[illegible]

TECHNICAL REPORT NO. 331

DDD – A Transformation System for Digital Design Derivation

by

Bhaskar Bose

May 1991

COMPUTER SCIENCE DEPARTMENT
INDIANA UNIVERSITY
Bloomington, Indiana 47405-4101

DDD - A Transformation System for Digital Design Derivation[*]
Reference Manual

by

Bhaskar Bose
Computer Science Department
Indiana University
Bloomington, Indiana

DDD - A Transformation System for Digital Design Derivation
Reference Manual

by

Bhaskar Bose

May, 1991

[*]Research reported herein was supported, in part, by the National Science Foundation under grants numbered DCR85-21497, MIP87-07067 and MIP89-21842.

Dear Reader,

The DDD system is constantly undergoing changes as researchers continue to study hardware design in the context of an algebraic framework. This manual is intended to document the current status of the system and acknowledges the dynamics of the system it attempts to characterize. It is the sincere hope of the author that this document provides the reader with an understanding of the system and continues to be updated as the system evolves.

Bhaskar Bose

Table of Contents

DDD - A Transformation System for Digital Design Derivation[*]
Reference Manual

Bhaskar Bose
Computer Science Department
Bloomington, Indiana

1 Introduction

DDD (Digital Design Derivation System) is a transformation system that implements a design algebra for synthesizing digital circuit descriptions from high level functional specifications. The system reflects a formal approach to digital design synthesis based on the algebraic manipulation of purely functional forms.

The system is intended to provide a well founded, mechanized, algebraic tool set for design synthesis. DDD is implemented in the Lisp dialect Scheme [11] as a collection of transformations that operate on s-expressions. Transformations are applied manually by the designer, either interactively or by creating a script, at various stages of the design process in order to derive hardware implementations. The hardware descriptions that are manipulated by DDD are written in Scheme and may be executed (with syntax extension) as Scheme programs.

DDD derives a technology independent set of digital circuit descriptions which are projected to binary representations. Boolean equations are then generated from these descriptions and are integrated with existing logic synthesis tools, such as boolean equation minimizers, PLD assemblers, and VLSI layout generators.

1.1 Design by Algebraic Transformation

In "Synthesis of Digital Designs from Recursion Equations" [9], Johnson defines a formal approach to hardware synthesis based on the algebraic manipulation of purely functional forms. In this framework, the discipline of applicative program design style is adapted to hardware synthesis.

Design is viewed as a translation of notation, starting with an *abstract specification* ranging over abstract data types and deriving an intended target description called an *implementation* and a physical object called a *realization*. This process may be viewed as a translation between dialects of recursive expressions, and can be expressed in the following diagram:

[*]Research reported herein was supported, in part, by the National Science Foundation under grants numbered DCR85-21497, MIP87-07067, and MIP89-21842.

$$D_0 \xrightarrow{\tau_0} D_1 \xrightarrow{\tau_1} \ldots \xrightarrow{\tau_{k-1}} D_k$$

D_0 represents a source description and D_k an implementation. The arcs may be enumerated from τ_0 to τ_{k-1} and represent applications of transformations. Thus design is defined as an initial specification D_0, and a derivation - the sequence of transformations $<\tau_0,..,\tau_{k-1}>$.

The first step in design is to write a functional specification. The design proceeds by applying a series of transformations on an evolving description. The final form is a circuit description ranging over binary values that are integrated with logic synthesis tools to generate hardware.

The method provides a *secure path to hardware*. In the formal discussion [9], each transformation is correctness preserving. The implementation is said to be correct by construction. The notion of "correctness" is defined as: Given a specification **S**, and a transformation τ, τ(**S**) results in an expression that will compute the same function as the initial specification.

The specifications are written in a functional programming language and are directly executable. Each derived form is also executable (with syntax extension). The source for synthesis is also the same object for *simulation*. For instance, the execution trace of the specification may also be used for *test vector generation*. An example of this may be found in [2]. A common source description, from which varying information is derived, reduces the chance of inconsistencies between the specification/implementation, and simulation model.

The notion of *abstraction* is fundamental. Control abstraction, data abstraction, and hierarchical abstraction define the design process. Control abstraction forms the basis of separating control and architecture from the initial specification. Data abstraction allows for the manipulation of architectural components. And finally, hierarchical abstraction allows for the translation to lower levels of description to the point of a realization.

The incorporation of *representations* may be introduced at any stage in the derivation. This allows the opportunity to postpone representation decisions to a later point in the design.

Verification is an interdependent facet of this methodology. Although mechanical derivation implements algebra that assures correct hardware - too many facets of design are unaccounted for in any given transformation system. In principle, the incorporation of a verification system will allow the engineer to call upon insight and experience to produce an efficient design, which is then simply certified by a computer. This is developed further in [3].

1.2 About this manual

Section 2 characterizes the derivation path in a design. Each of the sections (Section 3 to Section 10) discuss a facet of the derivation path. Each section has a heading which highlights which stage of derivation path is being addressed. Each section is divided into two parts.

First, an informal discussion on relevant strategic issues, and transformational algebra are presented. The second part, defines transformations that are implemented in DDD. Each transformation definition contains the name of the transformation and what its arguments are. A short description of the transformation, followed by an example application. Section 3 defines the input specification. Section 4 discusses the initial transformations which derive a structural description from a behavioral specification. Section 5 discusses algebra on structural descriptions. Section 6 discusses register transfer tables, a intuitive abstraction of structural descriptions. Section 7 discusses the projection of a description defined over an abstract basis to that of a description defined over some target basis. Section 8 discusses input/output extensions to Scheme. Section 9 discusses transducing scheme circuit descriptions into Daisy descriptions. Section 10 discusses the integration of DDD with logic synthesis tools.

Appendix A is a quick reference to all current DDD functions. Appendix B lists the forms of the objects that are manipulated by DDD. Appendix C contains two complete design examples. First, a simple single pulser circuit, SinglePulser, followed by a machine which implements the actions of a Black Jack dealer, BlackJack. Other examples not found in this text include a Stop-and-Copy garbage collector in VLSI [2], and an SECD machine [15].

In DDD, a sequence of transformations are applied to an initial specification defining a *derivation path* to an implementation. The path is expressed in the following diagram:

$$I \rightarrow C_1 \bullet S_1 \ldots \rightarrow C_n \bullet S_n \| F \xRightarrow{\rho_\beta} C_\beta \bullet S_\beta \| F$$

I represents an initial specification. It is iterative - the class of recursion schemata that characterizes sequential control. The initial specification is expressed in terms of a complex basis consisting of abstract operations and predicates, in addition to concrete operations and objects. Some examples are arrays of integers, memories, stacks, and arithmetic. The complex/concrete distinction is subjective and hierarchical. Thus I is defined at some intended level of description.

Initial transformations separate control and architecture, and derive a sequential system description: $C_1 \bullet S_1$. C denotes a descision combinator representing control, and S denotes a structural component representing architecture. The • operator denotes the composition of C and S. $C_1 \bullet S_1$ has the same complex basis as I, but the interpretation is based on the model of streams. Whereas before, variables ranged over values, they now denote sequences of values.

The algebra supported by DDD allows for the logical and physical decomposition of design. These design tactics alter the derivation path and sketch a complex design space with many possible paths between specification and implementation. The dotted notation ... in the ideogram denotes this one-to-many correspondence.

$C_n \bullet S_n \| F$ is system description at some level of refinement. Complex data types present in S_1 have been factored as a system of *abstract components* denoted by F. The $\|$ suggests a communicating system. As complex signals are factored, DDD generates signals to maintain the correct connectivity. Factorization is a central part of DDD's transformations.

$C_\beta \bullet S_\beta \| F$ is the projection of $C_n \bullet S_n \| F$ to a target representation, β. Representations are input to the system in the form of a projection function denoted by ρ_β.

Completing the path to hardware, abstract descriptions are projected to binary representations. Boolean equations are then generated. The equations can be implemented with MSI components directly, or can serve as inputs to a programmable logic device (PLD) programmer, or can be used as input to VLSI tools to generate PLAs, gate matrix, or standard cell layouts.

3 Specification

Specifications are written in Scheme. The specifications are written in a purely functional style where there are no side-effects. Descriptions are built from *applicative terms*, *constants*, *identifiers*, *conditional expressions*, *case statements*, and *function definitions*, and express synchronous systems. The specification is a control algorithm, as well as a description of architecture. Both control and architecture are derived from such specifications. DDD manipulates a concrete syntax of functional *s-expressions* in Scheme.

3.1 Scheme Syntax

Scheme is a statically scoped, applicative order, dialect of Lisp, which 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 language. An informal introduction to some of the basic forms are described in this section. A complete language definition can be found in [11].

Scheme supports operations on structured data such as *strings*, denoted with double quotes, *"abcd"*, *lists*, denoted by parenthesized sequences, $(l_1\ l_2\ \ldots\ l_n)$, 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.). A brief description of the forms used in DDD are described below.

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*.

Objects can be associated with a name at top level with 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*, which has the form

(if *test consequent alternative*)

which returns the *consequent* if the test is true, and the alternative otherwise; and a *case-*

expression of the form

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

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

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

(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 *exp1 exp2* ...

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

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

3.2 Hardware Specifications

A general form of the specification is given below. A circuit is defined by a set of mutually recursive function definitions, $\mathbf{S_0,...,S_Q}$, referred to as *state definitions*. Each state definition is a conditional expression representing a point of control. $\mathbf{(i_1\ i_2\ ...)}$ denotes a set of inputs to the circuit, and $\mathbf{(S_{init}\ r_{1init}\ r_{2init}\ ...\ r_{Ninit})}$ denotes the initial control point and state for the machine.

Each state definition has a uniform parameter list, $\mathbf{(r_1\ r_2\ ...\ r_N)}$ denoting a set of *registers*, defining the state of the machine, and i/o ports. However, since the level of abstraction is arbitrary, the notion of registers are abstract. The list of registers may contain arbitrary objects such as registers, memories, stacks, and communication channels. The set of registers is an initial estimation of architecture and represents architectural components with state.

```
(define CIRCUIT
  (lambda (i1 i2 ...)
    (letrec
      ((S0 (lambda (r1 r2 ... rN) exp0))
       (S1 (lambda (r1 r2 ... rN) exp1))
       ...
       (SQ (lambda (r1 r2 ... rN) expQ)))
      (Sinit r1init r2init ... rNinit))))
```

where **exp** can be:
a *let expression*

(let ((id val) ...) exp)

an *if statement*

(if pred $\exp_1$ $\exp_2$)

a *case statement*

(case pred (id_1 $\exp_1$) (id_2 $\exp_2$) ...)

or a *control point invocation*

(S val_1 ... val_N)

where **val** denotes an expression defined in the ground type at some intended level of abstraction. Valid terms for **val** are **?**, signals, constants, integers, booleans, arithmetic operations, boolean operations, routing primitives, and operations on abstract data types. **?** is a special character which denotes a "don't care" term.

The *let expression* provides a means of defining combinational signals, and associating a name with that expression.

The *if* and *case statement*, implement a conditional control construct dependent on a decision point **pred**. In the case of an *if statement*, **pred** can be any boolean expression defined in the ground type. In the case of a *case statement*, **pred** can be any expression defined in the ground type. These conditionals are implemented in control, while conditionals expressed within a *control point invocation* are implemented in the data path.

A *control point invocation*, **(S val_1 ... val_N)**, is a function invocation which denotes a parallel assignment to a set of registers and a transfer of control to state **S**. Operations expressed here are implemented in the data path.

Operations on *abstract data types* are expressed as

(op object x y ...)

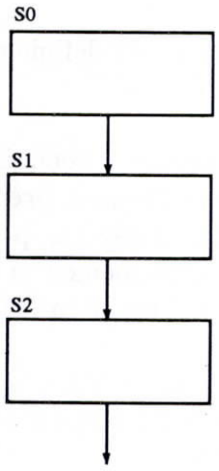
where **op** is an operation defined on the abstract data type **object**, and **x** and **y**, are arguments. For example, operations on a memory object, **MEM**, may be written with the expressions **(MemWrite MEM Addr Data)**, and **(MemRead MEM Addr)**. A stack object, **STACK**, may be written with the expressions **(Push STACK Data)**, **(Pop STACK)**, **(Top STACK)**, and **(Empty? STACK)**.

3.2.1 Expressing Control

The specification is *iterative* - each state definition, *S*, is a conditional expression in which the alternatives are tail-recursive, and is equivalent to the class of schemata associated with finite state machines. The correspondence between the iterative specification and finite state machines is illustrated in the mappings on the following page from partial state definitions to Algorithmic State Machine (ASM) [16] fragments. The ASM notation is derived from software flowchart notation, and provides a means of expressing abstract algorithms while supporting the conversion of the algorithm into hardware. Control flows through a sequence of states, denoted by a rectangle, based on the position in the control algorithm, and the values of the relevant status variables. Given a present state, the next state is determined unambiguously. To express operations on the architecture, operations are placed within the appropriate state rectangle. Conditional branches are denoted with a diamond. Operations on the architecture which occur conditionally are placed in ovals on the appropriate conditional branch. The following constructs show how a simple sequence of state transitions, a conditional branch, and a multi-way branch are expressed in the language. (note: Architectural details have been suppressed for the sake of clarity and are addressed subsequently.)

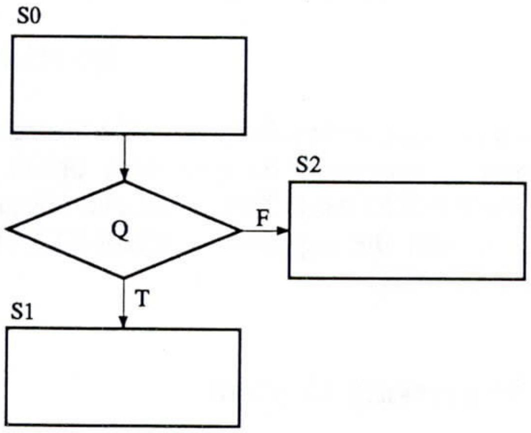
A sequence of state transitions:

```
...
(S0
  (lambda (...)
    (S1 ...)))
(S1
  (lambda (...)
    (S2 ...)))
...
```



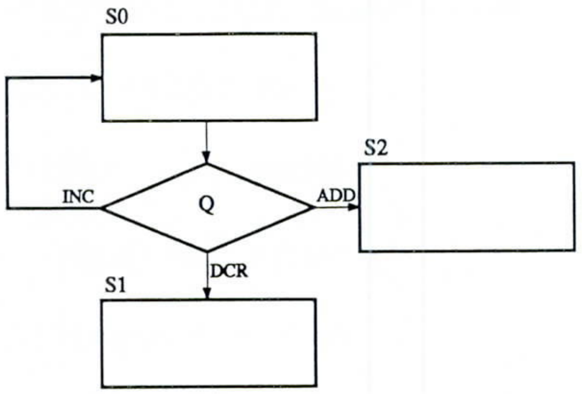
A conditional branch:

```
...
(S0
  (lambda (...)
    (if Q
        (S1 ...)
        (S2 ...))))
...
```



A multi-way branch:

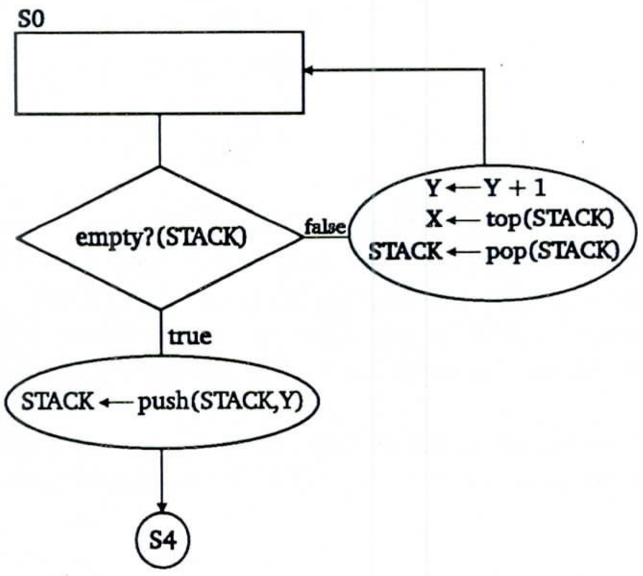
```
...
(S0
  (lambda (...)
    (case Q
      (INC (S0 ...))
      (DCR (S1 ...))
      (ADD (S2 ...)))))
...
```



3.2.2 Expressing Architecture

Architecture is expressed in the *formal parameters*, the *let expression **id/val** pairs*, and ***vals*** in the *control point invocations*. Consider the following state definition and ASM fragment which illustrates how the formal parameters and vals in the control point invocations are used to define the architectural components and operations:

```
...
(S0
 (lambda (X Y STACK)
   (if (empty? STACK)
      (S4 X Y (push STACK Y))
      (S0 (top STACK) (inc Y)
                  (pop STACK)))))
...
```



The formal parameters **(X Y STACK)** specify two registers **X** and **Y**, and a stack **STACK**. When **STACK** is empty, control is passed to state **S4**; the contents of **X** and **Y** remain unchanged; and the value of **Y** is pushed onto the stack. If **STACK** is not empty, control is passed to state **S0**; **X** is updated with the value of the top of stack; the contents of **Y** is incremented; and the stack is popped.

In order to execute the specification the operations **empty?**, **push**, **pop**, **top**, and **inc**, are defined in the *ground type* or *basis* at some level of abstraction. Suppose **X** and **Y** are defined over integers, and **STACK** is represented by a simple list. Then the above operations may be defined by the following function definitions:

```
(define initSTACK '())

(define empty? (lambda (s) (null? s)))

(define push (lambda (x s) (cons x s)))

(define inc (lambda (v) (+ v 1)))

(define top
  (lambda (s)
    (if (empty? s)
        (error 'top "stack is empty")
        (car s))))

(define pop
  (lambda (s)
    (if (empty? s)
        (error 'pop "stack is empty")
        (cdr s))))
```

The level of abstraction at which the ground type is defined is purely a matter of choice. The ground type may just have easily been described in terms of a binary representation, or a combination of different levels of abstractions - **X** may range over binary numbers, while **Y** may be defined over integers. In the course of a design, various levels of abstraction of the basis may be supplanted while the specification remains unaltered.

Combinational terms may also be associated with a signal name using the let expression. In the next example, the **(top STACK)** operation from the previous example is associated with the signal **Z**.

```
...
(S0 (lambda (X Y STACK)
     (let ((Z (top STACK)))
       (if (empty? STACK)
           (S4 X Y (push STACK) Y)
           (S0 Z (inc Y) (pop STACK))))))
...
```

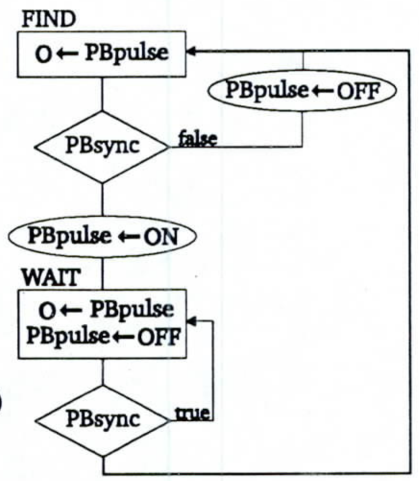
However, signals defined in let expressions are under specified. What is the value of **Z** outside the scope of the let expression in which it is defined? The DDD system defaults the value of **Z** to **nop** which denotes a "do nothing" operation.

3.3 An Example: *Single Pulser*

Consider the specification of the *single pulser* circuit as described in [16], and its corresponding ASM. The example, though simple, illustrates a complete specification. For further examples see [6,15,5,Appendix C]. In the next few sections, this example is taken through the derivation system.

> The SinglePulser senses the depression of the button, **PBsync**, and asserts an output signal, **PBpulse**, for a single clock pulse. Additional assertions of the output are not allowed until after the operator releases the button.

```
(define SinglePulser
  (lambda (PBsync)
    (letrec
      ((FIND
         (lambda (PBpulse)
           (let ((O (out PBpulse)))
             (if (PBsync) (WAIT ON) (FIND OFF)))))
       (WAIT
         (lambda (PBpulse)
           (let ((O (out PBpulse)))
             (if (PBsync) (WAIT OFF) (FIND OFF))))))
      (FIND OFF))))
```



The circuit is defined as a two-state machine: **FIND**, and **WAIT**, that takes a synchronized input assertion from a push button: **PBsync**. The output of the system is **O** which carries the signal **PBpulse**, a synchronized output assertion. The initial invocation of the single pulser algorithm, cycles in the **FIND** state, until a **true** signal is asserted on **PBsync**. Control is then transferred to the **WAIT** state, with the value **ON** being asserted on the **PBpulse** signal. The algorithm cycles in the **WAIT** state, until a **false** is asserted on the **PBsync** signal. Control is then transferred back to **FIND**.

4 Control Abstraction and Architecture

Initial transformations decompose the iterative specification into a control abstraction, called *Select*, and an architectural component, referred to as a *structural specification* defined as a set of mutually recursive stream equations. The control abstraction and structural specification are collectively referred to as a *sequential system*. The sequential system has the same abstract basis as the initial specification, but the interpretation is based on streams. Whereas before, variables ranged over instantaneous values, they now denote sequences of values over time.

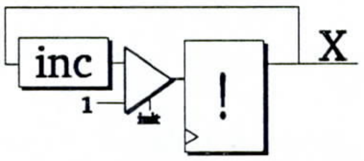
4.1 The Stream Model

Sequential systems are modeled by streams. The symbol **!** denotes an element that has state. In this discussion it is referred to as a *register*.

Consider a counter modeled as a stream.

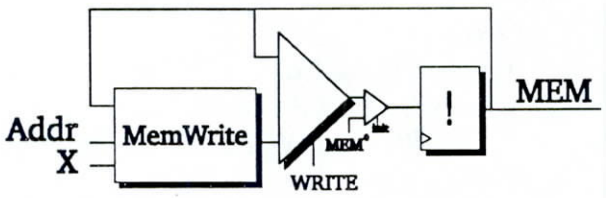
X = 1 ! inc(X)

X denotes a 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 whose output is **X+1**. The stream equation may be characterized by the following circuit:



Another example is a memory modeled as a stream.

MEM = MEM^0 ! (if WRITE (MemWrite MEM Addr X) MEM)



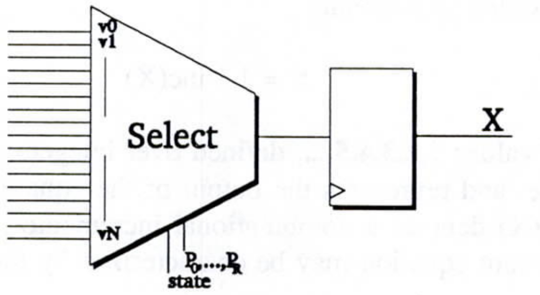
MEM denotes a stream of memories **MEM^0,MEM^1,MEM^2,MEM^3,...**
MEM^0 is the initial memory. If the **WRITE** signal is asserted, **MemWrite** will return a new memory object, with address, **Addr**, updated with value **X**. Otherwise, the memory is returned

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.

The characteristic stream equations derived by DDD have the form:

$$(X \Leftarrow (\text{Select state } P_0 \ldots P_R\ V_0\ V_1 \ldots V_N)) \qquad (a)$$
$$(Y = (\text{Select state } P_0 \ldots P_R\ V_0\ V_1 \ldots V_N)) \qquad (b)$$

The ! symbol together with an initial value are denoted with the $\Leftarrow$ symbol. Equations defined with $\Leftarrow$ have state. Equations with a simple = are combinational. **Select** is a combinational decision combinator which returns one of the possible values, $V_0,\ldots,V_N$ as a function of **state**, and a set of status predicates, $P_0,\ldots,P_R$. **Select** may be viewed as an N-input multiplexor, defined over abstract values, with control signals **state**, and $P_0,\ldots,P_R$. The associated schematic for *(a)* is shown below. The schematic for *(b)* is similar to *(a)* with the omission of the register.



4.2 Deriving a Sequential System

The derivation of a sequential system from the initial specification are outlined in this section. For a formal discussion refer to [9]. The **SinglePulser** specification on page 13 from the previous section is used to illustrate the initial set of transformations.

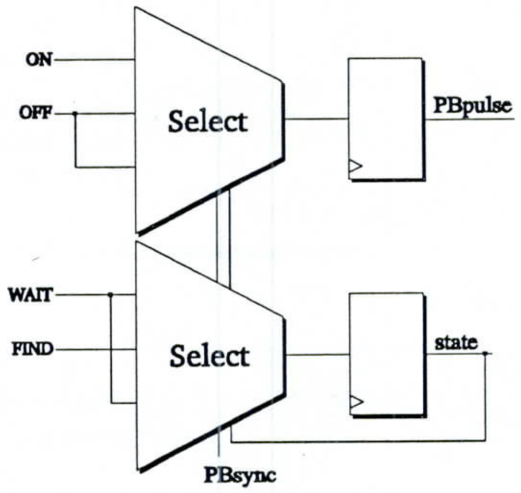
```
(define SinglePulser
  (lambda (PBsync)
    (letrec
      ((FIND
         (lambda (PBpulse)
           (let ((O (out PBpulse)))
             (if (PBsync) (WAIT ON) (FIND OFF)))))
       (WAIT
         (lambda (PBpulse)
           (let ((O (out PBpulse)))
             (if (PBsync) (WAIT OFF) (FIND OFF))))))
      (FIND OFF))))
```

The first step introduces a control token, **state**, to represent each of the state definitions **WAIT** and **FIND**. A *single loop* form of the initial specification is constructed by adding **state** to the formal parameter list, changing each function invocation to a recursive call to **SinglePulser**, and constructing a case statement encoding which function is in control.

```
(define SinglePulser
  (lambda (state PBpulse)
    (case state
      (FIND (let ((O (out PBpulse)))
              (if (PBsync)
                  (SinglePulser WAIT ON)
                  (SinglePulser FIND OFF))))
      (WAIT (let ((O (out PBpulse)))
              (if (PBsync)
                  (SinglePulser WAIT OFF)
                  (SinglePulser FIND OFF)))))))
```

A decision combinator, or control abstraction is then derived by factoring all predicates and function invocations from the single loop form. The result is a control specification, **Select**, and an architectural description, **SinglePulser**, defined as a set of stream equations.

```
(define Select
  (lambda (s p v0 v1 v2)
    (case s
      (FIND (if p v0 v1))
      (WAIT (if p v2 v1)))))

((state ⇐
   (Select state (PBsync) WAIT FIND WAIT))
 (PBpulse ⇐
   (Select state (PBsync) ON OFF OFF))
 (O =
   (Select state (PBsync) (out PBpulse) (out PBpulse)
           (out PBpulse))))
```



(ItrSys->SingleLoop *ItrSys*) → SingleLoop
Takes an iterative system specification and returns the single loop form.

```
DDD> (define sp
'(define singlepulser
   (lambda (pbsync)
      (letrec
        ([find
           (lambda (pbpulse)
              (let ([o (out pbpulse)])
                 (if (pbsync) (wait on) (find off))))]
         [wait
           (lambda (pbpulse)
              (let ([o (out pbpulse)])
                 (if (pbsync) (wait off) (find off))))])
      (find off))))

DDD> (ItrSys->SingleLoop sp)
(define singlepulser
   (lambda (state pbpulse)
      (case state
         [find
          (let ([o (out pbpulse)])
             (if (pbsync)
                 (singlepulser wait on)
                 (singlepulser find off)))]
         [wait
          (let ([o (out pbpulse)])
             (if (pbsync)
                 (singlepulser wait off)
                 (singlepulser find off)))])))
```

(SingleLoop->Select *SingleLoop*) → Select
Takes a single loop specification and returns the control abstraction, *Select*.

```
DDD> (SingleLoop->Select sl)
(define select
   (lambda (s p0 v0 v1 v2 v3)
      (case s
        [find (if p0 v0 v1)]
        [wait (if p0 v2 v3)])))
```

(SingleLoop->StrEqns *SingleLoop*) → StrEqns
Takes a single loop specification and returns the architectural component *StrEqns* - a set of stream equations.

```
DDD> (define sl (ItrSys->SingleLoop sp))
sl

DDD> (SingleLoop->StrEqns sl)
((state <= (select state (pbsync) wait find wait find))
 (pbpulse <= (select state (pbsync) on off off off))
 (o = (select state (pbsync) (out pbpulse) (out pbpulse)
              (out pbpulse) (out pbpulse))))
```

(ItrSys->SeqSys *ItrSys*) → [Select StrEqns]
Takes an iterative system specification and returns a sequential system. The sequential system is also optimized to eliminate redundant inputs to *Select.*

```
DDD> (define sp (ReadFile "SinglePulser"))
sp

> (ItrSys->SeqSys sp)
((define select
    (lambda (s p0 v0 v1 v2)
       (case s
         [find (if p0 v0 v1)]
         [wait (if p0 v2 v1)])))

 ((state <= (select state (pbsync) wait find wait))
  (pbpulse <= (select state (pbsync) on off off))
  (o = (select state (pbsync) (out pbpulse)
               (out pbpulse) (out pbpulse))))

DDD>
```

5 Algebra on Sequential Systems

At this stage in design, a sequential system consisting of a control specification, and an architectural component have been derived. Although this step in the derivation is mechanical, not every sequential system describes an implementable circuit. The identification of like terms may be necessary to reduce the complexity of the circuitry; signals may have to be merged in order to satisfy certain target constraints; abstract data types such as memories, stacks, alu's, or some subprocess may have to be factored from the description in order to construct a circuit defined over more concrete signals; and the partitioning of the design may be necessary to satisfy logical and/or physical aspects of the design. These decisions are made by the designer. DDD is a tool by which these decisions may be explored while preserving the integrity of the specification.

This section presents some of the algebra that has been implemented in DDD to manipulate sequential systems. The algebra is fairly simple, yet powerful, and manipulates design descriptions in a natural way. A set of transformations, *identification*, *merge*, *generalization*, and *distribution*, are defined. From this set, two transformations fundamental to factorization, *general factorization*, and *signal factorization*, are defined. A set of add hoc transformations to *add*, *rename*, *extract*, and *remove* stream equations are also defined. A transformation to *optimize* the sequential system is defined. A transformation to *partially evaluate* the control specification with respect to a stream equation is defined. A set of transformations to derive a *state generator* are defined.

5.1 Transformations

Identification is giving a name to an expression by adding a signal equation for it. Identification of like terms by a single equation has the effect of eliminating redundant circuitry.
Identifying **(inc X)** with **Z** in

(X ⇐ (Select p X (inc X) (inc X)))
(Y ⇐ (Select p Y (inc X) Y))

returns

(X ⇐ (Select p X Z Z))
(Y ⇐ (Select p Y Z Y))
(Z = (Select p ? (inc X) (inc X))

Merge Equations is a merging of signal equations by instantiating don't cares, which are denoted by **'?'**, and like terms. Merging allows multiple signals to share a common bus.
Merging **X** and **Y**

(X = (Select p Z ? X))
(Y = (Select p ? W X))

returns

(XY = (Select p Z W XY))

Generalization is the introduction of don't care arguments to normalize function calls across **Select**. Generalizing

(X = (Select p (f x y) (g u v w))))

returns

(X = (Select p (f' x y ?) (g u v w)))

with f extended to

(define f' (lambda (a b c) (f a b)))

Distribution is the distribution of **Select** over function application.
Distributing **Select** over **add**, and **sub** in

(X = (Select p (add W X) (sub Y Z)))

returns

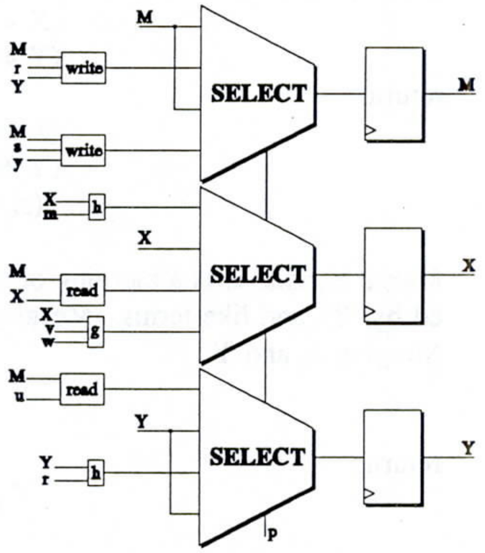
(X = ((Select p add sub) (Select p W Y) (Select p X Z)))

5.2 Factorization

A system factorization encapsulates a subsystem in order to remove some collection of operations from the description. The encapsulated subsystem is called an *abstract component*. The transformation maintains the correct connectivity between the system description and the factored component. This technique of encapsulation yields a circuit defined over more concrete signals.

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.

Consider the partial system description and its corresponding circuit characterization:



(M ⇐ (Select p M (write M r Y) M (write M s y))))
(X ⇐ (Select p (h X m) X (read M X) (g X v w)))
(Y ⇐ (Select p (read M u) Y (h Y r) Y))

5.2.1 General Factorization

Applying a general factorization on operations **h** and **g**

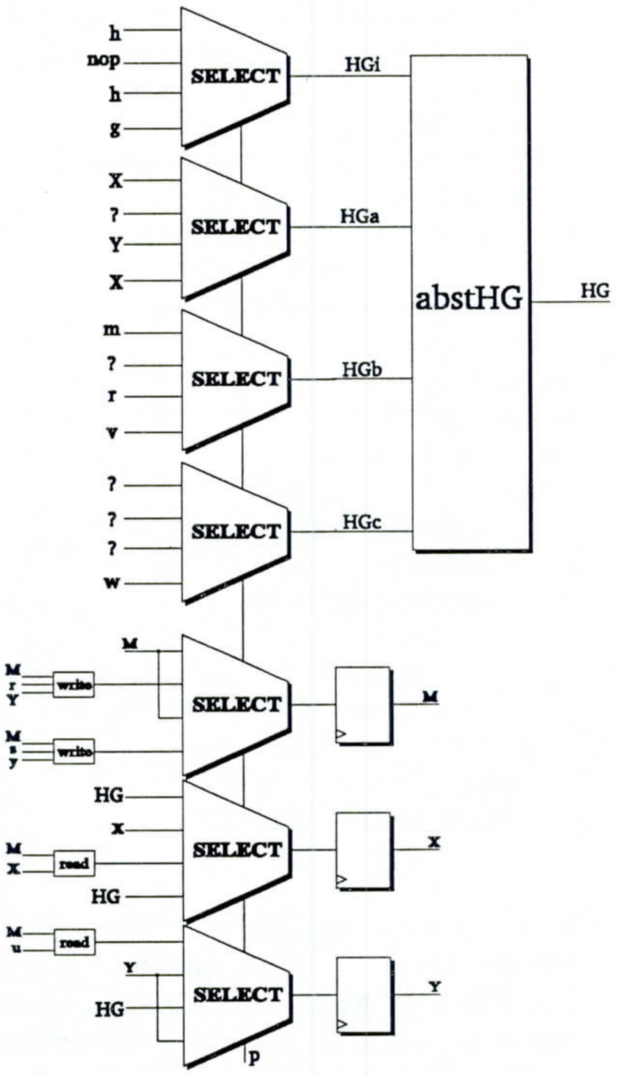
```
(M ⇐ (Select p M (write M r Y) M (write M s y)))
(X ⇐ (Select p (h X m) X (read M X) (g X v w)))
(Y ⇐ (Select p (read M u) Y (h Y r) Y))
```

yields

```
 (HG = (abstHG HGi HGa HGb HGc))
 (HGi = (Select p h nop h g))
(HGa = (Select p X ? Y X))
(HGb = (Select p m ? r v))
(HGc = (Select p ? ? ? w))
  (M ⇐ (Select p M (write M r Y)
                    M (write M s y)))
  (X ⇐ (Select p HG X (read M X) HG))
  (Y ⇐ (Select p (read M u) Y HG Y))
```

where

```
(define abstHG
  (lambda (i a b c)
    (case i
      (nop ?)
      (h (h a b))
      (g (g a b c)))))
```



The subject terms **(g X v w)**, **(h X m)** found in **X**, and **(h Y r)** found in **Y**, are identified and replaced with the output of the subsystem, **HG**. The subject terms are collated, and a set of combinational signals: an instruction signal, **HGi**, and three inputs to the subsystem, **HGa**, **HGb**, and **HGc**, are synthesized. A functional specification of the factored subsystem, **abstHG**, and its application in the description, **(abstHG HGi HGa HGb HGc)** are also synthesized.

The factored component, **abstHG**, becomes a "black box" from the perspective of the system description. The operations **h** and **g** have been encapsulated and only residual signals necessary to communicate with the black box are maintained within the description. The factorization of **h** and **g** represents a design decision which removes these operations from the description. The effect on the system description is a set of stream equations defined over more concrete signals.

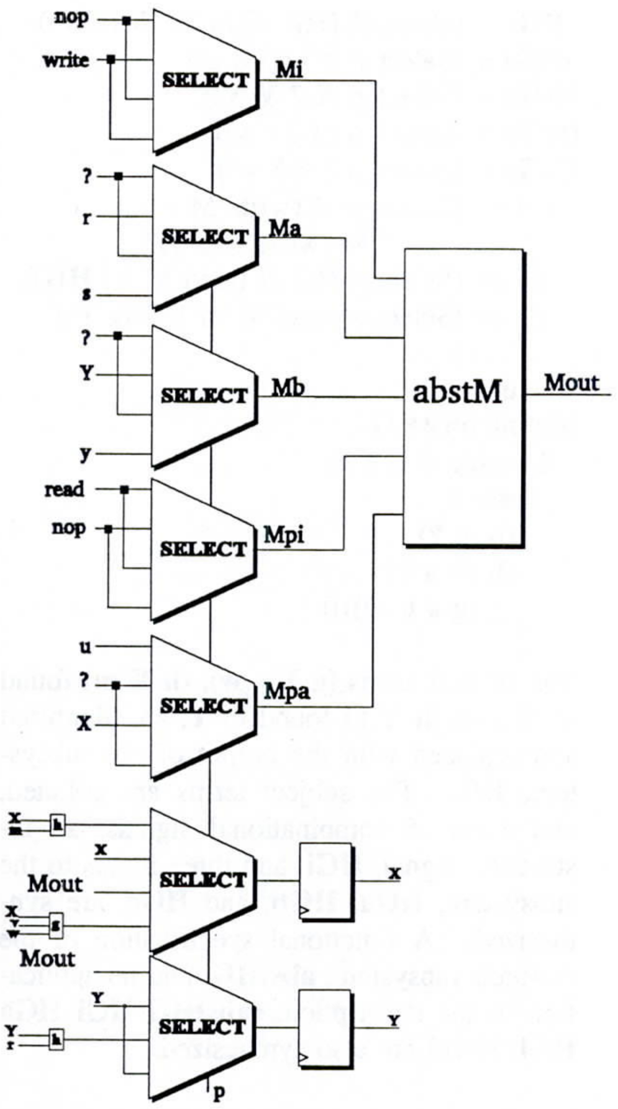
5.2.2 Signal Factorization

Consider again the partial system description. Applying a signal factorization on **M**

```
(M ⇐ (Select p M (write M r Y) M (write M s y))))
(X ⇐ (Select p (h X m) X (read M X) (g X v w)))
(Y ⇐ (Select p (read M u) Y (h Y r) Y))
```

yields

```
(Mout = (abstM Mi Ma Mb Mpi Mpa))
  (Mi = (Select p nop write nop write))
  (Ma = (Select p ? r ? s))
  (Mb = (Select p ? Y ? y))
 (Mpi = (Select p read nop read nop))
 (Mpa = (Select p u ? X ?))
   (X ⇐ (Select p (h X m) X Mout
                  (g X v w)))
   (Y ⇐ (Select p Mout Y (h Y r) Y))
```



where

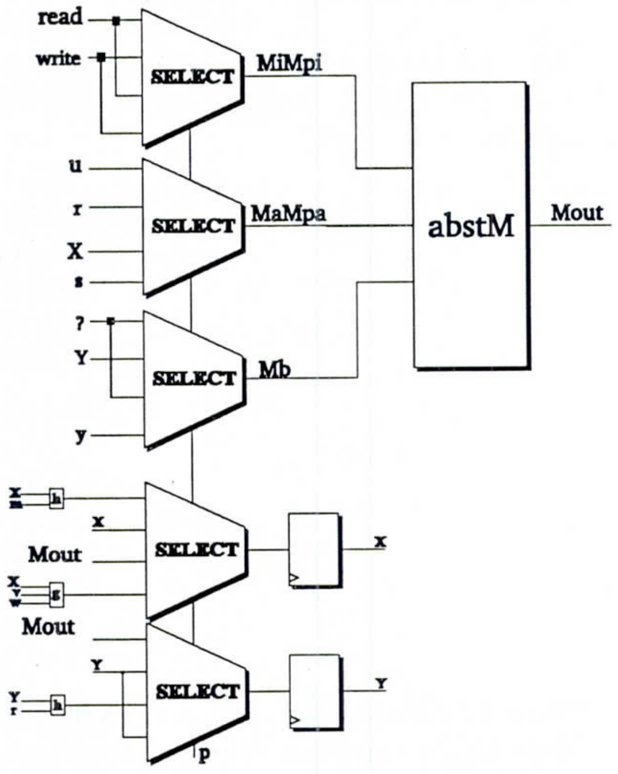
```
(define abstM
 (lambda (i a b c d)
   (letrec
     ((M (Interpret(M i a b)))
      (Interpret
        (lambda (m i a b)
           (case i
              (nop m)
              (write (write m a b))))))
    (case c
      (nop ?)
      (read (read M d))))))
```

Each of the operations found in the defining equation for **M**, such as, **(write M r Y)**, **(write M s y)**, are called *constructors*, since they return a new **M**. Each of the operations on **M** that occur outside the defining equation, such as, **(read M u)**, are called *probes*, since they return values from the object **M**. The factorization encapsulates the signal **M**. Three combinational constructor signals, **Mi**, **Ma**, and **Mb**, and two combinational probe signals, **Mpi**, and **Mpa** are synthesized to communicate with the abstract component. Each occurrence of a probe is substituted with the output of the abstract component **Mout**. A functional specification of the subsystem, **abstM**, and its application in the description, **(abstM Mi Ma Mb Mpi Mpa)**, are also synthesized.

The factorization of **M** leaves five residual signals, **Mi, Ma, Mb, Mpi**, and **Mpa** which communicate with the abstract component **abstM**. Since **abstM** will be implemented with a standard memory component, the **Mi** and **Mpi** represent the read/write instructions to memory, and **Ma, Mpa** represent the address, these signals can be collated into a single instruction, and address stream.

Merging the instructions **Mi, Mpi**, and addresses **Ma**, and **Mpa** returns

```
   (Mout = (abstM MiMpi MaMpa Mb))
  (MiMpi = (Select p read write
                      read write))
 (MaMpa = (Select p u r X s))
     (Mb = (Select p ? Y ? y))
      (X ⇐ (Select p (h X m) X Mout
                      (g X v w)))
      (Y ⇐ (Select p Mout Y (h Y r) Y))

where
(define abstM
 (lambda (i a b)
   (letrec
    ((M = (Interpret M i a b))
     (Interpret
       (lambda (m i a b)
         (case i
           (nop m)
           (read m)
           (write (write m a b))))))
     (read M a))))
```



Factorization is fundamental to the derivation of designs in DDD. The encapsulation of subsystems is a technique of information hiding analogous to an abstract data type. As complex objects are factored, the description is refined to more concrete terms. It is this stepwise refinement of the description which provides the mechanism by which designs are transformed to an implementation.

This design process continues until the design has been refined into a form which represents an implementable design. The notion of *implementable* depends on both the capabilities of the technology tools used to map the description to some target, and the designer's decision as to what to implement within the system description and what implement outside the system description.

5.3 Optimization

Optimizations on a design are done at both the boolean equation level[1], and the sequential system level. At the sequential system level, transformations may be applied to reduce the complexity of the design by identifying like terms, merging stream equations, minimizing stream equations, and minimizing selectors. Transformations to identify like terms, and merge stream equations are discussed in Section 5.1. Transformations to minimize selectors, and stream equations are discussed in the following sections.

5.3.1 Minimizing Selectors

Selectors are essentially conditional expressions. If-expressions in a selector are minimized using the conditional simplification rule: **(if p r r) → r.** For example, given the following selector:

```
(define Select
  (lambda (s p q v0 v1 v2)
    (case s
      (s0 (if p v0 v2))
      (s1 (if q v1 v1)))))
```

returns

```
(define Select
  (lambda (s p v0 v1 v2)
    (case s
      (s0 (if p v0 v2))
      (s1 v1))))
```

The conditional test **(if q v1 v1)** in **Select** simplifies to **(if q v1 v1)** which is simply **v1**. Consequently, the predicate **q** becomes unnecessary and is removed from the expression.

5.3.2 Minimizing Stream Equations

Just as **merge** allows for the merging of stream equations, it is useful to merge redundant inputs to a selector in order to minimize a set of stream equations.

Consider the following sequential system:

[1] Boolean equation minimization using 'espresso'.

```
(define Select
  (lambda (s p q v0 v1 v2)
    (case s
      (s0 (if p v0 (if q v1 v2)))
      (s1 v0))))

(X = (Select s p q X Z Z))
(Y = (Select s p q Y Q Q))
```

Looking at both **X** and **Y** simultaneously, notice that there are two instances where **X** and **Y** are updated with the values **Z** and **Q**, respectively. The last two inputs to **Select**, **v1** and **v2**, are the same. In order to eliminate the redundant **Z** term in **X**, and the redundant **Q** term in **Y**, the sequential system is rewritten as

```
(define Select
  (lambda (s p v0 v1)
    (case s
      (s0 (if p v0 v1))
      (s1 v0))))

(X = (Select s p X Z))
(Y = (Select s p Y Q))
```

The conditional test **(if q v1 v2)** in **Select** simplifies to **(if q v1 v1)** which is simply **v1**. The equations for **X** and **Y** reduce and the **q** predicate becomes unnecessary and is removed from the stream equations, as well as the selector.

The effectiveness of this optimization depends on the arrangement of possible values for all the stream equations associated with a particular selector. Redundant terms in a single equation may not be reduced using this technique if the redundancy does not occur in all the stream equations simultaneously.

Consider the following sequential system:

```
(define Select
  (lambda (s p q v0 v1 v2)
    (case s
      (s1 (if p v0 (if q v1 v2)))
      (s2 (if p v1 v2)))))

(X = (Select s p q Z Z X))
(Y = (Select s p q Y Q Q))
```

In this example there are redundant terms in each of the equations. However, when you consider the system as a whole, the inputs to **Select** are unique and cannot be merged. **v0** corresponds to **Z** and **Y**, **v1** to **Z** and **Q**, and **v2** to **X** and **Q**, in equations **X** and **Y** respectively.

The transformations to minimize stream equations and selectors are fundamentally different from the transformations previously described since it affects both the control specification, as well as the structural specification. Any changes to the selector may effect the stream equations, and any changes to the stream equations may effect the selector. The optimization is a global optimization on the sequential system.

5.4 Partitioning

Partitioning provides a powerful tool for reorganizing the design. The partitioning of a sequential system may be motivated by various aspects of design. Some of the considerations are the logical and physical organization of the design; the internal connectivity of the design; the communication specification of the design; and constraints imposed by the target technology. DDD supports the partitioning of a design along stream equation boundaries. How equations are grouped is determined by the designer and becomes a part of the design space which must be explored. Each partition is viewed as a distinct sequential system - each with is own selector and set of stream equations. Naturally, partitions with identical selectors can share the same selector. Since each partition is a distinct sequential system, they can be optimized independently.

Consider the sequential system:

```
(define Select
  (lambda (s p q v0 v1 v2)
    (case s
      (s0 (if p v0 (if q v1 v2)))
      (s1 v0))))

(X = (Select s p q X X Z))
(Y = (Select s p q X Y Y))
(Z = (Select s p q W Z Z))
```

Suppose **X** and **Y** are grouped together into one partition, and **Z** in another. This may be a reasonable grouping since **X** and **Y** share signals, and since **Z** is not dependent on either **X** nor **Y**. Then each partition may be optimized independently resulting in

```
(define Select
  (lambda (s p q v0 v1 v2)
    (case s
      (s0 (if p v0 (if q v1 v2)))                    Partition 1
      (s1 v0))))

(X = (Select s p q X X Z))
(Y = (Select s p q X Y Y))


(define Select_1
  (lambda (s p v0 v1)
    (case s
      (s0 (if p v0 v1))                               Partition 2
      (s1 v0))))

(Z = (Select_1 s p W Z))
```

An alternative partition is to group **Y** and **Z** together. Optimizing each of the partitions independently returns

```
(define Select_1
  (lambda (s p q v0 v1)
    (case s
      (s0 (if p v0 (if q v0 v1)))                     Partition 1
      (s1 v0))))

(X = (Select_1 s p q X Z))


(define Select_2
  (lambda (s p v0 v1 v2)
    (case s
      (s0 (if p v0 v1))                               Partition 2
      (s1 v0))))

(Y = (Select_2 s p X Y))
(Z = (Select_2 s p W Z))
```

Both of the partitions simplify. In general this is usually the case. However, the added complexity of two controllers, and the inter-partition communication may not offset the simplification of the equations. These questions are answered as designs are pushed through to hardware.

5.5 Partial Evaluation of Selectors

The partial evaluation of selectors with respect to a particular stream equation returns a specialized selector which computes the same function as the original equation. The transformation is an extension to the creation of a partition containing a single stream equation. This technique is commonly used to derive the classical *state generator*, and the various *instruction generators* found in a typical design.

5.5.1 Deriving a State Generator

Consider the selector, **Select**, and the **state**, stream from the **SinglePulser** sequential system from page 17:

```
(define Select
  (lambda (s p v0 v1 v2)
    (case s
      (FIND (if p v0 v1))
      (WAIT (if p v2 v1)))))

(...
 (state ⇐ (Select state (PBsync) WAIT FIND WAIT))
 ...)
```

Partial evaluation of **Select**, with respect to **state** returns

```
(define NextState
  (lambda (s p)
    (case s
      (FIND (if p WAIT FIND))
      (WAIT (if p WAIT FIND)))))

(state ⇐ (NextState state (PBsync)))
```

NextState is derived from **Select**. The **v0**, **v1**, and **v2** parameters to **Select** are instantiated with the constants **WAIT**, **FIND**, and **WAIT** respectively. **NextState** will return the values **WAIT** or **FIND** as a function of the present state and predicates. (Note: In this example, the case statement in the function definition **NextState** may be simplified. However, at present, DDD only simplifies if-expressions).

(Identify *Exp newStrEqnName StrEqns*) → StrEqns
Takes an expression, a new stream equation name, and a set of stream equations, and returns a set of stream equations. The function has the effect of identifying an expression with a stream equation name.

```
DDD> (define streqns
        '((x = (select p x (inc y) x (dcr x)))
          (y = (select p y y (dcr x) y)))
streqns

DDD> (Identify '(inc y) 'z streqns)
((x = (select p x z x (dcr x)))
 (y = (select p y y (dcr x) y))
 (z = (select p ? (inc Y) ? ?)))

DDD>
```

(Generalize *StrEqn*) → StrEqn
Takes a stream equation and returns a stream equation. The function has the effect of generalizing function calls within a stream equation. Don't care arguments, "?", are introduced to generalize the function calls across *Select*.

```
DDD> (define streqn
        '(x = (select p (f x y) (g u v w))))
streqn

DDD> (Generalize streqn)
(x = (select p (f x y ?) (g u v w)))

DDD>
```

(MergeOperations *newStrEqnName OpSet StrEqns*) → StrEqns
Takes a new stream equation name, a set of operations, and a set of stream equations, and returns a set of stream equations. The function has the effect of merging a set of operations into a stream equation. The new equation is given the name *newStrEqnName*. The set of operations are specified as a list of operation names in *OpSet*. If more than one operation must be done in parallel, multiple streams are created. The parallel operations are allocated using a naive allocation strategy.

```
DDD> (define streqns
        '((x = (select p ? (inc x) (dcr y)))
          (y = (select p y (inc x) ?))))

DDD> (MergeOperations 'z '(inc dcr) streqns)
((x = (select p ? z z))
 (y = (select p y z ?))
 (z = (select p ? (inc x) (dcr y))))
```

```
DDD> (define streqns
        '((x <= (select p ? (inc x) (dcr y)))
          (y <= (select p y (inc y) ?))))

DDD> (MergeOperations 'z '(inc dcr) streqns)
((x <= (select p ? z0 z0))
 (y <= (select p y z1 ?))
 (z0 = (select p ? (inc x) (dcr y)))
 (z1 = (select p ? (inc y) ?)))
```

(MergeStrEqns *StrEqn1 StrEqn2 newStrEqnName StrEqns*) → StrEqns
Takes two stream equations, a new stream equation name, and a set of stream equations, and returns a set of stream equations. The function has the effect of merging two stream equations.

```
DDD> (define streqns
        '((x = (select p ? ? w w ff))
          (y = (select p x ? w ? ?))))
streqns

DDD> (MergeStrEqns 'x 'y 'z streqns)
((z = (select p z ? w w ff)))
```

(Distribute *StrEqn*) → Exp
Takes a stream equation and returns an expression. The function has the effect of distributing *select* over function application.

```
DDD> (define streqn
        '(x = (select p (add w x) (sub y z))))
streqn

DDD>  (Distribute streqn)
(x = ((select p add sub) (select p w y) (select p x z)))
```

(AbstractOperations *AbsCompName OpSet StrEqns*) → StrEqns
Takes an abstract component name, a set of operations to be encapsulated, and a set of stream equations, and returns a set of stream equations. The function has the effect of implementing a *general factorization*. The abstract component name is the name given to the encapsulated subsystem. The collection of operations to be encapsulated is specified as a list of operation names in *OpSet*.

The factorization assumes that a single component, can perform only a single operation at any given time. If the set of operations identified in the specification require that more than one operation be performed at one time, multiple abstract components are generated.

```
DDD> (define streqns
  '((m <= (select p m (write m r y) m (write m s y)))
    (x <= (select p (h x m) x (read m x) (g x v w)))
    (y <= (select p (read m u) y (h y r) y))))
streqns

DDD> (AbstractOperations 'hg '(h g) streqns)
((m <= (select p m (write m r y) m (write m s y)))
 (x <= (select p hg x (read m x) hg))
 (y <= (select p (read m u) y hg y))
 (hg-i = (select p h nop h g))
 (hg-v0 = (select p x ? y x))
 (hg-v1 = (select p m ? r v))
 (Hg-v2 = (select p ? ? ? w)))
```

(AbstractStrEqn *StrEqnName StrEqns*) → StrEqns
Takes a stream equation name, a set of stream equations, and returns a set of stream equations. The function has the effect of implementing a *signal factorization.*

```
DDD> (AbstractStrEqn 'm streqns)
((m-i = (select p nop write nop write))
 (m-v1 = (select p ? r ? s))
 (m-v2 = (select p ? y ? y))
 (x <= (select p (h x m) x m-out (g x v w)))
 (y <= (select p m-out y (h y r) y))
 (m-probe-i = (select p read nop read nop))
 (m-probe-v1 = (select p u ? x ?)))
```

(AbstractMemory *EqnName StrEqns*) → StrEqns
Takes a memory stream equation name, and a set of stream equations, and returns a set of stream equations. The function has the effect of implementing a signal factorization, followed by a merging of signals for the constructors and probes to form single instruction and address.

```
DDD> (AbstractMemory 'm streqns)
((m-i = (select p read write read write))
 (m-a = (select p u r x s))
 (m-d = (select p ? y ? y))
 (x <= (select p (h x m) x m-out (g x v w)))
 (y <= (select p m-out y (h y r) y)))
```

(AddStrEqn *StrEqn StrEqns*) → StrEqns
Adds a stream equation to a set of stream equations.

```
DDD> (define streqn
        '(r = (select p u v w)))
streqn

DDD> (AddStrEqn streqn streqns)
((r = (select p u v w))
 (x = (select p x y z))
 (y = (select p y z x))
 (z = (select p z x y)))
```

(ExtractStrEqn *StrEqnName StrEqns*) → StrEqn
Takes a stream equation name, and a set of stream equations, and returns the named stream equation.

```
DDD> (define streqns
        '((x = (select p x y z))
          (y = (select p y z x))
          (z = (select p z x y))))
streqns

DDD> (ExtractStrEqn 'x streqns)
(x = (select p x y z))
```

(RemoveStrEqn *StrEqnName StrEqns*) → StrEqns
Removes a stream equation from a set of stream equations.

```
DDD> (RemoveStrEqn 'x streqns)
((y = (select p y z x))
 (z = (select p z x y)))
```

(RenameStrEqn *oldStrEqnName newStrEqnName StrEqns*) → StrEqns
Renames a stream equation in a set of stream equations.

```
DDD> (RenameStrEqn 'x 'newx streqns)
((newx = (select p newx y z))
 (y = (select p y z newx))
 (z = (select p z newx y)))
```

(OptimizeSEL *Select*) → Select
Takes a selector, and simplifies if-expressions within the selector using the conditional simplification rule: **(if p r r)** → **r**. At present, case statements are not simplified.

```
DDD> (define sel
        '(define select
            (lambda (s p q v0 v1 v2)
              (case s
                (S0 (if p v0 v2))
                (S1 (if p v0 (if q v1 v1)))))))

DDD> (OptimizeSEL sel)
(define select
   (lambda (s p v0 v1 v2)
      (case s
         (S0 (if p v0 v2))
         (S1 (if p v0 v1))))))
```

(OptimizeSeqSys *Select StrEqns*) → [Select StrEqns]
Takes a selector, and its corresponding set of stream equations, and returns a sequential system. The function has the effect of eliminating redundant data transfers. The function side-effects the file system by updating the **predinfo** (predicate information) file to reflect the reduced number of register transfers.

```
DDD> (define sel
        '(define select
            (lambda (s p q v0 v1 v2)
              (case s
                (s0 (if p v1 v0))
                (s1 (if p v0 (if q v1 v2)))))))
sel

DDD> (define streqns
        '((x = (select s p q x z z))
          (y = (select s p q y q q)))
streqns

DDD> (OptimizeSeqSys sel streqns)
((define select_1
   (lambda (s p v0 v1)
     (case s
       (s0 (if p v1 v0))
       (s1 (if p v0 v1)))))
 ((x = (select_1 s p x z))
  (y = (select_1 s p y q))))
```

(PartialEval *StrEqn Select*) → Select
Takes a stream equation and a selector, and returns a selector. The function has the effect of partially evaluating *Select* with respect to *StrEqn*. The function instantiates the **V** arguments in *Select* to the corresponding values from *StrEqn*, and returns a specialized selector with a new name composed of the name of the stream equation prefixed with **next**.

```
DDD> state
(state <== (select s (pbsync) wait find wait))

DDD> sel
(define select
  (lambda (s p v0 v1 v2)
     (case s
       [find (if p v0 v1)]
       [wait (if p v2 v1])))

DDD> (PartialEval state sel)
(define next-state
  (lambda (s p)
     (case s
       [find (if p wait find)]
       [wait (if p wait find)])))
```

(ExpandFuncDef *FuncDef StrEqns*) → StrEqns
Takes a function definition and a set of stream equations, and returns a set of stream equations. The function has the effect of replacing all applications of the defined function, *FuncDef*, with its body with arguments instantiated.

```
DDD> (define streqns
        '((x = (select p x y z))
          (y = (select p y (add x y) x))
          (z = (select p z z (add x z)))))

DDD> (Expand '(define add
                (lambda (a b) (+ a b))) streqns)
((x = (select p x y z))
 (y = (select p y (+ x y) x))
 (z = (select p z z (+ x z))))
```

(GroupStrEqns *StrEqns Order*) → [[StrEqns]...]
Takes a set of stream equations, and an order, and returns the set of stream equations grouped according to the ordering, **Order**. Where **Order** is a list of lists of stream equation names.

```
DDD> (define streqns
        '((x = (select p x y z))
          (y = (select p y (add x y) x))
          (z = (select p z z (add x z))))

DDD> (GroupStrEqns streqns '([x y] z))
([(x = (select p x y z))
  (y = (select p y (add x y) x))]
 [(z = (select p z z (add x z)))])
```

6 Register Transfer Table

A *register transfer table* (RTT) is an abstraction of the sequential system. It is a representation which reflects the abstraction by which boolean equations are derived within the system, and provides a framework in which designs may be reasoned.

A sequential system is composed of a set of mutually recursive stream equations, defining the architecture of the design, and a control abstraction, *Select* which computes the next value for each of the streams, based on a set of status signals. *Select* is applied to each of the stream equations simultaneously. *Select* takes as arguments, the common status signal of the system and a list of all the possible values that may occur for that a particular stream, and based on status, returns one of the possible values. Since Select is applied to all the stream equations, and the status is common, *Select* will return the ith possible value for each of the equations at the same time.

Consider the following example derived from the single pulser described on page 17.

```
(define Select
  (lambda (s p v0 v1 v2)
    (case s
      (FIND (if p v0 v1))
      (WAIT (if p v2 v1)))))
```

```
(state ⇐ (Select state (PBsync) 'WAIT 'FIND 'WAIT))
(PBpulse ⇐ (Select state (PBsync) 'ON 'OFF 'OFF))
```

Each of the signals **State**, and **PBpulse**, are evaluated simultaneously with the next value determined by **Select**. If **State** = **FIND**, and **PBsync** = **false**, **Select** will return the $\mathbf{V_1}$ value in both equations - **State** and **PBpulse** will be updated with the values **FIND** and **OFF** respectively. If **State** = **FIND**, and **PBsync** = **true**, **Select** will return the $\mathbf{V_0}$ value in both equations - **State** and **PBpulse** will be updated with the values **WAIT** and **ON** respectively. Similarly, **Select** will return the $\mathbf{V_2}$ value if **State** = **WAIT** and **PBsync** = **true**.

Since the status signals to *Select* are common to all the stream equations, *Select* will always return the ith value for all the stream equations. An appropriate transformation would be to abstract *Select*. The abstraction develops *Select* as a command generator, CMD. Whereas before, *Select* was a function of status, and returned the **i**th next value, it now simply returns a command signal **i**. A RTT is constructed by creating a column in the table for each stream equation, and assigning the corresponding **i** to each of the possible values in each of the stream equations. Each stream equation corresponds to a column in the RTT, and each **i** corresponds to a row in the RTT. The command generator, CMD, issues a command to the RTT which performs the corresponding register transfer.

In general, the abstraction to a register transfer table takes a sequential system of the form

```
(define Select
  (lambda (s $p_0$ $p_1$ ... $p_R$ $v_0$ $v_1$ ... $v_N$)
    (case s
      ($S_0$ (if $p_0$ $v_I$ $v_J$))
      ($S_1$ (if $p_1$ (if $p_2$ $v_K$ ...)))
      ...
      ($S_Q$ ...))))

(StrEqn$_1$ <= (Select status $W_0$ $W_1$ $W_2$ .... $W_N$))
(StrEqn$_2$ <= (Select status $X_0$ $X_1$ $X_2$ .... $X_N$))
...
(StrEqn$_M$ <= (Select status $Z_0$ $Z_1$ $Z_2$ .... $Z_N$))
```

And derives a command generator, and RTT of the form

```
(define CMD
  (lambda (s status)
    (case s
      (S0 (if p0 vI vJ))
      (S1 (if p1 (if p2 vK...)))
      ...
      (SQ ...))))
```

CMD	StrEqn1	StrEqn2	...	StrEqnM
v0 v1 v2 ... vN	W0 W1 W2 ... WN	X0 X1 X2 ... XN	... ...	Z0 Z1 Z2 ... ZN

Returning to the **SinglePulser** example: **CMD** is derived from **Select**. A RTT composed of two columns from the stream equations **State** and **PBpulse**, and a row for each of the possible register transfers **v0**, **v1**, and **v2**, is derived from the stream equations.

```
(define CMD
  (lambda (s p)
    (case s
      (Find (if p v0 v1))
      (Wait (if p v2 v1)))))
```

CMD	State	PBpulse
v0	Wait	On
v1	Find	Off
v2	Wait	Off

6.1 Algebra on RTTs (Register Transfer Tables)

In DDD, transformations have been developed to manipulate sequential systems. However, it is often useful to view these transformations as operating on RTTs. Consequently, transformations have been developed to translate from stream equations to RTTs. Transformations on sequential systems can be viewed as transformations on RTTs.

Identification introduces a new column in the RTT.
Identifying **(inc X)** with **Z** in

CMD	X	Y
0	X	Y
1	(inc X)	(inc X)
2	(inc X)	Y

returns

CMD	X	Y	Z
0	X	Y	?
1	Z	Z	(inc X)
2	Z	Y	(inc X)

Merge merges two columns in a RTT by instantiating don't cares (?), and like terms.
Merging **X** and **Y** in

CMD	X	Y
0	Z	?
1	?	W
2	X	X

returns

CMD	X_Y
0	Z
1	W
2	X_Y

Generalization introduces don't care arguments to normalize function calls in a column.
Generalizing **f** and **g** in **X**

CMD	X
0	(f x y)
1	(g u v w)

returns

CMD	X
0	(f′ x y ?)
1	(g u v w)

Merge operations merges a set of operations into a column.
Merging **h** and **g** in

CMD	M	X	Y
0	M	(h X m)	(read M u)
1	(write M r Y)	X	Y
2	M	(read M X)	(h Y r)
3	(write M s y)	(g X v w)	Y

returns

CMD	M	X	Y	HG
0	M	HG	(read M u)	(h X m)
1	(write M r Y)	X	Y	?
2	M	(read M X)	HG	(h Y r)
3	(write M s y)	HG	Y	(g X v w)

Abstract stream equation encapsulates a column and introduces a set of columns which specify communication with an abstract component.
Factoring **M** in

CMD	M	X	Y
0	M	(h X m)	(read M u)
1	(write M r Y)	X	Y
2	M	(read M X)	(h Y r)
3	(write M s y)	(g X v w)	Y

returns

CMD	Mi	Ma	Mb	Mpi	Mpa	X	Y
0	nop	?	?	read	u	(h X m)	M
1	write	r	Y	nop	?	X	Y
2	nop	?	?	read	X	M	(h Y r)
3	write	s	y	nop	?	(g X v w)	Y

Abstract operations encapsulates a set of operations and generates a set of columns to specify the communication with the abstract component.
Factoring **h** and **g** in

CMD	M	X	Y
0	M	(h X m)	(read M u)
1	(write M r Y)	X	Y
2	M	(read M X)	(h Y r)
3	(write M s y)	(g X v w)	Y

returns

CMD	M	X	Y	HGi	HGa	HGb	HGc
0	M	HG	(read M u)	h	X	m	?
1	(write M r Y)	X	Y	nop	?	?	?
2	M	(read M X)	HG	h	Y	r	?
3	(write M s y)	HG	Y	g	X	v	w

Optimizations on RTT's can eliminate redundant rows. These operations affect the **CMD** generator and it is updated appropriately.
Merging rows in

```
(define CMD
  (lambda (s p q)
    (case s
      (s0 (if p 0 (if q 1 2)))
      (s1 0))))
```

CMD	X	Y
0	X	Y
1	Z	Q
2	Z	Q

returns

```
(define CMD
  (lambda (s p)
    (case s
      (s0 (if p 0 1))
      (s1 0))))
```

CMD	X	Y
0	X	Y
1	Z	Q

The RTT may be *partitioned* along column boundaries. A set of columns may be grouped according to some partitioning strategy. Partitioning column **X** and columns **Y** and **Z**, and updating the respective CMD generators in

```
(define CMD
  (lambda (s p q)
    (case s
      (s0 (if p 1 (if q 2 3)))
      (s1 0))))
```

CMD	X	Y	Z
0	Z	X	Z
1	(inc X)	Y	?
2	Z	?	(inc Y)
3	Z	X	Z

returns

```
(define CMD_1
  (lambda (s p)
    (case s
      (s0 (if p 1 0))
      (s1 0))))
```

CMD_1	X
0	Z
1	(inc X)

CMD	Y	Z
0	X	Z
1	Y	?
2	?	(inc Y)
3	X	Z

Partially evaluating the CMD generator with respect to a column in the RTT has the effect of moving a column from the RTT to a CMD generator.

```
(define CMD
  (lambda (s p)
    (case s
      (s0 (if p 0 1))
      (s1 (if p 1 2)))))
```

CMD	X	Y
0	X	Y
1	?	?
2	Y	?

returns

```
(define CMD-x
  (lambda (s p)
    (case s
      (s0 (if p X ?))
      (s1 (if p ? Y)))))
```

CMD	Y
0	Y
1	?
2	?

(StrEqns->RTT *StrEqns OutFile*) → ()||OutFile
Takes a set of stream equations and creates an output file. *OutFile* is the formatted register transfer table.

```
DDD> (define streqns
       '((x = (select p x y z y))
         (y = (select p y z y z))
         (z = (select p z y x y))))
streqns

DDD> (StrEqns->RTT streqns "streqns.rtt")
The total line width of table = 6
()
```

streqns.rtt

```
    X Y Z
 0) X Y Z
 1) Y Z Y
 2) Z Y X
 3) Y Z Y
```

(RTT->StrEqns *Regs RTT*) → StrEqns
Takes a set of register names, and a register transfer table (in s-exp format), and returns a set of stream equations. The function loads the **predinfo** (predicate information) file to reconstruct the stream equations.

```
DDD> (define rtt
       '((x y z) (x y z) (y z y) (z y x) (y z y)))
rtt

DDD> (RTT->StrEqns '(x y) rtt)
((x <= (select p x y z y)
 (y <= (select p y z y z)
 (z  = (select p z y x y))
```

7 Projection

Projection is defined as a mapping of a sequential system defined over an abstract basis, given a target representation, to a sequential system defined over the target basis. The mapping is intended to be meaning preserving and defines a syntactic correspondence between each term in the abstract basis to corresponding terms in the target basis. This notion can be expressed in the following example:

Consider a simple addition expression **(+ a b)** denoted by **S**. Then a projection function **P: S → S′**, maps each term in **S** to a corresponding term in **S′**, defined by

$$\mathbf{P{:}\{a \rightarrow a',\ b \rightarrow b',\ \lambda xy.{+}xy \rightarrow \lambda xy.{+}'xy\}}$$

derives **(+′ a′ b′).**

DDD takes a sequential system $C_\pi \bullet S_\pi /\!/ F$, which is defined over some abstract basis, to a collection of sequential subsystems defined over a binary representation, $C_\beta \bullet S_\beta /\!/ F$. Transformations to project stream equations and selectors have been defined. The mapping from the abstract basis to the binary basis is input to the system and is developed by the designer. Representations, like the original specification, are assumed to be correct, thus, the correctness of the representations are not secured by DDD - DDD simply provides a tool by which the mechanical projection is automated.

It is at this stage in the derivation that representations necessary to move from one level of description to another are incorporated. However, the methodology does not preclude the declaration of representations at any prior stage in the design phase.

7.1 Projecting Selectors

Selectors, such as the command generator described in the RTT abstraction (Section 6), and the next-state generator described in Section 5.5, may also be projected. The projection of selectors to a binary representation is treated similar to the projection of stream equations. A projection map, similar to the one for stream equations, is defined to map the abstract values of the selectors to the target representation.

For example, given the command generator from Section 6 (page 44)

```
(define cmd
  (lambda (s p)
    (case s
      (s0 (if p v1 v2))
      (s1 v0))))
```

and a set of representations for **v0**, **v1**, and **v2**:

$$v0 \rightarrow [0\ 0]$$
$$v1 \rightarrow [0\ 1]$$
$$v2 \rightarrow [1\ 0]$$

projecting, returns two copies of **cmd**, one for each bit.

```
(define cmd_bit0                    (define cmd_bit1
  (lambda (s p)                       (lambda (s p)
    (case s                             (case s
      (s0 (if p 0 1))                     (s0 (if p 1 0))
      (s1 0))))                           (s1 0))))
```

7.2 Projecting Stream Equations

Consider the set of stream equations

$$(X = (\text{Select}\ p\ f\ g\ H))$$
$$(Y = (\text{Select}\ p\ u\ W\ u))$$

and suppose that **f**, **g**, and **u** are symbolic constants, and **X**, **Y**, **H**, and **W** are variables defined over some abstract base type. The following binary representation may be imposed on the equations: **f**, **g**, and **u** are bit constants with values 000, 001, and 1 respectively. **X** is a 3-bit signal and **Y** is a 1-bit signal. **H** and **W** are inputs to the system. This representation is encoded by a projection function that maps symbolic values in the abstract basis to values in a binary basis.

$$X \rightarrow [X_{bit2}\ X_{bit1}\ X_{bit0}]$$
$$Y \rightarrow [Y_{bit0}]$$
$$f \rightarrow [0\ 0\ 0]$$
$$g \rightarrow [0\ 0\ 1]$$
$$u \rightarrow [1]$$
$$H \rightarrow [H_{bit2}\ H_{bit1}\ H_{bit0}]$$
$$W \rightarrow [W]$$

The resulting set of stream equations are

$$(X_{bit0} = (\text{Select}\ p\ 0\ 1\ H_{bit0}))$$
$$(X_{bit1} = (\text{Select}\ p\ 0\ 0\ H_{bit1}))$$
$$(X_{bit2} = (\text{Select}\ p\ 0\ 0\ H_{bit2}))$$
$$(Y_{bit0} = (\text{Select}\ p\ 1\ W\ 1))$$

The mapping function may describe mappings for more complex terms than variables and symbolic constants. For example, the abstract stream equations may contain a bitwise **or** operation. The corresponding projection would be the binary representation of **or**.

$$(\text{or } Y\ W) \rightarrow [(Y_{bit0} + W)]$$
$$(\text{or } X\ H) \rightarrow [(X_{bit2} + H_{bit2})\ (X_{bit1} + H_{bit1})\ (X_{bit0} + H_{bit0})]$$

The entry into the projection table is more general. Operations are specified with the function name on the left hand side, and a lambda expression on the right hand side which takes the same arguments as the operation to be projected, and returns the expression or'ed in the correct target representation. Thus, the **or** operation is written as

```
or → (lambda (a b) (binary-OR a b))

(define binary-OR
  (lambda (x y)
    (cond
     ((null? x) nil)
     ((atom? x) (list x '+ y))
     (t (cons (list (car x) '+ (car y))
              (binary-OR (cdr x) (cdr y)))))))
```

where **binary-OR** is a function which returns the correct binary representation of **or** over arguments **a** and **b**.

Once selectors and stream equations are mapped to a binary representation, they are translated to boolean equations. Section 10, discusses the translation of projected streqns into boolean equations.

To incorporate a representation in DDD:

BitMap

Define a representation mapping by defining a "back-quotted" expression in the following form:

```
(define P
  `([ide_1 ,(lambda (...) exp_1)]
    ...
    [ide_R ,(lambda (...) exp_R]))
```

which is interpreted as a mapping from an abstract basis to a target basis

$$\mathbf{P:\{ide_1 \rightarrow exp_1, ..., ide_R \rightarrow exp_R\}}$$

$ide_1...ide_R$	Denotes <u>every</u> identifier in the abstract description.
$exp_1...exp_R$	Denotes the corresponding values in the target basis.

```
DDD> (define bitmap
        `((x ,(lambda () '[x.2 x.1 x.0])))
          (y ,(lambda () '[y.0])))
          (f ,(lambda () '[0 0 0])))
          (g ,(lambda () '[0 0 1])))
          (u ,(lambda () '[1])))
          (h ,(lambda () '[h.2 h.1 h.0])))
          (w ,(lambda () '[w])))))
```

(ProjectStrEqns *RepMap StrEqns N*) → StrEqns
Takes a representation map, a set of stream equations, and the number of bits, and returns a set of stream equations. The function has the effect of projecting a set of stream equations defined over an abstract basis to a set of stream equations defined over N-binary values. The integer, *N*, specifies how many bits to return.

```
DDD> (define streqns
      '((X = (select p f g h))
        (Y = (select p u w u))))
streqns

DDD> (define bitmap
        `((x ,(lambda () '[x.2 x.1 x.0]))
          (y ,(lambda () '[y.0]))
          (f ,(lambda () '[0 0 0]))
          (g ,(lambda () '[0 0 1]))
          (u ,(lambda () '[1]))
          (h ,(lambda () '[h.2 h.1 h.0]))
          (w ,(lambda () '[w]))))
bitmap

DDD> (ProjectStrEqns bitmap streqns 3)
((X.2 = (select p 0 1 h.2))
 (X.1 = (select p 0 0 h.1))
 (X.0 = (select p 0 0 h.0))
 (Y.0 = (select p 1 w 1)))
```

(ProjectSEL *RepMap Select*) → [Select0 ... SelectN]
Takes a representation map and a selector, and returns N selectors defined over N-binary values.

```
DDD> (define cmd-gen
       '(define cmd
          (lambda (s p)
            (case s
              (s0 (if p v1 v2))
                (s1 v0)))))
cmd-gen

DDD> (define bitmap
`((v0 ,(lambda () '[0 0]))
  (v1 ,(lambda () '[0 1]))
  (v2 ,(lambda () '[1 0]))))
bitmap

DDD> (ProjectSEL bitmap cmd-gen)
((define cmd.1
   (lambda (s p)
     (case s
       (s0 (if p v1 v2))
       (s1 v0))))
 (define cmd.0
    (lambda (s p)
      (case s
        (s0 (if p v1 v2))
        (s1 v0)))))
```

8 Input/Output

The DDD system is implemented as a collection of Scheme programs that implement a design algebra. Since the system is written in Scheme the entire set of Scheme i/o functions are available within the system. However, some simple i/o extensions have been defined to make certain routine operations simpler.

(open-output-file-prompt *OutFile*) → port
Returns a new output port. The function creates a new output port for the file specified by *OutFile*. If the file exists, the function prompts the user to either return an error if the file is not to be overwritten, or deletes the preexisting file and creates a new output port for the file. The OutFile must be a string.

(ReadFile *InFile*) → Exp
Returns first s-expression in the input file. *InFile* must be a string.

(WriteFile *Exp OutFile*) → ()||OutFile
Writes an expression to the specified output file in pretty-print format. If the file exists, the user is prompted as to whether the file should be deleted. *Exp* is any valid expression. *Out-File* must be a string.

(AppendFile *Exp OutFile*) → ()||OutFile
Appends the values of *Exp* to a file specified by *OutFile*. If the file exists, *Exp* is appended to the file. If the file does not exist, a new file is created. *Exp* can be any valid expression. *OutFile* must be a string.

9 Daisy Interface

Animation of design is a fundamental part of the design process. As an initial specification is developed at the algorithmic level, the execution of the specification helps the designer debug and understand the behavior of his design.

As the derivation process progresses through the various levels of abstraction, animation at each of these levels defines good design. The animation model in this research is developed around the notion of the executability of the specification. That is, each intermediate form of the circuit, is directly executable without any need for the development of extractors. The source for animation is the same for transformation.

Although the algorithmic specification is executable directly in SCHEME, the sequential system descriptions are executed in Daisy. Daisy's normal order evaluation, and stream constructs provide a good modeling environment for these designs. These features must be added to the SCHEME environment. This is under development and would provide a uniform environment.

A set of transformations have been implemented to transduce stream equations to Daisy. The transduction is only partial and requires manual editing of the Daisy code. There is no automated support to transduce the basis to Daisy, thus this must be done manually. This maybe a formidable task, unless a library of basis functions are established. Care must be taken in any manual intervention since errors are most likely to be introduced here. However animation is orthogonal to deriving correct hardware from a specification.

(StrEqns->Daisy *FuncName RegSet StrEqns OutFile*) → ()||OutFile

Takes a function name, a list of registers, a set of stream equations, and a file name, and generates the corresponding set of Daisy stream equations. The result is written out to the file name specified. The register set defines which of the signals is a register. The function name can be any valid Daisy symbol and defines the name of the set of Daisy stream equations.

```
DDD> S
((state <= (select state p q r f i g f i g))
 (x <= (select state p q r x x zz (dcr x) x
               (add (top s) x)))
  (s <= (select state p q r s s s (push s x) s (pop s)))
  (rdy <= (select state p q r nil rdy rdy rdy tt rdy))
  (go <= (select state p q r go go go go go go))))

DDD> (StrEqns->Daisy 'SUM '(state x s rdy) S "S.daisy")
()
```

S.daisy

```
SUM = ^\[initargs].
  REC:[[STATUS STATE X S RDY GO ]
 <    |STATUS
       XPS:<STATE P Q R >
      | STATE
        < @STATE ! SELECT:<STATUS F I G F I G >>
      | X
        < @X ! SELECT:<STATUS X X ZZ (DCR X) X (ADD (TOP S) X) >>
      | S
        < @S ! SELECT:<STATUS S S S (PUSH S X) S (POP S) >>
      | RDY
        < @RDY ! SELECT:<STATUS NIL RDY RDY RDY TT RDY >>
      | GO
        SELECT:<STATUS GO GO GO GO GO GO >
 >
 < STATUS STATE X S RDY GO > ]

TRACE = ^\[STATUS STATE X S RDY GO ].
<STATUS STATE X S RDY GO >
```

`initargs`, must be expanded to include the appropriate set of initial arguments. Function applications must be edited to conform to Daisy syntax. For example the term `(DCR X)` should be rewritten to be `DCR:<X>`

(StrEqns->DaisyConstants *StrEqns OutFile*) → ()||OutFile
Takes a set of stream equations, and a file name, and generates a set of constant stream definitions for every symbol. The result is written to the output file.

```
DDD> (StrEqns->DaisyConstants S "Sconst.daisy")
()

DDD>
```

Sconst.d

```
F = <"f" *>
I = <"i" *>
G = <"g" *>
ZZ = <"zz" *>
DCR = <"dcr" *>
ADD = <"add" *>
TOP = <"top" *>
PUSH = <"push" *>
POP = <"pop" *>
TT = <"tt" *>
```

The function does a brute force "streamification" of all identifiers. Some identifiers will have other definitions and its constant definition should be deleted.

(SEL->Daisy *Sel OutFile*) → ()||OutFile
Takes the combinator *Select* and generates a Daisy syntax version. The result is written to the output file.

```
DDD> (define SEL
       '(define select
           (lambda (s p0 p1 p2 v0 v1 v2 v3 v4 v5)
              (case s
                 [i (if p0 v0 v1)]
                 [f (if p1 v2 v3)]
                 [g (if p2 v4 v5)])))))

DDD> (SEL->Daisy SEL "sel.d")
()

DDD>
```

sel.d

```
select = ^\[[s p0 p1 p2 ] v0 v1 v2 v3 v4 v5 ] .
 if:<
     same?:<s "i"> if:<p0 v0 v1>
     same?:<s "f"> if:<p1 v2 v3>
     same?:<s "g"> if:<p2 v4 v5>>

SELECT = <select *>
```

10 Integrating with Logic Synthesis Tools

Logic synthesis refers to the low-level mapping of logic equations to a given target technology. The DDD system integrates with various logic synthesis tools in order to provide boolean minimization, and realization paths to MSI level Programmable Logic Devices (PLDs), and VLSI technology (PLA, Gate-Array, Standard-Cell, etc...).

10.1 Boolean Equation Generation

DDD generates a set of boolean equations from either selectors, or stream equations.

10.1.1 Generating Boolean Equations from Selectors

Selectors are compiled directly into a sum-of-products boolean equation. The selectors must have already been projected to bit-values. Consider the following command generator

```
(define cmd_bit0
  (lambda (s p)
    (case s
      (s0 (if p 0 1))
      (s1 0))))
```

In this example, **cmd_bit0**, returns a **0**, when **s0** and **p** is true, or when **s1** is true. The function returns a **1**, when **s0** is true, and **p** is false. The boolean equation for this selector is written as

```
cmd_bit0 = s0 & ~p;
```

Boolean equations for case-statements are generated to take advantage of the encodeing of the predicate, rather than expanding the expression to a series of if-statements. For example, looking at a memory instruction generator

```
(define mem-inst
  (lambda (s i p q)
    (case s
      (s0 (if p 0 1))
      (s1 (case i
            (i0 (if p 1 (if q 1 0)))
            (i1 (if q 0 1))
            (i2 (if p 0 1))))
      (s2 (if q 1 0)))))
```

The boolean equation generated is

```
mem-inst = q & s2
         | !p & i2 & s1
         | !q & i1 & s1
         | q & !p & i0 & s1
         | p & i0 & s1
         | !p & s0 ;
```

10.1.2 Generating Boolean Equations from Stream Equations

DDD generates either *D-type* or *Toggle-type* boolean equations to represent stream equations as described by Winkel in [17,18]. The D-type generates a single minterm for each input load, feedback load, 1 load, don't care load, and omits 0 loads, and is implemented with a D-type register. The Toggle-type generates two minterms for each input load, a single minterm for a 1, 0, or dontcare load, and omits a minterm for a feedback load, and is implemented with a Toggle-type register.

The correspondence between a stream equation and its boolean equation representation is tied closely with the register-transfer table (RTT) abstraction discussed in Section 6. Each of the possible values in a stream equation are indexed with a command code which is issued by the command generator derived from the selector. This command is incorporated into the boolean equations so that a minterm will represent the "on set" corresponding to the activation of the appropriate register transfer.

Consider the following stream equation and its corresponding RTT

(X = (Select status X Y 0 1 ?))

CMD	X
0	X
1	Y
2	0
3	1
4	?

A sequential assignment of command codes, 0 through 4 are assigned to each of the possible values in the stream equation. The possible values in a stream equation are categorized into four distinct types: a *feedback load* (or hold), a *new data load*, a *0 load*, a *1 load*, and a *don't care load.*

A D-type equation generates a minterm for a feedback load and a new data load by creating a term that is the product of the associated command and signal to be loaded. A 0 load is suppressed since it does not express a value in the on set. A 1 load is simply implemented by the command code, and a don't care load is implemented as the product of the associated command code and don't care symbol "?". Given the assignment of command codes, the corresponding D-type boolean function is derived:

```
X = cmd0 & X
  + cmd1 & Y
  + cmd3
  + cmd4 & ?;
```

A Toggle-type equation suppresses the minterm for a hold since that is the default of the logic. Two minterms are generated for a new data load by creating a term which implements the product of the associated command code with the exclusive-OR of the signal with the new data. A 0 load is implemented by the product of the command code and the signal, and a 1 load is implemented by the product of the command code and the complement of the signal. A don't care is implemented as the product of the associated command code and the don't care symbol. The corresponding Toggle-type boolean function may also be derived:

```
X = cmd1 & ~X & Y
  + cmd1 & X & ~Y
  + cmd2 & X
  + cmd3 & ~X
  + cmd4 & ?;
```

Both methods generate different equations. Depending on the type of registers available in the target technology, and the characteristics of the stream equations; the number of feedback loads, 1 loads, 0 loads, input loads, and dontcare loads, one type may lead to simpler logic than the other.

10.2 ESPRESSO Interface

ESPRESSO [12] is a heuristic based truth table minimizer which is part of the Berkeley VLSI Tools set. It takes as input a two-level representation of a multi-valued Boolean function (truth table), and produces a minimal equivalent representation. ESPRESSO automatically verifies that the minimized function is equivalent to the original function. The input is described as a character matrix with keywords imbedded in the input to specify the number of input variables, the number of output functions, and the number of product terms.

DDD derives truth tables in a format that is suitable for input to ESPRESSO, as described in the Berkeley 1986 VLSI Tools reference manual [12]. Each position in the input plane corresponds to an input variable where a "0" implies the corresponding variable appears complemented in the product term, a "1" implies the input variable appears uncomplemented in the product term, and a "-" implies the input variable does not appear in the product term. For each output, a "1" implies the minterm has a function value of 1, and a "-" implies that the minterm has a function value of dontcare or unspecified.

Once stream or boolean equations are converted to the truth table format compatible with ESPRESSO, they are input to the ESPRESSO system for simplification. ESPRESSO is applied to each of the truth tables individually. The minimized truth tables (output of ESPRESSO) may then be transduced to boolean equations. These boolean equations are in sum of product form and may be input to various tools.

The general ESPRESSO format generated by DDD has the form

```
% X                                ;name of the stream or boolean equation
% i_1 ... i_I                      ;inputs to the equation
% N                                ;number of command bits
.i I                               ;number of inputs
.o 1                               ;number of outputs (always 1)
...
cmd_{N-1} ... cmd_0 i_1 ... i_I 1  ;truth table
...
.end                               ;end marker
```

For example, converting either the stream equation, or the corresponding boolean equation

```
(X = (Select status X Y 0 1 ?))        X = cmd0 & X + cmd1 & Y
                                          + cmd3 + cmd4 & ?;
```

into an ESPRESSO format truth table results in

```
% x
% x y
% 3
.i 5
.o 1
0 0 0 1 - 1
0 0 1 - 1 1
0 1 0 - - -
0 1 1 - - -
.end
```

10.3 EQN Interface

EQN is a format for specifying boolean equations. The format is compatible with various logic synthesis tools, such as EQNTOTT and MISII. DDD generates EQN-format equations from either stream equations, or boolean equations.

The EQN-format has the following form:

```
NAME = filename ;
INORDER = i1 i2 i3 ... ;
OUTORDER = o1 o2 o3 ... ;
  o1 = ... ;
  o2 = ... ;
  ...
```

The NAME specifies the file name.

The INORDER and OUTORDER specifies the input and outputs, respectively, followed by a set of boolean equations.

For example, a set of boolean equations in DDD:

```
((sum ((!cin & !a & b) (!cin & a & !b)
       (cin & !a & !b) (cin & a & b)))
 (cout ((!cin & a & b) (cin & !a & b)
        (cin & a & !b) (cin & a & b)))))
```

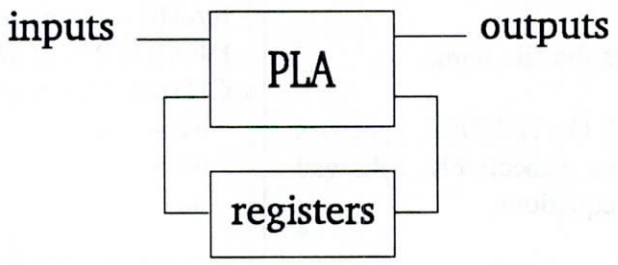
are translated into EQN-format:

adder.eqn

```
NAME = adder.eqn ;
INORDER = cin a b ;
OUTORDER = sum cout ;

sum = !cin & !a & b
    | !cin & a & !b
    | cin & !a & !b
    | cin & a & b;

cout = !cin & a & b
     | cin & !a & b
     | cin & a & !b
     | cin & a & b;
```

10.4 PLA Interface

A programmable logic array (PLA) provides a regular structure for implementing combinatorial sequential logic functions. Typically, PLA's compute some logic function of its inputs and yields outputs. Some of these outputs may be fed back to the inputs, thus forming a finite state machine as in figure 1.



The PLA uses an AND-OR structure. Latches are incorporated into the planes and correspond to half a latch. The basis for the PLA is a sum of products form of representation of boolean expressions. The product terms are formed in the AND plane, and the outputs are formed by ORing the appropriate product terms. Thus the height of the PLA is determined by the number of distinct product terms, and the width by the number of inputs and outputs.

The DDD system integrates with MPLA, a technology independent PLA generator, by generating a set of boolean equations, in EQN-format, compatible with EQNTOTT from a set of stream or boolean equations. DDD creates an EQN-format file. EQNTOTT generates a truth table suitable for PLA programming from a set of boolean equations. The truth table is then minimized with ESPRESSO and input to MPLA. Equations for both *static* and *dynamic* PLA's may be generated.

Static PLA's are the most straight forward design. The design uses a pseudo-nmos gate. Advantages are simplicity, small size, and the ability to handle feedback. The disadvantages are static power dissipation, and possible speed problems.

Dynamic PLA's generate less power and ground noise, but take a larger area and cannot implement function which requires feedback. A two-phase non-overlapping clocking strategy is required to implement the design. In this strategy, the AND stage is precharged during phase-1, and evaluated during phase-1. The OR state is precharged during phase-2 and evaluated during ~phase-2. This leads to an OR-NAND structure, in which case the equations implement the complement for all the outputs. Magic layouts for dynamic and static PLA's are then generated.

10.5 Altera Interface

The PLD (Programmable Logic Device) provides an inexpensive, rapid prototyping environment for hardware design. A PLD is an off the shelf component that can be programmed by the user to implement some logic function. Combinational circuitry, as well as latched function are typical in such devices. The logic, i/o pins, register count are specific to the device being programmed. PLD's have the inherent advantage over other target technologies (full custom, standard cell, gate array). They have a short development lead time. Low design costs and interchangeable inventory.

Input to PLD programmers is typically a fuse map. Higher level tools (Palasm, Altera) take a set of inputs, outputs, and boolean equations, and compute a fuse map for a specific device.

The DDD system generates a set of inputs, outputs, connectivity network, and a set of boolean equations from a set of boolean equations. Specifically, DDD generates an ADF (Altera Design File) file, that is input to the ALTERA EPLD system [1], which programs devices.

Altera provides CMOS EPLD (Erasable PLD) technology. Compared to bipolar fuse technology, CMOS provides lower power dissipation and a cooler operating temperature, which enables greater logic densities. The device uses an EPROM programming mechanism enabling reprogramming in the event of design changes.

The ADF file has the following form:

```
HEADER: filename
PART: AUTO
INPUTS:  i1 i2 i3 ...
OUTPUTS: o1 o2 o3 ...
NETWORK: ...
EQUATIONS:
  o1 = ...;
  o2 = ...;
     .
     .
     .
END$
```

The HEADER information is simply a comment line which contains the file name.

The INPUTS and OUTPUTS are derived from the right hand side, and left hand side of the equations, respectively.

The PART: AUTO parameter specifies that an appropriate ALTERA part will be chosen. However a manual specification of the part may be necessary if an appropriate part is not in the ALTERA compiler's library.

The NETWORK defines how signals are connected within the PLD. Currently supported Altera configurations are :

CONF	combinational output/no feedback
COCF	combinational output/combinational feedback
RONF	register output/no feedback
TONF	toggle output/no feedback
TOTF	toggle output/toggle feedback

(SEL->BoolEqns *Sel*) → BoolEqns
Takes a selector defined over bit-values and returns the boolean equation.

```
DDD> (define mem-inst
        '(define mem-inst
           (lambda (s i p q)
             (case s
               (s0 (if p 0 1))
               (s1 (case i
                     (i0 (if p 1 (if q 1 0)))
                     (i1 (if q 0 1))
                     (i2 (if p 0 1)))))))))
mem-inst

DDD> (SEL->BoolEqns mem-inst)
 (mem-inst ((!p & i2 & s1)
            (!q & i1 & s1)
            (q & !p & i0 & s1)
            (p & i0 & s1)
            (!p & s0)))
```

(StrEqns->BoolEqns *StrEqns RegType*) → BoolEqns
Takes a set of stream equations defined over bit-values, a register type (either 'd or 'toggle), and returns a set of boolean equations.

```
DDD> (define streqns
        '((x = (select status x y ? ? 0))
          (y = (select status y y x ? 1))))

DDD> (StrEqns->BoolEqns streqns 'd)
((x ((!p2 & !p1 & !p0 & x)
     (!p2 & !p1 & p0 & y)
     (!p2 & p1 & !p0)
     (!p2 & p1 & p0)))
  (y ((!p2 & !p1 & !p0 & y)
     (!p2 & !p1 & p0 & y)
     (!p2 & p1 & !p0 & x)
     (!p2 & p1 & p0)
     (p2 & !p1 & !p0))))
```

Auxiliary Functions:
(StrEqn->BoolEqn *StrEqn RegType*) → BoolEqn
Takes a stream equation, register type, and returns a boolean equation.

(StrEqns->ESPRESSO *StrEqns RegType Group*) → ()||TT.Group
Takes a set of stream equations, a register type (either **'d** or **'toggle**), and a group, and generates a truth table for each equation. The truth tables for each equation are written out to files with the equation name concatenated with the group name.

```
DDD> (define streqns
        '((x = (select status x y ? ? 0))
          (y = (Select status y y x ? 1))))
streqns

DDD> (StrEqns->ESPRESSO streqns 'd 'esp)
x.esp
y.esp
()
```

X.esp

```
% x
% x y
% 3
.i 5
.o 1
0 0 0 1 - 1
0 0 1 - 1 1
0 1 0 - - -
0 1 1 - - -
.end
```

Y.esp

```
% y
% y x
% 3
.i 5
.o 1
0 0 0 1 - 1
0 0 1 1 - 1
0 1 0 - 1 1
0 1 1 - - -
1 0 0 - - 1
.end
```

The *Group* may be any valid identifier. Appropriate naming conventions will allow the identification of equations belonging to the same set. For example a set of stream equations defining "bitslice 18" on a design may be appropriately grouped by the group name `bit18`. This will create a set of files corresponding to each equation in the bitslice, with each having the same file extension, `bit18`.

Auxiliary Functions:
(StrEqn->ESPRESSO_dtype *StrEqn*) → TT
(StrEqn->ESPRESSO_toggletype *StrEqn*) → TT
Takes a single stream equation and returns a truth table in s-exp form.

(BoolEqns->ESPRESSO *BoolEqn Group*) → ()||TT.Group
Takes a set of boolean equations in sum of product form, and a group name, and generates a set of truth tables corresponding to each equation. The truth tables for each equation are written out to files with the equation name concatenated with the group name. The s-exp notation for the sum of products form is a list of lists.

For example the equation **x = ¬a & b + a & ¬b** is represented by the list **(x ((!a & b) (a & !b)))**, where the first element denotes the name of the equation, and the second is a list denoting an OR of each of the minterms.

```
DDD> (define adder
        '((sum ((!cin & !a & b) (!cin & a & !b)
                (cin & !a & !b) (cin & a & b)))
          (cout ((!cin & a & b) (cin & !a & b)
                 (cin & a & !b) (cin & a & b)))))
adder

DDD> (BoolEqns->ESPRESSO adder 'adder)
sum.adder
cout.adder
()
```

sum.adder

```
% sum
% cin a b
% 0
.i 3
.o 1
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
.end
```

cout.adder

```
% cout
% cin a b
% 0
.i 3
.0 1
0 1 1 1
1 0 1 1
1 1 0 1
1 1 1 1
.end
```

Auxiliary Functions:
(BoolEqn->ESPRESSO *BoolEqn*) → TT
Takes a single boolean equation and returns a truth table in s-exp form.

(ESPRESSO *Input InputType Group* [*RegType*]) → BoolEqns
Takes an input (either a set of stream equations, or a set of boolean equations), an input type (either **'streqns** or **'booleqns**), a group name, and an optional argument specifing the register type (either **'d** or **'toggle**), and returns a set of minmized boolean equations. The optional argument, *RegType*, is only necessary when the input type is a set of stream equations.

```
DDD> (define adder
        '((sum ((!cin & !a & b) (!cin & a & !b)
                    (cin & !a & !b) (cin & a & b)))
           (cout ((!cin & a & b) (cin & !a & b)
                      (cin & a & !b) (cin & a & b)))))
adder

DDD> (ESPRESSO adder 'booleqns 'adder)
((cout ((cin & a) (cin & b) (a & b)))
 (sum ((cin & !a & !b)
       (!cin & a & !b)
       (!cin & !a & b)
       (cin & a & b))))

DDD> (define streqns
       '((x = (select p x y ? ? 0))
         (y = (select p y y x ? 1))))

DDD> (ESPRESSO streqns 'streqns 'bit32 'd))
((x ((!cmd.2 & cmd.0 & y) (!cmd.2 & !cmd.0 & x)))
 (y ((!cmd.2 & !cmd.1 & y)
     (!cmd.2 & cmd.1 & x)
     (cmd.2 & !cmd.1 & !cmd.0))))
```

The function side-effects the file system by generating an ESPRESSO-format truth table for each equation. These files have the individual stream equation names concatenated with the group name. In addition, the function creates two more files. The function makes a system call, and applies ESPRESSO to each of the truth tables and creates a single file with the output of ESPRESSO. This file is named with the group name concatenated with the file extention `.min`. The other file created is an s-exp format of the truth table output by ESPRESSO. This file is named with the group name concatenated with the file extention `.sch`.

In the first example, `(ESPRESSO adder 'booleqns 'adder)`, the resulting files will be:

sum.adder

```
% sum
% cin a b
% 0
.i 3
.o 1
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
.end
```

cout.adder

```
% cout
% cin a b
% 0
.i 3
.0 1
0 1 1 1
1 0 1 1
1 1 0 1
1 1 1 1
.end
```

adder.min

```
#% cout
#% cin a b
#% 0
.i 3
.o 1
.p 3
11- 1
1-1 1
-11 1
.e
#% sum
#% cin a b
#% 0
.i 3
.o 1
.p 4
100 1
010 1
001 1
111 1
.e
```

adder.sch

```
( (% cout )
(% ( cin a b ))
(% 0 )
(.i 3)
(.o 1)
(.p 3)
(1 1 -   1)
(1 - 1   1)
(- 1 1   1)
(.e))
( % sum )
(% ( cin a b ))
(% 0 )
(.i 3)
(.o 1)
(.p 4)
(1 0 0   1)
(0 1 0   1)
(0 0 1   1)
(1 1 1   1)
(.e))
```

(ESPRESSO->BoolEqns *TT*) → BoolEqns
Takes a set of truth tables in s-exp form, and returns the corresponding set of boolean equations.

```
DDD> (ESPRESSO->Sexp "adder.min" "adder.sch")
()

DDD> (ESPRESSO->BoolEqns (ReadFile "adder.sch"))
((cout ((cin & a) (cin & b) (a & b)))
 (sum ((cin & !a & !b)
       (!cin & a & !b)
       (!cin & !a & b)
       (cin & a & b))))
```

Auxiliary Functions:
(ESPRESSO->Sexp *InFile OutFile*) → ()
Converts ESPRESSO output to an s-exp form. The function writes the converted truth tables to the file specified by *OutFile*.

(StrEqns->EQN *StrEqns EQNtype OutFile*) → ()
Takes a set of stream equations, an EQN type (either **'standard** or **'inverter**), and an output file name, and generates a set of inputs, outputs, and boolean equations. The result is written to the file specified by OutFile and is compatible with EQNTOTT.

An ordering is placed on the INORDER and OUTORDER list to make routing of feedback minimal.

```
DDD> (define streqns
       '((x = (select status x y ? ? 0))
         (y = (select status y y x ? 1))))

DDD> (StrEqns->EQN streqns 'static "streqns.eqn")
()
```

streqns.eqn

```
NAME = streqns.eqn ;
INORDER = cmd2 cmd1 cmd0 x y ;
OUTORDER = y x ;

x = !cmd2 & !cmd1 & !cmd0 & x
  | !cmd2 & !cmd1 & cmd0 & y
  | !cmd2 & cmd1 & !cmd0 & ?
  | !cmd2 & cmd1 & cmd0 & ?;

y = !cmd2 & !cmd1 & !cmd0 & y
  | !cmd2 & !cmd1 & cmd0 & y
  | !cmd2 & cmd1 & !cmd0 & x
  | !cmd2 & cmd1 & cmd0 & ?
  | cmd2 & !cmd1 & !cmd0;
```

Auxiliary Functions:
(StrEqns->EQN_standard *StrEqns*) → EQN
(StrEqns->EQN_inverter *StrEqns* → EQN
Takes a set of stream equations, and returns either standard or inverted boolean equations in s-exp form.

(BoolEqns->EQN *BoolEqns EQNtype OutFile*) → ()
Takes a set of boolean equations, an EQN type (either **'standard** or **'inverter**), and an output file name, and generates a set of inputs, outputs, and boolean equations. The result is written to the file specified by OutFile and is compatible with EQNTOTT.

```
DDD> (define adder
       '((sum ((!cin & !a & b) (!cin & a & !b)
               (cin & !a & !b) (cin & a & b)))
         (cout ((!cin & a & b) (cin & !a & b)
               (cin & a & !b) (cin & a & b)))))
adder

DDD> (BoolEqns->EQN adder 'static "adder.eqn")
()
```

adder.mpla

```
NAME = adder.eqn ;
INORDER = cin a b ;
OUTORDER = sum cout ;

sum = !cin & !a & b
    | !cin & a & !b
    | cin & !a & !b
    | cin & a & b;

cout = !cin & a & b
     | cin & !a & b
     | cin & a & !b
     | cin & a & b;
```

Auxiliary Functions:
(BoolEqns->EQN_standard *BoolEqns*) → EQN
(BoolEqns->EQN_inverter *BoolEqns*) → EQN
Takes a set of boolean equations, and returns static or dynamic boolean equations in s-exp form.

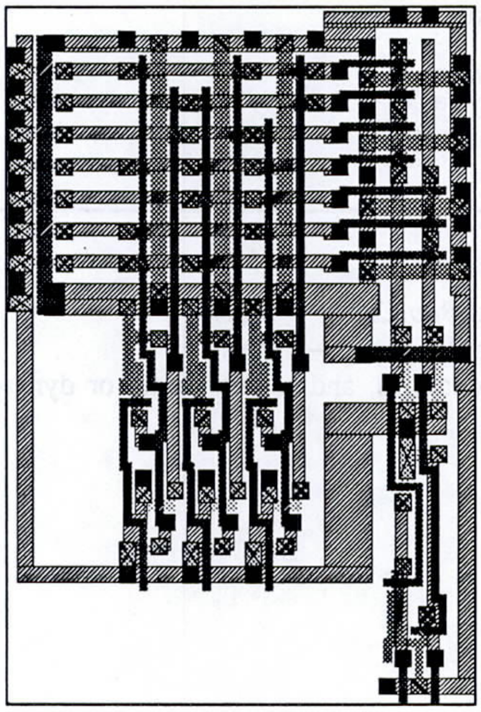
(MPLA *FileName PLAtype*) → ()
Takes a file containing boolean equations compatible with EQNTOTT, and a PLA type - either 'dynamic or 'static, and executes the unix shell scripts: "makestatic" or "makedynamic" on that file. "makestatic" and "makedynamic" generate Magic: static and dynamic PLA layouts, respectively. The scripts apply **eqntott, espresso**, and **mpla** sequentially to the original boolean equations (note: some sed filters are necessary to handle naming incompatibilities). The shell scripts may be executed directly from the Unix shell, or within DDD with the MPLA function.

makestatic: file
```
#! /bin/csh -f
sed 's/-/_/g' $1 | sed 's/+/|/g' | eqntott -f -.iolpte | sed 's/x/-/g' | espresso
                | mpla -s SCS3cis -G 10 -I -O -a -o $1
```

makedynamic: file
```
#! /bin/csh -f
sed 's/-/_/g' $1 | sed 's/+/|/g' | eqntott -f -.iolpte | sed 's/x/-/g' | espresso
                | mpla -s SCD3cis -G 10 -I -O -a -o $1
```

```
DDD> (MPLA "adder.mpla" 'static)
()
```

adder.mpla.mag



(BoolEqns->Altera *Regs BoolEqns OutFile*) → ()||OutFile
Takes a set of registers, a set of boolean equations, and a file name, and writes the result to OutFile. Signal names in the register list are, by default, implemented with D-type flipflops. Prefixing signal names with @ in the register list will implement them with toggle-type flipflops.

```
DDD> (define adder
       '((sum ((!cin & !a & b) (!cin & a & !b)
               (cin & !a & !b) (cin & a & b)))
         (cout ((!cin & a & b) (cin & !a & b)
                (cin & a & !b) (cin & a & b)))))
adder

DDD> (BoolEqns->Altera '(sum cout) adder "adder.adf")
()
```

adder.adf

```
HEADER: adder.adf
PART: AUTO

INPUTS: CLOCK, A, B, CIN
OUTPUTS: COUT, SUM

NETWORK:
CLOCK = INP (CLOCK)
A = INP (A)
B = INP (B)
CIN = INP (CIN)
SUM = RONF (SUMd,CLOCK,GND,GND,VCC)
COUT = RONF (COUTd,CLOCK,GND,GND,VCC)

EQUATIONS:
SUMd = !CIN & !A & B + !CIN & A & !B
     + CIN & !A & !B + CIN & A & B ;
COUTd = !CIN & A & B + CIN & !A & B
      + CIN & A & !B + CIN & A & B ;

END$
```

References

[1] Altera Corporation, *Altera Programmable Logic User System User Guide* (Version 4.0), Altera Corporation, Santa Clara, 1985.

[2] Boyer, C. David, Johnson, D. Steven, Using the Digital Derivation System: Case study of a VLSI garbage collector. In *Proceedings of the Ninth IFIP Symp. on Computer Hardware Description Languages* (CHDL89), J. Darringer and F. Ramming, Eds., Elsevier. Also published as Technical Report 274, Computer Science Department, Indiana University, April 1989.

[3] Johnson, Steven D., Wehrmeister, Robert, M., and Bose, Bhaskar, On the Interplay of Synthesis and Verification: Experiments with the FM8501 Processor Description, In *Applied Formal Methods For Correct VLSI Design* (Amsterdam, Netherlands, 1989), L. Claesen, Ed., IMEC, Elsevier, pp. 385-404. Also published as Technical Report 283, Computer Science Department, Indiana University, July 1989.

[4] Johnson, Steven D., Digital Design in a Functional Calculus. In *Formal Aspects of VLSI Design*, G. Milne and P.A. Subrahmanyam, Eds. North-Holland, Amsterdam, 1986, pp. 153-178. Also published as Technical Report 279, Computer Science Department, Indiana University, June 1989.

[5] Johnson, Steven D., and Bose, Bhaskar, DDD - A System for Mechanized Digital Design Derivation, Technical Report 323, Department of Computer Science, Indiana University, December 1990.

[6] Johnson, Steven D., Bose, Bhaskar and Boyer, C. David, A Tactical Framework for Digital Design. In *VLSI Specification, Verification and Synthesis*, G. Birtwistle and P.A. Subrahmanyam, Eds. Kluwer Academic Publishers, Boston, 1988 pp. 349-383. Also published as Technical Report 221, Department of Computer Science, Indiana Univeristy, May 1987.

[7] Johnson, Steven D., and Bose, Bhaskar. A System for Digital Design Derivation. Technical Report 289, Department of Computer Science, Indiana University, August 1989.

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

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

[10] Rath, Kamlesh, Ignacio Celis, and Robert M. Wehrmeister, RTBA: A Generic Bit-Sliced Bus Architecture for DataPath Synthesis, Technical Report 321, Department of Computer Science, Indiana University, December 1990.

[11] Rees, Jonathan and Clinger, William C., (eds.), The Revised^3 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices 21*, December 1986, 37-79. Also published as Technical Report 174, Department of Computer Science, Indiana University, June 1985.

[12] Scott, Walter S., Mayo, Robert N., Hamachi, Gordon and Ousterhout, John K., (eds.), 1986 VLSI Tools, Report No. UCB/CSD 86/272, Computer Science Division (EECS), University of California at Berkeley, (1985)

[13] Spickelmier, Rick L., Release Notes for Oct Tools Distribution 3.0, Electronics Research Laboratory, University of California, Berkeley, August 1989.

[14] Wand, M. Semantics-directed Machine Architecture. In *Proceedings of the Ninth ACM Symposium on Principles of Programming Languages* (1982).

[15] Wehrmeister, R. M. Derivation of an SECD Machine: Experience with a transformational approach to synthesis. Technical Report 290, Computer Science Department, Indiana University, September 1989.

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

[17] Winkel, David, What Next for PAL-Devices - The Second Generation Challenge, Technical Report 188, Department of Computer Science, Indiana University, February 1986.

[18] Winkel, David, The Use of PALS in CPU Design, Technical Report 204, Department of Computer Science, Indiana University, October 1986.

Appendix A: DDD Quick Reference

Control Abstraction and Architecture

Function	Arguments	Returns	Page
ItrSys→SingleLoop	[ItrSys]	SingleLoop	18
SingleLoop→Select	[SingleLoop]	Select	18
SingleLoop→StrEqns	[SingleLoop]	StrEqns	19
ItrSys→SeqSys	[ItrSys]	[Select StrEqns]	19

Algebra on Sequential Systems

Functions	Arguments	Returns	Page
Identify	[Exp newStrName StrEqns]	StrEqns	31
Generalize	[StrEqn]	StrEqn	31
MergeOperations	[newStrName OpSet StrEqns]	StrEqns	32
MergeStrEqns	[Str1 Str2 newStrName StrEqns]	StrEqns	32
AbstractOperations	[AbsCompName OpSet StrEqns]	StrEqns	33
AbstractStrEqn	[StrName StrEqns]	StrEqns	34
AddStrEqn	[StrEqn StrEqns]	StrEqns	35
ExtractStrEqn	[StrName StrEqns]	StrEqn	35
RemoveStrEqn	[StrName StrEqns]	StrEqns	35
RenameStrEqn	[oldStrName newStrName StrEqns]	StrEqns	36
OptimizeSEL	[Select]	Select	36
OptimizeSeqSys	[Select StrEqns]	[Select StrEqns]	37
PartialEval	[StrEqn Select]	Select	38
ExpandFuncDef	[FuncDef StrEqns]	StrEqns	38
GroupStrEqns	[StrEqns Order]	[[StrEqns]...]	39

Register Transfer Table

Functions	Arguments	Returns	Page
StrEqns->RTT	[StrEqns OutFile]	()‖OutFile	47
RTT→StrEqns	[RegSet RTT]	StrEqns	47

Projection

Functions	Arguments	Returns	Page
ProjectStrEqns	[RepMap StrEqns N]	StrEqns	53
ProjectSEL	[RepMap Select]	[Select0 ... SelectN]	54

Input/Output Extensions

Functions	Arguments	Returns	Page
open-output-file-prompt	[OutFile]	i/o port	55
ReadFile	[InFile]	1st s-exp in InFile	55
WriteFile	[Exp OutFile]	()‖OutFile	55
AppendFile	[Exp OutFile]	()‖OutFile	55

Daisy Interface

Functions	Arguments	Returns	Page
StrEqns→Daisy	[Name RegSet StrEqns OutFile]	()‖OutFile	58
StrEqns→DaisyConstants	[StrEqns OutFile]	()‖OutFile	59
SEL→Daisy	[Select OutFile]	()‖OutFile	60

Boolean Equations

Functions	Arguments	Returns	Page
StrEqns→BoolEqns	[StrEqns RegType]	BoolEqns	68
StrEqn→BoolEqn	[StrEqn]	BoolEqn	68
SEL→BoolEqns	[Select]	BoolEqns	68

Espresso Interface

Functions	Arguments	Returns	Page
StrEqns→Espresso	[StrEqns RegType Group]	()‖TT	69
StrEqn→Espresso_dtype	[StrEqn]	TT	69
StrEqn→Espresso_toggletype	[StrEqn]	TT	69
BoolEqns→Espresso	[BoolEqn Group]	TT	70
BoolEqn→Espresso	[BoolEqn]	TT	70
Espresso	[I Itype Group [RegType]]	BoolEqns	71
Espresso→Sexp	[InFile OutFile]	()‖OutputFile	73
Espresso→BoolEqns	[TT]	BoolEqns	73

EQN Interface

Functions	Arguments	Returns	Page
StrEqns→EQN	[StrEqns EQNtype OutFile]	()‖OutFile	74
StrEqns→EQN_standard	[StrEqns]	EQN	74
StrEqns→EQN_inverter	[StrEqns]	EQN	74
BoolEqns→EQN	[BoolEqns EQNtype OutFile]	()‖OutFile	75
BoolEqns→EQN_standard	[BoolEqns]	EQN	75
BoolEqns→EQN_inverter	[BoolEqns]	EQN	75

PLA Interface

Functions	Arguments	Returns	Page
MPLA	[InFile PLAtype]	()‖InFile.mpla	76
makestatic	[InFile]	InFile.mag	76
makedynamic	[InFile]	InFile.mag	76

Altera Interface

Functions	Arguments	Returns	Page
BoolEqns→ALTERA	[RegSet BoolEqns OutFile]	()‖OutFile	77

Appendix B: Type Classification

```
ItrSys := (define CIRCUIT
            (lambda ($i_1$ $i_2$ ...)
              (letrec
                 (($S_0$ (lambda ($r_1$ $r_2$ ... $r_N$) $exp_0$))
                  ($S_1$ (lambda ($r_1$ $r_2$ ... $r_N$) $exp_1$))
                  ...
                  ($S_Q$ (lambda ($r_1$ $r_2$ ... $r_N$) $exp_Q$)))
                    ($S_{init}$ $r_{1init}$ $r_{2init}$ ... $r_{Ninit}$))))

          where
          exp := (let ((id val) ...) exp)
               | (if pred $exp_1$ $exp_2$)
               | (case pred ($id_1$ $exp_1$) ($id_2$ $exp_2$) ...)
               | (S $val_1$ ... $val_N$)
          id := identifier
          val := expression
          pred := expression
```

```
SingleLoop := (define CIRCUIT
                (lambda (state $r_1$ $r_1$ ... $r_N$)
                  (case state
                    ($S_0$ $exp_0$)
                    ($S_1$ $exp_1$)
                    ...
                    ($S_Q$ $exp_Q$))))
```

```
Select := (define Select
            (lambda (s $p_0$ $p_1$ ... $p_R$ $v_0$ $v_1$ ... $v_N$)
              (case s
                ($S_0$ (if $p_0$ $v_I$ $v_J$))
                ($S_1$ (if $p_1$ (if $p_2$ $v_K$ ...)))
                ...
                ($S_Q$ ...))))
```

```
StrEqn := (StrEqn = (Select s $p_0$ $p_1$ ... $p_R$ $v_0$ $v_1$ ... $v_N$))
        | (StrEqn <= (Select s $p_0$ $p_1$ ... $p_R$ $v_0$ $v_1$ ... $v_N$))

StrEqns := (($StrEqn_1$ = (Select s $p_0$ $p_1$ ... $v_0$ $v_1$ ... $v_N$))
            ($StrEqn_2$ = (Select s $p_0$ $p_1$ ... $v_0$ $v_1$ ... $v_N$))
             ...
            ($StrEqn_M$ = (Select s $p_0$ $p_1$ ... $v_0$ $v_1$ ... $v_N$)))
```

Exp	:= expression
AbsCompName	:= identifier
Group	:= identifier
StrName	:= identifier
FileExt	:= string
InFile	:= string
OutFile	:= string
OpSet	:= list
RegSet	:= list
EQNtype	:= 'standard | inverter
Itype	:= 'streqns | 'booleqns
RegType	:= 'd | 'toggle
PLAtype	:= 'static | 'dynamic

FuncDef := (define FUNCTION
(lambda (args...) exp...))

Order := ((StrName$_I$ StrName$_J$ ...)
(StrName$_U$ StrName$_V$ ...)
...)

BoolEqn := (BoolEqn ((A & B) (C & D))) ;BoolEqn = A&B + C&D

BoolEqns := ((Eqn$_1$ ((A & B) (A & !B)))
(Eqn$_2$ ...)
...
(Eqn$_N$ ...))

RTT := ((StrEqn1 StrEqn2 ... StrEqnM)

```
        (v0 w0 x0 ... z0)
        (v1 w1 x1 ... z1)
        ...
        (vN wN xN ... zN))
```

CMD	StrEqn1	StrEqn2	...	StrEqnM
v0	w0	x0	...	z0
v1	w1	x1		z1
v2	w2	x2		z2
...	...	...		...
vN	wN	xN	...	zN

TT :=

```
((% name)
 (% output)
 (% no. of code
bits)
 (.i no. of inputs)
 (.o no. of outputs)
 (0 0 ...)
 (1 0 ...)
 (...)
 (1 1 0 1 )
 (.end))
```

```
% name
% output
% no. of code
bits
.i no. of inputs
.o no. of outputs
0 0 ...
1 0 ...
...
1 1 0 1
.end
```

EQN :=

```
NAME = filename ;
INORDER = i1 i2 i3 ... ;
OUTORDER = o1 o2 o3 ... ;
  o1 = ... ;
  o2 = ... ;
   ...
```

ADF :=

```
HEADER: filename
PART: AUTO
INPUTS:  i1 i2 i3 ...
OUTPUTS: o1 o2 o3 ...
NETWORK: ...
EQUATIONS:
   o1 = ...;
   o2 = ...;
    ...
END$
```

Example 1: A Single Pulser

```
;*********************************************************************
;*                            Specification                          *
;* The SinglePulser senses the depression of the button and asserts  *
;* an output signal for a single clock pulse.  Additional assertions *
;* of the output are not allowed until after the operator releases   *
;* the button.                                                       *
;*                                                                   *
;* The circuit is defined as a two-state machine: FIND, and WAIT,    *
;* that takes a synchronized input assertion from a push button,     *
;* PBsync.  The output of the system is O which carries the signal   *
;* PBpulse, a synchronized output assertion.  The initial invocation *
;* of the single pulser algorithm cycles in the FIND state until a   *
;* true signal is asserted on PBsync.  Control is then transferred   *
;* to the WAIT state with the value ON being asserted on the PBpulse *
;* signal.  The algorithm cycles in the WAIT state until  a false    *
;* is asserted on the PBsync signal.  Control is then transferred    *
;* back to FIND.                                                     *
;*********************************************************************

(define SinglePulser
  (lambda (PBsync)
    (letrec
      ((FIND
         (lambda (PBpulse)
            (let ((O (out PBpulse)))
              (if (PBsync) (WAIT ON) (FIND OFF)))))
       (WAIT
         (lambda (PBpulse)
            (let ((O (out PBpulse)))
              (if (PBsync) (WAIT OFF) (FIND OFF))))))
      (FIND OFF))))
```

```
;*********************************************************************
;*                         Derivation Path                           *
;* The derivation path illustrates the design from the iterative     *
;* specification, SP.ITR, to an ALTERA EPLD implementation, SP.ADF   *
;*********************************************************************
;
; SP.ITR
;   |
; SEQSYS
;   |
;   +-----------+
; SEL         STREQNS
;                |
;             STREQNS_1
;                |
;     +---------+-----------------------+
;     |                                 |
;   STATE                SEL           PBPULSE*
;     |                  | |            |
;     +---------------+ +---------------+
;          |                      |
;        STATE      REPMAP     PBPULSE*
;          |           ||         |
;          +----------++----------+
;               |           |
;            STATE.BIN  PBPULSE*.BIN
;               |           |
;               +-----------+
;                     |
;                  SP.BOOL
;                     |
;                SP.BOOL.MIN
;                     |
;                   SP.ADF
```

```
;*******************************************************************
;*                          Derivation Script                      *
;* The script details the derivation of the SinglePulser.  The     *
;* strategy employed in this derivation is to first, derive a      *
;* sequential system from the iterative specification.  Next, the  *
;* modelling interface is removed, and a signal is factored.  Next, *
;* the stream equations are partially evaluated with respect to the *
;* selector and projected to a binary representation.  Boolean     *
;* equations are then generated from the combinators and minimized *
;* using ESPRESSO.  Finally, the minimized equations are transduced *
;* into ADF-format from which an EPLD implementation is generated.  *
;*******************************************************************
;>>>Behavior to Structure<<<
(define SP.ITR  (ReadFile "SP.SS"))          ;The first step derives a
(define SEQSYS  (ItrSys->SeqSys SP.ITR))     ;sequential system, SEQSYS
(define SEL     (car SEQSYS))                ;from an iterative specifi-
(define STREQNS (cadr SEQSYS))               ;cation, SP.ITR.

;>>>Structure to Architecture<<<
(define STREQNS_1                            ;The modelling interface, O
  (RenameStrEqn 'PBPULSE-I 'PBPULSE*         ;is removed, the signal PB-
    (AbstractStrEqn 'PBPULSE                 ;PULSE is factored, and the
      (RemoveStrEqn 'O STREQNS))))           ;signal PBPULSE-I is renamed.

(define STATE                                ;The stream equations, STATE
  (PartialEval                               ;and PBPULSE, are extracted
    (ExtractStrEqn 'STATE STREQNS_1) SEL))   ;from the description and
(define PBPULSE*                             ;partially evaluated with
  (PartialEval                               ;respect to the selector, SEL.
    (ExtractStrEqn 'PBPULSE* STREQNS_1) SEL))

;>>>Architecture to Physical Organization<<<
(load "SP.REP")                              ;Load representations

(define STATE.BIN                            ;The combinators, STATE
  (OptimizeSEL                               ;and PBPULSE, are projected
    (car (ProjectSEL REP.MAP STATE))))       ;using the representations
(define PBPULSE*.BIN                         ;defined in REP.MAP.  The
  (OptimizeSEL                               ;combinators are then
    (car (ProjectSEL REP.MAP PBPULSE*))))    ;optimized.

;>>>Boolean Equations<<<
(define SP.BOOL                              ;Boolean equations are gen-
  (symbolic->bit                             ;erated and the symbolic val-
    (map SEL->BoolEqns                       ;ues, WAIT and FIND, are rep-
         (list STATE.BIN PBPULSE*.BIN)       ;laced with their binary proj-
   STATE.MAP)))                              ;ection defined in STATE.MAP.

;>>>Logic Synthesis<<<
(define SP.BOOL.MIN                          ;Boolean equations are mini-
  (ESPRESSO SP.BOOL 'booleqns 'TT))          ;mized with ESPRESSO.

(BoolEqns->ALTERA                            ;Transduce to ADF-format for
  '(STATE) SP.BOOL.MIN "SP.ADF")             ;input to ALTERA EPLD
                                             ;programmer.
```

```
;**********************************************************************
;*                      Binary Representations                        *
;* Representations for:   REP.MAP - WAIT, OFF = 0                      *
;*                                  FIND, ON = 1                       *
;*                      STATE.MAP - WAIT = 0 = !STATE                  *
;*                                  FIND = 1 = STATE                   *
;**********************************************************************
(define REP.MAP
  `((wait ,(lambda () '(0)))
    (find ,(lambda () '(1)))
    (off  ,(lambda () '(0)))
    (on   ,(lambda () '(1)))))
(define STATE.MAP
  `((wait ,(lambda () '(!state)))
    (find ,(lambda () '(state)))))


;**********************************************************************
;*                   Listing of Intermediate Forms                    *
;* SEQSYS                  - The initial sequential system             *
;* STREQNS_1               - The stream equations to be implemented    *
;* STATE, PBPULSE*         - The partially evaluated selectors         *
;* STATE.BIN, PBPULSE*.BIN - The projected combinators                 *
;* SP.BOOL                 - Boolean equations                         *
;* STATE.TT, PBPULSE*.TT   - ESPRESSO truth table format               *
;* SP.BOOL.MIN             - Minimized Boolean equations               *
;* SP.ADF                  - ADF-format of the boolean equations       *
;**********************************************************************
;----------------------------------------------------------------------
;                               SEQSYS
;----------------------------------------------------------------------
((define select
    (lambda (s p0 v0 v1 v2)
       (case s
         [find (if p0 v0 v1)]
         [wait (if p0 v2 v1)])))
 ((o = (select state (pbsync) (out pbpulse)
               (out pbpulse) (out pbpulse)))
  (pbpulse <= (select state (pbsync) on off off))
  (state <= (select state (pbsync) wait find wait))))


;----------------------------------------------------------------------
;                              STREQNS_1
;----------------------------------------------------------------------
((pbpulse* = (select state (pbsync) on off off))
 (state <= (select state (pbsync) wait find wait)))
```

```
;----------------------------------------------------------------------
;                            STATE & PBPULSE*
;----------------------------------------------------------------------
(define state
   (lambda (s p0)
      (case s
        [find (if p0 wait find)]
        [wait (if p0 wait find)])))
(define pbpulse*
   (lambda (s p0)
      (case s
        [find (if p0 on off)]
        [wait (if p0 off off)])))
```

```
;----------------------------------------------------------------------
;                        STATE.BIN & PBPULSE*.BIN
;----------------------------------------------------------------------
(define state
   (lambda (s p0)
     (case s
       [find (if p0 0 1)]
       [wait (if p0 0 1)])))
(define pbpulse*
   (lambda (s p0)
     (case s
       [find (if p0 1 0)]
       [wait 0])))
```

```
;----------------------------------------------------------------------
;                                SP.BOOL
;----------------------------------------------------------------------
((state ((!p0 & !state)
         (!p0 & state)))
 (pbpulse* ((p0 & state))))
```

```
;----------------------------------------------------------------------
;                         STATE.TT & PBPULSE*.TT
;----------------------------------------------------------------------
% state                              % pbpulse*
% p0 state                           % p0 state
% 0                                  % 0
.i 2                                 .i 2
.o 1                                 .o 1
0 0 1                                1 1 1
0 1 1                                .end
.end
```

```
;-----------------------------------------------------------------------
;                            SP.BOOL.MIN
;-----------------------------------------------------------------------
((state ((!p0)))
 (pbpulse* ((p0 & state))))


;-----------------------------------------------------------------------
;                              SP.ADF
;-----------------------------------------------------------------------
HEADER: SP.ADF
PART: AUTO

INPUTS: CLOCK, P0
OUTPUTS: PBPULSE*, STATE

NETWORK:
CLOCK = INP (CLOCK)
P0 = INP (P0)
STATE , STATE = RORF (STATEd,CLOCK,GND,GND,VCC)
PBPULSE* = CONF (PBPULSE*c,VCC)

EQUATIONS:
PBPULSE*c = P0 & STATE ;
STATEd = !P0 ;

END$
```

Example 2: A Black Jack Machine

```
;*******************************************************************
;*                          Specification                          *
;* The BlackJack machine simulates a dealer's actions in a black jack*
;* game, as specified in The Art of Digital Design, by Winkel and   *
;* Prosser.  An external agent presents cards, C, one at a time; a  *
;* full handshake is implemented for external synchronization - a hit*
;* signal, H, and a card ready signal, R.  The machine continues to *
;* play as long as its score, Score, is at most 21, or greater than *
;* 16.  The BlackJack machine may revalue an ace to either 1 or 11. *
;* A set of status signals, S - stand, and B - broke, H - hit,      *
;* communicate with the external agent.                             *
;*******************************************************************
(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))))
```

```
;**********************************************************************
;*                          Derivation Path                           *
;* The derivation path illustrates the design from the iterative      *
;* specification, BJ.ITR, to an implementation decomposed into three *
;* parts, PREDS.EQN, CONTROL.EQN, and SLICE_0..4.RTBA.               *
;**********************************************************************
;
;                          BJ.ITR
;                            |
;                          SEQSYS_1
;                            |
;                            +---------+
;                            |         |
;                          SEL_1   STREQNS_1
;                           | |    STREQNS_1.1
;                           | |    STREQNS_1.2
;STREQNS_1.4                | |    STREQNS_1.3
;   |                       | |    STREQNS_1.4
; STATE                     | |    STREQNS_1.5
;   |                       | |
;   +-----------------------+ + -------+
;             |                  |
;             |               SEQSYS_2
;             |                  |
;             |                  +-----------------+
;             |                  |                 |
;STATE.MAP  NEXTSTATE           SEL_2  CMD.MAP   STREQNS_2  STREQNS.MAP
;   |         |                  |        |       |             |
;   +---------+                  +--------+       +-------------+
;        |                       |                              |
;  NEXTSTATE.BIN  STATE.SYMMAP ENCODE.BIN  DATAPATH.ORG   STREQNS_2.BIN
;        |              | |  |                    |              |
;        +--------------+ +-----------+          +---------------+
;            |              |                            |
;       NEXTSTATE.BOOL  ENCODE.BOOL                STREQNS_2.ORG
;            |              |                            |
;       NEXTSTATE.MIN   ENCODE.MIN       +----------------------+
;            |              |            |                      |
;            |              |          SLICE_5.BOOL             |
;            |              |            |                      |
;            |              |          SLICE_5.MIN  SORB?B.BOOL |
;            |              |            |            |         |
;PREDS.BOOL  +--------------+------------+------------+         |
;    |                            |                             |
;PREDS.EQN                  CONTROL.EQN                 SLICE_0..4.RTBA
```

```
;*********************************************************************
;                          Derivation Script
;*********************************************************************
;Behavior to Structure
;~~~~~~~~~~~~~~~~~~~~~
(define BJ.ITR      (ReadFile "BJ.ss"))
(define SEQSYS_1    (ItrSys->SeqSys BJ.ITR))
(define SEL_1       (car SEQSYS_1))
(define STREQNS_1   (cadr SEQSYS_1))

;Structure to Architecture
;~~~~~~~~~~~~~~~~~~~~~~~~~
(define STREQNS_1.2                          ;Factor H, Go, O, Cd
  (RemoveStrEqns '(Go O Cd)
    (AbstractStrEqn 'H STREQNS_1)))

(define STREQNS_1.3                          ;Expand cancelace, addace
  (ExpandFunction
    '(define cancelace
       (lambda (score)
         (addto score -10ptace)))
    (ExpandFunction
      '(define addace
         (lambda (score)
           (addto score 10ptace))) STREQNS_1.2)))

(define STREQNS_1.4                          ;Factor addto
  (AbstractOperations
    'adder '(addto) STREQNS_1.3))

(define NEXTSTATE                           ;Derive nextstate combinator
  (PartialEval
    (ExtractStrEqn 'STATE STREQNS_1.4) SEL_1))

(define STREQNS_1.5                          ;Factor state, adder0-i,
  (RemoveStrEqns                             ;adder0-v0
    '(STATE ADDER0-I ADDER0-V0) STREQNS_1.4))

(define SEQSYS_2                             ;Optimize
  (OptimizeSEQSYS SEL_1 STREQNS_1.5))
(define SEL_2       (car SEQSYS_2))
(define STREQNS_2   (cadr SEQSYS_2))

;Architecture to Physical Organization
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
(load "bj.rep")                              ;Load representations

(define NEXTSTATE.BIN                        ;Map nextstate to binary
  (map OptimizeSEL                           ;representation.
    (ProjectSEL STATE.MAP
      (renameSEL 'STATE 'k NEXTSTATE))))

(define ENCODE.BIN                           ;Map select to binary
  (map OptimizeSEL                           ;representation.
    (ProjectSEL CMD.MAP
      (renameSEL 'SELECT 'CMD SEL_2))))
```

```
;-------------------------------------------------------------------
;                         Derivation Script (cont)
;-------------------------------------------------------------------
(define STREQNS_2.BIN                            ;Map stream equations to a
  (ProjectStrEqns                                ;binary representation.
     STREQNS.MAP STREQNS_2 (add1 WORDSIZE)))

(define STREQNS_2.ORG                            ;Organize stream equations
  (Group STREQNS_2.BIN DATAPATH.ORG))            ;into bit-slices.

;Logic Synthesis
;~~~~~~~~~~~~~~~
(define SLICE_5.BOOL                             ;Generate boolean equations
  (StrEqns->BoolEqns                            ;for SLICE_5, NEXTSTATE, and
    (slice 5 STREQNS_2.ORG) 'd))                 ;ENCODE.

(define NEXTSTATE.BOOL
  (symbolic->bit
    (map Sel->BoolEqns NEXTSTATE.BIN) STATE.SYMMAP))

(define ENCODE.BOOL
  (symbolic->bit
    (map Sel->BoolEqns ENCODE.BIN) STATE.SYMMAP))

(define SLICE_5.MIN                              ;Minimize equations
  (ESPRESSO SLICE_5.BOOL 'booleqns 'SLICE_5))
(define NEXTSTATE.MIN
  (ESPRESSO NEXTSTATE.BOOL 'booleqns 'NEXTSTATE))
(define ENCODE.MIN
  (ESPRESSO ENCODE.BOOL 'booleqns 'ENCODE))

;Generate RTBA cells for SLICE_0 to SLICE_4.
(StrEqns->RTBA SEL_2 (slice 0 STREQNS_2.ORG) "SLICE_0.RTBA")
(StrEqns->RTBA SEL_2 (slice 1 STREQNS_2.ORG) "SLICE_1.RTBA")
(StrEqns->RTBA SEL_2 (slice 2 STREQNS_2.ORG) "SLICE_2.RTBA")
(StrEqns->RTBA SEL_2 (slice 3 STREQNS_2.ORG) "SLICE_3.RTBA")
(StrEqns->RTBA SEL_2 (slice 4 STREQNS_2.ORG) "SLICE_4.RTBA")

(BoolEqns->EQN                                   ;Generate EQN-format to be
 (append ENCODE.MIN                              ;input to a PLA cell
         NEXTSTATE.MIN                           ;generator.
         SLICE_5.MIN
         SORB?.BOOL) 'static "CONTROL.EQN")

(BoolEqns->EQN                                   ;Generate EQN-format to be
 PREDS.BOOL 'static "PREDS.EQN")                 ;input to a STANDARD cell
                                                 ;generator.
```

```
;************************************************************************
;*                         Binary Representations                       *
;************************************************************************
(define wordsize 4)

(define streqns.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)))

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

```
;----------------------------------------------------------------------
;                    Binary Representations (cont.)
;----------------------------------------------------------------------
(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

(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.symmap
'((get ,(lambda () '(!k.1 & !k.0)))
  (add ,(lambda () '(!k.1 & k.0)))
  (use ,(lambda () '(k.1 & !k.0)))
  (tst ,(lambda () '(k.1 & k.0)))))
```

```
;********************************************************************
;*                    Listing of Intermediate Forms                 *
;********************************************************************
;--------------------------------------------------------------------
                              SEQSYS_1
;--------------------------------------------------------------------
((define select
    (lambda (s p0 p1 p2 p3 p4 p5 p6 v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10)
       (case s
          [get (if p0 (if p1 v0 (if p2 v1 v2)) v3)]
          [add (if p3 (if p4 v4 v5) v4)]
          [use v6]
          [tst (if p5 (if p6 (if p4 v7 v8) v9) v10)])))

 ((cd = (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) (cardvalue (swin)) (cardvalue (swin))
                (cardvalue (swin)) (cardvalue (swin)) (cardvalue (swin))
                (cardvalue (swin)) (cardvalue (swin)) (cardvalue (swin))
                (cardvalue (swin)) (cardvalue (swin))
                (cardvalue (swin))))
  (o = (select state r rd (or s b) (ace? c) a (sgt16? score)
               (sgt21? score) (output score h s b) (output score h s b)
               (output score h s b) (output score h s b)
               (output score h s b) (output score h s b)
               (output score h s b) (output score h s b)
               (output score h s b) (output score h s b)
               (output score h s b)))
  (rd <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                 (sgt21? score) r r r r r r r r r r r))
  (r <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) go go go go go go go go go go go))
  (a <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) a ff a a a a tt ff a a a))
  (score <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                    (sgt21? score) score zero score score
                    (addto score c) (addto score c) (addace score)
                    (cancelace score) score score score))
  (b <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) b ff ff b b b b b tt b b))
  (s <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) s ff ff s s s s s s tt s))
  (h <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) ff ff ff tt ff ff ff ff ff ff ff))
  (c <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) ? cd cd ? ? ?  c c c c c))
  (state <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                    (sgt21? score) get add add get tst use tst tst get
                     get get))
  (go = (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) (rin) (rin) (rin) (rin) (rin) (rin) (rin)
                (rin) (rin) (rin) (rin)))))
```

```
;-----------------------------------------------------------------------------------------------
                                  STREQNS_1.RTT
;-----------------------------------------------------------------------------------------------
    cd                  o                    rd r  a   score               b  s  h  c  state go
0)  (cardvalue (swin)) (output score h s b) r  go a   score               b  s  ff ?  get   (rin)
1)  (cardvalue (swin)) (output score h s b) r  go ff  zero                ff ff ff cd add   (rin)
2)  (cardvalue (swin)) (output score h s b) r  go a   score               ff ff ff cd add   (rin)
3)  (cardvalue (swin)) (output score h s b) r  go a   score               b  s  tt ?  get   (rin)
4)  (cardvalue (swin)) (output score h s b) r  go a   (addto score c)     b  s  ff ?  tst   (rin)
5)  (cardvalue (swin)) (output score h s b) r  go a   (addto score c)     b  s  ff ?  use   (rin)
6)  (cardvalue (swin)) (output score h s b) r  go tt  (addace score)      b  s  ff c  tst   (rin)
7)  (cardvalue (swin)) (output score h s b) r  go ff  (cancelace score)   b  s  ff c  tst   (rin)
8)  (cardvalue (swin)) (output score h s b) r  go a   score               tt s  ff c  get   (rin)
9)  (cardvalue (swin)) (output score h s b) r  go a   score               b  tt ff c  get   (rin)
10) (cardvalue (swin)) (output score h s b) r  go a   score               b  s  ff c  get   (rin)
```

```
;-----------------------------------------------------------------------
                              STREQNS_1.4
;-----------------------------------------------------------------------
((rd <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) r r r r r r r r r r r))
 (r <= (select state r rd (or s b) (ace? c) a (sgt16? score)
               (sgt21? score) go go go go go go go go go go go))
 (a <= (select state r rd (or s b) (ace? c) a (sgt16? score)
               (sgt21? score) a ff a a a a tt ff a a a))
 (score <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                   (sgt21? score) score zero score score adder0 adder0
                   adder0 adder0 score score score))
 (b <= (select state r rd (or s b) (ace? c) a (sgt16? score)
               (sgt21? score) b ff ff b b b b b tt b b))
 (s <= (select state r rd (or s b) (ace? c) a (sgt16? score)
               (sgt21? score) s ff ff s s s s s s tt s))
 (h-i = (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) ff ff ff tt ff ff ff ff ff ff ff))
 (c <= (select state r rd (or s b) (ace? c) a (sgt16? score)
               (sgt21? score) ? cd cd ? ? ? c c c c c))
 (state <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                   (sgt21? score) get add add get tst use tst tst get
                   get get))
 (adder0-i = (select state r rd (or s b) (ace? c) a (sgt16? score)
                     (sgt21? score) noop noop noop noop addto addto
                     addto addto noop noop noop))
 (adder0-v0 = (select state r rd (or s b) (ace? c) a (sgt16? score)
                      (sgt21? score) ? ? ? ? score score score score ?
                      ? ?))
 (adder0-v1 = (select state r rd (or s b) (ace? c) a (sgt16? score)
                      (sgt21? score) ? ? ? ? c c 10ptace -10ptace ?
                      ? ?)))

;-----------------------------------------------------------------------
                            STREQNS_1.4.RTT
;-----------------------------------------------------------------------
    rd r  a  score  b  s  h-i c  state adder0-i adder0-v0 adder0-v1
0)  r  go a  score  b  s  ff  ?  get   noop     ?         ?
1)  r  go ff zero   ff ff ff  cd add   noop     ?         ?
2)  r  go a  score  ff ff ff  cd add   noop     ?         ?
3)  r  go a  score  b  s  tt  ?  get   noop     ?         ?
4)  r  go a  adder0 b  s  ff  ?  tst   addto    score     c
5)  r  go a  adder0 b  s  ff  ?  use   addto    score     c
6)  r  go tt adder0 b  s  ff  c  tst   addto    score     10ptace
7)  r  go ff adder0 b  s  ff  c  tst   addto    score     -10ptace
8)  r  go a  score  tt s  ff  c  get   noop     ?         ?
9)  r  go a  score  b  tt ff  c  get   noop     ?         ?
10) r  go a  score  b  s  ff  c  get   noop     ?         ?
```

```
;-----------------------------------------------------------------
                         STREQNS_1.5
;-----------------------------------------------------------------
((rd <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) r r r r r r r r r r r))
 (r <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) go go go go go go go go go go go))
 (a <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) a ff a a a a tt ff a a a))
 (score <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) score zero score score adder0 adder0
                adder0 adder0 score score score))
 (b <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) b ff ff b b b b b tt b b))
 (s <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) s ff ff s s s s s s tt s))
 (h-i = (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) ff ff ff tt ff ff ff ff ff ff ff))
 (c <= (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) ? cd cd ? ? ? c c c c c))
 (adder0-v1 = (select state r rd (or s b) (ace? c) a (sgt16? score)
                (sgt21? score) ? ? ? ? c c 10ptace -10ptace ? ?
                ?)))

;-----------------------------------------------------------------
                        STREQNS_1.5.RTT
;-----------------------------------------------------------------
    rd r  a  score  b  s  h-i c  adder0-v1
0)  r  go a  score  b  s  ff  ?  ?
1)  r  go ff zero   ff ff ff  cd ?
2)  r  go a  score  ff ff ff  cd ?
3)  r  go a  score  b  s  tt  ?  ?
4)  r  go a  adder0 b  s  ff  ?  c
5)  r  go a  adder0 b  s  ff  ?  c
6)  r  go tt adder0 b  s  ff  c  10ptace
7)  r  go ff adder0 b  s  ff  c  -10ptace
8)  r  go a  score  tt s  ff  c  ?
9)  r  go a  score  b  tt ff  c  ?
10) r  go a  score  b  s  ff  c  ?
```

```
;----------------------------------------------------------------------
                                SEQSYS_2
;----------------------------------------------------------------------
((define select
   (lambda (s p0 p1 p2 p4 p5 p6 v0 v1 v2 v3 v4 v5 v6 v7 v8 v9)
      (case s
         [get (if p0 (if p1 v0 (if p2 v1 v2)) v3)]
         [add v4]
         [use v5]
         [tst (if p5 (if p6 (if p4 v6 v7) v8) v9)])))

 ((rd <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                 r r r r r r r r r r))
  (r <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                go go go go go go go go go go))
  (a <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                a ff a a a tt ff a a a))
  (score <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                    score zero score score adder0 adder0 adder0 score
                    score score))
  (b <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                b ff ff b b b b tt b b))
  (s <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                s ff ff s s s s s tt s))
  (h-i = (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                 ff ff ff tt ff ff ff ff ff ff))
  (c <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                ?  cd cd ?  ?  c c c c c))
  (adder0-v1 = (select state r rd (or s b) a (sgt16? score)
                       (sgt21? score) ? ? ? ? c 10ptace -10ptace ? ?
                       ?))))

;----------------------------------------------------------------------
                              STREQNS_2.RTT
;----------------------------------------------------------------------
    rd r  a  score  b  s  h-i c  adder0-v1
0)  r  go a  score  b  s  ff  ?  ?
1)  r  go ff zero   ff ff ff  cd ?
2)  r  go a  score  ff ff ff  cd ?
3)  r  go a  score  b  s  tt  ?  ?
4)  r  go a  adder0 b  s  ff  ?  c
5)  r  go tt adder0 b  s  ff  c  10ptace
6)  r  go ff adder0 b  s  ff  c  -10ptace
7)  r  go a  score  tt s  ff  c  ?
8)  r  go a  score  b  tt ff  c  ?
9)  r  go a  score  b  s  ff  c  ?
```

```
;---------------------------------------------------------------------
                         STREQNS_2.BIN
;---------------------------------------------------------------------
((score.0 <= (select state r rd (or s b) a (sgt16? score)
                  (sgt21? score) score.0 0 score.0 score.0 adder.0
                  adder.0 adder.0 score.0 score.0 score.0))
 (score.1 <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                  score.1 0 score.1 score.1 adder.1 adder.1 adder.1
                  score.1 score.1 score.1))
 (score.2 <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                  score.2 0 score.2 score.2 adder.2 adder.2 adder.2
                  score.2 score.2 score.2))
 (score.3 <= (select state r rd (or s b) a (sgt16? score)
                  (sgt21? score) score.3 0 score.3 score.3 adder.3
                  adder.3 adder.3 score.3 score.3 score.3))
 (score.4 <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
                  score.4 0 score.4 score.4 adder.4 adder.4 adder.4
                  score.4 score.4 score.4))
 (c.0 <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
              ? Cd.0 Cd.0 ? ? c.0 c.0 c.0 c.0 c.0))
 (c.1 <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
              ? Cd.1 Cd.1 ? ? c.1 c.1 c.1 c.1 c.1))
 (c.2 <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
              ? Cd.2 Cd.2 ? ? c.2 c.2 c.2 c.2 c.2))
 (c.3 <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
              ?  Cd.3 Cd.3 ? ? c.3 c.3 c.3 c.3 c.3))
 (adder-b.0 = (select state r rd (or s b) a (sgt16? score)
                   (sgt21? score) ?  ?  ?  ?  c.0 0 0 ?  ?  ?))
 (adder-b.1 = (select state r rd (or s b) a (sgt16? score)
                   (sgt21? score) ? ? ? ? c.1 1 1 ? ? ?))
 (adder-b.2 = (select state r rd (or s b) a (sgt16? score)
                   (sgt21? score) ? ? ? ? c.2 0 1 ? ? ?))
 (adder-b.3 = (select state r rd (or s b) a (sgt16? score)
                   (sgt21? score) ? ? ? ? c.3 1 0 ? ? ?))
 (rd <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
             r r r r r r r r r r))
 (r <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
            go go go go go go go go go go))
 (a <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
            a 0 a a a 1 0 a a a))
 (b <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
            b 0 0 b b b b 1 b b))
 (s <= (select state r rd (or s b) a (sgt16? score) (sgt21? score)
            s 0 0 s s s s s 1 s))
 (h* = (select state r rd (or s b) a (sgt16? score) (sgt21? score)
            0 0 0 1 0 0 0 0 0 0)))
```

```
;-------------------------------------------------------------------------------------------------------------
                                       STREQNS_2.BIN.RTT
;-------------------------------------------------------------------------------------------------------------
    score.0 score.1 score.2 score.3 score.4 c.0  c.1  c.2  c.3  adder-b.0 adder-b.1 adder-b.2 adder-b.3 rd r  a b s h*
0)  score.0 score.1 score.2 score.3 score.4 ?    ?    ?    ?    ?         ?         ?         ?         r  go a b s 0
1)  0       0       0       0       0       Cd.0 Cd.1 Cd.2 Cd.3 ?         ?         ?         ?         r  go 0 0 0 0
2)  score.0 score.1 score.2 score.3 score.4 Cd.0 Cd.1 Cd.2 Cd.3 ?         ?         ?         ?         r  go a 0 0 0
3)  score.0 score.1 score.2 score.3 score.4 ?    ?    ?    ?    ?         ?         ?         ?         r  go a b s 1
4)  adder.0 adder.1 adder.2 adder.3 adder.4 ?    ?    ?    ?    c.0       c.1       c.2       c.3       r  go a b s 0
5)  adder.0 adder.1 adder.2 adder.3 adder.4 c.0  c.1  c.2  c.3  0         1         0         1         r  go 1 b s 0
6)  adder.0 adder.1 adder.2 adder.3 adder.4 c.0  c.1  c.2  c.3  0         1         1         0         r  go 0 b s 0
7)  score.0 score.1 score.2 score.3 score.4 c.0  c.1  c.2  c.3  ?         ?         ?         ?         r  go a 1 s 0
8)  score.0 score.1 score.2 score.3 score.4 c.0  c.1  c.2  c.3  ?         ?         ?         ?         r  go a b 1 0
9)  score.0 score.1 score.2 score.3 score.4 c.0  c.1  c.2  c.3  ?         ?         ?         ?         r  go a b s 0
```

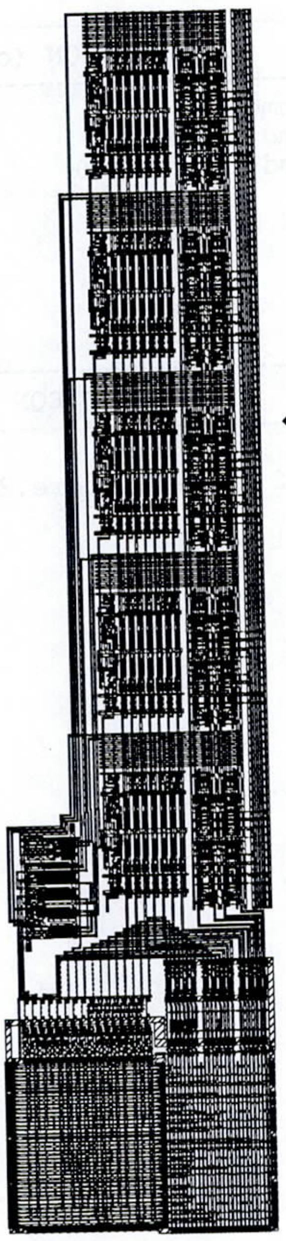
```
;-----------------------------------------------------------------
                            STREQNS_2.ORG
;-----------------------------------------------------------------
([(score.0 <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            0 score.0 score.0 adder.0 adder.0 adder.0 score.0 score.0
            score.0))
  (adder-b.0 =
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            ? ? ? c.0 0 0 ? ? ?))
  (c.0 <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            Cd.0 Cd.0 ? ? c.0 c.0 c.0 c.0 c.0))]

 [(score.1 <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
             0 score.1 score.1 adder.1 adder.1 adder.1 score.1 score.1
             score.1))
  (adder-b.1 =
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            ? ? ? c.1 1 1 ? ? ?))
  (c.1 <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            Cd.1 Cd.1 ? ? c.1 c.1 c.1 c.1 c.1))]

 [(score.2 <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            0 score.2 score.2 adder.2 adder.2 adder.2 score.2 score.2
score.2))
  (adder-b.2 =
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            ? ? ? c.2 0 1 ? ? ?))
  (c.2 <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            Cd.2 Cd.2 ? ? c.2 c.2 c.2 c.2 c.2))]

 [(score.3 <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            0 score.3 score.3 adder.3 adder.3 adder.3 score.3 score.3
            score.3))
  (adder-b.3 =
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            ? ? ? c.3 1 0 ? ? ?))
  (c.3 <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            Cd.3 Cd.3 ? ? c.3 c.3 c.3 c.3 c.3))]

 [(score.4 <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
            0 score.4 score.4 adder.4 adder.4 adder.4 score.4 score.4
            score.4))]
;-----------------------------------------------------------------------
                        STREQNS_2.ORG (cont)
;-----------------------------------------------------------------------
 [(a <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
             0 a a a 1 0 a a a))
  (b <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
             0 0 b b b b 1 b b))
  (s <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
             0 0 s s s s s 1 s))
  (h* =
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
             0 0 1 0 0 0 0 0 0))
  (rd <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
             r r r r r r r r r))
  (r <=
    (select state r rd (or s b) (ace? c) a (sgt16? score) (sgt21? score)
             go go go go go go go go go))])
```

```
;-----------------------------------------------------------------------
                                NEXTSTATE
;-----------------------------------------------------------------------
(define state
   (lambda (s p0 p1 p2 p3 p4 p5 p6)
      (case s
         [get (if p0 (if p1 get (if p2 add add)) get)]
         [add (if p3 (if p4 tst use) tst)]
         [use tst]
         [tst (if p5 (if p6 (if p4 tst get) get) get)])))

;-----------------------------------------------------------------------
                                NEXTSTATE.BIN
;-----------------------------------------------------------------------
((define k.1
    (lambda (s p4 p5 p6)
       (case s
          [get 0]
          [add 1]
          [use 1]
          [tst (if p5 (if p6 (if p4 1 0) 0) 0)])))
 (define k.0
    (lambda (s p0 p1 p3 p4 p5 p6)
       (case s
          [get (if p0 (if p1 0 1) 0)]
          [add (if p3 (if p4 1 0) 1)]
          [use 1]
          [tst (if p5 (if p6 (if p4 1 0) 0) 0)]))))
```

```
;---------------------------------------------------------------------
                                SEL_2
;---------------------------------------------------------------------
(define select
   (lambda (s p0 p1 p2 p4 p5 p6 v0 v1 v2 v3 v4 v5 v6 v7 v8 v9)
      (case s
         [get (if p0 (if p1 v0 (if p2 v1 v2)) v3)]
         [add v4]
         [use v5]
         [tst (if p5 (if p6 (if p4 v6 v7) v8) v9)])))


;---------------------------------------------------------------------
                              ENCODE.BIN
;---------------------------------------------------------------------
((define cmd.3
    (lambda (s p5 p6)
       (case s
          [get 0]
          [add 0]
          [use 0]
          [tst (if p5 (if p6 0 1) 1)])))
 (define cmd.2
    (lambda (s p5 p6)
       (case s
          [get 0]
          [add 1]
          [use 1]
          [tst (if p5 (if p6 1 0) 0)])))
 (define cmd.1
    (lambda (s p0 p1 p2 p5 p6)
       (case s
          [get (if p0 (if p1 0 (if p2 0 1)) 1)]
          [add 0]
          [use 0]
          [tst (if p5 (if p6 1 0) 0)])))
 (define cmd.0
    (lambda (s p0 p1 p2 p4 p5 p6)
       (case s
          [get (if p0 (if p1 0 (if p2 1 0)) 1)]
          [add 0]
          [use 1]
          [tst (if p5 (if p6 (if p4 0 1) 0) 1)]))))
```

```
;-----------------------------------------------------------------------
                              CONTROL.EQN
;-----------------------------------------------------------------------
NAME = CONTROL.EQN;
INORDER = p1 p4 p6 p0 p5 p3 go cmd.0 cmd.1 cmd.2 cmd.3 k.0 k.1
          a b r s p2 ;
OUTORDER = p2 s r b a k.1 k.0 cmd.3 cmd.2 cmd.1 cmd.0 h* rd ;

cmd.0 = !k.0 & p2 & !p1
      | k.1 & !p4 & p6
      | !k.0 & !p0
      | !p5 & k.1
      | k.1 & !k.0 ;

cmd.1 = !k.1 & !k.0 & !p2 & !p1
      | p6 & p5 & k.1 & k.0
      | !k.1 & !k.0 & !p0 ;

cmd.2 = p6 & p5 & k.0
      | k.1 & !k.0
      | !k.1 & k.0 ;

cmd.3 = k.1 & k.0 & !p6
      | !p5 & k.1 & k.0 ;

k.0 = p4 & p6 & p5 & k.0
    | !k.0 & !p1 & p0
    | p4 & !k.1 & k.0
    | !k.1 & k.0 & !p3
    | k.1 & !k.0 ;

k.1 = p4 & p6 & p5 & k.0
    | k.1 & !k.0
    | !k.1 & k.0 ;

a = !cmd.3 & !cmd.2 & !cmd.0 & a
  | !cmd.3 & !cmd.1 & !cmd.0 & a
  | !cmd.3 & cmd.1 & cmd.0 & a
  | cmd.3 & !cmd.2 & !cmd.1 & a
  | !cmd.3 & cmd.2 & !cmd.1 & cmd.0 ;

b = !cmd.2 & !cmd.1 & !cmd.0 & b
  | cmd.3 & !cmd.2 & !cmd.1 & b
  | !cmd.3 & cmd.2 & b
  | !cmd.3 & cmd.1 & cmd.0 & b
  | !cmd.3 & cmd.2 & cmd.1 & cmd.0 ;

h* = !cmd.3 & !cmd.2 & cmd.1 & cmd.0 ;

r = !cmd.2 & !cmd.1 & go
  | !cmd.3 & go ;

rd = !cmd.2 & !cmd.1 & r
   | !cmd.3 & r ;
```

```
;-----------------------------------------------------------------------
                             CONTROL.EQN (cont)
;-----------------------------------------------------------------------
s = !cmd.2 & !cmd.1 & !cmd.0 & s
  | cmd.3 & !cmd.2 & !cmd.1 & s
  | cmd.3 & !cmd.2 & !cmd.1 & !cmd.0
  | !cmd.3 & cmd.2 & s
  | !cmd.3 & cmd.1 & cmd.0 & s ;

p2 = s | b ;


;-----------------------------------------------------------------------
                                  PREDS.EQN
;-----------------------------------------------------------------------
NAME = PREDS.EQN;
INORDER = score.4 score.2 score.1 score.3 score.0 a c.3 c.2 c.1 c.0
          rd r ;
OUTORDER = p6 p5 p4 p3 p1 p0 ;

p6 = score.4 & score.2 & score.1
   | score.4 & score.3 ;

p5 = score.4 & score.3
   | score.4 & score.2
   | score.4 & score.1
   | score.4 & score.0 ;

p4 = a ;

p3 = c.3 & !c.2 & c.1 & c.0 ;

p1 = rd ;

p0 = r ;
```



Realization

The realization was decomposed into three physical blocks. Each block was generated using a different VLSI layout tool demonstrating a mixed layout solution. **BLOCK 1**: SLICE_0..4.RTBA: A RTBA implementation of Score, C, and addto. RTBA (Register Transfer Bus Architecture) [10] defines a layout style designed to maximize register transfers within a single bit slice, and the incorporation of arithmetic functional unit. **BLOCK 2**: PREDS.EQN: A standard cell implementation of the status predicates, sgt16?, gt21?, and ace?, were derived using OCT Tools MISII and the MSU standard cell library [13]. **BLOCK 3**: CONTROL.EQN: A PLA implementation of ENCODE, NEXTSTATE, A, B, S, H*, Rd, R, and the RTBA control logic. A PLA was generated using the Berkeley VLSI Tools PLA generator, eqntott and MPLA [12].