave residual address and content registers. Failures in factorization induce refinements to control, which are accomplished by *serializing* transformations on the behavioral description. These are discussed in Section 6.

4.3 Architecture to Physical Organization

Manipulations of the logical organization produce a target description that is much closer to the intended physical implementation. However, the architecture described may be more conceptual than practical. The final stages of derivation are concerned with correctly reorganizing the design for the purpose of logic synthesis. The goal is to partition the description into synthesizable subsystems. The final phase of a derivation introduces (more) concrete representations for the constants and operations of the ground type.

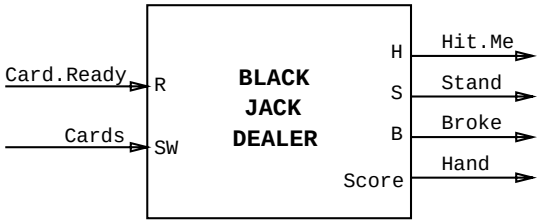
At the binary representation level, DDD generates a collection of boolean subsystems for implementing the design. This decomposition may be entirely orthogonal to the logical hierarchy. A common example, suggested at the right, is restructuring a portion of the data path into bit slices. Registers and certain combinational functions are projected to a single bit in each physical component. Though conceptually simple, this phase involves massive transformations of the design description. It is a task of the DDD system to sustain correctness during the pervasive reorganization.



The boolean subsystems are put into logic synthesis facilities, which perform low-level optimizations and assemble realizations. Working prototypes have been built in PLD, standard-cell, and PLA technologies; and we have developed a generic bit-slice module for rapid design [26].

5 First Example – A Black Jack Dealer

The first example is taken from a text book on digital design by Winkel and Prosser [25]. The **BJ** machine simulates a dealer's actions in a black jack game:



The environment presents cards one at a time; using a four-cycle handshake (`Hit.me`, `Card.ready`). The device plays until its score exceeds sixteen and looses should its score exceed twenty-one. It may have to revalue an ace, whose score is either one or eleven. Binary outputs `Stand` and `Broke` indicate the status of the game, and the machine and displays the dealer's score (`Hand`). The symbolic base type of the specification includes:

Objects	Tests
Card = playing cards	ace?: Count → Bool (test for an ace)
Count = numeric scores	16?: Count → Bool (does c exceed 16?)
Light = {on, off}	21?: Count → Bool (does c exceed 21?)
Bool = {tt, ff}	

Constants	Operations
zero ∈ Count	addto: Count^2 → Count (add scores)
+ace ∈ Count	addace: Count → Count (for 11-pt. aces)
-ace ∈ Count	cancelace: Count → Count (for 1-pt. aces)
	cardcount: Card → Count (value of a card)

Figures 1 through 9, dealing with this first derivation, are collected at the end of this section. Figure 1 shows the ASM control specification given by Winkel and Prosser for the black-jack machine. The corresponding DDD specification is shown in Figure 2. Both specifications refer to an architecture which is developed later in the derivation (See Figures 4 and 8). In the initial state `GET` the machine waits for a pulse on `R` while latching (the value of) a card into register `C`. It then updates its score and sets its status before returning to `GET`.

$\mathtt{BJ}_0$ is written to correspond to the ASM chart. Register `State` represents the control state. It is the functional analog of a `while-true` loop, but it might instead have been expressed as a system of four function definitions, one for each control state. For example, the `ADD` state (actually, `ADD` represents a decision and not a state of the flowchart) would be

```
ADD = λ (C H S B Score A R Rd)
  let Out = (display Score H S B)
      Go = (Rin)
      Cd = (cardvalue (SWin))
  in
     (if (ace? C)
        (if A
            (TST ? ff S B (addto Score C) A Go R)
            (USE ? ff S B (addto Score C) A Go R))
        (TST ? ff S B (addto Score C) A Go R)
```

and similarly for `GET`, `USE`, and `TST`. Although this yields a wordier specification, because of the repeated `let`-bindings, such systems are a common *target* form for higher level program transformations.

The specification could have been written more succinctly. For example, ADD reduces to

```
let
   ADDnext =   λ(C A) (if (and (ace? C) (not A)) use tst)
in
   (BJ (ADDnext C A) ? ff S B (addto Score C) A Go R)
```

This particular simplification is subsumed later in the derivation. As an example of nontrivial control manipulation, we might also fold USE into ADD. The body of ADD would be:

```
let
   Scorenext =   λ(C Score) (if (ace? C)
                                (addace Score)
                                Score)
in
   (BJ tst C ff S B (Scorenext C (addto Score C)) A Go R)
```

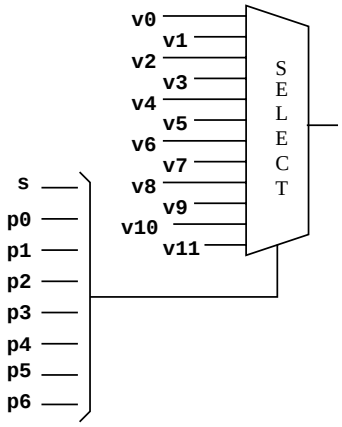
and the USE state would be eliminated. Again however, we chose the form $\mathtt{BJ}_0$ for its exact relationship to the ASM description of Figure 1.

The parameters Rin and SWin represent input ports, and Out displays the external status of the system. These entities are actually modeling interfaces. For expediency, we model communication using imperative input-output commands; Rin, SWin, and display are procedures that perform the I/O. In a moment, we will extract these impurities from the description. Although it adds more notational overhead to the specification, the right way to represent communication is to use streams [17]. This technique conforms to later stages of specification and consequently, supports a hierarchical decomposition of specifications. Since DDD places no special significance on operation symbols, limited used of imperative modeling techniques can coexist effectively with the derivation process.

5.1 Behavior to Structure

An initial system expression is derived automatically. Figure 3 shows the resulting system-expression. Details of the construction are given in [13, 14, 15]. The key step is to define a *selection combination* representing the control structure of the specification:

```
define select =
 λ((s p0 p1 p2 p3 p4 p5 p6)
    v0 v1 v2 v3 v4 v5
    v6 v7 v8 v9 v10 v11)
 (case s
  (get (if p0
           (if p1 v0 (if p2 v1 v2))
            v3))
  (add (if p3 (if p4 v4 v5) v6))
  (use v7)
  (tst (if p5
           (if p6 (if p4 v8 v9) v10)
           v11)))
```



Subsequent transformations are based on `select`. By distributing `select` over calls to `BJ`, traces of the individual parameters are obtained. The construction guarantees that in $\texttt{BJ}_1$, given a particular sequence of input events on `Rin` and `SWin`, mapping `display` over streams `Score`, `H`, `S`, and `B` produces the same output-event sequence as does the specification.

Figure 3 also shows is a *transfer table* representation of $\texttt{BJ}_1$. Its row numbers encode `select`'s twelve alternatives; its columns show information flow through individual signals. DDD produces transfer tables for analysis and visualization, but DDD transformations are always applied to expressions such as Figure 3. Certain realizations, such as PLAs, are fairly direct implementations of transfer tables. For such targets, DDD generates circuitry for encoding and decoding row numbers. The schematic in Figure 4 is a manually drawn graphical representation of $\texttt{BJ}_1$.

5.2 Structure to Architecture

The next task is to transform $\texttt{BJ}_1$ into a reasonable description of functional components for an implementation. The primary tool for this task is *system factorization,* which is used to impose logical organization [16]. DDD factorization encapsulates a subsystem in a combinator and, as a byproduct, generates the combinator definition. In the simplest cases, enclosure is simply the identification of a signal or abstraction of a group of terms. More generally it involves isolating the subsystem from the problem-dependent selector, using distributive laws.

In the Winkel-Prosser specification (Figure 1), the "hit me" signal, `H`, is an unregistered conditional output. The first factorization identifies the combinational input to the `H`-register:

```
 H ≐ H*
H* = (select Status ff ff ff tt ff ff ff ff ff ff ff ff)
```

The register `H` is moved outside $\mathtt{BJ}_1$ in a hierarchy of `system`-expressions. To preserve behavioral equivalence, the device's *environment* must now latch the `H` output.

Next, let us dispense with the modeling interface. Intuitively, the signal expressions `(Rin)` says, "always read from port `Rin`;" and similarly for `Sin` and `Out`. If we instead model `Rin`, `SWin` and `Out` as streams, these coercions can be eliminated.

A third factorization allocates the arithmetic operations on cards in the lower-left dashed box in Figure 4. Two combinator definitions are added, declaring that the `addace` and `cancelace` operations can be implemented by `addto`:

```
define cancelace = λ(S) (addto S -ace)
define    addace = λ(S) (addto S +ace)
```

Recall that `+ace` and `-ace` are constants. These definitions are expanded in place, resulting in a total of five occurrences of `addto`. The factorization superimposes these five occurrences by distributing `select` (shaded in Figure 4) through them:

```
Score   ≐ (select STATUS
                 Score zero Score Score BJADD-0 BJADD-0
                 BJADD-0 BJADD-0 BJADD-0 Score Score Score)

BJADD-0 = (addto BJADD-A BJADD-B)
BJADD-A = (select STATUS
                ? ? ? ? Score Score Score Score Score ? ? ?)
BJADD-B = (select STATUS
                ? ? ? ? C C C +ace -ace ? ? ?)
```

The result is the system $\mathtt{BJ}_2$ shown in the form of a transfer table in Figure 5. The factorizations have isolated that portion of the design intended for implementation on a device. Remnants of the behavioral model, the modeling

interface and the H register, have been moved up in the functional hierarchy. An outer-level description,

```
define BJenvironment = λ(Rin SWin)
  system
     Go = (Rin)
     Cd = (cardcount (SWin))
      O = (display Score H S B)
      H ≐ H*
     (Score H* S B) = (BJ2 Go Cd)
  in
     O
```

details how the peripheral environment behaves to maintain correct behavior in the BJ module. BJenvironment is functionally equivalent to $\mathtt{BJ}_0$, but the device we intend to build is $\mathtt{BJ}_2$. In other words, the environment description represents a set of conditions that must be verified of the surrounding system for $\mathtt{BJ}_2$ to be correctly integrated.

5.3 Architecture to Physical Organization

Of course, the black-jack machine is small enough to be wholly mapped to a number of programmable technologies. For example, Figure 9 shows PLA and standard-cell implementations of the entire circuit. The reorganization done in this section is simply to illustrate that aspect of derivation algebra necessary for larger systems. The specific tactic is to decompose the transfer table of Figure 5 into bit slices. The row numbers of the table encode the device's status. The select combination is reduced to an encoder that broadcasts a command to the bit-sliced transfer table, as depicted in Figure 8.

Transfer tables can be decomposed along columns; this is simply a partitioning of signals into groups. It is usually appropriate to separate the control sequencer from the remainder of the data path. The resulting two tables are shown in Figure 6.

Since it is just a constant mapping, the State transfer table can be dealt with using boolean minimization. A 2-bit binary assignment, (K_1, K_0), to the control tokens get (0, 0), add (0, 1), use (1, 0) and tst (1, 1), yields the

following boolean system for `State`:

$$K'_0 = \overline{K}_1 K_0 p_3 \overline{p}_4 + p_0 \overline{p}_1 \overline{K}_1 \overline{K}_0$$
$$K'_1 = p_4 K_1 p_5 p_6 + K_0$$

This is the `NEXTSTATE` component of Figure 8.

The table on the right in Figure 6 is reduced by eliminating identical rows and resynthesizing the command generator. The resynthesis is automatic. Other forms of optimization, such as reordering the rows to make better use of the state encoding, are manually guided.

Implementation of this transfer table also requires that representations be defined for the symbolic entries. In this case we can move directly to the boolean level. The declaration includes field descriptions and binary values for the constants. Although DDD can generate a binary encoding for many of the constants, these are more often declared manually as dictated by external considerations.

Figure 7 shows the declared binary representations of the ground constants at the upper-left. At the upper-right is a projection mapping for the bit-slice organization. With these declarations, the register paths of $\texttt{BJ}_2$ are decomposed into subsystems representing each bit slice. The result, expressed as transfer tables, is shown in Figure 7. Again, the actual transformations are performed on hierarchical `system` expressions.

Call the bit slices `SLICE0` through `SLICE5`. The global organization of the device is described by the `system` expression $\texttt{BJ}_3$, in Figure 8. In environment `BJenvironment`, $\texttt{BJ}_3$ is correct with respect to $\texttt{BJ}_1$, modulo the declared representation of the basis.

In order to complete the derivation, an implementation of the ground operations must be provided. This consists of boolean combinators for the primitive operations `or`, `16?`, `21?`, and `addto`.

5.4 Review of the Derivation

The derivation was done interactively by editing a command script in a window and feeding it piecewise to DDD. We estimate that a DDD expert, working alone, could complete the derivation in half an hour. This does not include the time spent composing the specification (Figure 2). It actually took about three hours of interaction—most of it between a DDD expert and his novice coauthor. Several design paths were tried; and we chose one

which best served our expository purposes. The CPU time to execute the full script is negligible (10 seconds on a SPARCstation 1). This measure of time is not very meaningful because derivation is incremental. No individual transformation takes more than a second.

The script is one page, about the same size as the structural description in Figure 3, which was generated by DDD. Thus, we can say that derivation script specifies the structural aspect of the design. Sixteen DDD transformations are needed to reach the target description in Figure 8. For larger designs that we have done, the number of transformations is not much greater, although file management adds appreciably to the size of the scripts. This derivation is summarized below:

$$\texttt{BJ}_0 \xrightarrow[\text{synthesis}]{\text{system}} \texttt{BJ}_1$$

Specification $\texttt{BJ}_0$, a behavioral description of the Black-Jack dealer, is shown in Figure 1. The first transformation automatically generates an initial structural description, shown in Figure 4.

$$\texttt{BJ}_1 \xrightarrow[\text{Rin, SWin, H}]{\text{factor:}} \texttt{BJ}_1{}' \xrightarrow[\text{cancelace, addace}]{\text{expand:}} \texttt{BJ}_1{}'' \xrightarrow[\text{addto}]{\text{factor:}} \texttt{BJ}_2$$

Using system factorizations, modeling interfaces and the `H` register are moved outside the description. The combinators `addace` and `cancelace` are expanded to terms involving the `addto` operation. A factorization allocates all occurrences of `addto` to single instance. The result is shown as a transfer table in Figure 5.

$$\texttt{BJ}_2 \xrightarrow[\text{decomposition}]{\text{transfer-table}} \texttt{BJ}_2{}' \bullet \texttt{NEXTSTATE} \bullet \texttt{ENCODE}$$

A control sequencer is isolated from the data path, which is then reduced by specializing the selector combination. A command encoder is synthesized for the simplified transfer table shown in Figure 6. This starts the development of the physical partitioning shown in Figure 8.

$$\texttt{BJ}_2{}' + \text{Declarations} \xrightarrow[\text{projection}]{\text{bit-slice}} \texttt{BJ}_3 = [\texttt{SLICE}_i] \bullet \texttt{addto} \bullet \cdots$$

Using declared representations and projection mappings, the subsystem of Figure 6 is decomposed into the bit-slice partitioning of Figure 7. Boolean

implementations of the base operations, `addto`, `ace?`, `21?` and `16?` are incorporated in the description.

$$\texttt{BJ}_3 \xrightarrow[\text{synthesis}]{\text{logic}} \text{realization}$$

Figure 9 shows three realizations produced by various logic-synthesis tools without manual intervention. The standard-cell realization, upper right, does not include registers. The PLA realization, lower right includes the full design except for feedback paths. The device at the left is in mixed technology, the data path was generated by a package we have developed and includes a serial scan path [26]. The PLA at the bottom implements control, and small standard-cell layout, mid-left, implements comparison operations. DDD's role at this stage is to preserve global correctness as the design is decomposed into synthesizable subsystems.

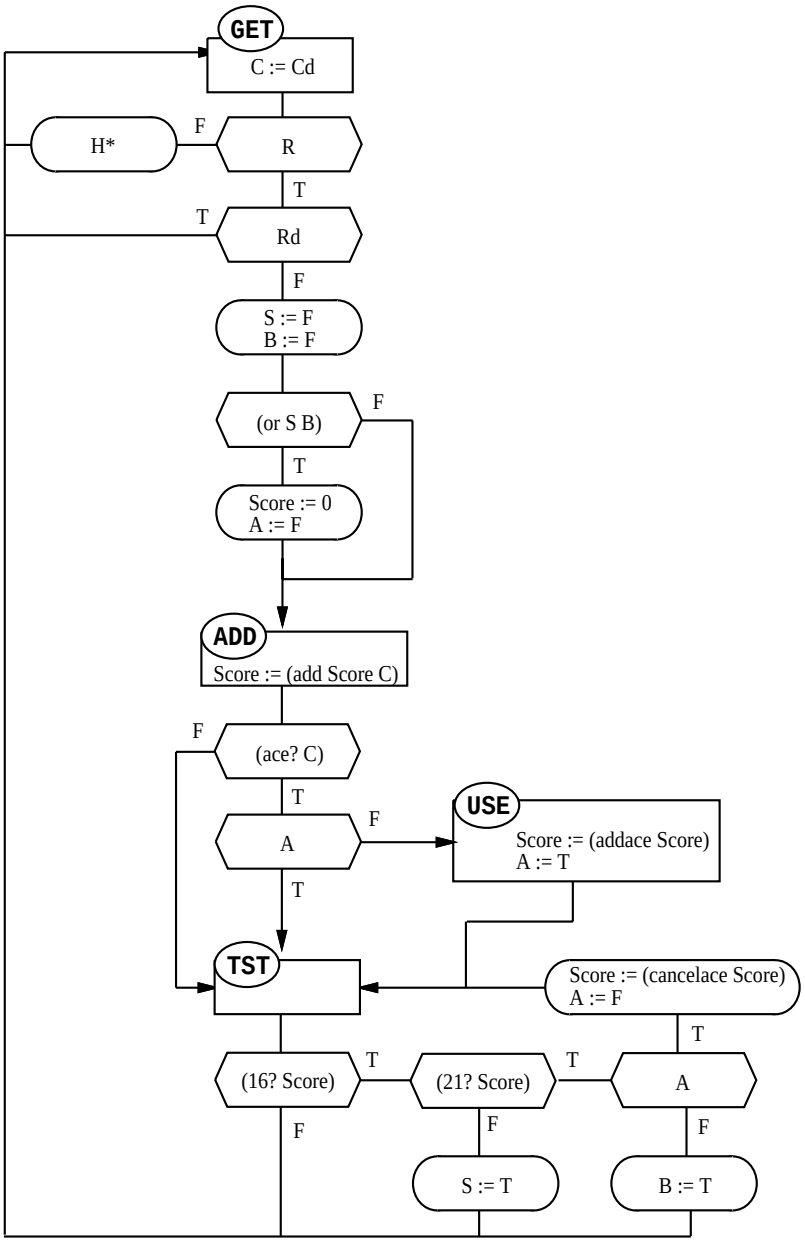


FIGURE 1. **ASM specification of the Black-Jack Machine[32]**

```
define BJ0 = λ (Rin SWin)
letrec
BJ = λ (State C H S B Score A R Rd)
  let  0 = (display Score H S B)
       Go = (Rin)
       Cd = (cardvalue (SWin))
  in
   (case State

     (get (if R
              (if Rd
                  (BJ get ? ff S B Score A Go R)
                  (if (or S B)
                      (BJ add Cd ff ff ff zero ff Go R)
                      (BJ add Cd ff ff ff Score A Go R)))
              (BJ get ?  tt S B Score A Go R)))

     (add (if (ace? C)
              (if A (BJ tst ? ff S B (addto Score C) A Go R)
                    (BJ use ? ff S B (addto Score C) A Go R))
              (BJ tst ? ff S B (addto Score C) A Go R)))

     (use (BJ tst C ff S B (addace Score) tt Go R))

     (tst (if (16? Score)
              (if (21? Score)
                  (if A
                      (BJ tst C ff S B (cancelace Score)
                          ff Go R)
                      (BJ get C ff S tt Score A Go R))
                  (BJ get C ff tt B Score A Go R))
              (BJ get C ff S B Score A Go R))))

in
  (BJ get 0 tt ff ff 0 ff ff ff)
```

FIGURE 2. **DDD Specification of the Black-Jack Machine**

```
define BJ1 = λ (Rin SWin)
 system
 Status = (list State R Rd (or S B) A (16? Score)
                (21? Score))
  State ≐ (select Status get add add get tst use
                          tst tst tst get get get)
     Rd ≐ (select Status R R R R R R R R R R R)
      R ≐ (select Status Go Go Go Go Go Go
                          Go Go Go Go Go Go))
      A ≐ (select Status A ff A A A A A tt ff A A A)
      B ≐ (select Status B ff ff B B B B B B tt B B)
      S ≐ (select Status S ff ff S S S S S S S tt S)
      H ≐ (select Status ff ff ff tt ff ff
                          ff ff ff ff ff ff)
      C ≐ (select Status ? Cd Cd ? ? ? ? S C C C C)
  Score ≐ (select Status Score zero Score Score
                          (addto Score c) (addto Score c)
                          (addto Score c) (addace Score)
                          (cancelace Score) Score Score Score)
    Out ≐ (display Score H S B)
     Go ≐ (Rin)
     Cd ≐ (cardvalue (SWin))
```

define BJ_1 = λ (Rin SWin)

	St.	C	H	S	B	Score	A	R	Rd	Cd	Go	Out
0	get	C	ff	S	B	Score	A	Go	R	t_1	t_2	t_3
1	add	Cd	ff	ff	ff	zero	ff	Go	R	t_1	t_2	t_3
2	add	Cd	ff	ff	ff	Score	A	Go	R	t_1	t_2	t_3
3	get	C	tt	S	B	Score	A	Go	R	t_1	t_2	t_3
4	tst	C	ff	S	B	(addto Score c)	A	Go	R	t_1	t_2	t_3
5	use	C	ff	S	B	(addto Score c)	A	Go	R	t_1	t_2	t_3
6	tst	C	ff	S	B	(addto Score c)	A	Go	R	t_1	t_2	t_3
7	tst	C	ff	S	B	(addace Score)	tt	Go	R	t_1	t_2	t_3
8	tst	C	ff	S	B	(cancelace Score)	ff	Go	R	t_1	t_2	t_3
9	get	C	ff	S	tt	Score	A	Go	R	t_1	t_2	t_3
10	get	C	ff	tt	B	Score	A	Go	R	t_1	t_2	t_3
11	get	C	ff	S	B	Score	A	Go	R	t_1	t_2	t_3

$t_1 \equiv$ (cardvalue (SWin)) $t_2 \equiv$ (Rin) $t_3 \equiv$ (display Score H S B))

FIGURE 3. **Initial System Description of BJ_0 and its Transfer-Table Representation**

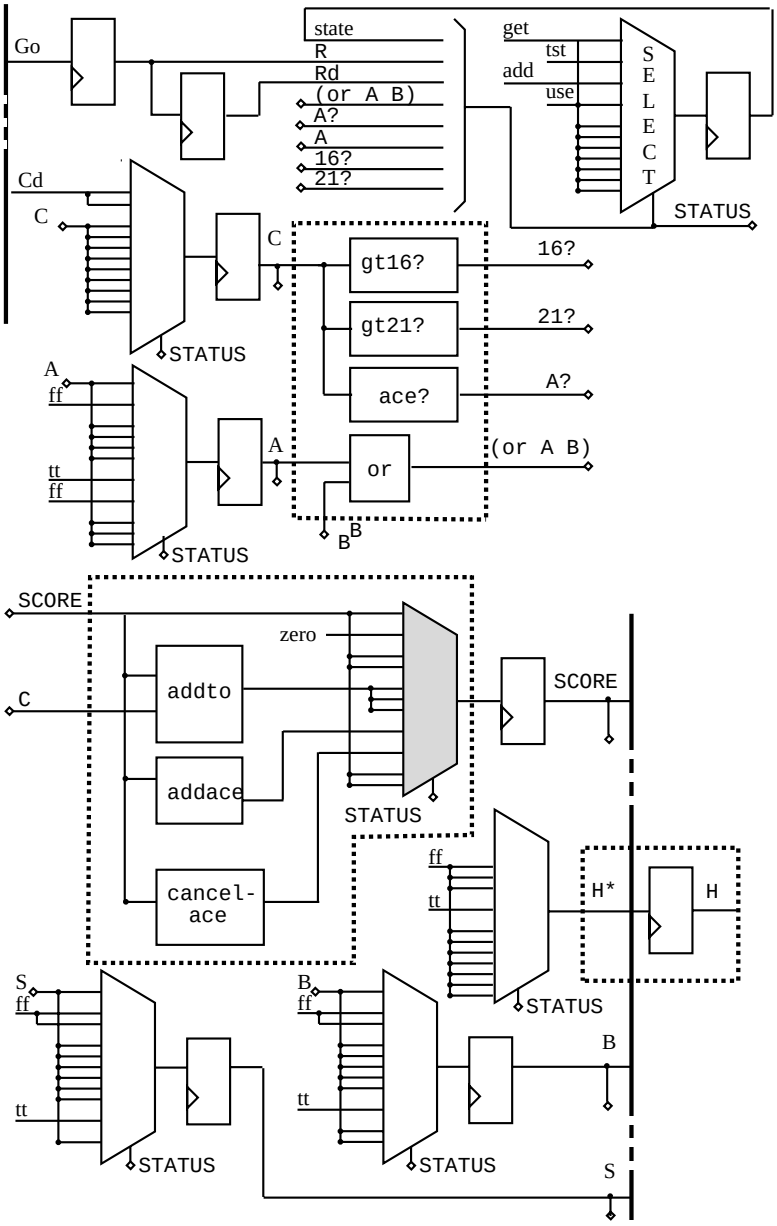


FIGURE 4. **Schematic of** $\mathtt{BJ}_1$

define BJ_2 = λ(Cd Go)

	State	C	H*	S	B	Score	BJADD-A*	BJADD-B*	A	R	Rd
0	get	C	ff	S	B	Score	?	?	A	Go	R
1	add	Cd	ff	ff	ff	zero	?	?	ff	Go	R
2	add	Cd	ff	ff	ff	Score	?	?	A	Go	R
3	get	C	tt	S	B	Score	?	?	A	Go	R
4	tst	C	ff	S	B	BJADD-0	Score	C	A	Go	R
5	use	C	ff	S	B	BJADD-0	Score	C	A	Go	R
6	tst	C	ff	S	B	BJADD-0	Score	C	A	Go	R
7	tst	C	ff	S	B	BJADD-0	Score	+ace	tt	Go	R
8	tst	C	ff	S	B	BJADD-0	Score	-ace	ff	Go	R
9	get	C	ff	S	tt	Score	?	?	A	Go	R
10	get	C	ff	tt	B	Score	?	?	A	Go	R
11	get	C	ff	S	B	Score	?	?	A	Go	R

BJADD-0 =(addto BJADD-A BJADD-B)

FIGURE 5. **Transfer-table representation of** BJ_2

	CTL
0	get
1	add
2	add
3	get
4	tst
5	use
6	tst
7	tst
8	tst
9	get
10	get
11	get

	C	H*	S	B	Score	BJADD-B*	A	R	Rd
0	C	ff	S	b	Score	?	A	Go	R
1	Cd	ff	ff	ff	zero	?	ff	Go	R
2	Cd	ff	ff	ff	Score	?	A	Go	R
3	C	tt	S	b	Score	?	A	Go	R
4	C	ff	S	b	BJADD-0	c	A	Go	R
5	C	ff	S	b	BJADD-0	+ace	tt	Go	R
6	C	ff	S	b	BJADD-0	-ace	ff	Go	R
7	C	ff	S	tt	Score	?	A	Go	R
8	C	ff	tt	b	Score	?	A	Go	R
9	C	ff	S	b	Score	?	A	Go	R

FIGURE 6. **Decomposition of the Transfer Table**

constant	value
ff	0
tt	1
on	1
off	0
+ace	01010
-ace	01101
zero	00000

	bit 0	bit 1	bit 2	bit 3	bit 4	bit 5
R						•
Rd						•
H*						•
S						•
B						•
A						•
C	•	•	•	•		
Score	•	•	•	•	•	
BJADD-A*	•	•	•	•	•	

	C_0	N_0	W_0*
0	C_0	N_0	?
1	Cd_0	0	?
2	Cd_0	N_0	?
3	C_0	N_0	?
4	C_0	V_0	C_0
5	C_0	V_0	0
6	C_0	V_0	1
7	C_0	N_0	?
8	C_0	N_0	?
9	C_0	N_0	?

	C_1	N_1	W_1*
0	C_1	N_1	?
1	Cd_1	0	?
2	Cd_1	N_1	?
3	C_1	N_1	?
4	C_1	V_1	C_1
5	C_1	V_1	1
6	C_1	V_1	0
7	C_1	N_1	?
8	C_1	N_1	?
9	C_1	N_1	?

	C_2	N_2	W_2*
0	C_2	N_2	?
1	Cd_2	0	?
2	Cd_2	N_2	?
3	C_2	N_2	?
4	C_2	V_2	C_2
5	C_2	V_2	0
6	C_2	V_2	1
7	C_2	N_2	?
8	C_2	N_2	?
9	C_2	N_2	?

	C_3	N_3	W_3*
0	C_3	N_3	?
1	Cd_3	0	?
2	Cd_3	N_3	?
3	C_3	N_3	?
4	C_3	V_3	C_3
5	C_3	V_3	1
6	C_3	V_3	1
7	C_3	N_3	?
8	C_3	N_3	?
9	C_3	N_3	?

	N_4	W_4*
0	N_4	?
1	0	?
2	N_4	?
3	N_4	?
4	V_4	C_4
5	V_4	0
6	V_4	0
7	N_4	?
8	N_4	?
9	N_4	?

	R	Rd	H*	S	B	A
0	Go	R	0	S	B	A
1	Go	R	0	0	0	0
2	Go	R	0	0	0	A
3	Go	R	1	S	B	A
4	Go	R	0	S	B	A
5	Go	R	0	S	B	1
6	Go	R	0	S	B	0
7	Go	R	0	S	B	A
8	Go	R	0	1	1	A
9	Go	R	0	S	B	A

FIGURE 7. **Binary Declarations and Bit-Slice Projections.** *Names* Score, BJADD-O *and* BJADD-B *have been shortened to* N, V *and* W, *respectively, to condense the tables.*

```
define BJ3 = λ (Go (Cd3 Cd2 Cd1 Cd0))
system
  Status  = (list R Rd (or S B) A (16? Score) (21? Score))
  State   = (NXTSTATE State Status)
  Score   = (list Score0 Score1 Score2 Score3 Score4)
  Cmd     = (ENCODE State Status)
  BJADD-B = (list BJADD-B0 BJADD-B1 BJADD-B2 BJADD-B3
                  BJADD-B4)

  (Score0 C0 BJADD-B0*) = (SLICE0 Cmd BJADD-O0 Cd0)
  (Score1 C1 BJADD-B1*) = (SLICE1 Cmd BJADD-O1 Cd1)
  (Score2 C2 BJADD-B2*) = (SLICE2 Cmd BJADD-O2 Cd2)
  (Score3 C3 BJADD-B3*) = (SLICE3 Cmd BJADD-O3 Cd3)
     (Score4 BJADD-B4*) = (SLICE4 Cmd BJADD-O4)
        (R Rd S B H* A) = (SLICE5 Cmd Go)

  (BJADD-O0 BJADD-O1 BJADD-O2 BJADD-O3 BJADD-O4)
          = (addto Score BJADD-B)
in
 (list H* S B Score)
```

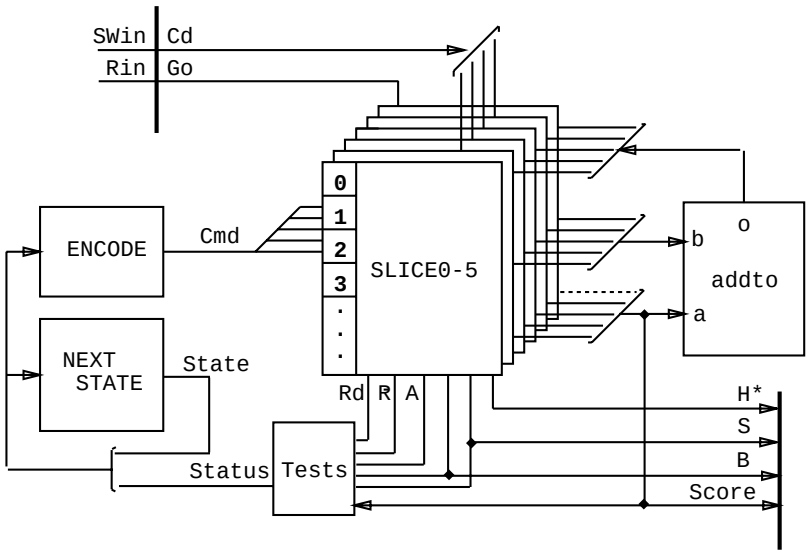
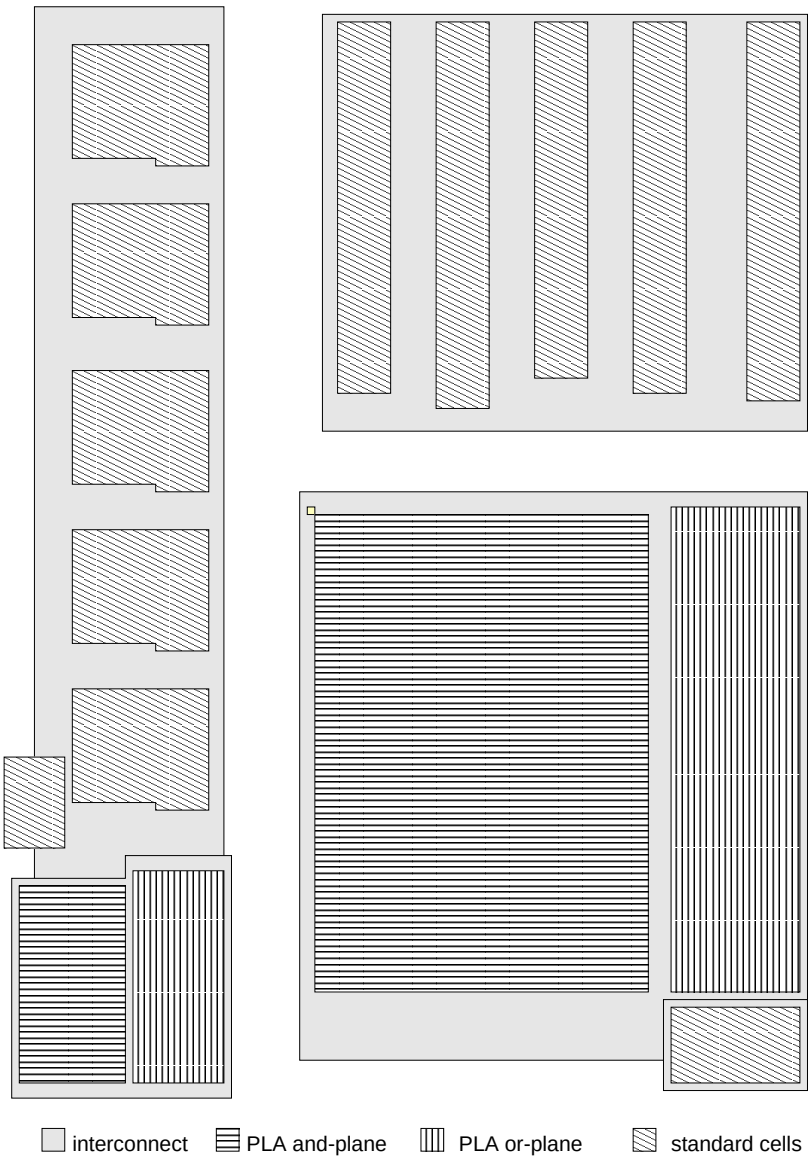


FIGURE 8. **Physical Organization of** BJ_3



[NOTE: *To save file space these floorplan diagrams replace the original figure, consisting of three VLSI layouts. Originals obtained by contacting the authors.*

FIGURE 9. **VLSI Realizations of the Black-Jack Machine.** *The standard-cell realization, upper right, does not include registers. The PLA realization, lower right, does not include feedback paths from the registers. The mixed layout realization, left, consists of a generic data path circuit [26] which is used to realize the* `score` *and* `C` *registers and the* `addto` *operation; the PLA below contains the* `ENCODE` *and* `NEXTSTATE` *functions and the remaining registers,* `SLICE 5`*; the tests* `gt21?`*,* `gt16?` *and* `ace?` *are implemented in standard-cell technology, just above the PLA on the left.*

6 Second Example – Fragment of a Memory Manager

This example begins with a fragment from a behavioral specification involving two random access memories. It is distilled from Boyer's derivation of a garbage collector [17, 3]. The device moves information from a source memory, organized as an array of two-byte words, to a target memory, organized as an array of bytes. The sort Memory involves two parameters, call them A and C, and operations for reading and writing:

$$\begin{aligned} \texttt{rd} &: \mathsf{Memory}(A, C) \times A \to C \\ \texttt{wt} &: \mathsf{Memory}(A, C) \times A \times C \to \mathsf{Memory}(A, C) \end{aligned}$$

The design will have two Memory objects which share a common Address space, one for the Byte-memory, the other for the Word-memory. They share a common Address sort. There is an increment operation for Address objects,

$$\texttt{inc}: \mathsf{Address} \to \mathsf{Address}$$

Two coercions extract bytes from a word:

$$\texttt{hi}, \texttt{lo}: \mathsf{Word} \to \mathsf{Byte}$$

Finally, there is a sort Count, with a decrementer and a test-for-zero:

$$\begin{aligned} \texttt{dcr} &: \mathsf{Count} \to \mathsf{Count} \\ \texttt{zero?} &: \mathsf{Count} \to \mathsf{Bool} \end{aligned}$$

The specification fragment $\texttt{MC}_0$, shown in Figure 10, copies `N` consecutive words from word-memory `M1` into byte-memory `M2`. The example is contrived to show how applicative formulations of complex objects are manipulated during derivation. Since memories are values of the ground type, the `LOOP` body can be can be written as a single expression. Hunt specified a microprocessor programmer's model in much the same fashion [12]. In

```
(wt (wt M2 P2 (lo (rd M1 P1))) (inc P2) (hi (rd M1 P1)))
```

The subterm $\mathcal{M} \equiv$ `(wt M2 P2 (lo (rd M1 P1)))` denotes the memory resulting from the first write, and so `(wt` $\mathcal{M}$ `(inc P2) (hi (rd M1 P1)))` denotes the effect of a second write to `M2`.

The first problem is to derive a more serial version of $\texttt{MC}_0$. Even for such a small example, the serialization constraints can be complicated. In no state should two operations be scheduled on the same memory, and there should be minimal combinational delay in setting up the memory operands. We might (as in this example) try to minimize the number of arithmetic devices, or alternatively, keep the memories busy in every state, (as was done in [17]).

The underlying algebra involves a sequence of *inverse substitutions* applied to the argument list of `LOOP`. At each step we need two lists of intermediate terms,

$$(S_N, S_{M1}, S_{M2}, S_{P1}, S_{P2}, S_T)$$

and

$$(T_N, T_{M1}, T_{M2}, T_{P1}, T_{P2}, T_T),$$

such that for $i, j \in \{\texttt{N}, \texttt{M1}, \texttt{M2}, \texttt{P1}, \texttt{P2}, \texttt{T}\}$, the substitutions $T_i[S_j/j]$ produce `LOOP`'s the original argument list,

```
((dcr N)
 M1
 (wt (wt M2 P2 (lo (rd M1 P1)))
     (inc P2)
     (hi (rd M1 P1)))
 (inc P1)
 (inc (inc P2)))
```

This process is repeated on one or both of the intermediate lists until the architectural constraints are satisfied.

We have experimented with strategies in which the designer provides architectural constraints in the form of a predicate, which accepts substitutions that meet architectural goals [3]. More recently, Zhu has developed an algebraic approach [33, 34, 35] having much in common a the treatment of Maadavaht, et.al, which based on DT0L languages [21]. The idea is to provide a partial set of architectural constraints, and then to determine automatically whether the given list of terms is serializable in that configuration. The DDD serializer provides several forms of constraint specification. The simplest is

a list of partial register transfers.

N	M1	M2	P1	P2	T
*	*	*	*	*	(rd M1 P1)
*	*	(wt M2 P2 (hi T))	*	*	*
*	*	(wt M2 P2 (lo T))	*	*	*
(dcr N)	*	*	P1	P2	*
N	*	*	(inc P1)	P2	*
N	*	*	P1	(inc P2)	*

The first row asserts that `(rd M1 P1)` may be transferred to register `T`. The last three rows together assert that at most one increment/decrement may occur at any time. The serializer composes these partial transfers to obtain trial substitutions. Using the constraints shown above, a serialized system description $\texttt{MC}_2$ in Figure 11 is obtained.

The derivation now proceeds along the path of the Black-Jack example. We shall carry it through the logical-organization phase to show some variations of factorization. The abstract `system`-expression shown in Figure 12 indicates the desired architecture. Four factorizations are applied to $\texttt{MC}_2$. The first creates a component, `WB`, for `hi` and `lo` operations. The second encapsulates all occurrences of `inc` and `dcr` in a component called `ALU`. Next, signal `M1` and the its `rd` operation are incorporated in a component called $\texttt{MEMORY}_1$. Finally, a second memory component, $\texttt{MEMORY}_2$ incorporates signal `M2`. The `system` expression in Figure 12 is the result.

As a byproduct, DDD generates a description of the factored subsystem. For instance, the synthesized description of $\texttt{MEMORY}_1$ is

```
define MEMORY_1 = λ(m1 I A)
  let
     Probe = λ(m i a) (case i
                          ((stet      ?)
                           ( rd (rd m a))))
```

```
in system
     M = (! m1 M)
   in (Probe M I A)
```

It is not hard to see that if this definition is expanded and then simplified in Figure 12, the original signal `M1` is recovered.

These examples show that DDD factorization is more general than the combinational abstraction done in the Black-Jack machine. They can incorporate state and introduce control. Factorizations are specified by giving a collection of *subject terms* to be incorporated [16]. There are several ways to abbreviate the subject terms. For example, there is a version of factorization which isolates all occurrences of a particular set of operators (e.g. `inc` and `dcr`) or all occurrences to a particular group of signals (e.g. `M1`).

As discussed in Section 3, the derivation has reduced $\mathtt{MC}_4$ to a simpler ground type of $\langle \mathsf{Address}, \mathsf{Content}, \mathsf{Count}, \mathsf{Bool} \rangle$. The more complex Memory entities have been encapsulated in modules. Thus, we can proceed in the manner of the previous section to reduce the design to hardware.

```
define MC0 λ(...)
  letrec

  LOOP = λ (N M1 M2 P1 P2)
    (if (zero? N)
        (LOOP N M1 M2 0 0)
        (LOOP (dcr N)
              M1
              (wt (wt M2 P2 (lo (rd M1 P1)))
                  (inc P2)
                  (hi (rd M1 P1)))
              (inc P1)
              (inc (inc P2))))
  in
    (LOOP ...)
```

FIGURE 10. **Specification fragment** $\mathtt{MC}_0$

```
define MC1 λ(...)
  letrec

    LOOP_0 = λ (N M1 M2 P1 P2 T)
      (if (zero? N)
          (LOOP_0 N M1 M2 P1 P2 ?)
          (LOOP_1 (dcr N) M1 M2 P1 P2 (rd M1 P1)))

    LOOP_1 = λ (N M1 M2 P1 P2 T)
          (LOOP_2 N M1 (wt M2 P2 (lsb T)) P1 (inc P2) T)

    LOOP_2 = λ (N M1 M2 P1 P2 T)
          (LOOP_3 N M1 (wt M2 P2 (msb T)) P1 (inc P2) ?)

    LOOP_3 = λ (N M1 M2 P1 P2 T)
          (LOOP_0 N M1 M2 (inc P1) P2 ?)
  in
    (LOOP_0 ...)
```

FIGURE 11. **Serialization of** $\mathtt{MC}_0$

```
system
  CTL := (select CTL Z L0 L1 L2 L3 L0)
    Z  = (zero? N)
    N := (select CTL Z N (dcr N) N N N)
   M1 := (select CTL Z M1 M1 M1 M1 M1))
   M2 := (select CTL Z M2 M2 (wt M2 P2 (lo T))
                             (wt M2 P2 (hi T)) M2))
   P1 := (select CTL Z P1 P1 P1 P1 (inc P1))
   P2 := (select CTL Z P2 P2 (inc P2) (inc P2) P2)
    T := (select CTL Z ? (rd M1 P1) T ? ?)
in ...
  system
     CTL := (select CTL Z L0 L1 L2 L3 L0)
       Z  = (zero? N)
       N := (select CTL Z N ALU-o N N N)
      P1 := (select CTL Z P1 P1 P1 P1 ALU-o)
      P2 := (select CTL Z P2 P2 ALU-o ALU-o P2)
       T := (select CTL Z ? M1-o T ? ?)
    M1-o  = (MEMORY1 M1-i M1-a)
    M1-i  = (select CTL Z stet rd stet stet stet)
    M1-a  = (select CTL Z ? P2 ? ? ?)
    M2-o  = (MEMORY2 M2-i M2-a M2-c)
    M2-i  = (select CTL Z stet stet wt wt stet)
    M2-a  = (select CTL Z ? ? P2 P2 ?)
    M2-c  = (select CTL Z ? ? BW-o BW-o ? ?)
   ALU-o  = (ALU ALU-i ALU-a)
   ALU-i  = (select CTL Z stet dcr inc inc inc)
   ALU-a  = (select CTL Z ? N P2 P2 P1)
    BW-o  = (BW BW-i T)
    BW-i  = (select CTL Z ? ? lo hi ?)
 in ...
```

FIGURE 12. **Initial (above) and Factored (below) System Expressions for** `MC`. *Factorization goals are indicated by the shadings.*

7 Summary and Conclusions

DDD was developed to help us explore a formalization of digital design. It reflects an approach that adapts a theoretical foundation of programming language semantics to the description and design of digital systems. In adopting a mathematical framework we also adopt its methodology. The mechanical aspect of engineering is characterized as a process of transformation, reducible to a taxonomy of basic algebraic laws. In this framework, our research is to identify abstractions used in digital engineering and learn how to manipulate them. The primary goal of DDD development is to create a flexible vehicle for performing these manipulations correctly.

By itself, the formalization says nothing about the creative and analytical aspects of engineering. Its first purpose is to provide means of explanation. Of course, we believe that with further development, a system like DDD is an appropriate basis for design automation. Since it operates exclusively on expressions, an engineer can inspect and interact with every transformation step. The simplicity of expression representation should also foster the development of specialized transformations to meet specialized needs. Over time, DDD's transformations will become more coarsely grained, so that less interaction is needed to obtain an implementation. However, by building transformations up from more elementary algebraic primitives, we hope to preserve the capability for interaction at lower levels of detail. The underlying challenge is to develop a *derivation language* that is powerful enough to express derivation tactics. The DDD language is not that powerful yet. With no provisions for design management, automatic decision making, branching, or backtracking, DDD is essentially a language of transformation commands, which are composed into "straight-line" derivation scripts.

Functional algebra is unlimited in its capacity for abstraction. Although the first-order DDD algebra does not exploit the full power of lambda calculus, it can evolve toward it. We have taken a bottom-up approach to implementing the algebra, making a priority of establishing a connection to real hardware. In principle, DDD integrates with software oriented program transformation methods, as developed for example in [30], to give a much higher level of hardware specification. In other words, we think that this approach can be seemlessly extended to higher levels of system description.

The first example presented in the paper, the Black-Jack machine in Section 5, shows how DDD is used to manipulate physical organization. The process involves factorization to develop a functional modularity, representa-

tion, and projection. Factorizations defined the boundary of the Black-Jack device, isolated the modeling interfaces, and allocated operations. Once a binary representation was incorporated for the symbolic basis of the specification, projections were used to reorganize the data path.

In Section 6, we looked briefly at algebra on behavioral descriptions and the manipulation of data hierarchies. Serialization is the *scheduling* component of high-level synthesis, but for DDD we want transformations that are general enough to handle multiple-value operations and, eventually, both architectural and temporal constraints.

Although the system is relatively flexible, getting the circuits we want sometimes involves micromanagement of the derivation process. This is a quality that seems common to many CAD systems. In order to improve DDD, we must develop more precise ways to specify the goal of an individual transformation. For example, the Memory factorizations in Section 6, synthesize a behavioral description of the memory process. Should this description be available beforehand—as we would expect for off-the-shelf components—it would in many cases be sufficient information to determine the factorization. The DDD serializer discussed in Section 7 offers a variety of ways to specify scheduling goals, but more experience is needed to determine how effective it is.

Most of the design exercises we have done with DDD are, like the examples here, data-path dominated. In our preliminary experimentation with control dominated designs, we find a lack of capability to manipulate sequential coprocesses except in trivial ways. We have not yet found an acceptable characterization of synchronization in the functional modeling language, and this is a priority in our present research.

References

[1] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* McGraw-Hill, 1985.

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

[3] C.D. Boyer and Steven D. Johnson. Using the digital design derivation system: Case study of a VLSI garbage collector. In Darringer and Ram-

ming, editors, *Ninth International Symposium on Computer Hardware Description Languages*, Amsterdam, 1989. IFIP WG 10.2, Elsevier.

[4] Robert S. Boyer and J. Struther Moore. *A Computational Logic*. Academic Press, 1979.

[5] Geoffrey M. Brown and Miriam E. Leeser. Synthesizing correct sequential circuits. In John A. Darringer and Franz J. Rammig, editors, *Computer Hardware Description Languages and their Applications*, pages 169–182. North-Holland, 1989. Proceedings of the IFIP WG 10.2 Ninth International Symposium on Computer Hardware Description Languages and their Applications.

[6] Raul Camposano. Behavior-preserving transformations for high-level synthesis. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, pages 106–128, New York, 1989. Proceedings of Mathematical Sciences Institute Workshop, Cornell University, Springer-Verlag.

[7] Daniel Friedman and Matthias Felleisen. *The Little LISPer*. MIT Press, third edition, 1988.

[8] J.A. Goguen. OBJ as a theorem prover with applications to hardware verification. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 218–267. Springer-Verlag, 1989.

[9] Ganesh C. Gopalakrishnan, Fichard M. Fujimoto, Vankatesh Akella, and Narayana S. Mani. HOP: A process model for synchronous hardware; semantics and experiments in process composition. *Integration, the VLSI journal*, 8:209–247, 1989.

[10] Ganesh C. Gopalakrishnan, David R. Smith, and Mandayam K. Srivas. An applicative specification language for verifiable VLSI designs and controller synthesis. Technical Report 85/4, Department of Computer Science, State University of New York at Stony Brook, January 1985.

[11] P. Hilfinger. A high-level language and silicon compiler for digital signal processing. In *Proceedings of the IEEE CICC Conference*, pages 213–216, 1985.

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

[13] Steven D. Johnson. Applicative programming and digital design. In *Proc. Eleventh Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming (POPL'84)*, pages 218–227, 1984.

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

[15] Steven D. Johnson. Digital design in a functional calculus. In Milne and Subramanyam, editors, *Formal Aspects of VLSI Design*, pages 153–178. North-Holland, Amsterdam, 1986.

[16] Steven D. Johnson. Manipulating logical organization with system factorizations. In Leeser and Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *LNCS*, pages 260–281. Springer, July 1989.

[17] Steven D. Johnson, B. Bose, and C.D. Boyer. A tactical framework for digital design. In Birtwistle and Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 349–383. Kluwer, Boston, 1988.

[18] Steven D. Johnson, R.M. Wehrmeister, and B. Bose. On the interplay of synthesis and verification: Experiments with the FM8501 processor description. In Claesen, editor, *Applied Formal Methods for Correct VLSI Design*, pages 385–404. Elsevier, 1989.

[19] Kurt Keutzer and Wayne Wolf. Anatomy of a hardware compiler. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 95–104. ACM, 1988.

[20] Eugene Kohlbecker. *Syntactic Extension in the Programming Language Lisp*. PhD thesis, Indiana University, Bloomington, Indiana, 1986.

[21] M. Mahmood, F. Mavaddat, M.I. Elmasry, and M.H.M. Cheng. A formal language model of local microcode synthesis. In Luc Claesen, editor, *Proceedings of The International Workshop on The Applied Formal Method for Correct VLSI Designs*, Leuven, Belgium, 1989. Elsevier Science Publishers B.V.

[22] 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.

[23] 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.

[24] John T. O'Donnell. HYDRA: Hardware description in a functional language using recursion equations and higher order combining forms. In G. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 309–328, Amsterdam, July 1988. Horth-Holland.

[25] Franklin P. Prosser and David E. Winkel. *The Art of Digital Design.* Prentice-Hall, second edition, 1987.

[26] K. Rath, I. Celis, and R. M. Wehrmeister. RTBA: A generic bit-sliced bus architecture for datapath synthesis. Technical Report 321, Department of Computer Science, Indiana University, December 1990.

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

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

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

[30] George Springer and Daniel P. Friedman. *Scheme and the Art of Programming.* McGraw-Hill, 1989.

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

[32] David Winkel and Franklin Prosser. *The Art of Digital Design.* Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1980.

[33] Zheng Zhu and Steven D. Johnson. An algebraic characterization of structural synthesis for hardware. In Claesen, editor, *Proceedings of The*

International Workshop on The Applied Formal Methods for Correct VLSI Designs. North-Holland, 1989.

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

[35] 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.

A DDD derivation of BlackJack

Specification (cf. Figure 2)

```
(define BJ
  (lambda (Rin SWin)
    (letrec
        ((get (lambda (C H S B Score A R Rd)
                (let ((O  (display Score H S B))
                      (Go (Rin))
                      (Cd (cardvalue (SWin))))
                  (if R
                      (if Rd
                          (get ? ff S B Score A Go R)
                          (if (or S B)
                              (add Cd ff ff ff zero ff Go R)
                              (add Cd ff ff ff Score A Go R)))
                      (get ?  tt S B Score A Go R)))))

         (add (lambda (C H S B Score A R Rd)
                (let ((O  (display Score H S B))
                      (Go (Rin))
                      (Cd (cardvalue (SWin))))
                (if (ace? C)
                    (if A
                        (tst ? ff S B (addto Score C) A Go R)
                        (use ? ff S B (addto Score C) A Go R))
                    (tst ? ff S B (addto Score C) A Go R)))))

         (use (lambda (C H S B Score A R Rd)
                (let ((O  (display Score H S B))
                      (Go (Rin))
                      (Cd (cardvalue (SWin))))
                (tst C ff S B (addace Score) tt Go R))))

         (tst (lambda (C H S B Score A R Rd)
                (let ((O  (display Score H S B))
                      (Go (Rin))
                      (Cd (cardvalue (SWin))))
                (if (Sgt16? Score)
                    (if (Sgt21? Score)
                        (if A
                            (tst C ff S B (cancelace Score) ff Go R)
                            (get C ff S tt Score A Go R))
                        (get C ff tt B Score A Go R))
```

```
                    (get C ff S B Score A Go R))))))
      (get C H S B Score A ff Rd))))
```

Representations (cf. Figures 5–7)

DDD introduces concrete representations by performing an alternate interpretation of specification syntax. The definitions below provide the basis for that interpretation. This is essentially the declarative part of the behavioral specification.

```
(define binary.map
`(
;DEFINED STREAM EQUATIONS
  (state     ,(lambda () (make-reg "k." 2)))
  (rd        ,(lambda () '(rd)))
  (r         ,(lambda () '(r)))
  (a         ,(lambda () '(a)))
  (score     ,(lambda () (make-reg "score." (add1 wordsize))))
  (b         ,(lambda () '(b)))
  (s         ,(lambda () '(s)))
  (h-i       ,(lambda () '(h*)))
  (c         ,(lambda () (make-reg "c." wordsize)))
  (adder0-v1 ,(lambda () (make-reg "adder-b." wordsize)))

;INPUTS
  (cardrdy   ,(lambda () '(cardrdy)))
  (sw        ,(lambda () (make-reg "sw." wordsize)))
  (adder0    ,(lambda () (make-reg "adder." (add1 wordsize))))
  (Go        ,(lambda () '(Go)))
  (Cd        ,(lambda () (make-reg "Cd." wordsize)))

;CONSTANTS
;state assignments
  (get ,(lambda () (nat->bv 0 2)))
  (add ,(lambda () (nat->bv 1 2)))
  (use ,(lambda () (nat->bv 2 2)))
  (tst ,(lambda () (nat->bv 3 2)))

;constants
  (10ptace  ,(lambda () '(1 0 1 0)))
  (-10ptace ,(lambda () '(0 1 1 0)))
  (zero     ,(lambda () '(0 0 0 0 0)))
  (?        ,(lambda () '(? ? ? ? ?)))
  (tt       ,(lambda () '(1)))
```

```
  (ff       ,(lambda () '(0)))))

(define datapath.org
  '((score.0 adder-b.0 c.0)
    (score.1 adder-b.1 c.1)
    (score.2 adder-b.2 c.2)
    (score.3 adder-b.3 c.3)
    (score.4)
    (a b s h* rd r)))
```

Implementation of control

```
(define SorB?.bool '((p2 ((s) (b)))));(or s b)
(define preds.bool
'((p6 ((score.4 & score.2 & score.1) ;sgt21?
       (score.4 & score.3)))
  (p5 ((score.4 & score.3)           ;sgt16?
       (score.4 & score.2)
       (score.4 & score.1)
       (score.4 & score.0)))
  (p4 ((a)))                         ;a
  (p3 ((c.3 & !c.2 & c.1 & c.0)))    ;(ace? c)
  (p1 ((rd)))                        ;rd
  (p0 ((r)))))                       ;r
```

Below are the encodings used to implement control.

```
(define cmd.map
`((v0  ,(lambda () (nat->bv 0 4)))
  (v1  ,(lambda () (nat->bv 1 4)))
  (v2  ,(lambda () (nat->bv 2 4)))
  (v3  ,(lambda () (nat->bv 3 4)))
  (v4  ,(lambda () (nat->bv 4 4)))
  (v5  ,(lambda () (nat->bv 5 4)))
  (v6  ,(lambda () (nat->bv 6 4)))
  (v7  ,(lambda () (nat->bv 7 4)))
  (v8  ,(lambda () (nat->bv 8 4)))
  (v9  ,(lambda () (nat->bv 9 4)))))

(define state.map
`((get ,(lambda () (mapBitPattern '(k.1 k.0) (nat->bv 0 2))))
  (add ,(lambda () (mapBitPattern '(k.1 k.0) (nat->bv 1 2))))
  (use ,(lambda () (mapBitPattern '(k.1 k.0) (nat->bv 2 2))))
  (tst ,(lambda () (mapBitPattern '(k.1 k.0) (nat->bv 3 2))))))
```

```
(define state.map.bit
`((get ,(lambda () '(0 0)))
  (add ,(lambda () '(0 1)))
  (use ,(lambda () '(1 0)))
  (tst, (lambda () '(1 1)))))
```

DDD derivation script for the BlackJack circuit

```
;Behavior to Structure
(define BJ_0              (ReadFile "BJ_0.ss"))
(define SEQSYS_1          (IterativeSystem->SequentialSystem BJ_0))
(define SEL_1             (car SEQSYS_1))
(define BJ_1              (cadr SEQSYS_1))

;Structure to Architecture
(load "predinfo")
(define BJ_1.1 (RemoveStrEqns '(Go 0 Cd)          ;Factor H, Go, 0, Cd
         (AbstractStrEqn 'H BJ_1)))
(define BJ_1.2                                    ;Expand cancelace, addace
  (ExpandFunction
    '(define cancelace
       (lambda (score)
         (addto score -10ptace)))
    (ExpandFunction
      '(define addace
         (lambda (score)
           (addto score 10ptace))) BJ_1.1)))
(define BJ_2                                      ;Factor addto
  (AbstractOperations 'adder '(addto) BJ_1.2))

(define SEQSYS_2                                  ;Optimization
  (OptimizeSEQSYS
    SEL_1 (RemoveStrEqns '(STATE ADDER0-I ADDER0-V0) BJ_2)))
(define SEL_2 (car SEQSYS_2))
(define BJ_2.1 (cadr SEQSYS_2))

;Architecture to Physical Organization
(define STATE (ExtractStrEqn 'STATE BJ_2))
(define NEXTSTATE  (map OptimizeSel
                     (ProjectSel state.map.bit
                       (renameSel 'STATE 'k
                         (PartialEval STATE SEL_1)))))
(define ENCODE (map OptimizeSel
                 (ProjectSel CMD.MAP
                   (renameSel 'SELECT 'CMD SEL_2))))
```

```
(define BJ_3 (Group (Project BJ_2.1 binary.map 5) datapath.org))

;Logic Synthesis
(StrEqns->RTBA (slice 0 BJ_3) "RTBA_0.streqns")
(StrEqns->RTBA (slice 1 BJ_3) "RTBA_1.streqns")
(StrEqns->RTBA (slice 2 BJ_3) "RTBA_2.streqns")
(StrEqns->RTBA (slice 3 BJ_3) "RTBA_3.streqns")
(StrEqns->RTBA (slice 4 BJ_3) "RTBA_4.streqns")
(define SLICE5.bool (StrEqns->BoolEqns (slice 5 BJ_3)))
(define ENCODE.bool (symbolic->bit (map Sel->BoolEqns ENCODE) state.map))
(define NEXTSTATE.bool (symbolic->bit (map Sel->BoolEqns NEXTSTATE) state.map))
(BoolEqns->MPLA
  (append ENCODE.bool NEXTSTATE.bool SLICE5.bool SorB?.bool) 'static "PLA.eqn")
```