ADA TO SILICON TRANSFORMATIONS: THE OUTLINE OF A METHOD

by

Lawrence A. Drenan[1] and Elliott I. Organick

Dept. of Computer Science
University of Utah
Salt Lake City, Utah 84112

This research was sponsored in part by the Defense Advanced Research Projects agency, DARPA contract No. MDA903-81-C-0414.

September 1982

---

[1]Presently employed by Western Digital Corp, 2445 McCabe Way, Irvine, CA 92715

ABSTRACT

This report explores the contention that a high-order language specification of a machine (such as an Ada program) can be methodically transformed into a hardware representation of that machine. One series of well-defined steps through which such transformations can take place is presented in this initial study.

The general method consists of a two-fold strategy:

1. Transform the high-level specification into a network of inter-communicating "state machine/data path pairs".

2. Through a catalogue method, map each state machine / data path pair into a circuit realization.

Four representational levels are utilized in the transformation process. Each inter-level transformation is discussed. The four levels are:

1. Ada specification of the algorithm.

2. Machine-description specification of the algorithm, consisting of a control part and a data part. This version is expressed in a stylized dialect of Ada developed for this study.

3. Protocol-definition specification of the algorithm, obtained by inserting constructs that define inter-program unit communication.

4. Storage/Logic Array (SLA) specification of the algorithm, which can be mapped directly to, and are regarded as equivalent to, circuit representations.

The transformation strategy relies upon exploiting a one-to-one correspondence between Ada instantiations of generic packages introduced in the level 2 representation and SLA "modules", which are composed of primitive SLA cells introduced at level 4.

The transformation methodology described in the paper has been demonstrated for a non-trivial Ada program example.

1. Introduction

This report reviews elementary principles applicable for methodically transforming a high-order language specification of a machine, such as an Ada program, into a hardware representation of that machine. In this initial study, we discuss one series of well-defined steps through which such transformations

can take place.

Research on automating Ada-to-Silicon transformations is currently underway at the University of Utah [9]. In this report, which does not attempt to document the specifics of the mainstream of that research, we outline a series of mappings for transforming individual Ada program units to equivalent integrated circuits. Our emphasis is on the feasibility of these transformations and is not concerned with finding a series of optimal transformation steps. Our purpose is to:

1. Demonstrate one (relatively straightforward) approach by which an Ada program can be mapped into a specification of an integrated circuit (IC) through adherence to rule-based techniques.

2. Examine the pros and cons inherent in the most straightforward, unoptimized approach.

The method presented follows the general transformation strategy suggested earlier [8]. The essence of this strategy is to represent each Ada program unit as a synchronous stored state machine part and a data path part. Circuits derived by following this approach have the general form pictured in Figure 1-1. The pairing of a state machine and a data path (i.e., an environment) is referred to as an "engine". The hardware realization of an entire Ada program, or of any subset of program units of that program, is actually a network of asynchronously intercommunicating engines, each having the form outlined in Figure 1-1. For the convenience of this report, individual Ada tasks are considered to be program units.

A transformation methodology is just beginning to be explored [11]. There is need to develop a well-defined set of rules through which such transformations can eventually become a mechanical process. Some guidelines that distinguish a set of rules as having the potential for eventual automation have been suggested [10].

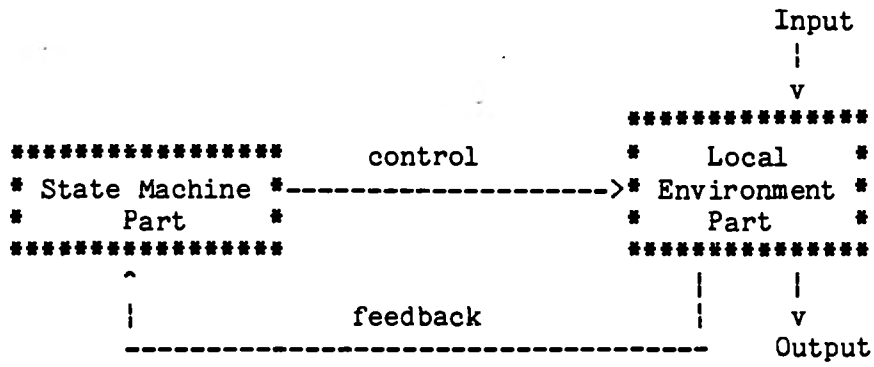

Figure 1-1: An Engine and Its Two Principal Components

The transformations presented here are considered to be extensions of those originally outlined in the following sense:

1. Not only is the high-level specification of a program unit expressed in Ada; intermediate levels of representation are also expressed in Ada. "Machine-description" and "Protocol-definition" styles of Ada programming are proposed to express intermediate transformation steps, permitting the algorithmic behavior to be checked through Ada program execution at all intermediate levels as well as the top level.

2. NMOS Storage Logic Array (SLA) technology [15] [14] is chosen for the low-level realization of the machine. (More practical versions of SLAs, called PPLs have been developed to serve as a target for this transformation process [9].) SLA "modules" give us a set of building blocks that fit the specific needs of this method. Utilization of other semi-custom integrated circuit components offers an opportunity for enrichment of this methodology into the VLSI range.

A high-order language Ada program is transformed in three steps to reach the level of representation from which integrated circuits may be produced directly. In this report, the four levels, counting the starting level, are called "stages". These stages are:

1. High-level Ada program
2. Machine-description Ada program
3. Protocol-definition Ada program
4. NMOS SLA program or equivalent

Characteristics of these stages and rules that guide the transformations between them are presented in succeeding sections. A case study that was performed following this method on a non-trivial Ada program is presented elsewhere [6].

[We again stress that circuit optimization (space or speed) is not a goal addressed in this paper. Thus, in situations where performance or circuit area or both are critical, the approach presented is unlikely to yield circuits with characteristics that are competitive with those produced by more custom methods, especially for many important, but special algorithms, e.g., those that lead to compact systolic arrays.]

2. Stage 1: High-Level Ada Program

The machines specified and realized by our transformation process are viewed as ensembles of interacting state machine/environment pairs (engines). The programming language Ada is well-suited for specifying such pairs. Thus, a strong correlation exists between data abstractions in Ada and data abstractions in certain views of integrated circuits; indeed we exploit this correlation.

An Ada program is composed of one or more program units [5] [2]. A program begins execution as a single thread of control in the main subprogram, but can initiate tasks, each of which has associated with it a separate thread of control. A program unit in this model is analogous to a machine that is initiated via a single "Go" button, but which is capable of delegating work among potentially concurrent sub-machines. In Ada, such sub-machines take the form of tasks. Ada also offers flexibility and control in specifying the communication between program units, i.e., in specifying the kind of interaction between units. Data abstractions represented as Ada packages, another form of program unit, are also transformable into individual engines whose operators either transform given instances of a data type or own and operate on individual instances. Shifting such an engine from idle to a particular active state

corresponds, at a higher level of abstraction, to the activation of an Ada package operation.

Information needed to represent an engine can be extracted from an Ada program unit for use in representing the local environment (data path) and the state machine (controller). This information is drawn both from the specification part and from the body part of the program unit being mapped to the next stage.

Stage 2 representation elaborates intra-program unit constructs while Stage 3 elaborates inter-program unit communication constructs. The language for Stage 2 is a stylized but legal form of Ada.

3. Stage 2: Machine-description-level Ada program

3.1. The Role of Stage 2

A Stage 2 program achieves two objectives:

1. Infers a collection of needed hardware modules from the declaration part of the program unit and identifies the needed modules through instantiation of generic packages.

2. Transforms infix expressions represented in the Stage 1 form into prefix form.

The distinction between the control flow and data flow of a program is sharpened by the transformation from Stage 1 to Stage 2. Thus, in its Stage 2 form, the program takes the form of a state machine and the data path it controls. The declarative part of the Stage 2 form represents a collection of hardware modules (a "data path") inferred from the declarative part of the Stage 1 form. The body part of the Stage 2 form represents a state machine whose structure is inferred from both the declarative and body parts of the Stage 1 form. The Stage 2 language style has two distinguishing features:

- extensive use of generic building blocks

- use of the "engine extension" style of representing states and state transitions

The terms "building block" and "module" have specific meanings below. A "building block" refers to a generic package instance introduced in Stage 2 to model a particular component of the data path. A "module" refers to a collection of SLA cells from which the full circuit will be constructed. Every generic package instance identified in the Stage 2 representation maps to a corresponding Stage 4 SLA module.

3.2. Stage 2 Examples

Figure 3-1 is an example of a generic package declaration for a building block representing a counter. An instantiation of this package (e.g., "package C is new Counter") corresponds to the module's "black box" representation (see Figure 3-2). The SLA program that corresponds to Figure 3-2 is presented in Figure 3-3.

```
generic
  lo_value: integer;
  hi_value: integer;
            -- allows one to instantiate
            -- counters of various sizes
package Counter is
    -- Function:
    --   a counter with load, lookup,
    --   increment, and decrement operations
  procedure Load(
      load_value: in integer );
  procedure Increment;
            -- Increment by 1 is implied.
  procedure Decrement;
            -- Decrement by 1 is implied.
  function Lookup return integer;
            -- Returns the current value.
end Counter;
```

Figure 3-1: Counter Building Block Package Specification

With a few exceptions (to be discussed below) all variables and operators in the Stage 1 program unit are transformed into instantiations of generic

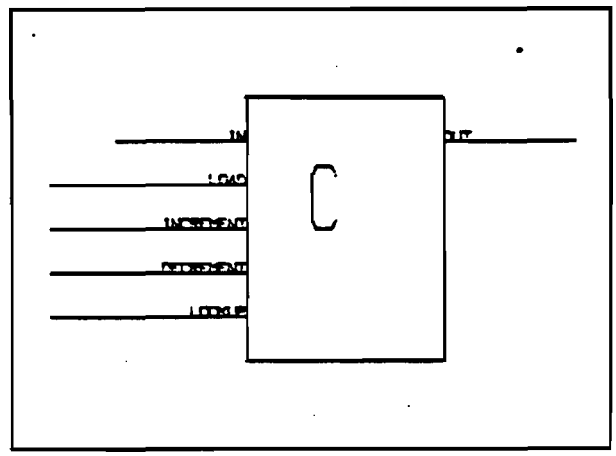

Figure 3-2: "Black Box" Representation of a Counter Module

packages. The Stage 2 code is then restricted to describing actions through the use of these instantiated packages. Stage 1 to Stage 2 transformations result in code that is composed primarily of function and procedure applications. For example, a line of code such as

```
A := B + C;
```

is transformed into

```
A.Write(Add.Go(B.Read, C.Read));
```

where A, B, C, and Add are previously instantiated packages. Thus, if the Stage 1 code includes the object declaration

```
A, B, C: integer;
```

the corresponding Stage 2 form would exhibit the instantiations

```
package A is new Register(word_length => integer);
package B is new Register(word_length => integer);
package C is new Register(word_length => integer);
```

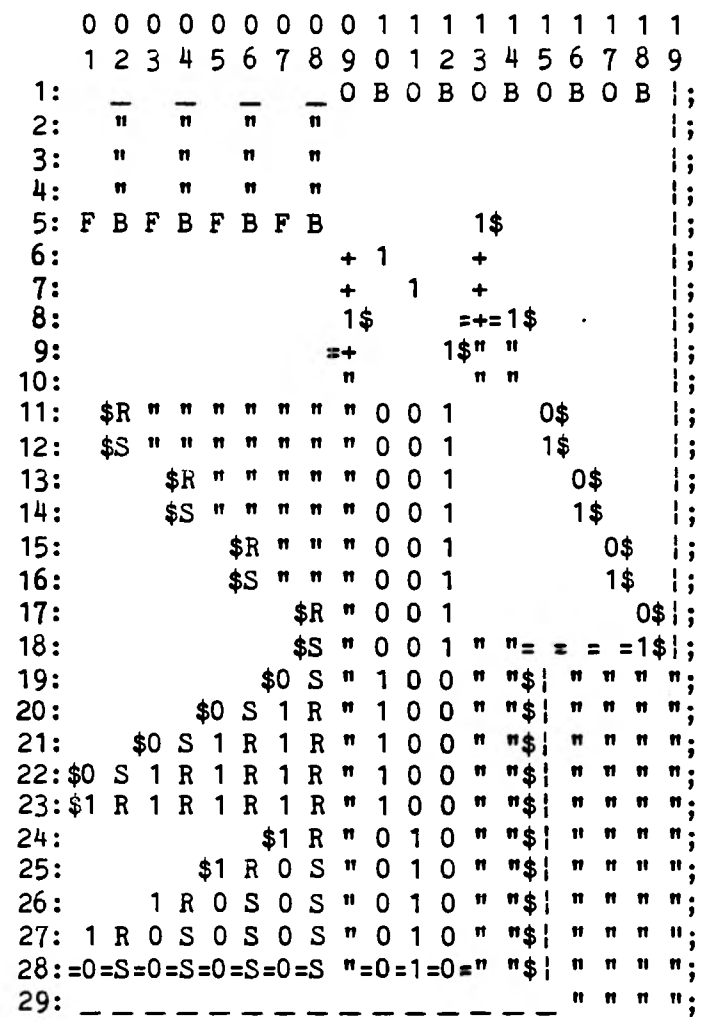

Figure 3-3: SLA Program for Counter Module Using the SCLED Notation

Furthermore, encountering "+" while parsing Stage 1 code would lead to the inclusion of

```
package Add is new Adder;
```

in the corresponding declarative part of the Stage 2 code. Hence, the code presented in this example would eventually map into a hardware structure abstractly presented in Figure 3-4.

The design of the building block set and the design of the SLA module set must be coordinated. As a possible means of enforcing the design discipline, a Stage 2 programmer is provided with one or more packages that specify the set of

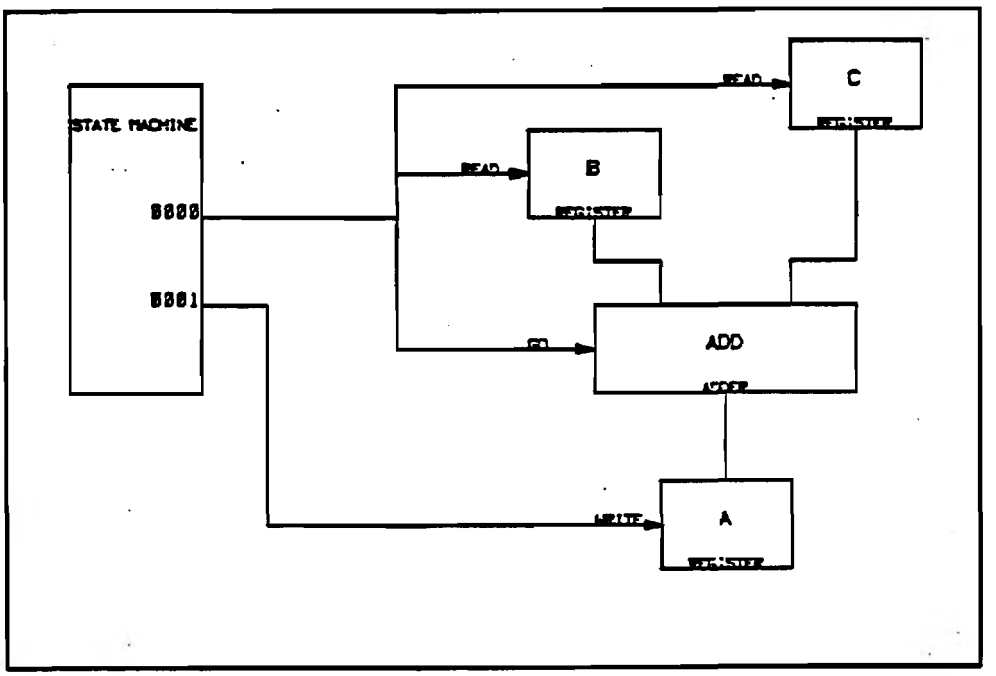

Figure 3-4: Hardware Realization of "A := B + C;"

generic packages available. The programmer can thereby be restricted to expressing algorithms with instantiations and use of the pre-defined generic packages.

3.3. The "Engine" Extension to Ada

The body part of a Stage 2 program is sub-divided into states denoted by labels. To represent the mutually independent actions that can occur in the same state of a state machine in standard Ada, one could use the "verbose form" that declares (and then initiates) a set of dynamically created tasks. A more succinct equivalent is possible if we were to include an "engine extension" for Ada to specify a similar objective. Used at Stage 2, the engine extension allows one to specify a sequence of Ada statements that can be translated into concurrent actions.

An engine clause has the structure illustrated in Figure 3-5. Within the scope of an engine clause, the sequence of statements bounded by two state

```
engine Example is
  begin
    <<State_Start>>  -- initial actions
                     -- executed in parallel


    <<State_1>>      -- actions to be
                     -- executed in parallel

    <<State_2>>      -- another set of actions which
                     -- can be executed in parallel

    <<State_stop>>   -- final state
                     null;
  end Example;
```

Figure 3-5: Structure of an Engine Clause for Representing "Transition Graph" of a State Machine

labels, e.g., <<State_1>> and <<State_2>> above, are actions that can occur in parallel. Execution of a "goto" statement within such a (labeled) sequence terminates the actions within that state (i.e., triggers a state transition). (To enhance readability, we follow the convention that the first node of every engine clause be labeled "State_Start" and the final node be labeled "State_Stop".)

Nesting of engines clauses follows Ada scoping rules. An engine may be declared local to another engine just as one procedure can be declared local to another procedure. Thus a local "sub-engine" may be called from its containing "main-engine". The effect of such a call is to transfer control to the label State_Start of the subengine at the time the subengine is called and to return control to the main engine when the subengine completes.

Note that this technique does not imply a relationship between state transitions and units of time. Although the particular SLA implementation chosen for Stage 4 in this work is synchronous, a syntax comparable to the engine extension has been be mapped to asynchronous implementations [4]. An algorithm used to determine the operations for which one can specify parallel execution, i.e., multiple actions within the same state, is presented in Section

5.

3.4. Building Blocks and Modules

For the purpose of this report, the following building blocks and modules have been designed [6]: Equals, Less_eq, Bool_eq, Counter, Loop_Counter, Register, Boolean_Register, Memory, and Two_D_Memory.

Building blocks and modules generally have parameters for specifying word lengths. Such specifications are provided by the Stage 2 programmer as part of an interactive design process. Thus, most generic package declarations contain the formal generic parameter

```
type word_length is range <>;
```

3.5. Three Intra-program Unit Communications Protocols

Three different intra-program unit protocols are defined, corresponding to the "function", "procedure", and "procedurE" Stage 2 subprogram declarations. These Stage 2 declarations convey assumptions about the number of states required for an operation to "complete its job". Different protocols may be utilized for invoking various operations within an implemented package. The corresponding SLA implementation is invoked with whichever protocol is appropriate. Protocols for communication between circuits representing separate Ada program units are discussed in Section 6.)

Operations are divided into two classes: those that return a value (e.g., a Read operation) and those that do not (e.g., a Write operation). Hardware implementation of the former requires that the module includes storage elements to hold the value of the output parameter (or function result). The protocols presented below ensure that such storage elements are sampled only after the correct values are loaded. In operations that do not return a value, the protocols ensure that the module completes its job (for example, modification of a global value) before a potentially conflicting operation can be initiated.

The distinguishing characteristics of operations adhering to each of the three protocols are as follows:

- "Function" protocol: The operation completes in the same state in which a request for the operation reaches the containing module. Two cases are implementable:

    1. The function result is always available.

    2. The request is received in phase Phi-1 of a given clock cycle, and causes the result to be available in phase Phi-2 of the same clock cycle.

  A function operation (such as the Lookup operation on a Counter module) does not need to issue an acknowledge to its requestor that it has performed its duty, because it can be assumed that the correct result will be available in a known state.

- "Procedure" protocol: The operation completes in the state immediately following the one in which the request reaches the module. As in the function protocol, it is not necessary for the procedure operation (such as the Increment operation on a Counter module) to inform the requestor that the desired action has been performed.

- "ProcedurE" protocol: For this operation, it cannot be assumed that the job will be completed in the same state in which the request is received, or even in the next state. Unlike the two previous protocols, it is necessary for the containing module to inform the requestor when execution of the desired action has been completed. The scenario is as follows: a requestor initiates a procedurE operation by issuing a "Go" signal; the procedurE in turn signals its caller, upon successful completion, with an "I'm done" signal. We call this convention the "Go/I'm done" protocol. Its use allows the introduction of arbitrary delays in the state transitions for clocked schemes that exhibit a single thread of control. The protocol, which is enforced by construction, is implemented as follows:

    * The requesting engine R sends a "Go" signal that invokes the type procedurE operation P of a containing module M and then enters a state where R waits for M to send an "I'm done" signal.

    * The initial state of M is a wait state for a "Go" signal. A Go for P causes the states the operation P to commence (transition to P). After the operation P completes M emits an "I'm done" signal before returning to its initial state.

  The protocol permits representation of a single thread of control that traverses from the requesting engine R to the host module M of the procedurE operation P and back again. The sequence of state transitions for every procedurE operation is local to one, and only one, engine. Hence, there is no possibility for contention. It is

this fact that allows us to use the simple "Go/I'm Done" protocol (instead of a somewhat more complex Request/Acknowledge) for intra-engine communication. The Read and Write operations on the Memory module are examples of the procedurE protocol.

4. Stage 1 to Stage 2 Transformations

4.1. Transforming Simple Expressions

Simple expressions are transformed in a straightforward way. Registers replace variables, comparators replace relational operators, adders replace plus signs, etc. Such transformations are syntax driven.

This style of transformation leads to the allocation of possibly redundant modules. Clearly, circuits produced by this method tend to be wasteful of "real estate". However, timing and communications are simplified in activating individual modules, since each Stage 2 call on a subprogram operation of a generic instantiation then corresponds to a unique control line in the hardware level. Some simple optimizations are possible within this framework; for example, use of counters where adders are not needed, and use of shift logic, where suitable, for multiplication or division.

4.2. Transforming Control Statements

The interpretation of control statements (e.g., loop, case, if, subprogram calls and task entry calls) lead to control flow changes. We discuss the required transformations for such constructs in this subsection on a case by case basis. In general, these transformations mimic well-understood strategies used by compilers [1].

Procedures, functions, and tasks The initial action to be performed in the body parts of procedure, function, and task entries with in parameters is the loading of the actual parameter values into the Registers that implement the corresponding formal parameters. Statements directing such actions must be

inserted into the Stage 2 program.

Out parameters also require instantiation of Register packages so their values can be loaded into these Registers as if they were local parameters and hence mimic the "copy-restore" parameter passing mechanism demanded (for the normal case) by Ada semantics. A similar treatment is required so that function values can be properly returned.

Building blocks that represent formal parameters of program units are derived in Stage 2. For example, if procedure P and function F are specified as:

```
procedure P(
    xx: integer;
    yy: integer);
function F(
    zz: integer)
   return real;
```

then four generic packages are instantiated:

```
package xx is new Register(word_length => in integer);

package yy is new Register(word_length => in integer);
                                                        -- For P.

package zz is new Register(word_length => in integer);

package f_result is new Register(word_length => real);
                                                        -- For F.
```

IF-STATEMENTS In the simplest case, if-statements are manifested in Stage 2 as structures of the form:

```
<<State_for_if>>  if  condition  then
                    goto State_X;
                  else
                    goto State_Y;
                  end if;
```

Missing but implicit else clauses are explicitly inserted. For example:

```
                  else
                    goto State_<the_state_where_the_2_branches_join>;
```

It is certainly possible, and in many cases advisable, to include actions in the branches before the goto statement, thereby reducing the total number of states specified in the machine description. For example,

```
if  mem_value = 0  then
  pointer := p_find;
  exit;
end if;
```

is transformed into

```
declare
  equals_result: boolean := false; -- Initialized to
   ...                             -- false.
begin
   ...
<<State_4>>  Equals.Test(
                 Mem_value.Lookup(), 0, equals_result);
             goto State_5;

<<State_5>>  if  equals_result  then
               Pointer.Write(P_find.Lookup());
               goto State_6;          -- Goes to exit.
             else                     -- Else is now explicit
               goto State_7;
             end if;
```

Notice the use of the boolean variable "equals_result" to represent the value of the condition. The rule followed is that the use of identifiers with "_result" as a suffix specifies Stage 4 routing to a storage element that is located within the module specified by the prefix (e.g., Equals). The storage element is loaded with the result of the operation. Every relational operator building block has such a "buddy" boolean variable. Out parameters in procedures and procedurEs, such as the value returned from a memory Read procedurE, are also treated this way.

BLOCKS A block is treated as a parameterless procedure.

FOR-LOOPS A generic Loop_Counter package that computes and holds the loop parameter value is instantiated for each Stage 1 for-loop. This package also

stores the value of the upper limit of the discrete range. In case the upper bound is a previously declared variable, e.g., Lim, a module that stores Lim's value already exists, so the extra storage element is redundant. This redundancy is accepted because, at the hardware level, the simplicity of communication and saving of extra communications lines appears to outweigh the use of extra storage space. Figure 4-1 shows the Stage 1 to Stage 2 transformation paradigm used for for-loops.

```
        STAGE 1                 |      STAGE 2
                                |
                                | -- Declaration part
                                | package Parameter is new Loop_Counter;
                                |                    -- Instantiation.
                                |                .
                                |                .
                                |                .
                                | -- Body part
                                | <<State_X>>   Parameter.Load (A, B);
                                |                   -- Load loop values.
                                |                   -- A is initial value.
                                |                   -- B is upper limit.
                                | <<State_Y>>   if Parameter.Test()  then
                                |                   -- Test the parameter
                                |                   -- versus upper bound.
for parameter in A..B           |                  goto State_Y+1;
                                |                   -- Go to the sequence
                                |                   -- of statements.
loop                            |                else
  Statement_1;                  |                  goto State_Z+1;
                                |                   -- Exit from loop.
                                |                end if;
  Statement_2;                  | <<State_Y+1>> Statement_1;
      .                         |
      .                         | <<State_Y+2>> Statement_2;
  Statement_N;                  |                   .
end loop;                       |                   .
                                | <<State_Y+N>> Statement_N;
                                |
                                | <<State_Z>>    Parameter.Increment();
                                |                goto State_Y;
                                |                   -- Go back to the test.
                                | <<State_Z+1>>
                                |                   -- Continue with the
                                |                   -- rest of the program.
```

Figure 4-1: A Paradigm For-Loop Transformation

WHILE-LOOPS While-loop transformations require the instantiation of as many building block packages as required to evaluate the while-loop condition. The Stage 2 expression of a while-loop whose condition is a simple equality test is modeled in Figure 4-2.

```
<<State_Y>>     Equals.Test(
                    first_operand, second_operand, equals_result);
                goto State_Y+1;
<<State_Y+1>>   if  equals_result  then
                  goto State_Y+2;
                else
                  goto State_Z+1;      -- Exit the loop.
                end if;

<<State_Y+2>>   Statement_1;           -- Begin loop body.
                    .
                    .
<<State_Y+N>>   Statement_N;           -- End loop body.

<<State_Z>>     goto State_Y;

<<State_Z+1>>   -- ...rest of program
```

Figure 4-2: Stage 2 Representation of a While-Loop

5. Thoughts towards a compiler

The method just presented informally emulates a multi-pass compiler that accepts as input a Stage 1 Ada program (i.e., a "normal" program confined only by restrictions we may choose to impose on the use of Ada) and produces a Stage 2 program, which is also legal, though "stylized" Ada code. This method is "compiler-like" in the sense that it is syntax driven and in that the transformations are viewed as production rules.

The Stage 1 to Stage 2 transformation involves several passes over a program unit. Backtracking within a given pass is sometimes necessary. For instance, a pass may begin by scanning the program unit and declaring the instantiation of all generic package objects that can be determined at that time, and may end with the declaration of more package objects that have been determined to be necessary while scanning the code. The passes can be organized as follows:

- Pass 1 - Transforms the declaration part of the program unit and the simple statements. Declares and instantiates packages that correspond to formal parameters and inserts code to write the actual parameter values into these packages.

- Pass 2/Part A - Transforms compound statements, that is, loops, if statements, accept statements and blocks. (Simple statements "exposed" in this step are also transformed.) Records situations that require backtracking. Also records situations that require new packages to be instantiated.

- Pass 2/Part B - Backtracks and replaces "temporary" state markers with appropriate state numbers.

- Pass 3 - Instantiates new packages whose need has been previously recorded. Transforms expressions that involve relational operators and expressions that similarly involve an increase in the number of states.

5.1. Determining concurrency within a state

Determining which actions may take place in parallel is an important part of the methodology. Reasoning can be applied to specific cases based on the function, procedure, and procedurE specifications. However, a general rule is desirable. The following principles (constraints) are adhered to:

1. At the Stage 2 level no two operations of a given package instance may be called within a given state. This applies both to multiple calls on a single subprogram contained in a generic package instance and to single calls on different subprograms of the same package. Thus, the calls

```
Point.Load;
Point.Test;
```

must be invoked in separate states, whereas

```
Point.Load;
Slot.Test;
```

or

```
Point.Load;
Slot.Load;
```

may be initiated concurrently.

2. After receiving an appropriate "Go" signal, a module M (executing a type procedurE operation) will not recognize another "Go" signal sent from a module N until after M raises the matching "I'm done" signal. If a module N were to send such a signal, its "Go" signal will be

ignored and the action that N requests of M would never take place. Furthermore, N runs the risk of mistakenly viewing the "I'm done" signal M sends upon completion of the previous operation as intended for N and will therefore proceed in error.

3. The hardware modules developed in this report have no underlying storage resource management: they allow for only one "activation record" at a given time. Thus, overlapping invocations will result in undefined behavior.

The rule is sufficient for our purposes to ensure proper behavior but no claim is made that it is always necessary. (Note that Ada semantics permit concurrent activations of operations within a package, although such permissiveness can lead to non-deterministic behavior.) The fact that a unique module is created in hardware for every variable, every computation (e.g., addition), and every comparison, suggests that control line conflicts will be avoided as long as no module is presented with more than one command at a time.

6. Stage 3: Protocol-definition Ada program

An Ada task defines a distinct thread of control. Ordinary subprogram calls by a task T are regarded as traversals along this thread of control. Since contention for subprogram activation has been eliminated by the constraints we have imposed, Go/I'm done protocols can be used safely in such cases. Inter-task communication is more complex since two separate threads of control are involved and since contention is possible. Such communication is, therefore, implemented with a four-cycle Request/Acknowledge protocol. Implementation details for both kinds of communication are introduced in the transformation from Stage 2 to Stage 3.

6.1. Motivation for Stage 3

Like its predecessor, the Protocol-definition stage is specified in legal Ada code. The discipline introduced in Section 3 is extended. The Protocol-definition stage realizes two goals:

1. New states are inserted and "Line" packages are instantiated to

specify protocols for communication between the program units expressed in the Stage 1 code.

Note that the transformations presented thus far have been concerned with communications within a given Stage 1 program unit. Since each of the original program units maps into a unique state machine/data path pair (engine), task entry calls, procedure calls, and function calls between these units cannot be represented by simple control line assertions. Instead, such communication must be implemented either using Request/Acknowledge or Go/I'm Done protocols.

2. State label numbers are converted to binary numbers, primarily to facilitate the encoding of the Stage 3 body part as an SLA state machine, which takes place in Stage 4.

In the transformation to Stage 3, the list of declared hardware modules is completed and the state machine is reduced to a sequence of if-statements, goto statements, and subprogram calls representing control line assertions.

6.2. Implementing Inter-Program Unit Communications Protocols

Stage 3 inserts protocols only for those program units that are originally specified in Stage 1. Protocols are already defined (in Stage 2) for program units that are introduced as a result of building block generic package instantiations.

In hardware representation each inter-engine communication requires two communications lines. Each line (i.e., wire) is realized by the instantiation of the generic package named "Line". The specification part for Line is:

```
generic
package Line is
  procedure Lift;
    -- Function:
    --   Assigns the logical value 1.
  procedure Lower;
    -- Function:
    --   Assigns the logical value 0.
  function Test return boolean;
    -- Function:
    --   Returns true if wire has logical value 1,
    --   else returns false.
end Line;
```

An instance of this package corresponds to a physical line whose level may be lowered, raised, or tested.

6.2.1. Transforming Procedure and Function Calls

A procedure or function X is mapped from Stage 2 to Stage 3 as follows:

1. Line packages X.Go and X.Done are instantiated.
2. The decision "if X_Go.Test()" is inserted as the initial state. (The machine remains in this state until X_Go.Test becomes true. Lines are always initialized to the logical value 0, regarded here as false.)
3. "X_Done.Lift" is made the action of the final state. The state machine of X takes the necessary actions to allow the caller to "see" the return values at the same time X_Done is sensed true.

Program units that contain procedure and function calls to other program units must also be transformed to reflect the calling protocol. For example, the action:

```
<<State_1>>  X(some_arguments);     -- Call on X
             goto State_2;
```

is transformed into:

```
<<State_1>>  X_Go.Lift;
             X(some_arguments);    -- The original action.
             goto State_2;

<<State_2>>  if  X_Done.Test  then
               -- Load the out parameters/function result
               -- into proper register(s).
               goto State_3;
             else
               goto State_2;
             end if;
```

Notice that the original invocation of X is left in the code.

6.2.2. Transforming Task Entry Calls and Accept Statements

The transformation of tasks is similar to that for subprograms. The scheme outlined in the previous subsection is followed, although "X_Req" is substituted for "X_Go" and "X_Ack" is substituted for "X_Done". Additionally, a Line

package is instantiated for each entry statement of the task. This Line and the X_Req Line are "raised" concurrently by the calling task (via a calls to the respective Lift procedures). Each accept alternative in the receiving task tests the tasks request line and the corresponding entry statement line before performing the desired operation. As an example, consider the task named "Storage" that models a Read/Write memory. Storage is specified in Stage 1 as:

```
task Storage is
  entry Read(
    address: integer;
    value:   out integer);

  entry Write(
    address:  integer;
    value:    integer);
end Storage;
```

The instantiations

```
package Storage_Req   is new Line;
package Storage_Ack   is new Line;
package Storage_Read  is new Line;
package Storage_Write is new Line;
```

must be visible to Storage and all tasks which can call it.

The body of Storage is realized as:

```
<<State_0000>>  if  Storage_Read.Test() and
                       Storage_Req.Test()  then
                  goto State_0001;
                elsif Storage_Write.Test() and
                         Storage_Req.Test()  then
                  goto State_0100;
                end if;

<<State_0001>>  accept Read(
                    address: integer;
                    value:   out integer)
                do
                    -- Perform read operation.
                    -- This may take several steps
                    -- in the general case but here
                    -- we simplify to one step.
                end Read;
                goto State_0010;

<<State_0010>>  Storage_Read.Lower();
                goto State_0110;

                  .
                  .

<<State_0100>>  accept Write(
                    address: integer;
                    value:   integer);
                do
                    -- Perform write operation .
                    --
                end Write;
                goto State_0101;

<<State_0101>>  Storage_Write.Lower();
                goto State_0110;


<<State_0110>>  Storage_Ack.Lift();
                    -- Raise the acknowledge line.
                goto State_0111;

<<State_0111>>  if  Storage_Req.Test()  then
                    -- Keep Ack high until Req is lowered.
                  Storage_Ack.Lift();
                  goto State_0111;
                else
                  Storage_Ack.Lower();
                  goto State_<some_next_state>;
                end if;
                  .
                  .
```

A Stage 1 call on the Storage write operation such as

```
<<State_4>>     Storage.Write(
                    1,
                    Some_Value.Read());
                goto State_5;
```

is realized in Stage 3 as:

```
<<State_1000>>  Storage_Req.Lift();        -- Raise request line.
                Storage_Write.Lift();      -- Raise write accept line.
                Storage.Write(
                    1,
                    Some_Value.Read());
                goto State_1001;

<<State_1001>>  if  Storage_Ack.Test()  then
                  Storage_Req.Lower();     -- Test acknowledge line.
                  goto State_<some_next_state>;
                else
                  Storage_Req.Lift();
                  goto State_1001;
                end if;
```

Note that the effects of these transformations are to:

1. Force tasks to follow standard Request/Acknowledge protocol.

2. Create an implicit case statement which directs the proper accept alternative choice (e.g., State_0000 above).

6.3. Transformation to Binary Numbers

In Stage 4, states are encoded as a series of "0" and "1" cells that are connected to SR flip-flops. For example, <<State_0110>> is realized by placing "0", "1", "1", and "0" cells in the same row (AND plane) in adjoining columns a matrix called and SLA. The level associated with this row is "raised" whenever that sequence of values 0110 is stored collectively in the flip-flops. We regard raising this row's level as equivalent to being in State 0110.

To facilitate this encoding, state label numbers are transformed to binary representations as the last action of Stage 3. With the completion of the state

expansions outlined earlier in this section, the state machine is fully specified.

In summary, Stage 2 to Stage 3 transformations can be performed in two passes. The first pass inserts the necessary state and package instantiations to specify the communications protocols. The second pass converts the state label numbers to binary numbers.

7. Stage 4: SLA Program

This section discusses SLA programs and their derivation from Stage 3.

7.1. Background and Use of SLA Programs

SLA is an acronym for Storage Logic Array. SLA methodology lends itself to the realization of interacting state machine/environment pairs; they are used to describe both the state machine and the data path components. The SLA concept was originally conceived by S. Patil [15] [14], extended by Patil and Welch [12] [13], and further extended by K. Smith [18]. Simply put, SLAs are "folded" Programmable Logic Arrays (PLAs) in which column and row breaks in both the AND and OR planes allow the design of independent arrays in the same circuit. "Programming" an SLA involves the placement of symbolic elements (with the help of an editor) in a manner that may result in representing an arbitrary number of independent finite state machines whose interconnection is specified by the SLA program. These symbolic elements may then be automatically translated into IC layout masks in the appropriate circuit technology. The translation of the SLA program into an integrated circuit can be viewed as the actual placement of finite SLA machines onto the active area of the chip. SLA programs make it easy for the designer to visualize the physical layout of the circuit from its logical description. A designer who thinks primarily in terms of the functional description effectively specifies the physical layout as well. Smith and co-workers have designed SLAs in I^2L, NMOS, and CMOS technologies [18]. More recent work by Smith's group has extended the SLAs based on a new

concept for cell set design. The new circuits, called PPLs, are being primarily applied in the design of asynchronous state machines [4].

Our method uses SLAs in two ways:

1. The SLA modules previously developed are treated as hardware components that replace the Stage 3 generic packages. Note that no formal method is employed for the design of the SLA modules. However, each module has been simulated independently to test its correctness.

2. The state machines, including control and feedback lines, are encoded as SLAs [13].

We use SLA cells to build a library of composite "macros", which are the Stage 4 modules described in Section 3. These modules comprise the data path and are inserted using a cell substitution approach. In this sense our use of SLAs is similar to the use of macro cells [3] and Associative Logic [7].

The particular cell set employed in this work was the 5 micron NMOS set described in [17]. An SLA editor (SCLED [20]) and a SLA simulator (NSIM [19]) were built and tested at Utah; both were used extensively in this study.

7.2. Encoding of State Machines

The Stage 3 specification of a state, say, State 0110, results in the connection of the appropriate SLA cells such that the row corresponding to State 0110 goes high at the proper time. Further, in each state the levels on columns "connected" to the row of a given state are raised when the SLA is in that state. These columns are the sources of the control lines, which correspond to the operations to be initiated in that state. A two-pass method is employed to accomplish the desired encoding. This technique is presented by referring to a simple example. Consider the Stage 1 if-statement construct:

```
if  A = B  then
  C := C + 1;
else
  A := B + 1;
end if;
```

With the assumptions that "A" maps into a Register while "B" and "C" map into Counters, this construct could be specified in Stage 3 as:

```
<<State_0000>> Equals.Go(A.Read, B.Lookup, equals_result);
               goto State_0001;

<<State_0001>> if  equals_result  then
                 goto State_0010;
               else
                 goto State_0011;
               end if;

<<State_0010>> C.Increment;
               goto State_0110;

<<State_0011>> B.Increment;
               goto State_0100;

<<State_0100>> A.Write(B.Lookup);
               goto State_0101;

<<State_0101>> B.Decrement;
               goto State_0110;

<<State_0110>> null;
```

In the first pass, the states of Stage 3 are scanned sequentially. Every function and procedure call on a generic package instantiation in Stage 3 is transformed into the raising of a control line when the row corresponding to the given state "goes high". If-statements are transformed into two rows, one for each possible result of the if. The state machine layout rules employed are:

1. For simplicity, columns representing test inputs and control line outputs that are used to communicate with other state machines (program units) are placed on the left of the state machine and those that communicate to local modules are placed on the right.

2. Rows and columns are annexed as needed as the Stage 3 states are scanned. When a new Stage 3 subprogram call is discovered, a column is designated to carry the corresponding control line.

Figure 7-1 presents the result of the initial encoding pass over the Stage 3 code presented above.

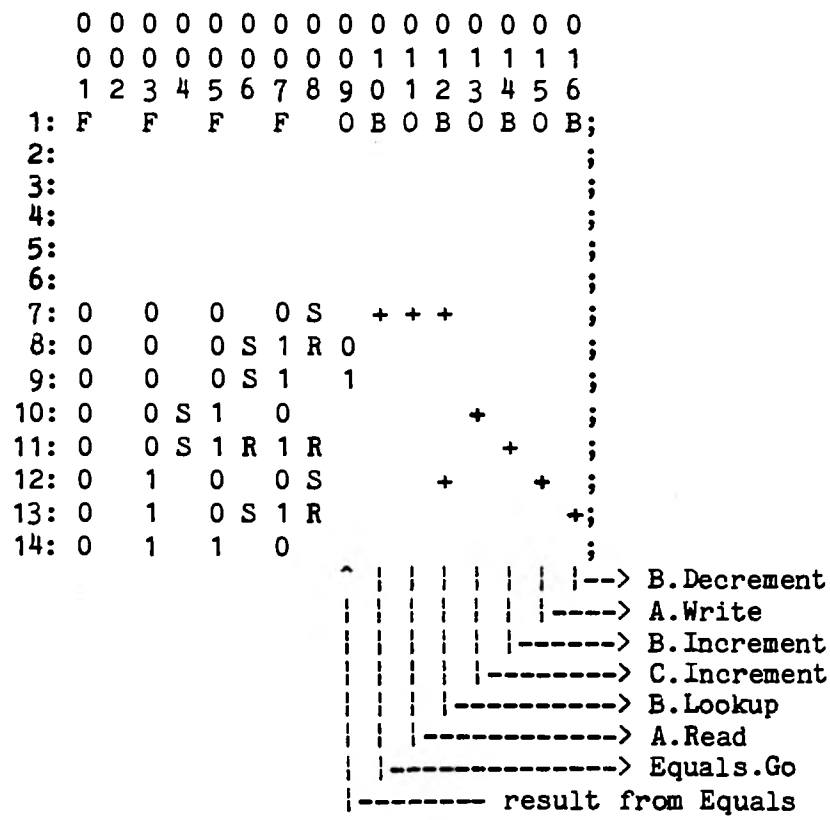

Figure 7-1: First Pass Stage 4 Encoding

Note how state 0000 (row 7) raises columns 10, 11, and 12. This row corresponds to the "Equals.Go(A.Read, B.Lookup,...)" operations specified for state 0000 in the Stage 3 code above. State 0001 (rows 8 and 9) corresponds to the if-statement. Row 8 "goes high" if the result from the comparator carried in column 9 is false (i.e. a /= b). Row 9 goes high if the result is true (a = b). Note how new columns are added on the right as new procedure and function calls are scanned in the Stage 3 code. Note also how the B.Lookup (column 12) is raised in State 0000 (row 7) and in State 0100 (row 12). The second time "B.Lookup" is scanned in the Stage 3 code we remember that a column was already dedicated to this control line; we don't dedicate another. Since this simple circuit does not communicate with other state machines, all control line firings are on the right side.

In the first pass the "+", "1", and "0" cells are placed only as the need for them is discovered. A dispersed layout often results. The second manual pass re-arranges the control lines to group lines that are directed to the same module. Thus, the second pass merely clusters the control lines, arranging them according to their destination. The effect of the second pass is to simplify routing of the control lines to the modules. Figure 7-2 presents the result of re-arranging of the columns of Figure 7-1. Note how commands going to the same module are now on adjacent columns.

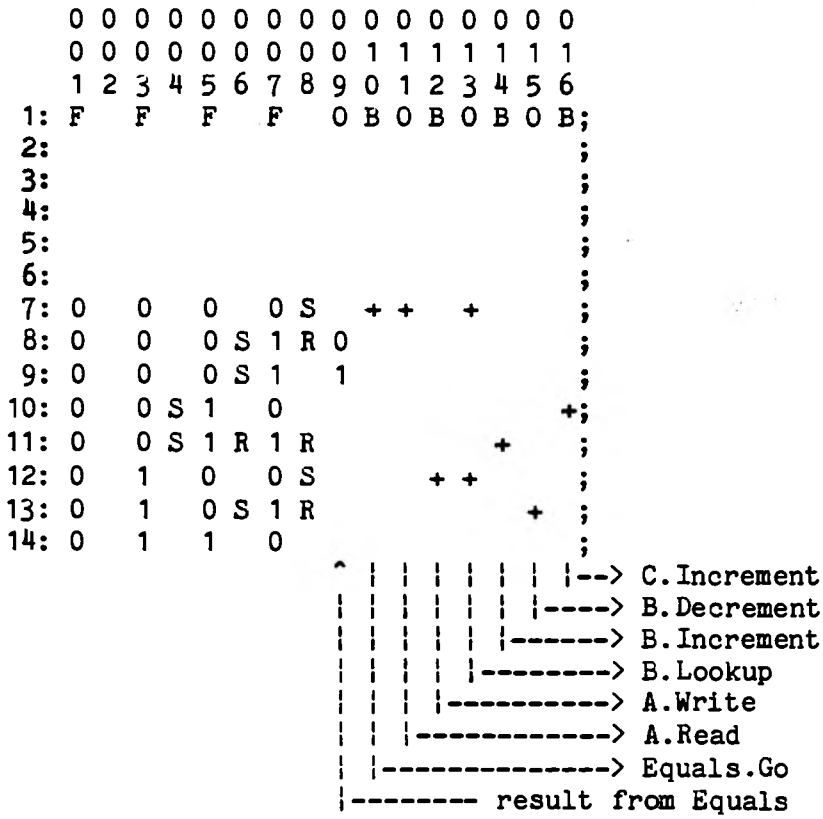

Figure 7-2: Second Pass Stage 4 Encoding

7.3. Layout, Routing and Busing Issues

An algorithmic method for cell layout and routing has not yet been incorporated into our method. Reference [6] discusses a simple manual routing method that utilizes the fact that the declaration part of a given Stage 3 program unit specifies the modules utilized by that unit.

As mentioned earlier, engines that are physical representations of tasks communicate through the use of the Request/Acknowledge protocol. In the hardware realm, such engines communicate via buses. A circuit derived by our method may include several buses, which may be private (non-contention) or public (with potential for contention between the users). Both types support the Request/Acknowledge protocol. It is well-known that a Request/Acknowledge protocol strategy will not work on a contention bus without some sort of arbitration mechanism. The Request/Acknowledge protocol implemented here closely follows the scheme outlined by Seitz [16], and appears to be adaptable to his arbitration scheme. Bus issues are detailed further in [6].

8. Conclusions

The transformation methodology described in the preceeding sections was developed and exercised in conjunction with an extensive and non-trivial case study [6]. The algorithm developed for that exercise is a possible model for the behavior of the Ada selective wait statement, itself initially specified as an Ada program consisting of a set of intercommunicating Ada server and requestor tasks. The transformation rules were only applied to a subset of the program. Application of the rules resulted in two SLA programs whose behavior was tested with the simulator NSIM.

The case study [6] provided a "real" example of rule-based transformations which covers the significant portion of the Ada-to-Silicon "spectrum". No theoretical stumbling blocks were encountered in this process, which suggests that there is nothing in principle to invalidate the concept that such transformations may be automated. On the other hand, we have not yet formalized these transformation rules as concrete algorithms. There is the additional challenge of reaching practical and competitive circuits with this approach.

We have experimented the intriguing concept of using Ada itself as an intermediate language in the mapping process. For this purpose we have found

important ways to exploit Ada's abstraction features:

1. In mapping Ada program variables to instantiations of generic packages to pre-defined IC modules.

2. In mapping Ada subprogram and task calls to specific hardware protocols.

The end result of successful research in this area can be that the traditional hardware logic design activity will become increasingly a programming activity that is keyed to the use of high-order programming languages for system specification. Such an evolution will progress, however, only as rapidly as we succeed in evolving a new class of high-quality compilers for hardware.

REFERENCES

1. Aho, A., and Ullman, J., Principles of Compiler Design, Addison-Wesley, Reading, Mass., 1977.

2. Barnes, J., Programming in Ada, Addison-Wesley Publishers Ltd., 53 Bedford Square, London, WC1B 3DZ, International Computer Science Series, 1982.

3. Carey, J. and Blood, B., "Macrocell Arrays-An Alternative to Custom LSI," Proceedings Semi-Custom Integrated Circuit Technology Symposium, Institute for Defense Analysis, Science and Technology Division, May 1981, pp. 19-37.

4. Carter, T., "ASSASSIN: An Assembly, Specification and Analysis System for Speed-Independent Control Unit Design in Integrated Circuits Using Path Programmable Logic (PPL)," Master's thesis, University of Utah Computer Science Dept., June 1982.

5. U.S. Department of Defense, Military Standard ADA Programming Language, U.S. Department Of Defense, Washington D.C., 1980.

6. Drenan, L., "On Transforming Ada to Silicon," Master's thesis, University of Utah Computer Science Dept., August 1982.

7. Greer, D., "An Associative Logic Matrix," IEEE Journal of Solid State Circuits, Vol. SC-11, October 1976, pp. 679-691.

8. Organick, E., "Programmer's Introduction to Hardware Design", unpublished course notes used at the University of Utah

9. Organick, E., " Semiannual Technical Report: Transformation of Ada Programs Into Silicon," Tech. report UTEC-82-020, University of Utah Computer Science Dept., March 1982, DARPA Order No. 4305.

10. Organick, E.; Boll, S.; Davis, A.; Griss, M.; Hayes, A.; Hollar, L.; Huber, R.; Lindstrom, G.; Rushforth, C.; Smith, K.; and Subrahmanyam, P., "Transformations of Ada Programs into Silicon : A Research Proposal to Defense Advanced Research Projects Agency," University of Utah, March 1981.

11. Organick, E., and Lindstrom, G., "Mapping High-Order Language Units Into VLSI Structures," Proc. COMPCON 82, IEEE, Feb. 1982, pp. 15-18.

12. Patil, S. and Welch, T., "A Programmable Logic Approach for VLSI," IEEETrans, Vol. C-28, Sept 1979, pp. 594-601.

13. Patil, S., "On Testability of Digital Systems Designed with Storage/Logic Arrays," IEEE International Conference on Circuits and Computers, 1980, IEEE, New York, 1980.

14. Patil, S., "Micro-control for Parallel Asynchronous Computers," 1975 Proceedings Euromicro, Euromicro, 1975, North-Holland Publishing Company.

15. Patil, S., "An Asynchronous Logic Array," Tech. report TM-62, MIT, May 1975, Project Mac.

16. Seitz, C., "Ideas About Arbiters ," LAMBDA, Vol. 1, No. 1, First Quarter 1980, pp. 10-14.

17. Smith, K., "Design of Integrated Circuits with Structured Logic Using the Storage Logic Array (SLA) Definition and Implemantation," PhD dissertation, University of Utah, March 1982.

18. Smith, K.; Carter, T.; and Carter, T., "Structured Logic Design of Integrated Circuits Using the Storage/Logic Array (SLA)," IEEE Transactions on Electron Devices, Vol. ED-29, No. 4, April 1982, pp. 765-776.

19. Nelson, B., NSIM User's Manual: University of Utah VLSI Research Group, 1981.

20. Nelson, B., SLED User's Manual: University of Utah VLSI Research Group, 1981.

Table of Contents

List of Figures