A Synthesis-for-Test Design System

LaNae J. Avra, Laurent Gerbaux, Jean-Charles Giomi,
Francoise Martinolle, Edward J. McCluskey

94-3

(CSL TR 94-622)

May 1994

Center for Reliable Computing
ERL 460
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract:

Hardware synthesis techniques automatically generate a structural hardware implementation given an abstract (e.g., functional, behavioral, register transfer) description of the behavior of the design. Existing hardware synthesis systems typically use cost and performance as the main criteria for selecting the best hardware implementation, and seldom even consider test issues during the synthesis process. We have developed and implemented a computer-aided design tool whose primary objective is to generate the lowest-cost, highest-performance hardware implementation that also meets specified testability requirements. By considering testability during the synthesis process, the tool is able to generate designs that are optimized for specific test techniques. The input to the tool is a behavioral VHDL specification that consists of high-level software language constructs such as conditional statements, assignment statements, and loops, and the output is a structural VHDL description of the design. Implemented synthesis procedures include compiler optimizations, inter-process analysis, high-level synthesis operations (scheduling, allocation, and binding) and control logic generation. The purpose of our design tool is to serve as a platform for experimentation with existing and future synthesis-for-test techniques, and it can currently generate designs optimized for both parallel and circular built-in self-test architectures.

Funding:

This work was supported in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and administered through the Office of Naval Research under Contract No. N00014-92-J-1782, and by the National Science Foundation under Grant No. MIP-9107760.

A Synthesis-for-Test Design System

LaNae J. Avra and Edward J. McCluskey
CRC Technical Report No. 94-3
(CSL TR 94-622)
May 1994

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA, USA 94305-4055

ABSTRACT

Hardware synthesis techniques automatically generate a structural hardware implementation given an abstract (e.g., functional, behavioral, register transfer) description of the behavior of the design. Existing hardware synthesis systems typically use cost and performance as the main criteria for selecting the best hardware implementation, and seldom even consider test issues during the synthesis process. We have developed and implemented a computer-aided design tool whose primary objective is to generate the lowest-cost, highest-performance hardware implementation that also meets specified testability requirements. By considering testability during the synthesis process, the tool is able to generate designs that are optimized for specific test techniques. The input to the tool is a behavioral VHDL specification that consists of high-level software language constructs such as conditional statements, assignment statements, and loops, and the output is a structural VHDL description of the design. Implemented synthesis procedures include compiler optimizations, inter-process analysis, high-level synthesis operations (scheduling, allocation, and binding) and control logic generation. The purpose of our design tool is to serve as a platform for experimentation with existing and future synthesis-for-test techniques, and it can currently generate designs optimized for both parallel and circular built-in self-test architectures.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

1 INTRODUCTION

Hardware synthesis techniques automatically generate a structural hardware implementation given an abstract (e.g., functional, behavioral, register transfer) description of the behavior of the design. A *behavioral description* is specified in terms of high-level software language constructs such as conditional statements, assignment statements, and loops. The behavioral description need not contain any information that implies how it should be implemented in hardware, such as the specification of clock signals or the assignment of variables to registers. Synthesis techniques that operate on behavioral descriptions either make certain assumptions about the characteristics of the output hardware description (e.g., synchronous vs. asynchronous, minimum cost vs. maximum performance, specific technology) or allow the user to specify those characteristics.

Many different hardware designs can implement a given behavioral description, a subset of which also meet specified requirements such as cost, performance, and testability. Existing hardware synthesis systems (see [Brayton 87], [McFarland 88], [De Micheli 94]) typically use cost and performance as the main criteria for selecting the best hardware implementation, and seldom even consider test issues during the synthesis process. We have developed and implemented a computer-aided design tool, named Odin, whose primary objective is to generate the lowest-cost, highest-performance hardware implementation that also meets specified testability requirements. By considering testability during the synthesis process, Odin is able to generate designs that are optimized for specific test techniques. The purpose of our design tool is to serve as a platform for experimentation with existing and future synthesis-for-test techniques, and it can currently generate designs optimized for both parallel and circular built-in self-test (BIST) architectures.

In this report, we describe the current capabilities of Odin. Section 1.1 introduces the VHDL language hierarchy and describes some of the VHDL statements that are typically used in behavioral and structural hardware descriptions. Section 1.2 is a brief overview of the Odin design tool. The input to the tool is a behavioral VHDL specification that consists of high-level software language constructs such as conditional statements, assignment statements, and loops, and the output is a structural VHDL description of the design. Section 2 describes the VHDL input and output files for the design system: the input behavioral description, the component library that Odin accesses when performing synthesis operations, and the output structural description. Section 3 contains a description of the pre-synthesis procedures that are applied to the input behavioral description: transformations that allow for simpler internal data structures, generation of the internal data structure, the data and control flow graph (DCFG), inter-process analysis, and compiler optimizations. Section 4 describes the synthesis techniques: scheduling, allocation, binding, and control logic generation. These techniques access area, delay, and control information from a VHDL component library that contains specifications of the hardware components that can be used in the generated design.

1.1 VHDL DESCRIPTIONS

Table 1-1 illustrates the VHDL hardware specification hierarchy for a 2-to-1 multiplexer. At the top level of the hierarchy is the *entity* statement, which defines the input and output signals of the specification. Associated with each entity are one or more *architectures*, each representing a different description of the same design. Table 1-1 illustrates three types of architectures that are used in our synthesis tool: behavioral, register transfer level (RTL), and structural. The behavioral architecture description contains algorithmic statements such as conditional statements, assignment statements, and loops, that functionally describe the behavior of the design. These algorithmic statements are included within one or more *process* statements. The statements within a process statement describe sequential behavior, and the process statement represents behavior that operates concurrently with the behavior of other process statements in the architecture. The RTL architecture consists of high-level hardware constructs such as Boolean equations and memory elements. The structural architecture, similar to a netlist description, consists of components and their interconnections. Each component can also have an associated entity and architecture description. The behavioral architecture is the input to high-level synthesis; the RTL architecture, containing the same functionality but in a different format, is the output of high-level synthesis and the input to logic synthesis; and the structural architecture is the output of logic synthesis.

The sequential statements within the process statement make up the algorithmic description of VHDL behavioral specifications. Table 1-2 shows the two possible types of process statements: a process with a sensitivity list, and a process with one or more wait statements. Processes communicate with each other by accessing values associated with the *signals* in the architecture. A process statement is activated following a change in value of one of the signals in either its sensitivity list or the sensitivity list of one of the wait statements contained within the process. When activated, the statements within the process statement execute sequentially. The process may assign new values to signals which could then activate other processes. After either all of the statements within the process statement execute or another wait statement is encountered, the process suspends until there is another change in value of one of the signals in the sensitivity list. All activated processes within an architecture execute concurrently.

Table 1-1 VHDL hardware specification hierarchy for 2-to-1 multiplexer.

```
entity mux is
port (selA,A,B   : in    bit;
       Dout        : out  bit );
end mux;

architecture Behavioral of mux is begin
   process (A, B, selA) begin
      if (selA = '1') then
         Dout <= A;
      else
         Dout <= B;
      end if;
   end process;
end Behavioral;

architecture RTL of mux is begin
   Dout <= (A and selA) or (B and not  selA);
end RTL;

architecture Structural of mux is begin
   NOT (selA_b, selA);
   AND (t1, selA, A);
   AND (t2, selA_b, B);
   OR (Dout, t1, t2);
end Structural;
```

Table 1-2 VHDL process statements.

Types of Process Statements	Types of Sequential Statements

process (sensitivity list) begin
   [sequential statements]
end process;
	sequential statements =
	   assignment
	   procedure call

process begin
   [wait statements
      or sequential statements]
end process;
	if-then-else
	case
	while loop
	for loop

1.2 SYNTHESIS OVERVIEW

Figure 1-1 illustrates the flow of tasks that are performed by the Odin design system. The input to Odin is a VHDL intermediate format (VIF) description that represents the behavioral description of the design. A commercial VHDL compiler, developed by Vantage Analysis Systems, is used to create VIF from the input behavioral description. Several transformations are applied to the VIF description in order to simplify Odin's internal data structures. VIF access routines supplied by Vantage are used to transform the VIF description into a data and control flow graph (DCFG). Compiler optimizations are applied to the DCFG in order to minimize the amount of code that must be mapped to hardware. If the behavioral VHDL description contains multiple process statements, inter-process communication can be analyzed [Martinolle 91] in order to extract all potential functional parallelism in the description so that the most efficient hardware can be synthesized. The first task in high-level synthesis is scheduling, where operations (e.g., addition, comparison, multiplication) in the behavioral description are assigned to specific clock cycles based upon data dependencies and the delays of the hardware components that are used to implement the operations. Operations and variables are then mapped to hardware components from a user-specified VHDL component library. Control logic that generates control signals for the components in the data path logic is synthesized based upon the selected schedule and the binding of operations and variables to hardware components. The output is structural VHDL descriptions of control logic and data path logic that implement the input behavioral description.

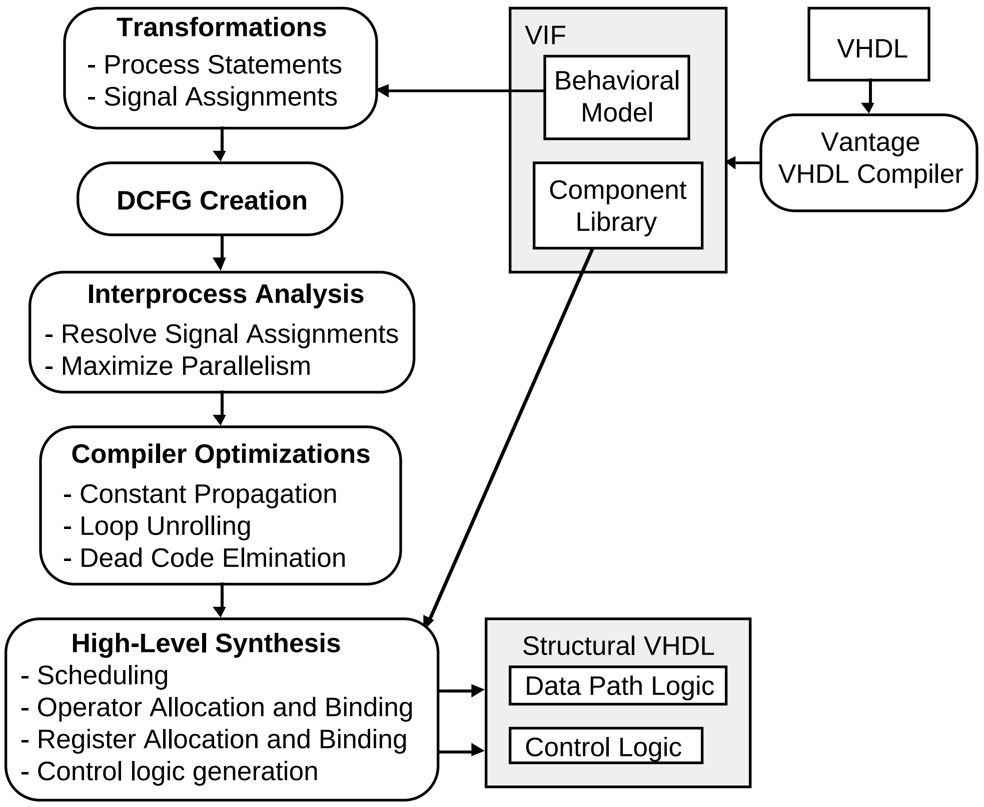


Figure 1-1 Odin design system overview.

2 DESIGN SYSTEM INPUT AND OUTPUT

As shown in Fig. 1-3, the input to Odin consists of two VIF data structures, generated by the Vantage compiler from VHDL source code. The data structures represent a behavioral model of the design and a library of hardware components used by the synthesis techniques to generate a structural implementation.

2.1 VHDL DESCRIPTIONS

The VHDL behavioral description of the design to be synthesized consists of algorithmic, high-level software constructs such as conditional statements, assignment statements, and loops. The behavioral description does not necessarily contain information, such as clock signal designations, that implies how it should be implemented in hardware. Table 2-1, a process statement from the *DiffEq* benchmark circuit description in [HLSW 92], is an example of the type of behavioral description that Odin accepts as input. A grammar specifying the subset of VHDL that Odin currently supports is given in Appendix A.

Table 2-1 Example VHDL behavioral description: *DiffEq* from [HLSW 92].

```
process (Aport, DXport, Xinport, Yinport, Uinport)
  variable x_var,y_var,u_var, a_var, dx_var: integer ;
  variable x1, y1, t1,t2,t3,t4,t5,t6: integer ;
begin
  x_var := Xinport;  y_var := Yinport; u_var := Uinport;
  a_var := Aport; dx_var := DXport;
  while (x_var < a_var) loop
    t1 := u_var * dx_var;
    t2 := 3 * x_var;
    t3 := 3 * y_var;
    t4 := t1 * t2;
    t5 := dx_var * t3;
    t6 := u_var - t4;
    u_var := t6 - t5;
    y1 := u_var * dx_var;
    y_var := y_var + y1;
    x_var := x_var + dx_var;
  end loop;
  Xoutport <= x_var;
  Youtport <= y_var;
  Uoutport <= u_var;
end process;
```

The VHDL structural descriptions of the control and data path logic generated by Odin are netlists that consist of component instantiations and signal interconnections. All components in the structural description are specified in the component library which is described in Sec. 2.2. A portion of an example VHDL structural description of data path logic generated by Odin is shown in Table 2-2. Three components (*ADD*, *SEL2*, and *SUB*) and their signal interconnections (*port map* statement) are shown. The *generic map(16)* statement specifies that the data path for that component is 16 bits, so, for example, the two data input signals, *INTERNAL14* and *INTERNAL9* for the *ADD* component are each 16 bits.

Table 2-2 Example VHDL structural description of data path logic generated by Odin.

```
architecture STRUCTURAL of DATA_1 is
use work.COMPONENT_PKG.all;
.....
G14: ADD
  generic map(16)
  port map(INTERNAL8,INTERNAL14,INTERNAL9,cIn);
G15: SEL2
  generic map(16)
  port map(INTERNAL7,SEL(13),SEL(14),INTERNAL13,INTERNAL20);
G16: SUB
  generic map(16)
  port map(INTERNAL6,INTERNAL18,INTERNAL7,cIn);
.....
end STRUCTURAL;
```

2.2 COMPONENT LIBRARY

The VHDL structural description generated by Odin consists of hardware components that are specified by the user in a VHDL component library. Odin assumes that any operation in the behavioral description can be implemented by one or more components in the component library. The library must also contain register components and multiplexer or selector components for implementing the data path logic, and basic logic gates (e.g., NAND, NOR, INVERT) and bistables for implementing the control logic. The user is able to guide the synthesis process by means of the types of components available in the library and the characteristics associated with each available component. For example, by including in the library components that have been optimized for scan architectures, the user can minimize the overhead of any scan designs that Odin generates.

Associated with each component in the component library are attributes that specify its area, delay, and control information. The area attribute is used by Odin to estimate the cost of the component, and the delay attribute is used to estimate its performance. Since a single component may implement multiple operations (e.g., ALUs, comparators), the control information attributes specify the control signal values required for the component to implement the associated operation. Components that implement operations with multiple inputs also have an attribute that specifies whether or not the operation is commutative. The inputs to a commutative component can sometimes be switched to reduce the number of interconnections in the data path logic.

3 DESIGN SYSTEM FRONT-END

In this section, we describe the operations that are applied to the input behavioral description prior to performing hardware synthesis. These operations include transformation of the VHDL description to a consistent format in order to simplify the internal data structures, generation of the data and control flow graph, inter-process analysis, and compiler optimizations

3.1 TRANSFORMATIONS

The first step in the synthesis process is to apply various transformations to the input description, the goal being to eliminate unnecessary code in the description, thereby preventing the synthesis of hardware to implement that unnecessary code.

Odin currently supports four types of concurrent VHDL statements: the concurrent procedure call, the concurrent signal assignment statement, the generate statement, and the process statement. The concurrent component instantiation statement is passed through, unmodified, to the output structural VHDL model. We assume that instantiated components in the behavioral description represent existing hardware, such as custom multipliers or memory components, that need not be synthesized. The concurrent assertion statement is not addressed by the synthesis operations because it is typically implemented as a passive operation to flag error conditions during simulation, such as unexpected signal values or unmet timing constraints.

Process statements with sensitivity lists, concurrent procedure calls, and concurrent signal assignment statements are transformed, as described in the VHDL Language Reference Manual [IEEE 88], to an equivalent process statement that contains a single wait statement as its last statement, since the wait statement version of the process statement is the more general form. These transformations are illustrated in Tables. 3-1, 3-2, 3-3, and 3-4.

Table 3-1 Process statement transformation.

Process With Sensitivity List:
```
process ([list of signals])
begin
  [sequential statements]
end process;
```
Equivalent Process:
```
process
begin
  [sequential statements]
  wait until [list of signals];
end process;
```

Table 3-2 Concurrent procedure call transformation.

```
Concurrent Procedure Call:
  mux_procedure (A, B, selA, Dout);

Equivalent Process:
  process
  begin
    mux_procedure (A, B, selA, Dout);
    wait on A, B, selA;
  end process;
```

Table 3-3 Conditional signal assignment statement transformation.

```
 Conditional Signal Assignment:
 Dout <=  A when selA='1' else
          B when selB='1' else
          C;

Equivalent Process:
 process
 begin
   if (selA='1') then
     Dout <= A;
   elsif (selB='1') then
     Dout <= B;
   else
     Dout <= C;
   end if;
   wait on A, B, C, selA, selB;
 end process;
```

Table 3-4 Selected signal assignment statement transformation.

```
 Selected Signal Assignment:
     with selA select
       Dout <=    A when '1',
                  B when '0';

Equivalent Process:
     process
     begin
       case selA is
         when '1' =>   Dout <= A;
         when '0' =>   Dout <= B;
       end case;
       wait on A, B, selA;
     end process;
```

The concurrent generate statement, which is a coding convenience in that it uses either a loop or conditional statement format to describe multiple concurrent statements with a single statement, is transformed by mapping the generated concurrent statements into equivalent process statements as previously described. The generated process statements must also include sequential control statements that are equivalent to the control constructs used in the generate statement. This is illustrated in Table 3-5.

The next transformation that Odin performs is to move all sequential signal assignment statements to the location in the process statement that is just prior to the next wait statement in the code. The reason for this is that this new location represents the accurate location for a sequentially-executed signal assignment statement. When a signal is assigned a value within a process statement, the signal doesn't actually receive the new value until after the process suspends operation due to a wait statement. Thus, any uses of a signal in a sequential statement that appears after an assignment to that signal actually use the old value of the signal rather than the newly-assigned value. After applying the signal assignment transformation, the DCFG can be easily extracted from VIF, and standard data flow analysis techniques, such as those described in [Aho 86], can be used to perform the compiler optimizations described in Sec. 3.4. The sequential signal assignment transformation is illustrated in Table 3-6.

Table 3-5 Generate statement transformation.

Concurrent Generate Statement:

```
for i in 0 to 15 generate
  Dout(i) <= (selA and A(i)) or (not selA and B(i));
end generate;
```

Equivalent Process Statement:

```
process
  variable i:   integer;
begin
  for i in 0 to 15 loop
    Dout(i) <= (selA and A(i)) or (not selA and B(i));
  end loop;
  wait on A, B, selA;
end process;
```

Table 3-6 Sequential signal assignment transformation.

Before:
```
process (clk)
begin
  s1 <= (not EN) or D;
  if (clk = ‘1’) then
    out <= s1;        -- old value of s1
  end if;
end process;
```

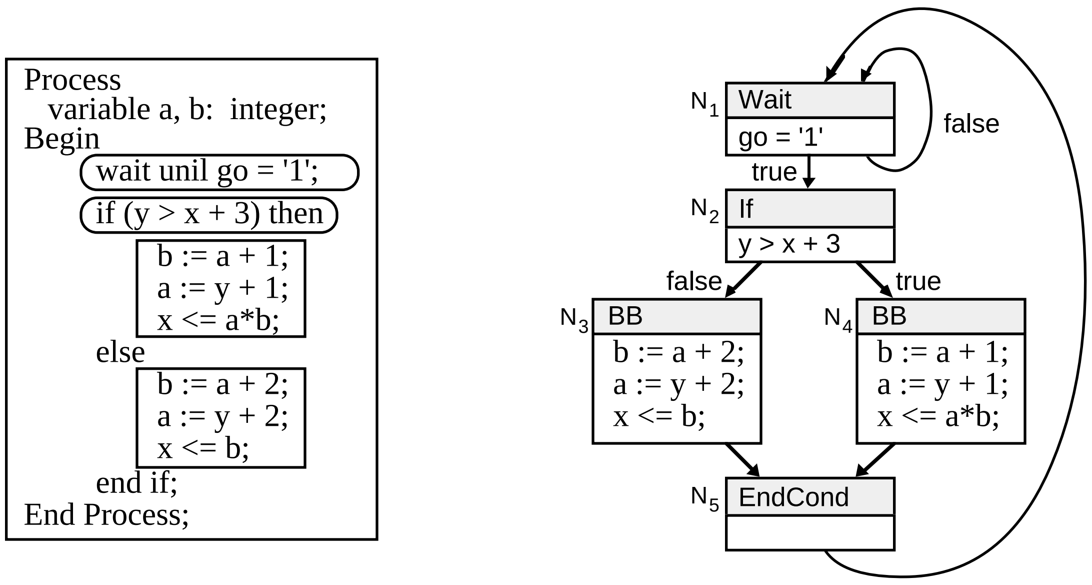
After:
```
process (clk)
  variable var_s1 : [same type as s1];
  variable var_out : [same type as out];
begin
  var_s1 := s1;
  var_out := out;
  var_s1 := (not EN) or D;
  if (clk = ‘1’) then
    var_out := s1;
  end if;
  s1 <= var_s1;
  out <= var_out;
end process;
```

3.2 DATA AND CONTROL FLOW GRAPH GENERATION

The data and control flow graph (DCFG) is generated from the VHDL intermediate format after the transformations described in Sec. 3.1 have been applied. The DCFG provides an accurate and concise means of conveying the information flow of the behavioral description to the inter-process analysis, compiler optimization, and hardware synthesis operations. The DCFG has two levels of hierarchy. The top level is created by partitioning the behavioral description into basic blocks and control statements. A *basic block* is a sequence of consecutive assignment statements from the behavioral description. Figure 3-1a shows an example VHDL behavioral description that is partitioned into two control statements (circled), a *wait* and an *if*, and two basic blocks.

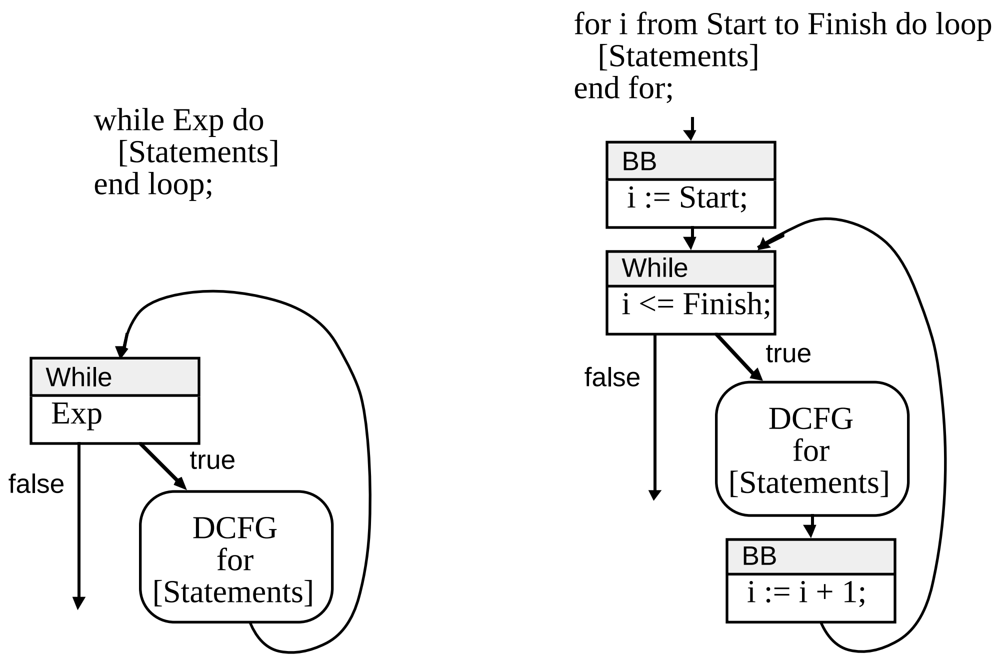
The DCFG has six different types of nodes: *Wait*, *If*, and Case, which correspond to wait, if, and case statements, respectively, *While* for loop statements, *EndCond* for joining the mutually-exclusive branches of conditional statements, and *BB* for basic blocks. Figure 3-1b shows the top level of the DCFG for the process statement given in Fig. 3-1a and illustrates how *Wait* (node N_1), *If* (node N_2), *BB* (nodes N_3 and N_4), and *EndCond* (node N_5) nodes are implemented in the DCFG. Edges in the DCFG represent the branching of operation between the behavioral statements represented by the nodes. A value associated with an edge represents the condition under which that branch is taken. For example, in the behavioral description, execution is suspended at a wait statement until the wait condition becomes true. This behavior is specified in the DCFG by an edge, labeled *False*, from a *Wait* node to itself, and by an edge, labeled *True*, from the *Wait* node to the node representing the next statement in the behavioral description.

The *While* node for the *while* loop statement is illustrated in Fig. 3-2a. The *for* loop statement is transformed into an equivalent *while* loop statement, which is implemented in the DCFG as shown in Fig. 3-2b. If it is possible to unroll the *for* loop statement as described in Sec. 3.4, it is implemented in the DCFG with a *BB* node.



```
Process
  variable a, b:  integer;
Begin
      wait unil go = '1';
      if (y > x + 3) then
            b := a + 1;
            a := y + 1;
            x <= a*b;
      else
            b := a + 2;
            a := y + 2;
            x <= b;
      end if;
End Process;
```

Figure 3-1 DCFG example: (a) partitioned behavioral description; (b) top level of the DCFG.



(a) **(b)**

Figure 3-2 *While* node implementation for loops: (a) while loop; (b) for loop.

A data flow graph (DFG) is associated with each node in the DCFG except for *EndCond* nodes. The data flow graph is a directed, acyclic graph that specifies the operations and the data dependencies of a basic block or an expression. Each node in the DFG represents an operation in the behavioral description. A directed edge from node N_i to node N_j specifies that the output of the operation represented by node N_i is an input to the operation represented by node N_j. Each DFG has a source node (*SRC*) which only has outgoing edges that represent variables that are outputs of other DFGs, and a sink node (*SNK*) which only has incoming edges that represent variables that are inputs of other DFGs. The DFG for basic block nodes specifies how the variables defined by the statements in the basic block (a variable is *defined* if it appears on the left-hand-side of an assignment statement) are used in subsequent statements of the block (see Fig. 3-3a). The DFG for control nodes (*Wait*, *For*, *While*, *If*, *Case*) represents the conditional expression associated with the control statement. For example, the DFG associated with an *If* node represents the conditional expression of the *If* statement (see Fig. 3-3b).

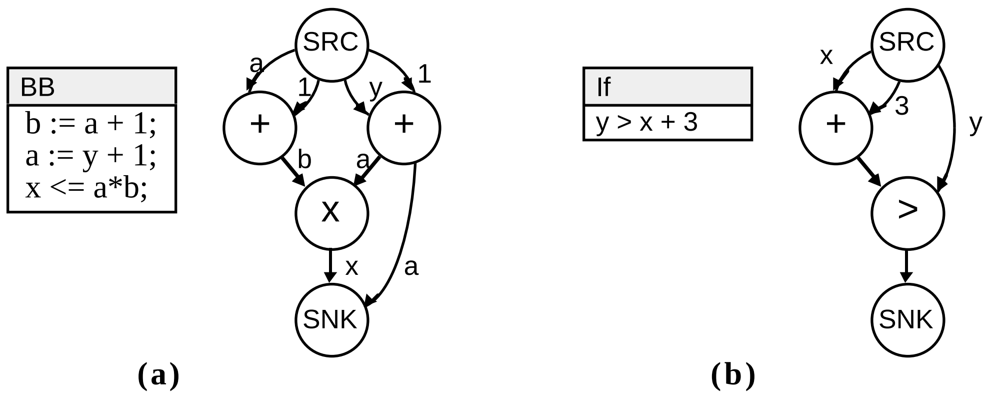


Figure 3-3 Data flow graphs: (a) *BB* node DFG; (b) *If* node DFG.

3.3 INTER-PROCESS ANALYSIS

Many existing high-level synthesis techniques operate on a data flow description extracted from an algorithmic specification of the system to be synthesized. In order to handle very complex behavioral descriptions, designers often decompose the description into several interconnected and concurrently-executing processes, where a sequential, algorithmic description is contained within each individual process. The synthesis tools then perform synthesis separately on each process, ignoring possible optimizations across process boundaries. This is not an issue for synthesis systems for which the input behavioral description language is a procedural language such as C or Pascal, since these languages do not support the specification of concurrent operation. It is, however, a significant compromise when the input description language is VHDL, since one of the major advantages of VHDL as a hardware description language is its ability to specify the concurrent execution of multiple, interacting processes.

The purpose of inter-process analysis, mentioned in [Avra 90] and described in detail in [Martinolle 91], is to analyze the communication mechanisms between processes and to merge interacting processes into a single process. This analysis is necessary in order to extract all potential, functional parallelism in the description so that the most efficient hardware can be synthesized. The result of inter-process analysis is a set of processes that operate independently of each other. This allows high-level synthesis to perform global rather than local scheduling, allocation, and binding since the execution of interacting processes is integrated into a single process.

Inter-process analysis extracts the information relative to inter-process communication (signal assignments, sensitivity lists, and conditions on signal assignments) from the DCFG of each process to determine which processes should be merged. The merging operation, called *fusion*, uses this information to determine the global function of all interacting processes and to merge these processes into a single composition process. There are two types of fusion operations: parallel composition and sequential composition. *Parallel composition* is applied to two wait blocks if at least one signal is common to both wait block sensitivity lists, since an event on that signal will cause both wait blocks to execute concurrently. Parallel composition is also applied to all wait blocks sensitive to signals that are assigned values under a single condition. *Sequential composition* is applied to two wait blocks if a signal that is assigned a value in one wait block is included in the sensitivity list of the other. A *wait block* is defined as the group of sequential statements that are within a *Wait* node loop in the DCFG. The criteria for performing fusion is summarized in Table 3-7. Detailed descriptions of the fusion algorithms and comparisons of designs synthesized with and without fusion can be found in [Martinolle 91].

Table 3-7 Criteria for fusing multiple processes into a single process.

Definitions:

Wi	wait block i
SLi	sensitivity list of Wi
ALi	the list of the signals assigned within Wi
si	signal name
->	sequential composition
//	parallel composition
Ci(sj)	condition in Wi under which sj is assigned a value

Sequential composition:

Wj -> Wi is applied iff SLi $\cap$ ALj $\neq \emptyset$

Parallel composition:

Wi // Wj is applied under either of the following conditions:

1. SLi $\cap$ SLj $\neq \emptyset$
2. (si $\in$ ALk $\cap$ SLi) & (sj $\in$ ALk $\cap$ SLj) & (Ck(si) = Ck(sj))

3.4 COMPILER OPTIMIZATIONS

Odin uses the DCFG to perform several basic compiler optimizations on the input behavioral description in order to minimize the amount of code that must be mapped to hardware and to improve the performance of the final design. The first optimization performed is constant propagation, where each use of a constant is replaced with its value. This simplifies expressions in the behavioral description and reduces the cost of the hardware that implements those expressions. For example, the expression *x=y* can be implemented in hardware with *n* exclusive-NOR gates and one *n*-input AND gate, where *x* is an *n*-bit variable, and *y* is an *n*-bit constant with value 17. When constant propagation is applied, the resulting expression, *x=17*, can be implemented with just one *n*-input AND gate.

Next, Odin analyzes the flow of data in the DFGs in order to identify dead code (statements that can be removed without affecting the behavior of the description) that can be eliminated from the behavioral description. Since dead code does not contribute to the behavior of the design, Odin removes it before synthesis so that it is not mapped to hardware. Dead code may result from previously-applied compiler optimizations, such as constant propagation, or may be inadvertently introduced by the designer.

The final optimization applied to the DCFG is to unroll loops with definite limits, where the limits do not exceed a specified maximum. The statements within a loop are executed sequentially for each iteration of the loop. When the data dependencies between the statements within the loop are such that they can be executed in parallel, unrolling the loop reduces the number of clock cycles required to execute the loop statement.

4 HIGH-LEVEL SYNTHESIS FOR TESTABLE DESIGNS

This section describes the high-level synthesis operations that are implemented in the Odin synthesis system.

4.1 SCHEDULING AND OPERATION BINDING

The first high-level synthesis task that Odin performs is scheduling, where the operations in each DFG are assigned to execute in specific clock cycles. We have implemented Paulin's forced-directed scheduling algorithm [Paulin 89] which attempts to minimize the area of the data path logic by evenly distributing the number of operations executed in each clock cycle without increasing the total number of clock cycles for the DFG. The scheduling algorithm uses the delay and area attributes of the components in the component library to perform this operation. After the schedule has been defined, operations in the DFG are bound to specific function blocks by a greedy algorithm that simply binds each operation in a clock cycle to the first available function block that performs that operation. For example, for the first clock cycle in Fig. 4-1a, Odin first binds one of the addition operations to the *ADD1* function block, then binds the second addition operation to the *ADD2* function block. We intend to investigate more sophisticated operation binding techniques that consider parallel BIST implementation requirements.

4.2 REGISTER BINDING FOR PARALLEL BIST

The register allocation and binding algorithms are implemented as described in [Avra 91], where the number of self-adjacent registers is minimized when a parallel BIST architecture is specified by the user. We have also implemented a register allocation and binding algorithm that allows self-adjacent registers in the data path logic, the only difference being that testability conflict edges are not added to the register conflict graph. Having both algorithms allows us to compare parallel BIST implementations with implementations that have been optimized without considering testability.

4.3 MULTIPLEXER BINDING

Odin generates multiplexers for the inputs of registers and function blocks in order to accommodate the flow of data required by the scheduled, bound DFG. Multiplexer generation for register inputs is performed in a straightforward manner by using the DFG to determine the source of data for each register at each clock cycle. For example, the register in Fig. 4-1a that receives input data from variable *c* in the first clock cycle receives input data from the output of *ADD2* in the second clock cycle and from the output of *ADD1* in the third clock cycle. This register therefore requires a 3-to-1 multiplexer on its input as shown in Figs 4-1b and 4-1c. Generation of multiplexers for function block inputs is performed in the same manner except that, when the operation is commutative, Odin attempts to reduce the size of the input multiplexers by permuting the function block inputs. For example, Fig. 4-1b shows that two, 3-to-1 multiplexers are required for the inputs to *ADD1* when the left and right inputs of the addition operation in the DFG correspond to the left and right inputs of the function block in the data path logic. Since the addition operation is commutative, however, the left and right inputs to *ADD1* can be switched for one or more clock cycles. For this example, by switching the inputs for the second clock cycle, Odin generates two 2-to-1 multiplexers for the inputs of *ADD1* (Fig. 4-1c).

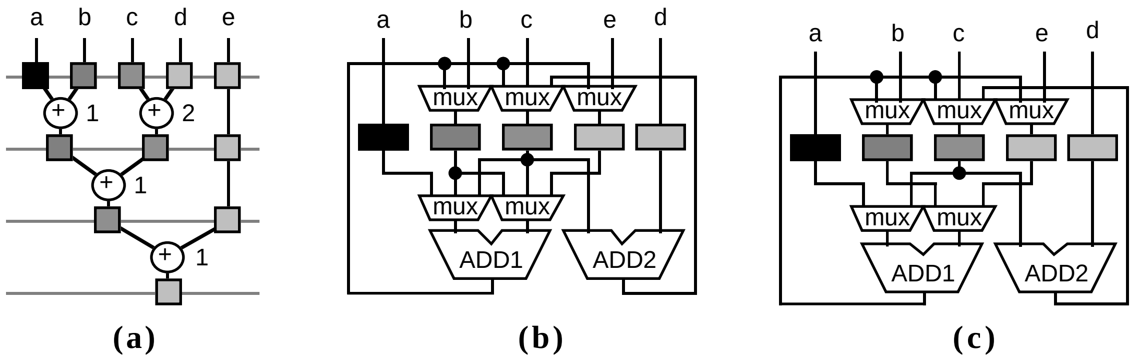


Figure 4-1 High-level synthesis example: (a) scheduled, bound data flow graph; (b) data path logic; (c) data path logic with optimized multiplexers.

4.4 CONTROL LOGIC SYNTHESIS

The last synthesis operation that Odin performs is to generate a control logic state machine that supplies register enable signals, multiplexer select signals, and function block control signals to the data path logic. Each block of data path logic corresponds to the DFG of a node in the DCFG and can be controlled by a mod-*m* counter, where *m* is the total number of clock cycles for the scheduled DFG. The DFG counters are reset, enabled, and disabled by control logic that is generated from information in the DCFG edges. Odin combines all of the counters and the DCFG control logic into a single state machine. Figure 4-2 is a possible control logic state machine description for the process statement in Fig. 3-1, where Ni_j represents clock cycle *j* of the scheduled DFG of DCFG node N_i. For example, DCFG node N_2 in Fig. 3-1b represents the *if* statement in Fig. 3-1a. Assume that the output of the *if* statement conditional expression, $y > x+3$, is generated in two clock cycles in data path logic. Then, when the control logic state machine in Fig. 4-2 is in state $N2_1$, the first clock cycle of the DFG for $y > x+3$ is executed, and when the machine is in state $N2_2$, the second clock cycle is executed. Odin generates a *KISS* format description of the control logic state machine, then uses procedures from the *SIS* logic synthesis tool [Sentovich 92] to perform state assignment and logic optimization, and to generate a circular BIST state machine implementation as described in [Avra 93].

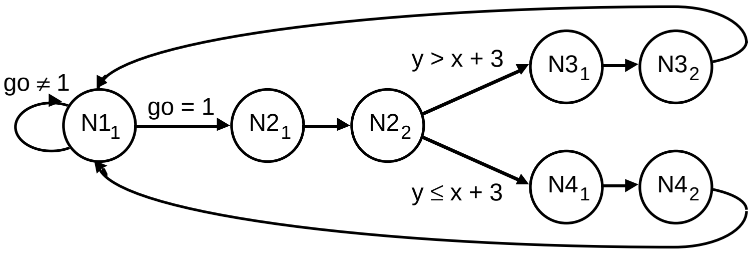


Figure 4-2 Control logic state machine description for Fig. 3-1.

5 CONCLUSIONS

This report describes Odin, a synthesis tool that automatically creates a synchronous, self-testable hardware design given a behavioral VHDL description of that design. Data path logic is synthesized for the parallel BIST architecture, and control logic is synthesized for the circular BIST architecture. We have implemented in Odin all of the major design system algorithms necessary for the generation of data path and control logic given a behavioral VHDL description: VHDL input and output file processing algorithms, internal data structure creation and manipulation algorithms, and high-level synthesis algorithms. Emphasis was placed on ease of modification when implementing the system in order to encourage the addition of future test synthesis and synthesis-for-test techniques.

ACKNOWLEDGMENTS

The authors wish to thank Hong Hao and Piero Franco for their useful comments and suggestions. The authors also wish to thank Vantage Analysis Systems, Inc. for donating the software that was used to implement Odin. This work was supported in part by Digital Equipment Corporation, Low-End Diagnostics Group, and in part by the Innovative Science and Technology Office of the Strategic Defense Initiative Organization and administered through the Office of Naval Research under Contract Nos. N00014-85-K-0600 and N00014-92-J-1782, and in part by the National Science Foundation under grant Nos. MIP-8709128 and MIP-9107760.

REFERENCES

[Aho 86] Aho, A. V., R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Reading, MA, 1986.

[Avra 90] Avra, L., and E. J. McCluskey, "Behavioral Synthesis of Testable Systems with VHDL," *COMPCON Spring 90 Digest of Papers,* San Francisco, pp. 410-415, Feb. 26-Mar. 2, 1990.

[Avra 91] Avra, L., “Allocation and Assignment in High-Level Synthesis for Self-Testable Data Paths,” *Int. Test Conf.*, Nashville, TN, USA, pp. 463-472, October 26-30, 1991.

[Avra 93] Avra, L. J., and E. J. McCluskey, “Synthesizing for Scan Dependence in Built-In Self-Testable Designs,” *Int. Test Conf.*, Baltimore, MD, USA, pp. 734-743, October 17-21, 1993.

[Brayton 87] Brayton, R. K., R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, “MIS: A Multiple-Level Logic Optimization System,” *IEEE Trans. on Comput.-Aided Des.*, Vol. CAD-6, No. 6, pp. 1062-1081, November 1987.

[De Micheli 94] De Micheli, Giovanni, “Synthesis and Optimization of Digital Circuits,” McGraw-Hill, Inc., Hightstown, NJ, USA, 1994.

[HLSW 92] 1992 High-Level Synthesis Workshop Benchmark Circuits, available via anonymous ftp from mcnc.mcnc.org.

[IEEE 88] IEEE Standard 1076-1987, "IEEE Standard VHDL Language Reference Manual," IEEE Standards Board, 345 East 47th Street, New York, NY 10017, 1988.

[Konemann 79] Konemann, B., J. Mucha, and G. Zwiehoff, "Built-In Logic Block Observation Techniques," *1979 IEEE Test Conference*, Cherry Hill, NJ, pp. 37-41, 1979.

[Martinolle 91] Martinolle, F., “Fusion of VHDL Processes,” *Center for Reliable Computing Technical Report 91-7*, CSL-TN-91-384, Computer Systems Laboratory, Stanford University, Stanford, CA, USA, December 1991.

[McFarland 88] McFarland, M. C., A. C. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," *25th ACM/IEEE Design Automation Conference*, Anaheim, CA, pp. 330-336, June 12-15, 1988.

[Paulin 89] Paulin, P. G., and J. P. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASIC's," *IEEE Transactions on Computer-Aided Design*, Vol. 8, No. 6, pp. 661-679, June 1989.

[Sentovich 92] Sentovich, E.M., J. K. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Sequential Circuit Design Using Synthesis and Optimization,” *Int. Conf. on Comput. Des.*, Los Alamitos, CA, USA, pp. 328-333, 1992.

APPENDIX A

VHDL SUBSET SUPPORTED BY ODIN

```
Architecture ::=
  ARCHITECTURE BEHAVIOR OF
            Identifier IS
  BEGIN
    PROCESS
      {VariableDecl}
    BEGIN
      Statements
    END PROCESS;
  END BEHAVIOR;

VariableDecl ::=
  VARIABLE IdentifierList : Type;

IdentifierList ::=
  Identifier
  {, Identifier}

Type ::=
  INTEGER
  | BOOLEAN

Statements ::=
  {Statement}

Statement ::=
  IfStatement
  | CaseStatement
  | LoopStatement
  | WaitStatement
  | AssignmentStatement

IfStatement ::=
  IF Expression THEN
    Statements
  {ELSE Statements}
  END IF;

CaseStatement ::=
  CASE Expression IS
    CaseStatementsAlternative
    {CaseStatementsAlternative}
  END CASE;

CaseStatementsAlternative ::=
  WHEN Choices =>
    Statements

Choices ::=
  Choice
  { | Choice}

Choice ::=
  SimpleExpression

LoopStatement ::=
  IterationScheme LOOP
    Statements
  END LOOP;

IterationScheme ::=
  WHILE Expression
  | FOR LoopParameterSpecification

LoopParameterSpecification ::=
  Identifier IN DiscreteRange

DiscreteRange ::=
  Expression TO Expression
  | Expression DOWNTO Expression

WaitStatement ::=
  WAIT UNTIL Expression;

AssignmentStatement ::=
  VariableAssignment
  | SignalAssignment

VariableAssignement ::=
  Identifier := Expression;

SignalAssignment ::=
  Identifier <= Expression;

Expression ::=
  SimpleExpression
  | EnumLiteral
  | Identifier
  | Expression Operator Expression
  | Operator Expression

SimpleExpression ::=
  Number
  | BitOrCharAggregate
```