ABSTRACT

Capturing Finite State Machines (FSMs) with Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) is explored. An overview of programmable logic design methods and languages is given. Types of FSMs are described. Methods of representing FSMs in behavioral VHDL are described, including specific requirements of the Exemplar GALILEO, the PICA VCOMP and VSIM, and the ALLIANCE tools. Pitfalls and ways to avoid them are discussed. A set of guidelines for describing FSMs with state diagrams is presented. Useful behavioral VHDL output formats are presented including styles used by the set of VHDL tools examined. A subset of the TROFF PIC file format for recording graphics is described. A set of data structures for storing FSM design data bases is described and a method of parsing graphical information into them is presented. As part of this thesis, a new computer program, BRUSEY20, is designed and implemented to convert PIC state machine drawings into behavioral VHDL. The BRUSEY20 tool is presented with an example design run.

PART 1

INTRODUCTION AND HISTORICAL REVIEW

The field of digital hardware design has advanced substantially in the past two decades. The primary design vehicles in the 1970s were discrete logic, small– and medium–scale integration, and simple programmable array logic (PAL) devices. Today, complex programmable logic designs with equivalent gate counts in the tens of thousands are commonplace. Design capture using boolean equations or a few schematic sheets was sufficient for the complexity of many programmable logic designs in the 1970s and 1980s, but today sophisticated design capture tools are not only more convenient, but are becoming essential. Designers require object–oriented tools capable of multiple levels of detail hiding and simplification. High–level design languages (HDLs) similar to those used for computer software are gaining ground as the tools of choice.

One of the key languages which satisfies today's programmable logic design needs is Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL). This language provides a method for both low–level and hierarchical capture in structural and behavioral modes. Originally VHDL was mainly useful for simulation, but today VHDL is increasingly supported by synthesis vendors. These vendors offer tools which can compile a behavioral design into a structural representation and synthesize PLD and FPGA programming files or ASIC floorplans. This paper describes the motivation for the creation of VHDL, and the application of VHDL to programmable logic design, especially the design of Finite State Machines (FSMs). Many hardware designers are not familiar with or prefer not to enter into the software designer mind set required to capture behavioral design descriptions directly with VHDL. The capture of FSMs can be particularly trouble-

some. Although the templates for this process are relatively straight–forward given a target tool, the mechanics of translating an idea into VHDL statements can become as troublesome as compiling an FSM into gates by hand.

The purpose of this effort is to devise a method to allow graphical capture of FSMs using the familiar state diagram and provide VHDL output suitable for use with specific simulation and synthesis tools. Graphical capture is performed using a tool such as XFIG, a free drawing program developed by Supoj Sutanthavibul, et al, which can export a subset of the TROFF PIC format[1]. This format was chosen because it is simple to understand and parse, it can be edited with a free tool, and it can be converted easily for inclusion in design documentation.

The VHDL tools considered in this paper are

- Exemplar's GALILEO synthesis tool,
- the Universite Pierre et Marie Curie ALLIANCE suite, and
- the University of Pittsburgh Integrated Circuit Analysis (PICA) Lab's VCOMP / VSIM simulator.

These tools were chosen because they are resentative of tools currently in use in industry and academia.

PART 2

PROGRAMMABLE LOGIC DESIGN

A. Entry Methods

The classic way to design logic is with schematics. In the past, digital designs were simple enough to be expressed using a single level of hierarchy with discrete component symbols connected by signal runs, possibly across a small number of sheets. Schematics can be used in a hierarchical way with custom symbol block diagrams and multiple levels of decomposition. Functional blocks may be populated with text–documented designs such as PAL Assembly (PALASM) language files. These files may be as simple as to contain only logic equations, or may contain FSM descriptions and truth tables. A relatively new mode of design which can fill in the functional block is graphical entry. This is different from schematic entry in that design behavior is captured instead of structure. This type of entry can take the form of state diagrams, waveform timing diagrams, flow charts, data flow diagrams, and so on.

B. High–Level Design Languages

It is debated that the use of a text HDL is the wave of the future. By the late 1980s and early 1990s, ”less than five percent of all hardware engineers used any HDL at all[2]”. The largest number of engineers use schematics for design capture. Many have not yet begun to employ the hierarchical design style described above. Despite these attitudes, the complexity of designs has driven increased use of HDLs to cleanly partition design responsibilities and avoid errors. Following are presented a few representative logic design languages and their capabilities.

PALASM. This language was pioneered originally in the late 1970s by Monolithic Memories (MMI), a PLD vendor, for use in capturing PAL equations for their devices. The basic concept of designing with PALASM is to record combinational and sequential equations for the PAL. Few provisions are made for behavioral capture. The designer is responsible for "compiling" FSMs, decoders, and functions into equations in a process analogous to early software programming using assembly language. Design block hierarchy is not supported. PALASM2 saw the introduction of rudimentary functional simulation capability. Although this language was introduced to handle a specific vendor's family of PLDs, PALASM is now used as an intermediate format for structural description of digital circuits because of its simplicity and stability.

ABEL. ABEL is a language introduced in the 1980s by Data I/O, a PLD programming vendor. This language offers an increased emphasis on behavioral specification and is mostly vendor and device independent. Popular digital design elements such as FSMs and truth tables are supported in relatively rigid formats. If a design's behavior does not initially fit into one of the supported categories it must be forced to. Functional simulation is supported for both combinational and sequential logic for design verification. Synthesis is supported for multiple target devices and technologies.

VHDL. In the late 1970s and early 1980s, the U. S. Department of Defense funded the Very High Speed Integrated Circuit (VHSIC) program to push digital design technologies. In 1981, the VHSIC Hardware Description Language (VHDL) was proposed to allow the VHSIC program members and vendors communicate designs in a common format. The Department of Defense issued Requirement 64 of MIL–STD–454 requiring the use of VHDL in military projects. Since its beginnings, VHDL has grown to be very com-

plex, and incorporates features from software programming languages such as Ada and other programmable logic design languages. It has progressed from IEEE STD 1076–1987 and –1993. There are also offshoot standards for synthesis, libraries, and analog extensions to VHDL. It allows description in free form structural and behavioral modes, making it flexible enough to carry the design process from concept to implementation. Because VHDL is an IEEE standard, many vendors support it for capture, simulation, and synthesis. With its flexibility comes some unpredictability in results from one vendor to the next, but as the understanding of VHDL and its simulation and synthesis standards increases, this language will mature into an all–purpose design tool.

Certainly there are many more proprietary and generic digital design languages than those listed here. The purpose of this section is merely to introduce a flavor for what levels of HDL capabilities exist and how they can be used.

C. Synthesis

Once a design is captured using either a combination of the methods above or other methods, the programmable logic must be synthesized. Two stages of synthesis can be identified: (a) the conversion of a behavioral design description into gates, flip–flops, or other macro cells, or (b) the optimized combination of these macro cells and the required signal routing into a specific floorplan within a particular silicon architecture. In a simple PAL or Programmable Logic Device (PLD), these steps may be so integrally linked as to merge. For a more complex Field Programmable Gate Array (FPGA) or Application Specific Integrated Circuit (ASIC), the work needed in each step and the types of tasks required may merit a completely separate notion of the two stages, or even the use of more than two stages.

PART 3

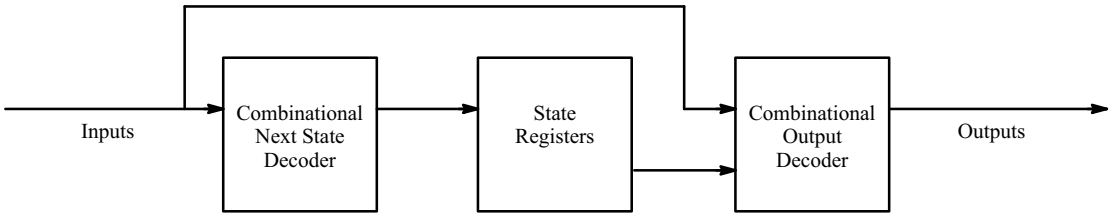
BEHAVIORAL VHDL FINITE STATE MACHINE DESCRIPTION

A. Definitions

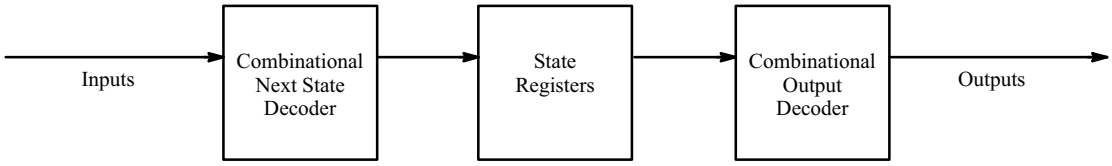
A Finite State Machine is an agglomeration of combinational logic and memory cells clocked through multiple states based on input conditions. The state of the machine is maintained by memory elements, and each output of the machine may be a function of the inputs and the machine state. Typically, FSMs are said to fall into two categories, Moore and Mealy. A Moore FSM is one for which all outputs depend only on the current state. A Mealy FSM is one for which outputs depend on the state and the inputs. These definitions can be somewhat open to interpretation. Fletcher[3] further qualifies FSM types with letter designations. Mealy and Moore FSMs are Class A and B respectively. A Fletcher Class C FSM is a Moore FSM with its outputs taken directly from the outputs of the memory elements. An additional classification, a Mealy FSM with registered outputs, may be denoted as class A1. A summary of these classes is given in Table I. Illustrations of Fletcher FSM classes A, B, and C and the additional class A1 are given in Figure 1. FSMs may also be a mix of classes. Specifically, some outputs may be registered, and others may be combinational. In addition, some outputs may depend directly on the inputs while others are pure functions of the state. In such cases, the outputs can be classified with letter designations in keeping with the convention presented.

Table I. Finite State Machine Classes

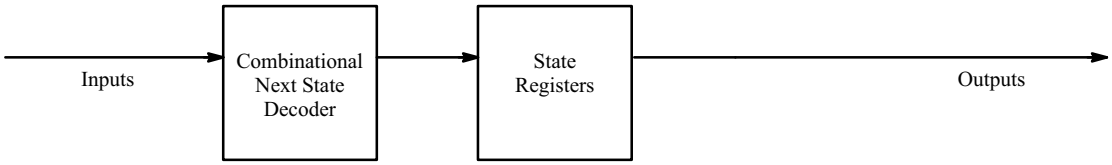
Class A	Combinational–Output Mealy FSM
Class B	Combinational–Output Moore FSM
Class C	Registered–Output Moore FSM
Class A1	Registered–Output Mealy FSM



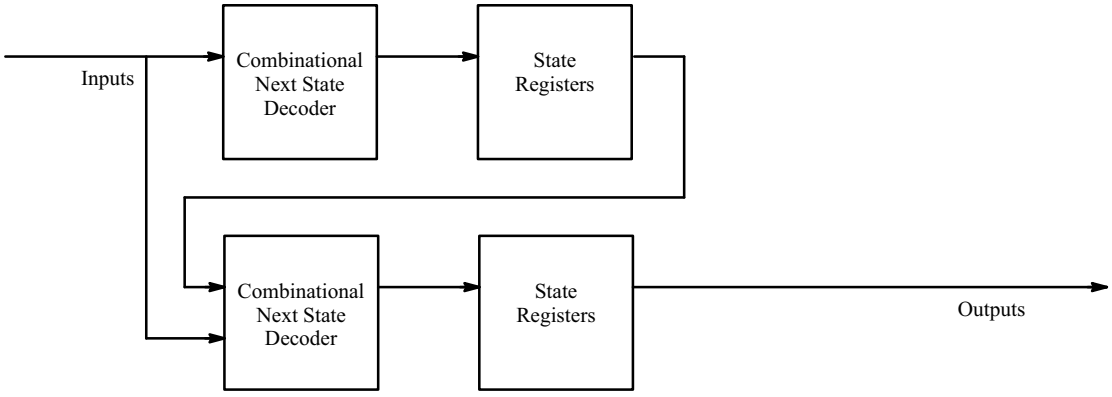
Class A (combinational output Mealy) FSM



Class B (combinational output Moore) FSM



Class C (registered output Moore) FSM



Class A1 (registered output Mealy) FSM

Figure 1. Finite State Machine Classifications

FSMs are useful in implementing complex sequences of events. These sequences may be of a playback type, where the FSM is triggered by a small number of inputs to provide a complex set of output waveforms, or of an action–reaction type, where there are few inputs and outputs, but the FSM will traverse many states. Another dimension to FSM types is the proportion of required fast responses to required slow responses, in other words the FSM may have to wait for long periods between intervals of fast activity.

B. The Finite State Machine Design Process

The typical starting place for a FSM design is to define the interfaces, which signals are inputs and which are outputs. Next, the required relationships between the signals are considered, not only the order of events, but the response times. A quick turn–around time between input changes and output responses will drive a higher FSM clock frequency or will drive the FSM to be of Mealy type. There may be long delays for other outputs which justify a counter to awaken the FSM. Next, the steps required to perform the task at hand must be identified: What happens first? What responses are required at what times? Are there priorities which should divert the FSM from its current action? Next, it is useful to draw a timing diagram, showing inputs, the state of the FSM, and its outputs with time. Once a satisfactory notion of FSM sequencing is obtained, a state diagram can be drawn. With this step completed, the designer must synthesize the combinational and sequential logic to perform the task. One way is to record the diagram using a language like VHDL and feed this format into an automated design process. Another way is to draw the state diagram using a drawing tool and have the balance of the process performed automatically.

C. State Encoding

Another concern in FSM design is the encoding of the state with the memory elements used. Several encoding methods are in use, each with benefits and drawbacks. Binary encoding uses the fewest registers for a given number of states and is most beneficial for PAL and PLD designs where registers are at a premium. Each state is numbered and represented by the binary equivalent. For example, ”000”, ”001”, ”010”, ”011”, ”100”, and so on. In addition to being conservative with registers, this scheme may be easier to interpret during debugging. Gray–code is used to minimize glitching of combinational functions of the state bits, which can be particularly helpful in class B FSMs. Only one bit toggles for each change in state, for example ”000”, ”001”, ”011”, ”111”, ”110”, ”100”, etc. The drawback with this scheme is that it can be wasteful of bits, especially in FSMs where there is a web of possible transitions between states. One–hot encoding is a scheme which minimizes additional logic required to decode the next state and is most useful in FPGAs where registers are plentiful. Only one register is active at a time, so each possible transition needs only consider one state bit instead of all of the bits. An example of one–hot encoding is ”001”, ”010”, ”100”, etc.

D. Asynchronous Reset

It is very important to provide a reset signal to the synchronous processes in a VHDL FSM design. Although simulators and some technologies may be consistent in how registers power up, it is usually not guaranteed in hardware. It is possible to avoid using a reset if unused states transition back to known states and initial glitches are tolerable at start up.

E. State Diagrams

The method described here for drawing state diagrams is similar to the methods presented by Fletcher[4], Blakeslee[5], and Mano[6]. Moore and Mealy implementations of a bounce suppression design are shown shown in Figures 2 and 3. These FSMs can be implemented with registered or combinational outputs resulting in the four possible FSM classes. The differences in functionality are illustrated in Figure 4. In practice, glitching occurs on the output for non–registered Class A and B implementations. States are represented by circles of arbitrary size with the name of the state denoted by the text string closest to the center of the circle.

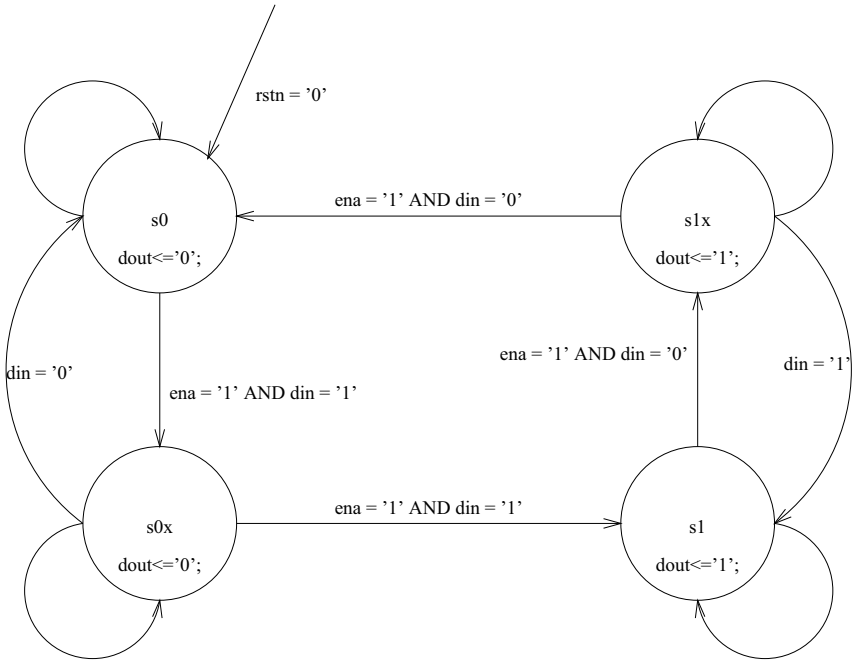


Figure 2. Moore Bounce Suppression Example

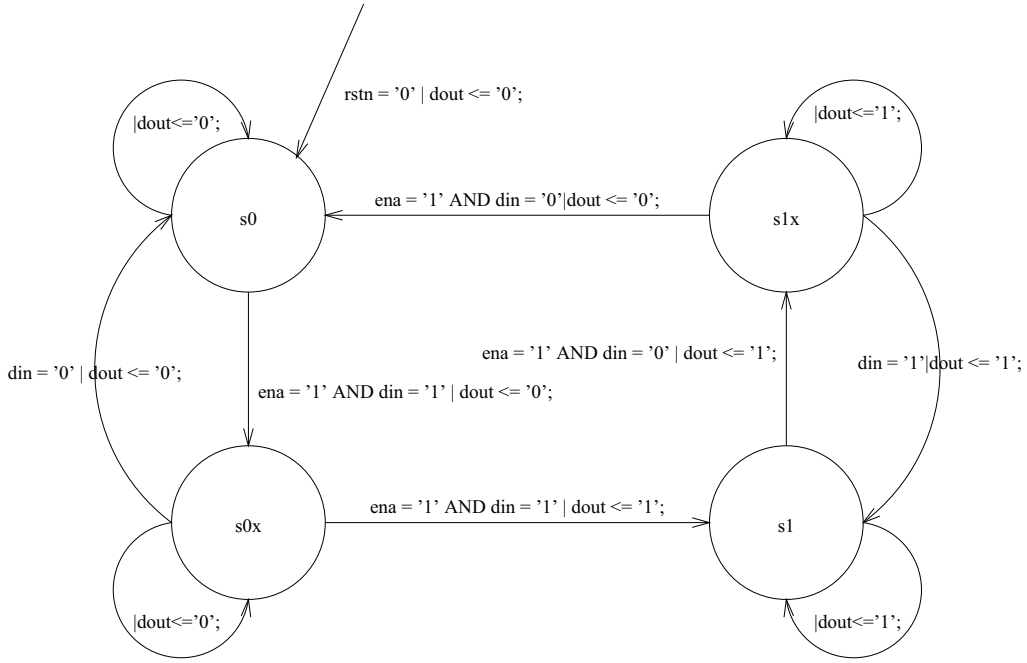


Figure 3. Mealy Bounce Suppression Example

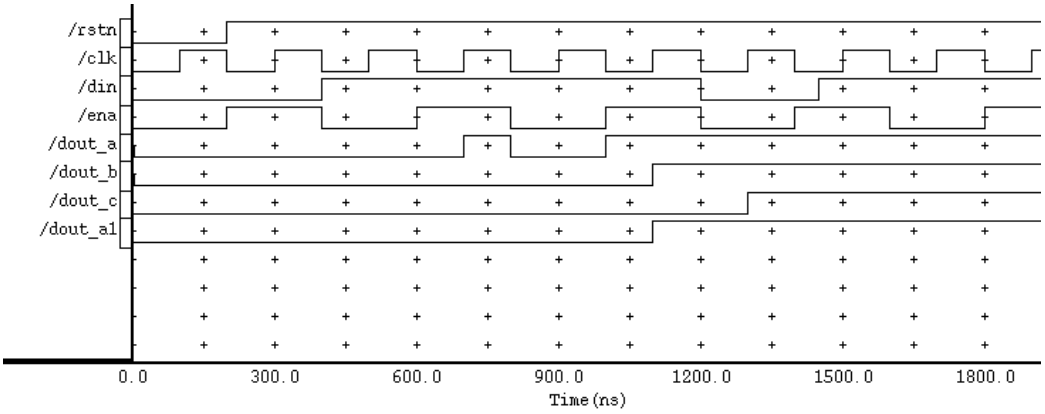


Figure 4. Bounce Suppression FSM Class Functionality

Transitions between states are represented by lines or arcs with arrowheads at the destination state. The asynchronous reset is denoted by a transition with no source state and a reset signal in its conditional expression. All other transitions are synchronous with respect to an implicit clock. Conditional expressions based on the FSM's inputs cause transi-

tions. A conditional expression is specified as a text string at the midpoint of the transition's line or arc. Conditional expressions must be specified such that only one may be true for transitions leaving a given state. One exit transition without a condition is allowed per state, denoting the default transition. If no default is given, it is assumed that the FSM remains in the current state if no transitions are valid.

Both Moore FSM classes and registered–output Mealy class FSMs are supported by the state diagram methodology presented here. Output assignments may be made in states or with transitions. For a given output, if assignments are made only in states, the output will be a function of the state, denoting a Moore FSM. From the diagram, it is not discernable whether or not output assignments in states are registered. This must be specified externally. If at least one assignment for an output is made with a transition, the FSM is implied to be Mealy. Outputs which change with transitions are specified in the same text string as the associated input condition, separated by a vertical bar (pipe). In the case of a state's default transition, an output assignment may still be specified following an initial vertical bar. For the purposes of the BRUSEY20 computer program presented in this paper, both conditional expressions and output assignments must use valid VHDL syntax as in Figure 5.

Figure 5. Conditional Expression / Output Assignment

```
din = '0' AND ena = '1' | dout <= '0';
```

State encoding is implicit and left to the downstream synthesis tool. Therefore, a binary coded value is not needed with a state.

F. Implementation of Finite State Machines with VHDL

Most tools which accept behavioral VHDL input are similar in their expectations of how an FSM is described. Exemplar Logic is a company which specializes in compilation of behavioral inputs to target specific PLD, FPGA, and ASIC technologies. The VHDL coding style expected by Exemplar[7] is used as a baseline for discussion of style differences.

The format expected by The University of Pittsburgh PICA Suite[8] is slightly different. Notably:

- States must be declared as `bit_vectors`.
- Logic variables and signals are of type `bit` instead of `std_logic`.
- Labels must be declared.
- Instead of `'EVENT`, `'RISING` or `'FALLING` must be used for clock edges.

The most encumbering of these differences is the requirement of `bit_vector` state declaration. Fortunately, the context where this stands to do the most harm is gate–level synthesis, which is not part of the PICA process.

The Alliance toolset has been developed and is supported by the CAO–VLSI team at Laboratoire MASI, Universite Pierre et Marie Curie (PARIS VI) in Paris, France. One of the tools available with Alliance is SYF[9], a FSM behavioral to structural synthesizer. It operates on a very specialized subset of VHDL which they have logically dubbed FSM[10]. Notable nuances of this VHDL subset are:

- Logic variables and signals are of type `bit` instead of `std_logic`.
- SYF is assisted via pragmas in identifying the clock and state variables.
- Reset is synchronous.
- Instead of `'EVENT`, `'STABLE` is used.

Unfortunately, Alliance does not directly support registered FSM outputs. The outputs can be encoded into the state definition, but this is done through an external coding file.

VHDL Design Units

The starting place for capturing an FSM with VHDL is the *Entity* which contains the functionality of the FSM. This design unit will declare the inputs and outputs to the FSM. Care must be taken to use the appropriate signal types expected by the downstream synthesis tool. Specifically, Exemplar works best with the IEEE standard 1164 `std_logic` and `std_logic_vector` types, while PICA and Alliance expect `bit` and `bit_vector` types. Figure 6 shows an entity suitable for use with the examples in Figures 2 and 3.

Figure 6. Finite State Machine Entity

```
ENTITY lpf IS
  PORT ( clk : IN std_logic;
    ena : IN std_logic;
    rstn : IN std_logic;
    din : IN std_logic;
    dout : OUT std_logic
  );
END lpf;
```

The VHDL *Architecture* is the implementation of the entity's "black box." It contains internal signals, variables, and most importantly, processes. Most synthesis tools, including Exemplar and Alliance, allow the use of enumerated types for state declaration. In this case, the architecture is setup in VHDL as a type as in Figure 7.

Figure 7. Enumerated Type State Architecture

```
ARCHITECTURE exemplar OF lpf IS
  TYPE state_type IS ( s0, s0x, s1, s1x );
  SIGNAL current_state, next_state : state_type;
  -- other signals omitted
BEGIN
  -- FSM body omitted
END exemplar;
```

To coerce the synthesis tool to use a desired state encoding scheme, the designer can usually explicitly declare the state register output values as in Figure 8.

Figure 8. Explicit State Architecture

```
ARCHITECTURE explicit OF lpf IS
  SIGNAL current_state, next_state :
    std_logic_vector ( 1 DOWNTO 0 );
  CONSTANT s0 : std_logic_vector ( 1 DOWNTO 0 ) := "00";
  CONSTANT s0x : std_logic_vector ( 1 DOWNTO 0 ) := "01";
  CONSTANT s1 : std_logic_vector ( 1 DOWNTO 0 ) := "10";
  CONSTANT s1x : std_logic_vector ( 1 DOWNTO 0 ) := "11";
  -- other signals omitted
BEGIN
  -- FSM body omitted
END explicit;
```

With PICA, it is easiest to make the state register type `bitvec` and use literal constants in the FSM body because neither type definition nor constants are supported.

The *Process* is the VHDL design unit which allows sequentially executing instructions. An architecture without a process is reduced to a structural description. To enable downstream synthesis tools to optimize the implementation, an FSM is best captured as a single entity with processes comprising the internal functionality. The key to behavioral VHDL FSMs is the sensitivity lists of processes. To a VHDL simulator, a change on a signal in a processes sensitivity list triggers the functionality of the process. For synthesis, the way signals in the sensitivity list are used implies combinational and sequential logic. A process with a clock in its sensitivity list and internal dependence on the edges of this clock will synthesize as sequential logic. Figure 9 illustrates this principle.

Figure 9. Sequential Process

```
ARCHITECTURE exemplar OF lpf IS
-- declarations omitted
BEGIN
  registered : PROCESS ( rstn, clk )
  BEGIN
    IF ( rstn = '0' ) THEN
      current_state <= s0;
    ELSIF ( clk'EVENT AND clk = '1' ) THEN
      current_state <= next_state;
    END IF;
  END PROCESS;
  -- other processes ommited
END exemplar;
```

A process with multiple random signals in its sensitivity list with internal boolean expressions involving their instantaneous values will synthesize as combinational logic.

The implication of latches can be a pitfall when unintentional. The designer must make sure to assign a value to each of a combinational process's "output" signals with each execution of the process, otherwise combinational loops may be created. The code in Figure 10 shows how this can happen.

Figure 10. Unintentional Combinational Loop

```
ARCHITECTURE loop OF simple IS
BEGIN
  wrong : PROCESS ( data_in )
  BEGIN
    IF ( data_in = '1' ) THEN
      data_out <= '0';
    END IF;
  END PROCESS;
END loop;
```

This is somewhat subtle, but note that whenever the process is triggered due to `data_in` becoming zero, there is no action taken. This usually synthesizes to an OR gate with output `data_out` and inputs `data_in` and `data_out`. The feedback in this implementation is undesirable.

VHDL Finite State Machine Topologies

For FSM design, many process topologies yield desirable results given the ideal synthesis tool. With the specific tools addressed by this paper, there are specific topologies which are recommended. Typically two processes are used, one combinational and one sequential.

For a class A (combinational–output Mealy) FSM, the combinational process is used to decode the next state and the current outputs given the current state and the inputs. The synchronous process is used to clock the state bits. This type of FSM is not generated by the BRUSEY20 computer program presented in this paper. When BRUSEY20 encounters output assignments in transitions as in Figure 3, it assumes the outputs to be registered. Exemplar, PICA, and Alliance all support it. Examples of the Exemplar, PICA, and Alliance styles for a class A implementation of the FSM in Figure 3 are given in Figures 11, 12, and 13 below.

Figure 11. Exemplar Class A (combinational–output Mealy) FSM

```
ARCHITECTURE exemplar_a OF lpf IS

  TYPE state_type IS ( s0x, s0, s1x, s1 );
  SIGNAL current_state, next_state : state_type;

BEGIN

  registers : PROCESS ( clk, rstn )
  BEGIN
    IF ( rstn = '0' ) THEN
      current_state <= s0;
    ELSIF ( clk'EVENT AND clk = '1' ) THEN
      current_state <= next_state;
    END IF;
  END PROCESS;

  transitions : PROCESS ( current_state, ena, din )
  BEGIN
    CASE current_state IS
      WHEN s0x =>
        IF ( ena = '1' AND din = '0'  ) THEN
          dout <= '0';
          next_state <= s0;
        ELSIF ( ena = '1' AND din = '1'  ) THEN
          dout <= '1';
          next_state <= s1;
        ELSE
          dout<='0';
          next_state <= s0x;
        END IF;
      WHEN s0 =>
        IF ( ena = '1' AND din = '1'  ) THEN
          dout <= '0';
          next_state <= s0x;
        ELSE
          dout<='0';
          next_state <= s0;
        END IF;
      WHEN s1x =>
        IF ( ena = '1' AND din = '1' ) THEN
          dout <= '1';
          next_state <= s1;
        ELSIF ( ena = '1' AND din = '0' ) THEN
          dout <= '0';
          next_state <= s0;
        ELSE
          dout<='1';
          next_state <= s1x;
        END IF;
      WHEN s1 =>
        IF ( ena = '1' AND din = '0'  ) THEN
```

```
          dout <= '1';
          next_state <= s1x;
        ELSE
          dout<='1';
          next_state <= s1;
        END IF;
    END CASE;
  END PROCESS;
END exemplar_a;
```

Figure 12. PICA Class A (combinational–output Mealy) FSM

```
ARCHITECTURE pica_a OF lpf IS
  SIGNAL present_state, next_state : bitvec;
  LABEL registers, transitions;
BEGIN

  registers : PROCESS ( clk, rstn )
  BEGIN
    IF ( rstn = '0' ) THEN
      present_state <= "00"; -- s0
    ELSIF ( clk'RISING ) THEN
      present_state <= next_state;
    END IF;
  END PROCESS;

  transitions : PROCESS ( present_state, ena, din )
  BEGIN
    CASE present_state IS
      WHEN "00" => -- s0
        IF ( ena = '1' AND din = '1' ) THEN
          dout <= '0';
          next_state <= "01"; -- s0x
        ELSE
          dout <= '0';
          next_state <= "00"; -- s0
        END IF;
      WHEN "01" => -- s0x
        IF ( ena = '1' AND din = '1' ) THEN
          dout <= '1';
          next_state <= "10"; -- s1
        ELSIF ( ena = '1' AND din = '0' ) THEN
          dout <= '0';
          next_state <= "00"; -- s0
        ELSE
          dout <= '0';
          next_state <= "01"; -- s0x
        END IF;
      WHEN "10" => -- s1
        IF ( ena = '1' AND din = '0' ) THEN
          dout <= '1';
          next_state <= "11"; -- s1x
```

```
        ELSE
          dout <= '1';
          next_state <= "10"; -- s1
        END IF;
      WHEN "11" => -- s1x
        IF ( ena = '1' AND din = '0' ) THEN
          dout <= '0';
          next_state <= "00"; -- s0
        ELSIF ( ena = '1' AND din = '1' ) THEN
          dout <= '1';
          next_state <= "10"; -- s1
        ELSE
          dout <= '1';
          next_state <= "11"; -- s1x
        END IF;
    END CASE;
  END PROCESS;
END pica_a;
```

Figure 13. Alliance Class A (combinational–output Mealy) FSM

```
ARCHITECTURE alliance_a OF lpf IS
  TYPE state_type IS ( s0, s0x, s1, s1x );

  -- pragma CLOCK clk
  -- pragma CUR_STATE current_state
  -- pragma NEX_STATE next_state

  SIGNAL current_state, next_state : state_type;
BEGIN
--
--  Synchronous State Registers
--  (Note synchronous reset)
--
  registers : PROCESS ( clk )
  BEGIN
    IF ( clk = '1' AND NOT clk'STABLE ) THEN
      current_state <= next_state;
    END IF;
  END PROCESS;
--
--  Combinational State Transitions
--
  transitions : PROCESS ( rstn, current_state, ena, din )
  BEGIN
    -- process to update the current state;
    IF ( rstn = '0' ) THEN
      dout <= '0';
      next_state <= s0;
    ELSE
      CASE current_state IS
        WHEN s0 =>
```

```
            IF ( ena = '1' AND din = '1' ) THEN
              dout <= '0';
              next_state <= s0x;
            ELSE
              dout <= '0';
              next_state <= s0;
            END IF;
          WHEN s0x =>
            IF ( ena = '1' AND din = '1' ) THEN
              dout <= '1';
              next_state <= s1;
            ELSIF ( ena = '1' AND din = '0' ) THEN
              dout <= '0';
              next_state <= s0;
            ELSE
              dout <= '0';
              next_state <= s0x;
            END IF;
          WHEN s1 =>
            IF ( ena = '1' AND din = '0' ) THEN
              dout <= '1';
              next_state <= s1x;
            ELSE
              dout <= '1';
              next_state <= s1;
            END IF;
          WHEN s1x =>
            IF ( ena = '1' AND din = '0' ) THEN
              dout <= '0';
              next_state <= s0;
            ELSIF ( ena = '1' AND din = '1' ) THEN
              dout <= '1';
              next_state <= s1;
            ELSE
              dout <= '1';
              next_state <= s1x;
            END IF;
        END CASE;
      END IF;
    END PROCESS;

END alliance_a;
```

For a class B (combinational–output Moore) FSM, the combinational process decodes the next state based on current state and inputs, but only depends on the state to decode the current outputs. It is supported by Exemplar, PICA, and Alliance in a style specific to each tool. The Exemplar style of this type of FSM can be generated by the BRUSEY20 computer program described in this paper. When the BRUSEY20 tool encounters output assignments in states, it assumes them to be combinational. Examples of the Exemplar, PICA, and Alliance style implementations for the state diagram in Figure 2 are shown in Figures 14, 15, and 16, respectively.

Figure 14. Exemplar Class B (combinational–output Moore) FSM

```
ARCHITECTURE exemplar_b OF lpf IS

  TYPE state_type IS ( s0x, s0, s1x, s1 );
  SIGNAL current_state, next_state : state_type;

BEGIN

  registers : PROCESS ( clk, rstn )
  BEGIN
    IF ( rstn = '0') THEN
      current_state <= s0;
    ELSIF ( clk'EVENT AND clk = '1' ) THEN
      current_state <= next_state;
    END IF;
  END PROCESS;

  transitions : PROCESS ( current_state, ena, din )
  BEGIN
    CASE current_state IS
      WHEN s0x =>
        dout<='0';
        IF ( ena = '1' AND din = '0' ) THEN
          next_state <= s0;
        ELSIF ( ena = '1' AND din = '1' ) THEN
          next_state <= s1;
        ELSE
          next_state <= s0x;
        END IF;
      WHEN s0 =>
        dout<='0';
        IF ( ena = '1' AND din = '1' ) THEN
          next_state <= s0x;
```

```
        ELSE
          next_state <= s0;
        END IF;
      WHEN s1x =>
        dout<='1';
        IF ( ena = '1' AND din = '1' ) THEN
          next_state <= s1;
        ELSIF ( ena = '1' AND din = '0' ) THEN
          next_state <= s0;
        ELSE
          next_state <= s1x;
        END IF;
      WHEN s1 =>
        dout<='1';
        IF ( ena = '1' AND din = '0' ) THEN
          next_state <= s1x;
        ELSE
          next_state <= s1;
        END IF;
    END CASE;
  END PROCESS;
END exemplar_b;
```

Figure 15. PICA Class B (combinational–output Moore) FSM

```
ARCHITECTURE pica_b OF lpf IS
  SIGNAL present_state, next_state : bitvec;
  LABEL registers, transitions;
BEGIN

  registers : PROCESS ( clk, rstn )
  BEGIN
    IF ( rstn = '0' ) THEN
      present_state <= "00"; -- s0
    ELSIF ( clk'RISING ) THEN
      present_state <= next_state;
    END IF;
  END PROCESS;

  transitions : PROCESS ( present_state, ena, din )
  BEGIN
    CASE present_state IS
      WHEN "00" => -- s0
        dout <= '0';
        IF ( ena = '1' AND din = '1' ) THEN
          next_state <= "01"; -- s0x
        ELSE
          next_state <= "00"; -- s0
        END IF;
      WHEN "01" => -- s0x
        dout <= '0';
        IF ( ena = '1' AND din = '1' ) THEN
```

```
          next_state <= "10"; -- s1
        ELSIF ( ena = '1' AND din = '0' ) THEN
          next_state <= "00"; -- s0
        ELSE
          next_state <= "01"; -- s0x
        END IF;
      WHEN "10" => -- s1
        dout <= '1';
        IF ( ena = '1' AND din = '0' ) THEN
          next_state <= "11"; -- s1x
        ELSE
          next_state <= "10"; -- s1
        END IF;
      WHEN "11" => -- s1x
        dout <= '1';
        IF ( ena = '1' AND din = '0' ) THEN
          next_state <= "00"; -- s0
        ELSIF ( ena = '1' AND din = '1' ) THEN
          next_state <= "10"; -- s1
        ELSE
          next_state <= "11"; -- s1x
        END IF;
    END CASE;
  END PROCESS;
END pica_b;
```

Figure 16. Alliance Class B (combinational–output Moore) FSM

```
ARCHITECTURE alliance_b OF lpf IS
  TYPE state_type IS ( s0, s0x, s1, s1x );

  -- pragma CLOCK clk
  -- pragma CUR_STATE current_state
  -- pragma NEX_STATE next_state

  SIGNAL current_state, next_state : state_type;
BEGIN
--
--  Synchronous State Registers
--  (Note synchronous reset)
--
  registers : PROCESS ( clk )
  BEGIN
    IF ( clk = '1' AND NOT clk'STABLE) THEN
      current_state <= next_state;
    END IF;
  END PROCESS;
--
--  Combinational State Transitions
--
  transitions : PROCESS ( rstn, current_state, ena, din )
  BEGIN
```

```
    IF ( rstn = '0' ) THEN
      dout <= '0';
      next_state <= s0;
    ELSE
      CASE current_state IS
        WHEN s0 =>
          dout <= '0';
          IF ( ena = '1' AND din = '1' ) THEN
            next_state <= s0x;
          ELSE
            next_state <= s0;
          END IF;
        WHEN s0x =>
          dout <= '0';
          IF ( ena = '1' AND din = '1' ) THEN
            next_state <= s1;
          ELSIF ( ena = '1' AND din = '0' ) THEN
            next_state <= s0;
          ELSE
            next_state <= s0x;
          END IF;
        WHEN s1 =>
          dout <= '1';
          IF ( ena = '1' AND din = '0' ) THEN
            next_state <= s1x;
          ELSE
            next_state <= s1;
          END IF;
        WHEN s1x =>
          dout <= '1';
          IF ( ena = '1' AND din = '0' ) THEN
            next_state <= s0;
          ELSIF ( ena = '1' AND din = '0' ) THEN
            next_state <= s1;
          ELSE
            next_state <= s1x;
          END IF;
      END CASE;
    END IF;
  END PROCESS;

END alliance_b;
```

With a class C (registered–output Moore) FSM, description is possible with only one sequential process which controls the outputs and steps the state bits. This type of FSM is supported by Exemplar and PICA and is not directly supported by Alliance. The BRUSEY20 program presented in this paper does not support this class of FSM. When BRUSEY20 encounters output assignments in states such as in Figure 2, it assumes them to be combinational. The Exemplar and PICA styles of the FSM in Figure 2 are shown in Figures 17 and 18.

Figure 17. Exemplar Class C (registered–output Moore) FSM

```
ARCHITECTURE exemplar_c OF lpf IS

  TYPE state_type IS ( s0x, s0, s1x, s1 );
  SIGNAL current_state, next_state : state_type;
  SIGNAL next_dout : std_logic;

BEGIN

  registers : PROCESS ( clk, rstn )
  BEGIN
    IF ( rstn = '0') THEN
      current_state <= s0;
      dout <= '0';
    ELSIF ( clk'EVENT AND clk = '1' ) THEN
      current_state <= next_state;
      dout <= next_dout;
    END IF;
  END PROCESS;

  transitions : PROCESS ( current_state, ena, din )
  BEGIN
    CASE current_state IS
      WHEN s0x =>
        next_dout<='0';
        IF ( ena = '1' AND din = '0' ) THEN
          next_state <= s0;
        ELSIF ( ena = '1' AND din = '1' ) THEN
          next_state <= s1;
        ELSE
          next_state <= s0x;
        END IF;
      WHEN s0 =>
        next_dout<='0';
        IF ( ena = '1' AND din = '1' ) THEN
          next_state <= s0x;
```

```
        ELSE
          next_state <= s0;
        END IF;
      WHEN s1x =>
        next_dout<='1';
        IF ( ena = '1' AND din = '1' ) THEN
          next_state <= s1;
        ELSIF ( ena = '1' AND din = '0' ) THEN
          next_state <= s0;
        ELSE
          next_state <= s1x;
        END IF;
      WHEN s1 =>
        next_dout<='1';
        IF ( ena = '1' AND din = '0' ) THEN
          next_state <= s1x;
        ELSE
          next_state <= s1;
        END IF;
    END CASE;
  END PROCESS;
END exemplar_c;
```

Figure 18. PICA Class C (registered–output Moore) FSM

```
ARCHITECTURE pica_c OF lpf IS
  SIGNAL present_state, next_state : bitvec;
  SIGNAL next_dout : bit;
  LABEL registers, transitions;
BEGIN

  registers : PROCESS ( clk, rstn )
  BEGIN
    IF ( rstn = '0' ) THEN
      dout <= '0';
      present_state <= "00"; -- s0
    ELSIF ( clk'RISING ) THEN
      dout <= next_dout;
      present_state <= next_state;
    END IF;
  END PROCESS;

  transitions : PROCESS ( present_state, ena, din )
  BEGIN
    CASE present_state IS
      WHEN "00" => -- s0
        next_dout <= '0';
        IF ( ena = '1' AND din = '1' ) THEN
          next_state <= "01"; -- s0x
        ELSE
          next_state <= "00"; -- s0
        END IF;
```

```
      WHEN "01" => -- s0x
        next_dout <= '0';
        IF ( ena = '1' AND din = '1' ) THEN
          next_state <= "10"; -- s1
        ELSIF ( ena = '1' AND din = '0' ) THEN
          next_state <= "00"; -- s0
        ELSE
          next_state <= "01"; -- s0x
        END IF;
      WHEN "10" => -- s1
        next_dout <= '1';
        IF ( ena = '1' AND din = '0' ) THEN
          next_state <= "11"; -- s1x
        ELSE
          next_state <= "10"; -- s1
        END IF;
      WHEN "11" => -- s1x
        next_dout <= '1';
        IF ( ena = '1' AND din = '0' ) THEN
          next_state <= "00"; -- s0
        ELSIF ( ena = '1' AND din = '1' ) THEN
          next_state <= "10"; -- s1
        ELSE
          next_state <= "11"; -- s1x
        END IF;
    END CASE;
  END PROCESS;
END pica_c;
```

A class A1 (registered output Mealy) FSM may have two processes, a sequential process for the states and outputs, and a combinational process for transitions. Exemplar and PICA support this class, while Alliance does not. This type of FSM may be generated by the BRUSEY20 computer program presented in this paper. When BRUSEY20 encouters output assignments in transitions, it assumes them to be registered. The Exemplar style is shown in Figure 19, while the PICA style is shown in Figure 20. In this implementation, an interim combinational signal is declared to feed a register for a given output.

Figure 19. Exemplar Class A1 (registered–output Mealy) FSM

```
ARCHITECTURE exemplar_a1 OF lpf IS

  TYPE state_type IS ( s0x, s0, s1x, s1 );
  SIGNAL current_state, next_state : state_type;
  SIGNAL next_dout : std_logic;

BEGIN

  registers : PROCESS ( clk, rstn )
  BEGIN
    IF ( rstn = '0' ) THEN
      dout <= '0';
      current_state <= s0;
    ELSIF ( clk'EVENT AND clk = '1' ) THEN
      dout <= next_dout;
      current_state <= next_state;
    END IF;
  END PROCESS;

  transitions : PROCESS ( current_state, ena, din )
  BEGIN
    CASE current_state IS
      WHEN s0x =>
        IF ( ena = '1' AND din = '0'  ) THEN
          next_dout <= '0';
          next_state <= s0;
        ELSIF ( ena = '1' AND din = '1'  ) THEN
          next_dout <= '1';
          next_state <= s1;
        ELSE
          next_dout<='0';
          next_state <= s0x;
        END IF;
      WHEN s0 =>
        IF ( ena = '1' AND din = '1'  ) THEN
```

```
          next_dout <= '0';
          next_state <= s0x;
        ELSE
          next_dout<='0';
          next_state <= s0;
        END IF;
      WHEN s1x =>
        IF ( ena = '1' AND din = '1' ) THEN
          next_dout <= '1';
          next_state <= s1;
        ELSIF ( ena = '1' AND din = '0' ) THEN
          next_dout <= '0';
          next_state <= s0;
        ELSE
          next_dout<='1';
          next_state <= s1x;
        END IF;
      WHEN s1 =>
        IF ( ena = '1' AND din = '0'  ) THEN
          next_dout <= '1';
          next_state <= s1x;
        ELSE
          next_dout<='1';
          next_state <= s1;
        END IF;
    END CASE;
  END PROCESS;
END exemplar_a1;
```

Figure 20. PICA Class A1 (registered–output Mealy) FSM

```
ARCHITECTURE pica_a1 OF lpf IS
  SIGNAL present_state, next_state : bitvec;
  SIGNAL next_dout : bit;
  LABEL registers, transitions;
BEGIN

  registers : PROCESS ( clk, rstn )
  BEGIN
    IF ( rstn = '0' ) THEN
      dout <= '0';
      present_state <= "00"; -- s0
    ELSIF ( clk'RISING ) THEN
      dout <= next_dout;
      present_state <= next_state;
    END IF;
  END PROCESS;

  transitions : PROCESS ( present_state, ena, din )
  BEGIN
    CASE present_state IS
```

```
        WHEN "00" => -- s0
          IF ( ena = '1' AND din = '1' ) THEN
            next_dout <= '0';
            next_state <= "01"; -- s0x
          ELSE
            next_dout <= '0';
            next_state <= "00"; -- s0
          END IF;
        WHEN "01" => -- s0x
          IF ( ena = '1' AND din = '1' ) THEN
            next_dout <= '1';
            next_state <= "10"; -- s1
          ELSIF ( ena = '1' AND din = '0' ) THEN
            next_dout <= '0';
            next_state <= "00"; -- s0
          ELSE
            next_dout <= '0';
            next_state <= "01"; -- s0x
          END IF;
        WHEN "10" => -- s1
          IF ( ena = '1' AND din = '0' ) THEN
            next_dout <= '1';
            next_state <= "11"; -- s1x
          ELSE
            next_dout <= '1';
            next_state <= "10"; -- s1
          END IF;
        WHEN "11" => -- s1x
          IF ( ena = '1' AND din = '0' ) THEN
            next_dout <= '0';
            next_state <= "00"; -- s0
          ELSIF ( ena = '1' AND din = '1' ) THEN
            next_dout <= '1';
            next_state <= "10"; -- s1
          ELSE
            next_dout <= '1';
            next_state <= "11"; -- s1x
          END IF;
      END CASE;
    END PROCESS;
  END pica_a1;
```

As can be seen, the variations on the theme of FSM behavioral description are many. The formats presented here are those recommended by the authors of their respective target tools. It is undoubtedly true that there are many more ways to describe the same FSM with identical or better results with these and other tools.

PART 4

THE MECHANICS OF CONVERSION

A. PIC Graphics Description Input Format

The input to the BRUSEY20 computer program presented in this paper is a subset of the TROFF PIC file format. The major object types used are listed below with their meanings in the context of this process. Each line in the PIC input is a separate statement unlike VHDL where multi–line statements are terminated with a semicolon. All locations and measurements in the PIC input are specified in fixed point format. A circle in the PIC input defines a state in the VHDL output. The format of the PIC circle statement is given in Figure 21.

Figure 21. Circle

```
circle at XC,YC rad R
```

where `XC,YC` is the center of the circle and `R` is the radius. An arc or line in the input defines a transition in the output. Only single–segment arcs and lines are supported. The format of the arc statement is given in Figure 22.

Figure 22. Arc

```
arc {<-,->} at XC,YC from XS,YS to XE,YE [cw]
```

where `<-` denotes that the arrow head is at the ”from” end in the statement and `->` denotes the arrow head is at the ”to” end. `XC,YC` is the center of the circle of which the arc is part, `XS,YS` is the first endpoint of the arc, and `XE,YE` is the second endpoint. If `cw` appears at the end of the statement, the arc swings clockwise between the first and second endpoints instead of counter–clockwise. The format of the line statement is given in Figure 23.

Figure 23. Line

```
line {<-,->} from XS,YS to XE,YE
```

where <– denotes that the arrow head is at the ”from” end in the statement and –> denotes the arrow head is at the ”to” end. `XS,YS` is the first endpoint of the line and `XE,YE` is the second endpoint. The format of the text string statement is given in Figure 24.

Figure 24. Text String

```
”[\sN][\fF]S[\fF]” at XO,YO [{ljust,rjust}]
```

where `N` is the font size, `F` is a font style, and `XO,YO` is the origin of the string. `rjust` or `ljust` indicate right or left justification with respect the origin with a default of center justification if neither is given.

B. Internal Finite State Machine Data Base Structures

A set of data structures are used to store the FSM data base during BRUSEY20 execution. Four structures types store states, transitions, and strings as follows.

Table II. State Structure

string structure pointer for state name	initially blank
center location	
radius	
string structure pointer for first assignment in linked list for the state	initially blank
first transition pointer in linked list for this state	initially blank
next state pointer in overall linked list of states	initially blank

Table III. Transition Structure

Center location	
”From” location	
”From” state pointer	initially blank
”To” location	
”To” state pointer	initially blank
string structure pointer for conditional expression / output assignment	initially blank
next transition pointer in overall linked list of transitions	initially blank
next transition pointer in the linked list for the state	initially blank

Table IV. String Structure

origin location	
justification (left, center, right)	
text string	
next string structure pointer in overall linked list of strings	initially blank
next string pointer in linked list for the state	initially blank

Table V. Input and Output Signal Structure

identifier text string	flag indicating whether the input will be the asynchronous reset signal or whether the output will be registered

As an instance of each of the object type is encountered in the PIC input, memory is allocated to store it. Additional instances are added in a linked list format. This scheme allows the most flexible storage of the largest and most complicated FSMs possible in the memory space available. There is a slight penalty in speed of execution, but no space is wasted with empty object structures.

C. The Conversion Process

With the input and output formats and the internal data base structures presented, what remains to be described is the process employed by BRUSEY20 to generate behavioral VHDL from a PIC drawing.

In order to translate the input PIC file into VHDL, several steps are taken. First, the input is read by the *parser*, the part of the BRUSEY20 program which recognizes syntax and is triggered by language constructs. The parser populates the design data base with the graphical input information, and then control is passed to the *filler*, which makes calculations and relates the data base elements to one another. Next, the *IO finder* sifts through all strings in the design to extract inputs and outputs using a VHDL lexical analysis and grammar specification. The *traverser* operates next, traversing through all structure instances and ultimately writing output based on the generated data base. The details of these steps are expounded below.

Parsing PIC Graphics Description Input Format

The PIC input is parsed using a *Flex*– and *Bison*–based parser. Flex is a computer program which takes in a description of the tokens to be found in the input and desired actions and puts out C code to perform this lexical analysis. Bison is a computer program that takes in a description of the grammar which the tokens are arranged in and puts out C code to perform the parsing. The code generated by Bison calls the code generated by Flex. Once the code has been generated, it may be compiled and included in a main program. This portion of the process creates the linked lists of data structures as it parses the input. Only the geometrical information available in the PIC input is stored during this step.

Filling the Design Data Base

The next step is to associate the instances created in the previous step with one another. First, strings are associated with states based on the distance between the origin of the string and the center of the state's circle. The ratio between the distance from the center to the string and the radius must be below a fixed value.

Next, conditional expression / output assignment strings are associated with transitions. The origin of the string must be within a fixed distance from the midpoint of the arc or line. The midpoint of the arc or line is calculated using geometrical methods and is guaranteed to be on the curve or line. If a string is not found for a given transition, that transition is later flagged as the default for its "from" state.

The transitions ends are then matched with states. The end of the transition must be inside a circle concentric with the state's circle, but with a radius larger by a fixed amount. Connectivity is checked as described above in the description of state diagrams.

Next, each Moore output string is associated with a state which encompasses its origin location. Strings which are not associated with states or transitions are ignored.

Identifying Inputs and Outputs

Each state and transition is processed by sending each associated string to a secondary parser. This parser processes VHDL expressions and assignments and is based on different Flex lexical analysis and Bison grammar files. In other words, there are two parsers used by the BRUSEY20 program, a PIC parser and a VHDL parser. The VHDL parser is given a starting token which identifies whether a given string is associated with a state or a transition. The parser then scans transition and state strings seeking design inputs and outputs. If an output is found in a transition, it is marked as registered to support a class A1

(registered–output Mealy) FSM. If an input is found in the asynchronous reset transition, it is marked for inclusion in the sensitivity list of the registered process of the VHDL architecture. Outputs found in states are marked as non–registered to support a class B (combinational–output Moore) FSM.

Generating VHDL Output

Only the generation of Exemplar–style behavioral code is described in this section, as this is the only style supported for now. Generation of the other styles of behavioral code would be similar.

The first step in generating the behavioral VHDL output is to print the entity header. The input and output signals are written, including the implicit clock signals with their proper types (`IN`, `OUT`, or `INOUT`). `INOUT` types are generated when a signal appears in both the list of inputs and the list of outputs generated above.

Next, the architecture is printed. An enumerated type is defined with names of each of the states in the linked list. Signals for the current state and the next state are declared, followed by any combinational output terms required for registered outputs.

The process used to reset the FSM and clock the registers is printed next. Any input signals that were found in the reset transition are printed in the sensitivity list. Statements to advance the state are printed. If the FSM is a class A1 (registered–output Mealy) FSM, statements to clock the associated output registers are printed.

Next, the process to decode the next state and next outputs is printed. Within a `CASE` statement, a `WHEN` statement is printed for each state. Within the `WHEN` statement, an algorithm is used to correctly print `IF`, `THEN`, `ELSE` statements to cover all transitions leaving the given state and all associated outputs. Outputs are printed within `IF` statements

if the FSM is class A1, and outside the `IF` statement if the FSM is class B for those outputs. If a default transition is found, its processing is deferred until all other transitions from the state have been handled. This way, the default transition is printed within an `ELSE` statement or alone if there is only one transition. The first transition is printed with an `IF` statement, whereas additional transitions (except the default) are printed with `ELSIF` statements. Once all transitions have been handled, the `IF` statement (if any) is capped off with an `END IF`. Finally, the `CASE` statement and the process are closed and the architecture is ended.

PART 5

RESULTS

A. Description

The state diagram for the test case is the example in Figure 25. This design is a class A1 (registered–output Mealy) implementation of a TPLH–TPHL stretcher. This design delays turn–on and turn–off of the output based on the state of the input. Turn–on delay is controlled by the number of states in the left column in the state diagram and turn–off delay is controlled by the right column.

The test case state diagram was drawn with XFIG and exported to PIC format. Next, the BRUSEY20 computer program presented in this paper was run on the PIC file to yield a behavioral VHDL file. The design was then synthesized using Exemplar's GALILEO to yield a structural VHDL output file. GALILEO was then used to synthesize the design with one–hot state encoding in an Actel ACT2 device. Then, a schematic representation of the structure was printed. As can be seen from the schematics below, the FSM output, "O", is registered and there is one register for each state.

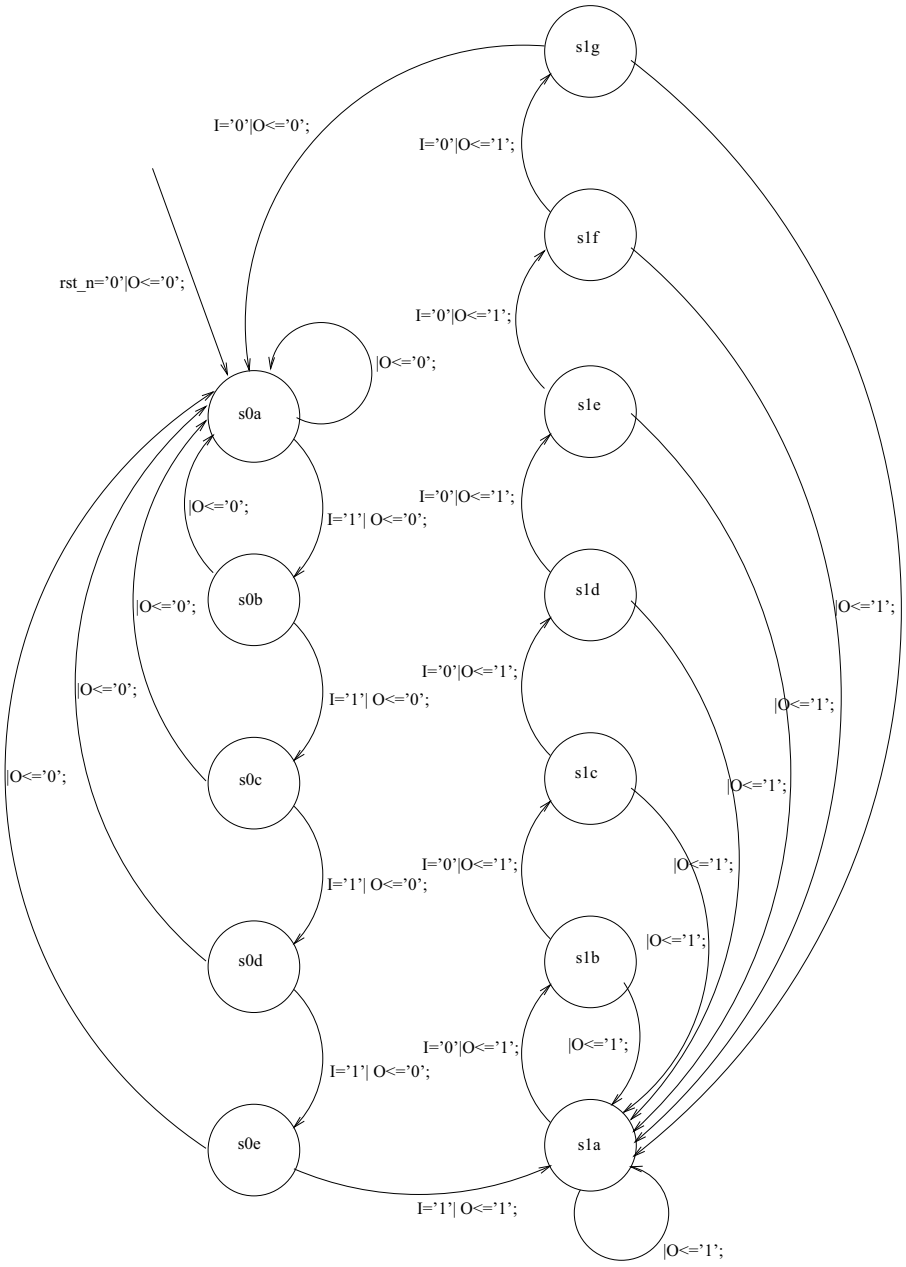


Figure 25. State Diagram for Test Case

B. PIC File Representation

The PIC file representation is shown in Figure 26.

Figure 26. PIC file for Test Case

```
.PS
.ps 10
arc  -> at 2.126,5.992 from 2.657,6.529 to 2.657,5.455 cw
arc  -> at 2.127,4.559 from 2.657,5.095 to 2.657,4.022 cw
arc  -> at 2.126,3.127 from 2.657,3.664 to 2.657,2.590 cw
arc  -> at 2.125,1.695 from 2.657,2.232 to 2.657,1.157 cw
arc  -> at 3.420,5.267 from 1.969,3.861 to 1.969,6.673 cw
arc  -> at 3.755,4.620 from 1.969,2.455 to 1.969,6.786 cw
arc  -> at 3.929,3.927 from 1.969,0.992 to 2.025,6.899 cw
arc  -> at 5.188,1.732 from 4.658,1.195 to 4.658,2.269 cw
arc  -> at 5.187,3.165 from 4.658,2.628 to 4.658,3.701 cw
arc  -> at 5.188,4.597 from 4.658,4.060 to 4.658,5.134 cw
arc  <- at 2.556,6.024 from 2.025,6.561 to 2.025,5.487
arc  -> at 5.189,8.841 from 4.658,8.304 to 4.658,9.379 cw
arc  -> at 5.189,6.029 from 4.658,5.491 to 4.658,6.566 cw
arc  -> at 5.143,7.463 from 4.612,6.925 to 4.612,8.000 cw
arc  -> at 4.449,7.433 from 4.612,9.600 to 2.306,7.069
arc  <- at 4.595,1.868 from 5.139,1.334 to 5.232,2.287
arc  -> at 3.631,3.107 from 2.655,0.832 to 4.658,0.855
arc  <- at 1.991,5.201 from 5.310,0.931 to 5.288,9.488
arc  <- at 2.367,4.520 from 5.319,1.042 to 5.288,8.025
arc  -> at 2.742,3.919 from 5.288,6.730 to 5.308,1.125 cw
arc  -> at 3.285,3.244 from 5.288,5.268 to 5.288,1.219 cw
arc  -> at 4.317,2.561 from 5.288,3.805 to 5.231,1.275 cw
arc  <- at 2.868,7.045 from 2.474,7.081 to 2.676,6.698 cw
arc  <- at 5.213,0.487 from 5.286,0.848 to 4.892,0.668 cw
circle at 2.343,6.707 rad 0.358
circle at 2.343,5.276 rad 0.358
circle at 2.343,3.844 rad 0.358
circle at 2.343,2.410 rad 0.358
circle at 2.343,0.979 rad 0.358
circle at 4.971,1.016 rad 0.358
circle at 4.971,2.448 rad 0.358
circle at 4.971,3.880 rad 0.358
circle at 4.971,8.125 rad 0.358
circle at 4.971,9.557 rad 0.358
circle at 4.971,5.313 rad 0.358
circle at 4.971,6.745 rad 0.358
line <- from 2.136,7.036 to 1.551,8.645
"\s10\fRs0b\fP" at 2.318,5.266
"\s10\fRs0c\fP" at 2.318,3.860
"\s10\fRs0d\fP" at 2.318,2.442
"\s10\fRs0e\fP" at 2.306,1.017
"\s10\fRs1b\fP" at 4.950,2.468
"\s10\fRs1c\fP" at 4.939,3.908
```

```
"\s10\fRs1d\fP" at 4.950,5.337
"\s10\fRs1e\fP" at 4.961,6.788
"\s10\fRs1f\fP" at 4.950,8.087
"\s10\fRs1g\fP" at 4.950,9.584
"\s10\fRI='0'|O<='1';\fP" at 4.415,1.761 rjust
"\s10\fRI='0'|O<='1';\fP" at 4.405,4.697 rjust
"\s10\fRI='1'| O<='0';\fP" at 2.930,4.481 ljust
"\s10\fRI='0'|O<='1';\fP" at 4.382,6.064 rjust
"\s10\fRI='1'| O<='0';\fP" at 2.915,3.036 ljust
"\s10\fRI='0'|O<='1';\fP" at 4.404,3.195 rjust
"\s10\fRI='0'|O<='1';\fP" at 4.354,7.481 rjust
"\s10\fRs0a\fP" at 2.312,6.717
"\s10\fR|O<='0';\fP" at 0.960,4.556 ljust
"\s10\fR|O<='0';\fP" at 0.409,3.881 ljust
"\s10\fRI='1'| O<='0';\fP" at 2.937,1.574 ljust
"\s10\fR|O<='1';\fP" at 5.876,2.599 rjust
"\s10\fR|O<='1';\fP" at 6.090,3.195 rjust
"\s10\fR|O<='1';\fP" at 6.506,3.814 rjust
"\s10\fR|O<='1';\fP" at 6.878,4.444 rjust
"\s10\fR|O<='1';\fP" at 7.350,5.209 rjust
"\s10\fR|O<='1';\fP" at 5.269,1.789 rjust
"\s10\fRrst_n='0'|O<='0';\fP" at 1.804,7.740 rjust
"\s10\fRs1a\fP" at 4.950,1.022
"\s10\fRI='1'| O<='1';\fP" at 3.656,0.684
"\s10\fRI='0'|O<='0';\fP" at 2.812,8.897 rjust
"\s10\fRI='0'|O<='1';\fP" at 4.444,8.897 rjust
"\s10\fRI='1'| O<='0';\fP" at 2.925,6.028 ljust
"\s10\fR|O<='0';\fP" at 3.263,7.153 ljust
"\s10\fR|O<='0';\fP" at 1.800,6.028 ljust
"\s10\fR|O<='0';\fP" at 1.406,5.240 ljust
"\s10\fR|O<='1';\fP" at 5.344,0.234 ljust
.PE
```

C. BRUSEY20 Output

The BRUSEY20 output with full debugging turned on is given in Figure 27, and the VHDL output is given in Figure 28.

Figure 27. BRUSEY20 Debug Output for Test Case

```
-- BRUSEY20 - PIC to VHDL Parser - v2.1
-- Copyright (C) 1995 by Tom Mayo
--
-- To contact the author:  tcmayo@dsinfo.psf.lmco.com
--
--                         Tom Mayo
--                         67 Wilson St.
--                         Pittsfield, MA  01201
--
```

```
-- This program is free software; you can redistribute it and/or
-- modify
-- it under the terms of version 2 of the GNU General Public License
-- as
-- published by the Free Software Foundation.
--
-- This program is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
-- GNU General Public License for more details.
--
-- You should have received a copy of the GNU General Public License
-- along with this program; if not, write to the Free Software
-- Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
--
-- Parsing PIC file...
-- Transition at (2.881, 5.992): (2.657, 6.529) -> (2.657, 5.455).
-- Transition at (2.881, 4.559): (2.657, 5.095) -> (2.657, 4.022).
-- Transition at (2.881, 3.127): (2.657, 3.664) -> (2.657, 2.590).
-- Transition at (2.881, 1.695): (2.657, 2.232) -> (2.657, 1.157).
-- Transition at (1.400, 5.267): (1.969, 3.861) -> (1.969, 6.673).
-- Transition at (0.948, 4.620): (1.969, 2.455) -> (1.969, 6.786).
-- Transition at (0.400, 3.961): (1.969, 0.992) -> (2.025, 6.899).
-- Transition at (4.434, 1.732): (4.658, 1.195) -> (4.658, 2.269).
-- Transition at (4.434, 3.165): (4.658, 2.628) -> (4.658, 3.701).
-- Transition at (4.434, 4.597): (4.658, 4.060) -> (4.658, 5.134).
-- Transition at (1.801, 6.024): (2.025, 5.487) -> (2.025, 6.561).
-- Transition at (4.433, 8.841): (4.658, 8.304) -> (4.658, 9.379).
-- Transition at (4.433, 6.029): (4.658, 5.491) -> (4.658, 6.566).
-- Transition at (4.387, 7.463): (4.612, 6.925) -> (4.612, 8.000).
-- Transition at (2.842, 8.897): (4.612, 9.600) -> (2.306, 7.069).
-- Transition at (5.354, 1.794): (5.232, 2.287) -> (5.139, 1.334).
-- Transition at (3.659, 0.632): (2.655, 0.832) -> (4.658, 0.855).
-- Transition at (7.399, 5.215): (5.288, 9.488) -> (5.310, 0.931).
-- Transition at (6.929, 4.541): (5.288, 8.025) -> (5.319, 1.042).
-- Transition at (6.535, 3.932): (5.288, 6.730) -> (5.308, 1.125).
-- Transition at (6.133, 3.244): (5.288, 5.268) -> (5.288, 1.219).
-- Transition at (5.895, 2.526): (5.288, 3.805) -> (5.231, 1.275).
-- Transition at (3.218, 7.230): (2.676, 6.698) -> (2.474, 7.081).
-- Transition at (5.366, 0.152): (4.892, 0.668) -> (5.286, 0.848).
-- State at (2.343, 6.707): radius 0.358.
-- State at (2.343, 5.276): radius 0.358.
-- State at (2.343, 3.844): radius 0.358.
-- State at (2.343, 2.410): radius 0.358.
-- State at (2.343, 0.979): radius 0.358.
-- State at (4.971, 1.016): radius 0.358.
-- State at (4.971, 2.448): radius 0.358.
-- State at (4.971, 3.880): radius 0.358.
-- State at (4.971, 8.125): radius 0.358.
-- State at (4.971, 9.557): radius 0.358.
-- State at (4.971, 5.313): radius 0.358.
-- State at (4.971, 6.745): radius 0.358.
```

```
-- Transition at (1.844, 7.840): (1.551, 8.645) -> (2.136, 7.036).
-- String at (2.318, 5.266): "s0b".
-- String at (2.318, 3.860): "s0c".
-- String at (2.318, 2.442): "s0d".
-- String at (2.306, 1.017): "s0e".
-- String at (4.950, 2.468): "s1b".
-- String at (4.939, 3.908): "s1c".
-- String at (4.950, 5.337): "s1d".
-- String at (4.961, 6.788): "s1e".
-- String at (4.950, 8.087): "s1f".
-- String at (4.950, 9.584): "s1g".
-- String at (4.415, 1.761): "I='0'|O<='1';".
-- String at (4.405, 4.697): "I='0'|O<='1';".
-- String at (2.930, 4.481): "I='1'| O<='0';".
-- String at (4.382, 6.064): "I='0'|O<='1';".
-- String at (2.915, 3.036): "I='1'| O<='0';".
-- String at (4.404, 3.195): "I='0'|O<='1';".
-- String at (4.354, 7.481): "I='0'|O<='1';".
-- String at (2.312, 6.717): "s0a".
-- String at (0.960, 4.556): "|O<='0';".
-- String at (0.409, 3.881): "|O<='0';".
-- String at (2.937, 1.574): "I='1'| O<='0';".
-- String at (5.876, 2.599): "|O<='1';".
-- String at (6.090, 3.195): "|O<='1';".
-- String at (6.506, 3.814): "|O<='1';".
-- String at (6.878, 4.444): "|O<='1';".
-- String at (7.350, 5.209): "|O<='1';".
-- String at (5.269, 1.789): "|O<='1';".
-- String at (1.804, 7.740): "rst_n='0'|O<='0';".
-- String at (4.950, 1.022): "s1a".
-- String at (3.656, 0.684): "I='1'| O<='1';".
-- String at (2.812, 8.897): "I='0'|O<='0';".
-- String at (4.444, 8.897): "I='0'|O<='1';".
-- String at (2.925, 6.028): "I='1'| O<='0';".
-- String at (3.263, 7.153): "|O<='0';".
-- String at (1.800, 6.028): "|O<='0';".
-- String at (1.406, 5.240): "|O<='0';".
-- String at (5.344, 0.234): "|O<='1';".
-- 12 states, 25 transitions, 37 strings.
-- Filling data structures...
-- Name for state at (2.343, 6.707) is "s0a".
-- Name for state at (2.343, 5.276) is "s0b".
-- Name for state at (2.343, 3.844) is "s0c".
-- Name for state at (2.343, 2.410) is "s0d".
-- Name for state at (2.343, 0.979) is "s0e".
-- Name for state at (4.971, 1.016) is "s1a".
-- Name for state at (4.971, 2.448) is "s1b".
-- Name for state at (4.971, 3.880) is "s1c".
-- Name for state at (4.971, 8.125) is "s1f".
-- Name for state at (4.971, 9.557) is "s1g".
-- Name for state at (4.971, 5.313) is "s1d".
-- Name for state at (4.971, 6.745) is "s1e".
```

```
-- Condition for transition at (2.881, 5.992) is "I='1'| O<='0';".
-- Starting new transition chain for state "s0a".
-- Transition "I='1'| O<='0';" from state "s0a".
-- Transition "I='1'| O<='0';" from state "s0b".
-- Condition for transition at (2.881, 4.559) is "I='1'| O<='0';".
-- Starting new transition chain for state "s0b".
-- Transition "I='1'| O<='0';" from state "s0b".
-- Transition "I='1'| O<='0';" from state "s0c".
-- Condition for transition at (2.881, 3.127) is "I='1'| O<='0';".
-- Starting new transition chain for state "s0c".
-- Transition "I='1'| O<='0';" from state "s0c".
-- Transition "I='1'| O<='0';" from state "s0d".
-- Condition for transition at (2.881, 1.695) is "I='1'| O<='0';".
-- Starting new transition chain for state "s0d".
-- Transition "I='1'| O<='0';" from state "s0d".
-- Transition "I='1'| O<='0';" from state "s0e".
-- Condition for transition at (1.400, 5.267) is "|O<='0';".
-- Transition "|O<='0';" from state "s0a".
-- Adding link 1 to transition chain for state "s0c".
-- Transition "|O<='0';" from state "s0c".
-- Condition for transition at (0.948, 4.620) is "|O<='0';".
-- Transition "|O<='0';" from state "s0a".
-- Adding link 1 to transition chain for state "s0d".
-- Transition "|O<='0';" from state "s0d".
-- Condition for transition at (0.400, 3.961) is "|O<='0';".
-- Transition "|O<='0';" from state "s0a".
-- Starting new transition chain for state "s0e".
-- Transition "|O<='0';" from state "s0e".
-- Condition for transition at (4.434, 1.732) is "I='0'|O<='1';".
-- Starting new transition chain for state "s1a".
-- Transition "I='0'|O<='1';" from state "s1a".
-- Transition "I='0'|O<='1';" from state "s1b".
-- Condition for transition at (4.434, 3.165) is "I='0'|O<='1';".
-- Starting new transition chain for state "s1b".
-- Transition "I='0'|O<='1';" from state "s1b".
-- Transition "I='0'|O<='1';" from state "s1c".
-- Condition for transition at (4.434, 4.597) is "I='0'|O<='1';".
-- Starting new transition chain for state "s1c".
-- Transition "I='0'|O<='1';" from state "s1c".
-- Transition "I='0'|O<='1';" from state "s1d".
-- Condition for transition at (1.801, 6.024) is "|O<='0';".
-- Transition "|O<='0';" from state "s0a".
-- Adding link 1 to transition chain for state "s0b".
-- Transition "|O<='0';" from state "s0b".
-- Condition for transition at (4.433, 8.841) is "I='0'|O<='1';".
-- Starting new transition chain for state "s1f".
-- Transition "I='0'|O<='1';" from state "s1f".
-- Transition "I='0'|O<='1';" from state "s1g".
-- Condition for transition at (4.433, 6.029) is "I='0'|O<='1';".
-- Starting new transition chain for state "s1d".
-- Transition "I='0'|O<='1';" from state "s1d".
-- Transition "I='0'|O<='1';" from state "s1e".
```

```
-- Condition for transition at (4.387, 7.463) is "I='0'|O<='1';".
-- Transition "I='0'|O<='1';" from state "s1f".
-- Starting new transition chain for state "s1e".
-- Transition "I='0'|O<='1';" from state "s1e".
-- Condition for transition at (2.842, 8.897) is "I='0'|O<='0';".
-- Transition "I='0'|O<='0';" from state "s0a".
-- Starting new transition chain for state "s1g".
-- Transition "I='0'|O<='0';" from state "s1g".
-- Condition for transition at (5.354, 1.794) is "|O<='1';".
-- Transition "|O<='1';" from state "s1a".
-- Adding link 1 to transition chain for state "s1b".
-- Transition "|O<='1';" from state "s1b".
-- Condition for transition at (3.659, 0.632) is "I='1'| O<='1';".
-- Adding link 1 to transition chain for state "s0e".
-- Transition "I='1'| O<='1';" from state "s0e".
-- Transition "I='1'| O<='1';" from state "s1a".
-- Condition for transition at (7.399, 5.215) is "|O<='1';".
-- Transition "|O<='1';" from state "s1a".
-- Adding link 1 to transition chain for state "s1g".
-- Transition "|O<='1';" from state "s1g".
-- Condition for transition at (6.929, 4.541) is "|O<='1';".
-- Transition "|O<='1';" from state "s1a".
-- Adding link 1 to transition chain for state "s1f".
-- Transition "|O<='1';" from state "s1f".
-- Condition for transition at (6.535, 3.932) is "|O<='1';".
-- Transition "|O<='1';" from state "s1a".
-- Adding link 1 to transition chain for state "s1e".
-- Transition "|O<='1';" from state "s1e".
-- Condition for transition at (6.133, 3.244) is "|O<='1';".
-- Transition "|O<='1';" from state "s1a".
-- Adding link 1 to transition chain for state "s1d".
-- Transition "|O<='1';" from state "s1d".
-- Condition for transition at (5.895, 2.526) is "|O<='1';".
-- Transition "|O<='1';" from state "s1a".
-- Adding link 1 to transition chain for state "s1c".
-- Transition "|O<='1';" from state "s1c".
-- Condition for transition at (3.218, 7.230) is "|O<='0';".
-- Adding link 1 to transition chain for state "s0a".
-- Transition "|O<='0';" from state "s0a".
-- Transition "|O<='0';" from state "s0a".
-- Condition for transition at (5.366, 0.152) is "|O<='1';".
-- Adding link 1 to transition chain for state "s1a".
-- Transition "|O<='1';" from state "s1a".
-- Transition "|O<='1';" from state "s1a".
-- Condition for transition at (1.844, 7.840) is
-- "rst_n='0'|O<='0';".
-- Transition "rst_n='0'|O<='0';" from state "s0a".
-- Reset transition is "rst_n='0'|O<='0';".
-- Finding inputs and outputs...
-- Parsing strings for state "s0a".
-- Parsing string 1: " trans: I='1'| O<='0';".
-- Passing 22 bytes to the lexer.
```

```
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 2: " trans: |O<='0';".
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state "s0b".
-- Parsing string 3: " trans: I='1'| O<='0';".
-- Passing 22 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 4: " trans: |O<='0';".
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state "s0c".
-- Parsing string 5: " trans: I='1'| O<='0';".
-- Passing 22 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
```

```
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 6: " trans: |O<='0';".
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state "s0d".
-- Parsing string 7: " trans: I='1'| O<='0';".
-- Passing 22 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 8: " trans: |O<='0';".
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state "s0e".
-- Parsing string 9: " trans: |O<='0';".
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 10: " trans: I='1'| O<='1';".
-- Passing 22 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
```

```
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state "s1a".
-- Parsing string 11: " trans: I='0'|O<='1';".
-- Passing 21 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 12: " trans: |O<='1';".
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state "s1b".
-- Parsing string 13: " trans: I='0'|O<='1';".
-- Passing 21 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 14: " trans: |O<='1';".
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state "s1c".
-- Parsing string 15: " trans: I='0'|O<='1';".
-- Passing 21 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
```

```
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 16: " trans: |O<='1';".
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state "s1f".
-- Parsing string 17: " trans: I='0'|O<='1';".
-- Passing 21 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 18: " trans: |O<='1';".
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state "s1g".
-- Parsing string 19: " trans: I='0'|O<='0';".
-- Passing 21 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier "I"
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier "O"
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
```

```
-- Parsing string 20: ” trans: |O<=’1’;”.
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier ”O”
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state ”s1d”.
-- Parsing string 21: ” trans: I=’0’|O<=’1’;”.
-- Passing 21 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier ”I”
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier ”O”
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 22: ” trans: |O<=’1’;”.
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier ”O”
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing strings for state ”s1e”.
-- Parsing string 23: ” trans: I=’0’|O<=’1’;”.
-- Passing 21 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: identifier ”I”
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier ”O”
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Parsing string 24: ” trans: |O<=’1’;”.
-- Passing 16 bytes to the lexer.
-- lexer: ok, starting a transition string.
-- lexer: |
-- lexer: identifier ”O”
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
```

```
-- Passing 0 bytes to the lexer.
-- Parsing string 25: ” async: rst_n='0'|O<='0';”.
-- Passing 25 bytes to the lexer.
-- lexer: ok, starting a async transition string.
-- lexer: identifier ”rst_n”
-- Adding input signal rst_n.
-- lexer: =
-- lexer: literal
-- lexer: |
-- lexer: identifier ”O”
-- lexer: less-than-or-equal or assignment operator
-- lexer: literal
-- lexer: ;
-- Passing 0 bytes to the lexer.
-- Generating VHDL...
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
-- default transition, deferring.
```

Figure 28. BRUSEY20 VHDL Output for Test Case

```
--
-- The following VHDL code was generated by
-- BRUSEY20 - PIC to VHDL Parser - v2.1
-- Copyright (C) 1995 by Tom Mayo
--
-- To contact the author:  tcmayo@dsinfo.psf.lmco.com
--
--                         Tom Mayo
--                         67 Wilson St.
--                         Pittsfield, MA  01201
--
-- This program is free software; you can redistribute it and/or
-- modify
-- it under the terms of version 2 of the GNU General Public License
-- as
-- published by the Free Software Foundation.
--
-- This program is distributed in the hope that it will be useful,
-- but WITHOUT ANY WARRANTY; without even the implied warranty of
-- MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
-- GNU General Public License for more details.
--
-- You should have received a copy of the GNU General Public License
-- along with this program; if not, write to the Free Software
-- Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
--
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY your_entity IS
  PORT (
    clk : IN std_logic;
    I : IN std_logic;
    rst_n : IN std_logic;
    O : OUT std_logic
  );
END your_entity;

ARCHITECTURE exemplar OF your_entity IS

  TYPE state_type IS ( s0a, s0b, s0c, s0d, s0e, s1a, s1b, s1c, s1f,
s1g, s1d, s1e );
  SIGNAL current_state, next_state : state_type;
  SIGNAL next_O : std_logic;

BEGIN

  registers : PROCESS ( clk, rst_n )
  BEGIN
    IF ( rst_n='0') THEN
      O<='0';
```

```
    current_state <= s0a;
  ELSIF ( clk'EVENT AND clk = '1' ) THEN
    O <= next_O;
    current_state <= next_state;
  END IF;
END PROCESS;

transitions : PROCESS ( current_state, I )
BEGIN
  CASE current_state IS
    WHEN s0a =>
      IF ( I='1' ) THEN
        next_O<='0';
        next_state <= s0b;
      ELSE
        next_O<='0';
        next_state <= s0a;
      END IF;
    WHEN s0b =>
      IF ( I='1' ) THEN
        next_O<='0';
        next_state <= s0c;
      ELSE
        next_O<='0';
        next_state <= s0a;
      END IF;
    WHEN s0c =>
      IF ( I='1' ) THEN
        next_O<='0';
        next_state <= s0d;
      ELSE
        next_O<='0';
        next_state <= s0a;
      END IF;
    WHEN s0d =>
      IF ( I='1' ) THEN
        next_O<='0';
        next_state <= s0e;
      ELSE
        next_O<='0';
        next_state <= s0a;
      END IF;
    WHEN s0e =>
      IF ( I='1' ) THEN
        next_O<='1';
        next_state <= s1a;
      ELSE
        next_O<='0';
        next_state <= s0a;
      END IF;
    WHEN s1a =>
      IF ( I='0' ) THEN
```

```
    next_O<='1';
    next_state <= s1b;
  ELSE
    next_O<='1';
    next_state <= s1a;
  END IF;
WHEN s1b =>
  IF ( I='0' ) THEN
    next_O<='1';
    next_state <= s1c;
  ELSE
    next_O<='1';
    next_state <= s1a;
  END IF;
WHEN s1c =>
  IF ( I='0' ) THEN
    next_O<='1';
    next_state <= s1d;
  ELSE
    next_O<='1';
    next_state <= s1a;
  END IF;
WHEN s1f =>
  IF ( I='0' ) THEN
    next_O<='1';
    next_state <= s1g;
  ELSE
    next_O<='1';
    next_state <= s1a;
  END IF;
WHEN s1g =>
  IF ( I='0' ) THEN
    next_O<='0';
    next_state <= s0a;
  ELSE
    next_O<='1';
    next_state <= s1a;
  END IF;
WHEN s1d =>
  IF ( I='0' ) THEN
    next_O<='1';
    next_state <= s1e;
  ELSE
    next_O<='1';
    next_state <= s1a;
  END IF;
WHEN s1e =>
  IF ( I='0' ) THEN
    next_O<='1';
    next_state <= s1f;
  ELSE
    next_O<='1';
```

```
          next_state <= s1a;
        END IF;
    END CASE;
  END PROCESS;
END exemplar;
```

D. Exemplar Output

The Exemplar output is given in Figure 29, and the schematic representation is given in Figure 30.

Figure 29. Exemplar VHDL Output for Test Case

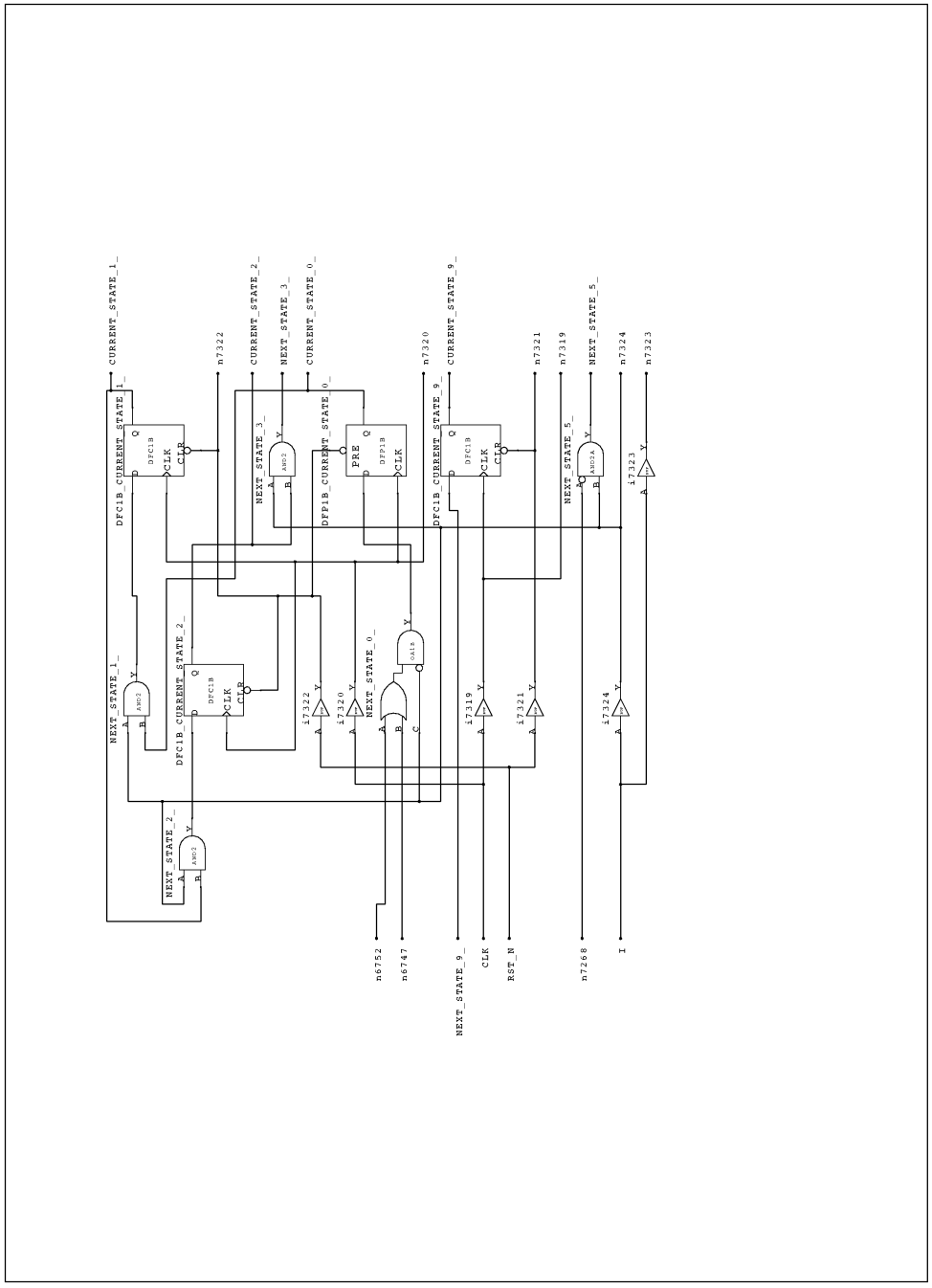
```
--
-- Program
-- gc woj.vhd woj.rtl -input_format=VHDL -target=behav
-- -output_format=VHDL -are
-- a -effort=Standard -macro -wire_tree=Worst -report=slack_table
-- -report=cell_
-- usage -report=device_util -encoding=OneHot -VHDL_93
-- -modgen_library=generic
-- -status_pipe=8
-- Version V3.0.2
-- Definition of  YOUR_ENTITY
--
-- VHDL Concurrent Statements, created by
-- Exemplar Logic's Galileo
--      Tue Jul 25 10:53:00 1995
--
--
--

library IEEE ;
use IEEE.STD_LOGIC_1164.all ;
library EXEMPLAR ;
use EXEMPLAR.EXEMPLAR_1164.all ;

entity YOUR_ENTITY is
   port (
      CLK : IN std_logic ;
      I : IN std_logic ;
      RST_N : IN std_logic ;
      O : OUT std_logic) ;
end YOUR_ENTITY ;

architecture EXEMPLAR of YOUR_ENTITY is
   signal
      CURRENT_STATE_11, n8, CURRENT_STATE_10, CURRENT_STATE_9,
      CURRENT_STATE_8, CURRENT_STATE_7, CURRENT_STATE_6,
```

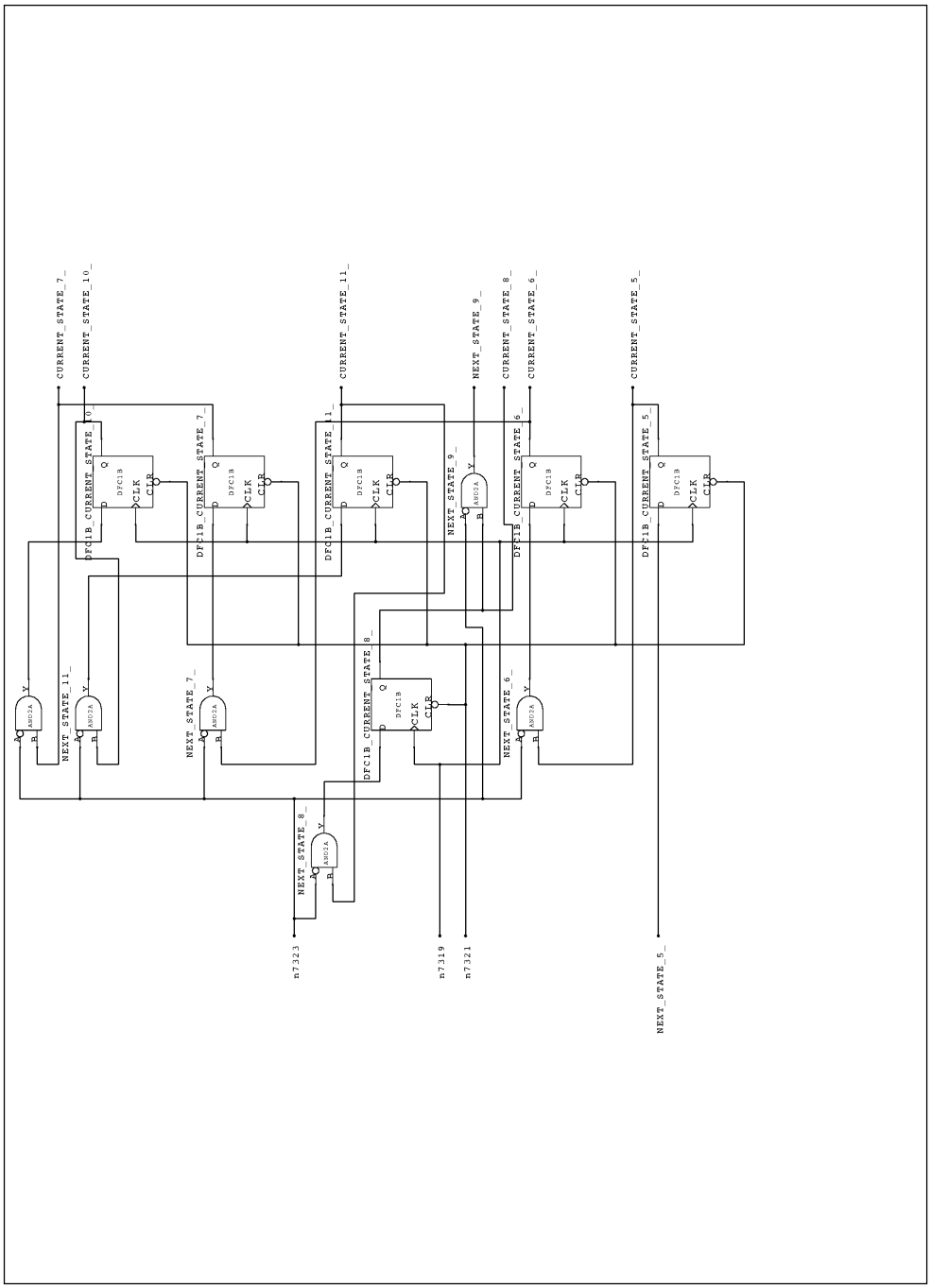
```
CURRENT_STATE_5,
      CURRENT_STATE_4, CURRENT_STATE_3, CURRENT_STATE_2,
CURRENT_STATE_1,
      CURRENT_STATE_0, n126, NEXT_O, NEXT_STATE_5, NEXT_STATE_0,
n596, n597,
      n598, n599, n600, n601, n602, n603, n604, n605, n745, n749,
n750, n751
      : std_logic ;

begin
   O <= n126 ;
   n8 <= (not RST_N) ;
   NEXT_O <= (I and n745) or (n751) or (n750) or (n749) ;
   NEXT_STATE_5 <= (I and n751) or (I and n750) or (I and n749) or
(I and
   n745) ;
   NEXT_STATE_0 <= (not I and n745) or (not I and CURRENT_STATE_0)
or (not I
    and CURRENT_STATE_1) or (not I and CURRENT_STATE_2) or (not I
and
   CURRENT_STATE_3) ;
   n596 <= (I and CURRENT_STATE_0) ;
   n597 <= (I and CURRENT_STATE_1) ;
   n598 <= (I and CURRENT_STATE_2) ;
   n599 <= (I and CURRENT_STATE_3) ;
   n600 <= (not I and CURRENT_STATE_5) ;
   n601 <= (not I and CURRENT_STATE_6) ;
   n602 <= (not I and CURRENT_STATE_11) ;
   n603 <= (not I and CURRENT_STATE_8) ;
   n604 <= (not I and CURRENT_STATE_7) ;
   n605 <= (not I and CURRENT_STATE_10) ;
   n745 <= (CURRENT_STATE_4) or (CURRENT_STATE_9) ;
   n749 <= (CURRENT_STATE_10) or (CURRENT_STATE_11) ;
   n750 <= (CURRENT_STATE_7) or (CURRENT_STATE_8) ;
   n751 <= (CURRENT_STATE_5) or (CURRENT_STATE_6) ;
   DFFPCE (data=>n605,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_11) ;
   DFFPCE (data=>n604,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_10) ;
   DFFPCE (data=>n603,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_9) ;
   DFFPCE (data=>n602,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_8) ;
   DFFPCE (data=>n601,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_7) ;
   DFFPCE (data=>n600,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_6) ;
   DFFPCE
(data=>NEXT_STATE_5,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_5) ;
   DFFPCE (data=>n599,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_4) ;
```

```
   DFFPCE (data=>n598,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_3) ;
   DFFPCE (data=>n597,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_2) ;
   DFFPCE (data=>n596,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_1) ;
   DFFPCE
(data=>NEXT_STATE_0,preset=>n8,clear=>'0',enable=>'1',clk=>CLK,q=>
      CURRENT_STATE_0) ;
   DFFPCE
(data=>NEXT_O,preset=>'0',clear=>n8,enable=>'1',clk=>CLK,q=>n126
   ) ;

end EXEMPLAR ;
```

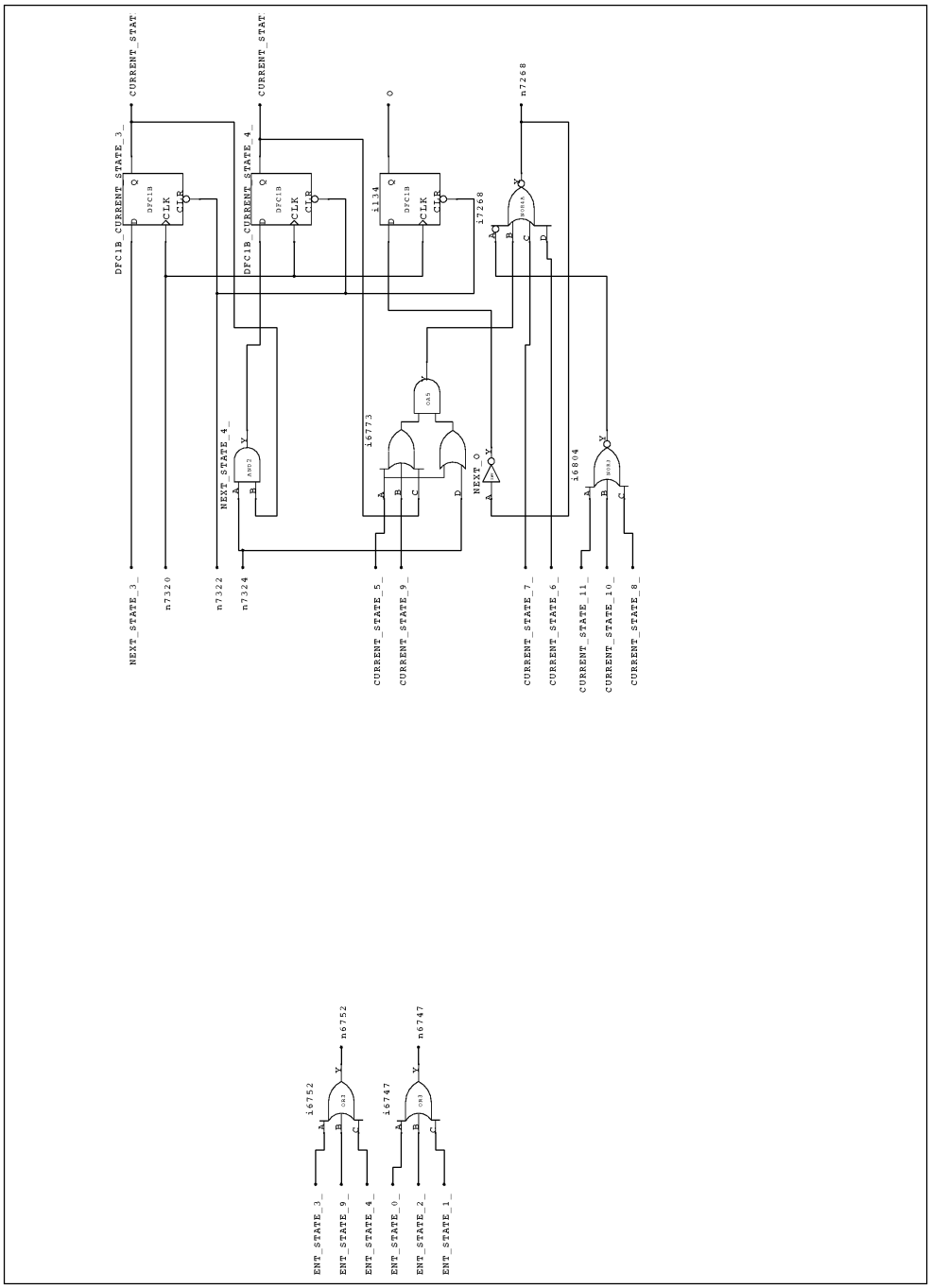
Figure 30. Exemplar Schematic Output for Test Case

See the following three pages.


Library: woj Cell: woj View: Synthesis Page: 1
created by tcmayo@eng112 - 25 Jul 1995, 09:54
CURRENT_STATE_1_
DFC1B_CURRENT_STATE_1_
n7322
CURRENT_STATE_2_
NEXT_STATE_3_
CURRENT_STATE_0_
DFP1B_CURRENT_STATE_0_
n7320
CURRENT_STATE_9_
DFC1B_CURRENT_STATE_9_
n7321
n7319
NEXT_STATE_5_
n7324
n7323
NEXT_STATE_1_
DFC1B_CURRENT_STATE_2_
NEXT_STATE_2_
NEXT_STATE_0_
n6752
n6747
NEXT_STATE_9_
CLK
RST_N
n7268
I



Library: woj Cell: woj View: Synthesis Page: 2
created by tcmayo@eng112 - 25 Jul 1995, 09:54


Library: woj Cell: woj View: Synthesis Page: 3
created by tcmayo@eng112 - 25 Jul 1995, 09:55
DFC1B_CURRENT_STATE_3_
DFC1B_CURRENT_STATE_4_
i1134
i7268
n7268
i6773
i6804
NEXT_STATE_4_
NEXT_O
NEXT_STATE_3_
n7320
n7322
n7324
CURRENT_STATE_5_
CURRENT_STATE_9_
CURRENT_STATE_7_
CURRENT_STATE_6_
CURRENT_STATE_11_
CURRENT_STATE_10_
CURRENT_STATE_8_
i6752
n6752
i6747
n6747
ENT_STATE_3_
ENT_STATE_9_
ENT_STATE_4_
ENT_STATE_0_
ENT_STATE_2_
ENT_STATE_1_

PART 6

DISCUSSION AND CONCLUSIONS

A. Use of the Algorithm

Using the BRUSEY20 program presented in this paper stands to make the FSM design and documentation process easier. The designer need only draw the state diagram once. The diagram can then be converted using BRUSEY20 into behavioral VHDL and used in the actual design. The diagram can also be included without modification in design documentation.

B. Limitations

The BRUSEY20 program, as implemented at this time, has several limitations which restrict its usefulness. These limitations are listed in the order they appear in the program as written.

- The geometric associations in parsing the PIC input are not ideal. For example, the state name must be in the center of the state circle, and transition expressions must be within a fixed distance from the midpoint of the transition.
- There is no way to specify prioritization of transitions. This would reduce expression complexity.
- Signals in expressions and assignments may only be logic type, i.e. no vectors, integers, enumerated types, etc.
- Output FSMs of registered–output Moore and combinational–output Mealy types are not supported.
- The output styles for PICA's VCOMP and VSIM and for Alliance have not been implemented yet.

In order to improve the deficiencies above, the steps below could be taken.

- Tolerances used for association of strings with states and transitions and of transitions with states could be dynamically based on the dimensions of the objects in question.
- String syntax or line style ordering could be used to denote transition prioritization.

- The VHDL parser and the I/O data structure could be enhanced to account for different types (i.e. vectors, integers, enumerated types) of inputs and outputs.
- The VHDL–generating traverser could be enhanced to support class A and C FSMs. Registered–output Moore FSMs could be supported by adding an internal combinational signal and clocking outputs in the registered process. Combinational–output Mealy FSMs could be supported by eliminating the steps in the algorithm which add the internal signal.
- Additional downstream synthesis and simulation tool VHDL output styles could be supported by modifying the traverser and adding a run time parameter which indicates what style is desired.

C. Future Directions

The user interface to the BRUSEY20 program is somewhat cumbersome in terms of input and output file names as well as run time options. Better handling in this regard could be achieved with a graphical push–button type interface. The user could fill in a form with the desired input and output names and activate check boxes to select options. The name of the entity and architecture could also be specified.

There are many State Machine capture tools such as BRUSEY20 available commercially. The capabilities of these commercial tools are more complete than those of the BRUSEY20 program, mainly with respect to integration with the actual drawing program. It would be beneficial to improve the BRUSEY20 tool in this regard.

The steps in the FSM capture process with BRUSEY20 are currently

- Draw the FSM using XFIG,
- Run BRUSEY20,
- Run a gate–level synthesis or simulation tool (Exemplar, PICA, Alliance, or another tool), and
- Run a silicon floorplanner to generate fuse files or an ASIC map.

Integration of these steps into one user interface would make this process easier to follow.

PART 7

LITERATURE CITED

[1]Brian W. Kernighan, "PIC – A Graphics Language for Typesetting User Manual," *Bell Laboratories Computing Science Technical Report No. 116*, (May 1991).

[2]Robert Mendes da Costa, "Teaching Engineers a New Design Paradigm," *Electronic Design*, December 5, 1991, p. 60.

[3]William I. Fletcher, *An Engineering Approach to Digital Design*, (Englewood Cliffs, NJ: Prentice–Hall, 1980), pp. 293–95.

[4]Fletcher, p. 336.

[5]Thomas R. Blakeslee, *Digital Design with Standard MSI and LSI*, (New York: John Wiley & Sons, 1975), pp 117–120.

[6]Morris M. Mano, *Computer Engineering Hardware Design*, (Englewood Cliffs, NJ: Prentice Hall, 1988), pp. 137–138.

[7]Exemplar Logic, Inc., *HDL Synthesis Reference Manua*l, (Berkeley, CA: Exemplar Logic, Inc., 1994), pp. 3–14 – 3–17.

[8]Alan R. Martello, *VCOMP Manual Pages*, (1988), pp. 1–3. and Martello, VSIM Manual Page, (1988), pp. 1–6.

[9]CAO–VLSI team at Laboratoire MASI, Universite Pierre et Marie Curie (PARIS VI), *SYF Manual Pages*, (Paris: electronic, 1993), pp. 1–2.

[10]CAO–VLSI team at Laboratoire MASI, Universite Pierre et Marie Curie (PARIS VI), *FSM Manual Pages*, (Paris: electronic, 1993), pp. 1–5.

APPENDIX

BRUSEY20 MANUAL PAGE

```
BRUSEY20(1)              USER COMMANDS              BRUSEY20(1)


NAME
     brusey20, zzz - Convert TROFF PIC state diagrams into
     behavioral VHDL

SYNOPSIS
     brusey20 design

     zzz [ -h ] [ -dO ... ] [ -sO ... ] [ -ve ]

DESCRIPTION
     zzz parses a state diagram in TROFF PIC format and creates
     behavioral VHDL suitable for simulation and synthesis by
     downstream tools.  PIC input is read from standard input,
     VHDL output is written to standard output, and error and any
     debug output is written to standard error.

     brusey20 runs zzz with full debugging turned on and
     design.pic as standard input, design.vhd as standard output,
     and design.out as standard error.

OPTIONS
     -h   Print help information and quit.

     -dO  Turn on the debugging specified by O.  (See below.)

     -da  Turn all debugging on.

     -dp  Turn PIC parse debugging on.

     -df  Turn data structure fill debugging on.

     -de  Turn expression parse debugging on.

     -di  Turn I/O find debugging on.

     -dv  Turn VHDL code generation debugging on.

     -sO  Turn on synchronizing specified by O.  (See below.)

     -sr  Reset synchronously.  Not yet implemented.

     -so  If an output is Moore, make it registered.  If Mealy,
          make it combinational.  Not yet implemented.
```

-ve Generate Explicit default state transitions. Not yet implemented.

SEE ALSO

Thomas Clayton Mayo, Converting State Diagrams into Synthesizable VHDL, August, 1995.

BUGS

The geometric associations in parsing the PIC input are not ideal. For example, the state name must be in the center of the state circle, and transition expressions must be within a fixed distance from the midpoint of the transition.

There is no way to specify prioritization of transitions. This would reduce expression complexity.

Signals in expressions and assignments may only be logic type, i.e. no vectors, integers, enumerated types, etc.

Output FSMs of registered-output Moore and combinational-output Mealy types are not yet supported.

The output styles for PICA's VCOMP and VSIM and for Alliance have not been implemented yet.

DIAGNOSTICS

Many.

WARNING

brusey20 overwrites design.vhd and design.out without confirmation.