

Investigating Back Annotation of Timing Information into Dataflow Descriptions

by

Zainalabedin Navabi*, Susan Day*, and Mehran Massoumi†

*Electrical and Computer Engineering Department
Northeastern University
409 Dana Research Center
Boston, Massachusetts 02115

†ViewLogic Systems Incorporated
239 Boston Post Road West
Marlboro, Massachusetts 01752
mehr@viewlogic.com

ABSTRACT

This paper presents a study on obtaining high level behavioral models that have accurate timing behavior. The primary concentration is on Finite State Machines (FSM), and we will use timing information from gate-level implementation of these state machines to back-annotate the high level models. The implementation of Mealy and Moore machines is studied, and encoded state assignments versus one-hot state assignments for these machines are compared. Feasibility of timing back annotation of high level FSM descriptions with the information obtained from gate level is analyzed.

1. INTRODUCTION

Efficient simulation is an important feature of the functional or high level models of digital systems. On the other hand, the gate level models which simulate much slower than their functional equivalents offer a much greater accuracy. The timing information required to write accurate models at any level of abstraction is not available until after the circuit is realized. Consequently, the design process of digital circuits involves a down-flow of design information, which includes requirements, constraints, etc., from abstract models, such as functional or dataflow, to the circuit realization and an up-flow of information from the circuit realization to the higher level models. Clearly, the up-flow of timing information can be carried to gate level models without much difficulty. However, to gain better simulation speed, we shall investigate the feasibility of back annotating behavioral models with timing information.

Since considerable research effort has been channeled into identifying synthesizable subsets of VHDL [1, 2], there has been an emergence of various subsets of VHDL [3, 4] each more suitable for describing a certain class of circuits than if the language is used freely in its entirety. Some of these subsets are in the form of a template which the modeler can use and fill in the blanks. In this paper, we shall consider the subset of VHDL which is designed for describing Finite State Machines effectively and efficiently. This choice is made as the focus of our study primarily because State Machines are used in nearly all digital circuits, and their influence on the overall timing of a circuit is more pronounced. Moreover, the style used in this paper is an accepted style by many VHDL tool developers.

Using VHDL, we shall present our findings on high level and gate level simulation tradeoffs, and explore the possibility of timing back-annotation of high level models. We begin by discussing our modeling methods. We will then concentrate on analysis of the simulation of various FSMs and their implementations. The models considered in this study are a Mealy and Moore machines implemented by encoded state assignment and one-hot state assignment

2. BACK ANNOTATION

The process of extracting information from low level descriptions of a circuit, and using this information to correct inaccuracies of upper level models is back-annotation. If the information extracted from a gate level description is related to timing, then the upper level model is said to have been back-annotated with gate level timing information.

A gate level description consists of multiple levels of nesting of gates and hardware components. The simulation, therefore, involves many component invocations and signal assignments. This causes simulation of the gate level descriptions to

generally run slower than their high level counterparts. A high level description that is back-annotated with the information from gate level descriptions can replace the gate level model in simulations where speed is important.

2.1. BACK-ANNOTATING FINITE STATE MACHINES

Gate level descriptions contain information on timing, fan-out, and load dependencies for every node of the circuit. These parameters appear as timing delays and possibly positive or negative short glitches on the final output(s) of the circuit. For back-annotating FSM descriptions with information from gate level simulation, the issues that we will deal with are *delays*, and *hazards*.

Delays: Because of the gate delays, the outputs of gate level descriptions lag behind dataflow or behavioral models in which delays are ignored. In general, the amount of delay depends on the state of the circuit, the status of input and output lines, and the gate level implementation of the circuit. In order to correct high level models, in most cases, the worst case delays for transitions on the outputs of the circuit are used to back-annotate high level models. Depending on the type of the FSM, and its exact implementation, this method may or may not result in an accurate high level model. Obtaining the values of worst case delays is tedious, and may require exhaustive simulation of the gate level models.

Hazards: Hazards may be generated when two inputs of a gate are changing while its output is to remain unchanged, and there is a short time that none of the inputs can drive the output to its required value. In such a case a 0 or a 1 glitch will appear on the output of the gate. Hazards are less predictable than delays, and they heavily depend on the gate level implementation of a structural model. In a small circuit if the exact case of a hazard is known, a behavioral description can be coded such that for the exact input combination and for the exact timing a glitch appears on the output. For larger circuits, however, number of input signals and states of the circuit make the coding of exact instances of hazards prohibitively complex. Therefore, we suggest no solution for back-annotating hazard related information from the gate level to behavioral level. We will, however, suggest implementations of FSMs in which hazards appear in low level as well as high level models.

3. STATE MACHINE DESCRIPTION STYLES

In order to study the ability to back-annotate FSM descriptions and implementations, we will use the *sequence_detector* of Figure 1. The circuit has a *clk* input, an *x_in* input, and a *z_out* output. The *clk* input will be used for synchronization, while data into the circuit will arrive via the *x_in* input. The detector circuit searches for a synchronized sequence of 110 or 101 on the *x_in* input. When a correct sequence is found, a 1 will be placed on the output line.

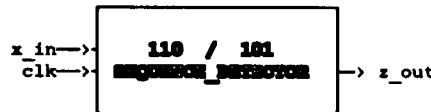


Figure 1. Sequence Detector Example

3.1. MEALY IMPLEMENTATION OF FINITE STATE MACHINE

The functioning of the *sequence_detector* circuit can be described by use of a Mealy machine as shown in Figure 2. This state machine allows for overlapping sequences.

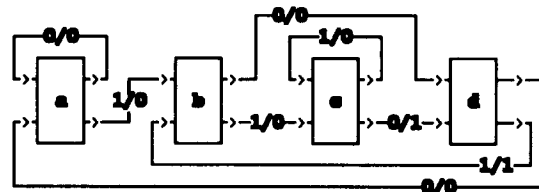


Figure 2. Mealy State Diagram for the Sequence Detector

The VHDL description of the *sequence_detector* has a close correspondence to the state diagram of Figure 2, and is shown in Figure 3. In the declaration part, the enumerate *state* type is defined and is used for declaring the states of the machine (i.e. *a*, *b*, *c*, *d*). A signal named, *current*, of type *state* is used to store the current state of the machine. This signal is initialized to *a*, in order to start the machine in this state.

A process statement in the statement part of the *mealy* architecture of *sequence_detector* implements the state diagram of Figure 2. This process is sensitive to *clk* and an if-statement detects the rising edge of the clock. Assignment of values to the output of the circuit, *z_out*, is done by a conditional signal assignment which uses the signal *current* and input *x_in* for the condition of assigning a '1' to the output.

```

ENTITY detector IS
  PORT (x_in, clk : IN BIT; z_out : OUT BIT);
END detector;
ARCHITECTURE mealy OF detector IS
  TYPE state IS (a, b, c, d);
  SIGNAL current : state := a;
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk = '1' THEN
      CASE current IS
        WHEN a =>
          IF x_in = '1' THEN current <= b; ELSE current <= a; END IF;
        WHEN b =>
          IF x_in = '1' THEN current <= c; ELSE current <= d; END IF;
        WHEN c =>
          IF x_in = '0' THEN current <= d; ELSE current <= c; END IF;
        WHEN d =>
          IF x_in = '1' THEN current <= b; ELSE current <= a; END IF;
      END CASE;
    END IF;
  END PROCESS;
  z_out <= '1' WHEN (current = c AND x_in = '0') OR (current = d AND x_in = '1') ELSE '0';
END mealy;

```

Figure 3. Description of the Mealy Sequence Detector

3.2. MOORE IMPLEMENTATION OF FINITE STATE MACHINE

Moore implementation of the *sequence_detector* is shown in Figure 4. When the circuit reaches states *d* or *f*, the output of the circuit becomes '1' and stays at this level for exactly one clock period. While the output is '1', changes on the input cannot effect the output. The state machine allows for overlapping sequences.

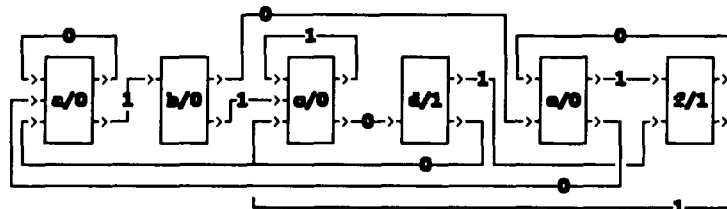


Figure 4. Moore State Diagram for the Sequence Detector

The VHDL description of the Moore version of the *sequence_detector* is shown in Figure 5. The interface description is not shown in Figure 5 because it is the same as that of the *mealy* model. The description of Figure 5 has the same structure as that of Figure 3 except for the output expression which is not dependent on the *x_in* input. This description requires two more states than the description of Figure 3. This is due to the fact that outputs of Moore machines are synchronized with the clock, and the assignment of values to the outputs must be done independent of the inputs.

```

ARCHITECTURE moore OF detector IS
  TYPE state IS (a, b, c, d, e, f);
  SIGNAL current : state := a;
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk = '1' THEN
      CASE current IS
        WHEN a =>
          IF x_in = '1' THEN current <= b; ELSE current <= a; END IF;
        WHEN b =>
          IF x_in = '1' THEN current <= c; ELSE current <= e; END IF;
        WHEN c =>
          IF x_in = '0' THEN current <= d; ELSE current <= c; END IF;
        WHEN d =>
          IF x_in = '1' THEN current <= f; ELSE current <= a; END IF;
        WHEN e =>
          IF x_in = '1' THEN current <= f; ELSE current <= a; END IF;
        WHEN f =>
          IF x_in = '1' THEN current <= a; ELSE current <= e; END IF;
      END CASE;
    END IF;
  END PROCESS;
  z_out <= '1' WHEN (current = d OR current = f) ELSE '0';
END moore;

```

Figure 5. Description of Moore Sequence Detector

3.3. COMPONENTS OF STRUCTURAL DESCRIPTION

A simple gate level implementation for either version of the *sequence_detector* circuit (Mealy or Moore) can be obtained by using standard logic design methods. A minimum set of components needed for this purpose are an inverter, a two-input AND gate, a two-input OR gate, and a d-type flip-flop. Figure 6 shows these structural level component descriptions, in which typical delay values are instilled for rising and falling delays. These delay values are the source of discrepancies between the structural and behavioral level descriptions, and provide an insight into the back annotation of the behavioral model.

```

ENTITY inv IS
  PORT (a : IN BIT; z : OUT BIT);
  CONSTANT rise_delay : TIME := 5 NS;
  CONSTANT fall_delay : TIME := 3 NS;
END inv;
ARCHITECTURE primitive OF inv IS
  BEGIN
    PROCESS (a)
    BEGIN
      IF a = '0' THEN
        z <= '1' AFTER rise_delay;
      ELSE
        z <= '0' AFTER fall_delay;
      END IF;
    END PROCESS;
  END primitive;

ENTITY or2 IS
  PORT (a, b : IN BIT; z : OUT BIT);
  CONSTANT rise_delay : TIME := 7 NS;
  CONSTANT fall_delay : TIME := 5 NS;
END or2;
ARCHITECTURE primitive OF or2 IS
  BEGIN
    PROCESS (a, b)
    BEGIN
      IF a = '1' OR b = '1' THEN
        z <= '1' AFTER rise_delay;
      ELSE
        z <= '0' AFTER fall_delay;
      END IF;
    END PROCESS;
  END primitive;

ENTITY and2 IS
  PORT (a, b : IN BIT; z : OUT BIT);
  CONSTANT rise_delay : TIME := 7 NS;
  CONSTANT fall_delay : TIME := 5 NS;
END and2;
ARCHITECTURE primitive OF and2 IS
  BEGIN
    PROCESS (a, b)
    BEGIN
      IF a = '1' AND b = '1' THEN
        z <= '1' AFTER rise_delay;
      ELSE
        z <= '0' AFTER fall_delay;
      END IF;
    END PROCESS;
  END primitive;

ENTITY d_flipflop IS
  PORT (d, set, rst, clk : IN BIT; q : OUT BIT);
  CONSTANT rise_delay : TIME := 15 NS;
  CONSTANT fall_delay : TIME := 13 NS;
END d_flipflop;
ARCHITECTURE primitive OF d_flipflop IS
  SIGNAL state : BIT := '0';
  BEGIN
    PROCESS (rst, set, clk)
    BEGIN
      IF set = '1' THEN
        state <= '1' AFTER rise_delay;
      ELSEIF rst = '1' THEN
        state <= '0' AFTER fall_delay;
      ELSEIF clk = '1' AND clk'EVENT THEN
        IF d = '1' THEN
          q <= '1' AFTER rise_delay;
        ELSE
          q <= '0' AFTER fall_delay;
        END IF;
      END IF;
    END PROCESS;
    q <= state;
  END primitive;

```

Figure 6. Primitive Gate Level Components

4. BACK ANNOTATION IN MEALY MACHINES

For analyzing back annotation of timing information into behavioral Mealy machine descriptions, we will use the *mealy_sequence_detector* of Section 3, and implement it using the hardware structures of Figure 6. Two hardware implementations for this machine will include encoded state assignment, and one-hot state assignment. In the following subsections, the structural models of the *mealy_sequence_detector* will be presented, simulation results of these models will be compared, and the back annotation of the Mealy behavioral model will be discussed.

4.1. ENCODED STATE ASSIGNMENT MODEL

The encoded state assignment 00, 01, 11, 10 is used for states *a*, *b*, *c*, and *d* respectively. Based on this assignment, the gate level implementation for the *mealy_sequence_detector* is shown in Figure 7. This design consists of two rising-edge D-type flip-flops with set/reset capabilities. AND and OR gates are used for implementing the input equations of the flip-flops and the logic for the output circuit.

The VHDL description of the circuit of Figure 7 can be written by wiring the components that have been shown in Section 3. Figure 8 shows the structural description of the *mealy_sequence_detector*.

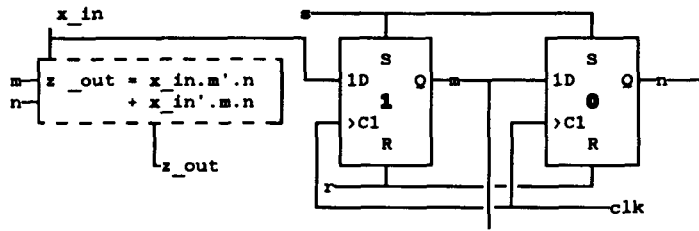


Figure 7. Encoded State Assignment Implementation

```

ENTITY detector IS
    PORT (x_in, clk : IN BIT; z_out : OUT BIT);
END detector;

ARCHITECTURE mealy_encoded OF detector IS
    COMPONENT dff PORT (d, c : IN BIT; q : OUT BIT); END COMPONENT;
    COMPONENT and2 PORT (a, b : IN BIT; z : OUT BIT); END COMPONENT;
    COMPONENT or2 PORT (a, b : IN BIT; z : OUT BIT); END COMPONENT;
    COMPONENT inv PORT (a : IN BIT; z : OUT BIT); END COMPONENT;
    FOR ALL : dff USE ENTITY WORK.d_sr_flipflop (behavioral);
    FOR ALL : and2 USE ENTITY WORK.and2 (primitive);
    FOR ALL : or2 USE ENTITY WORK.or2 (primitive);
    FOR ALL : inv USE ENTITY WORK.inv (primitive);
    SIGNAL node_m, node_n, node_o, node_p, node_q, node_r, node_s : BIT;
    SIGNAL s, r : BIT := '0';
BEGIN
    d1: dff PORT MAP (x_in, s, r, clk, node_m);
    d2: dff PORT MAP (node_m, s, r, clk, node_n);
    i1: inv PORT MAP (node_m, node_o);
    i2: inv PORT MAP (x_in, node_p);
    a1: and2 PORT MAP (node_o, x_in, node_q);
    a2: and2 PORT MAP (node_m, node_p, node_r);
    o2: or2 PORT MAP (node_q, node_r, node_s);
    a3: and2 PORT MAP (node_s, node_n, z_out);
END mealy_encoded;

```

Figure 8. Structural Description of the Mealy Encoded Detector

4.2 ONE-HOT STATE ASSIGNMENT MODEL

In the *one-hot* state assignment for the *mealy* machine of Figure 2, the state assignment 0001, 0010, 0100, 1000 is used for states *a*, *b*, *c*, and *d* respectively. This design implements one flip-flop per state for a total of four D-type set/reset flip-flops. As in the *mealy_encoded* AND gates and OR gates are used for implementing the Boolean equations of the states and output of *mealy_onehot*.

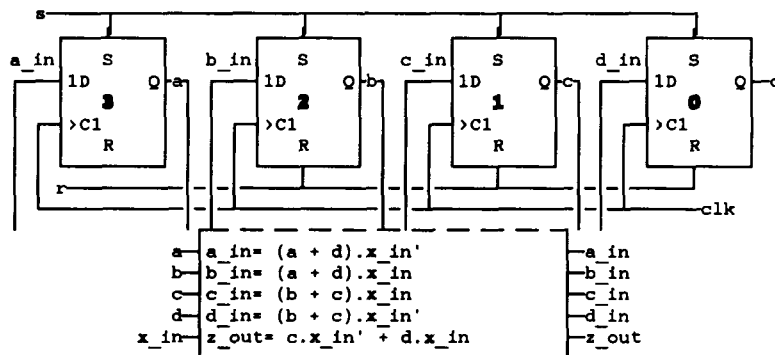


Figure 9. One-Hot State Assignment Implementation

The VHDL description that corresponds to the hardware of Figure 9 is shown in Figure 10.

4.3 TEST BENCH AND SIMULATION IN VHDL

Shown in Figure 11 is the VHDL description used for testing all three versions of the *mealy_sequence_detector* in parallel. The interface description of the test bench contains a *scale* factor for scaling the clocking speed and the rate of change of the input signal. In the architectural declaration part of this description the *detect* component has been declared. This is followed by a configuration specification which binds the instantiation of *detect* (labeled *machine*) to the architectural description of the

detector that is identified as *mealy* (Figure 3). A second configuration specification binds the instantiation of *detect* (labeled *encoded*) to the description of Figure 8. Lastly, the instantiation of *detect* (labeled *onehot*) is bound to the description of Figure 10.

```

ARCHITECTURE mealy_onehot OF detector IS
-- ... same as Figure 8 ... --
SIGNAL s : BIT := '1';
SIGNAL r : BIT := '0';
SIGNAL node_a,node_b,node_c,node_d : BIT;
SIGNAL node_o,node_p,node_r,node_s,node_t,node_u : BIT;
SIGNAL node_v,node_w,node_q : BIT;
-BEGIN
s <= '0' AFTER 25 NS;
d1: dff PORT MAP (node_q, s, r, clk, node_a);
d2: dff PORT MAP (node_r, r, s, clk, node_b);
d3: dff PORT MAP (node_t, r, s, clk, node_c);
d4: dff PORT MAP (node_u, r, s, clk, node_d);
i1: inv PORT MAP (x_in, node_p);
o1: or2 PORT MAP (node_a, node_d, node_o);
a1: and2 PORT MAP (node_o, node_p, node_q);
a2: and2 PORT MAP (node_o, x_in, node_r);
o2: or2 PORT MAP (node_b, node_c, node_s);
a3: and2 PORT MAP (node_s, x_in, node_t);
a4: and2 PORT MAP (node_s, node_p, node_u);
a5: and2 PORT MAP (node_c, node_p, node_v);
a6: and2 PORT MAP (node_d, x_in, node_w);
o3: or2 PORT MAP (node_v, node_w, z_out);
-END mealy_onehot;

```

Figure 10. Structural Description of the Mealy One-Hot Detector

In the body of the architectural specification of the *test_bench*, a conditional signal assignment statement is used for the generation of clock pulses. The statements following the assignments to *x* are instantiations of the three versions of the *mealy_sequence_detector*. The internal signals *c* and *x* are connected to the *clk* and *x_in* inputs of the models of the *mealy_sequence_detector* respectively. The output signals are named *z_machine*, *z_encoded*, and *z_onehot*.

```

ENTITY test_bench IS
GENERIC (scale : INTEGER := 10);
-END test_bench;
-ARCHITECTURE mealy OF test_bench IS
COMPONENT detect PORT (x_in,clk: IN BIT; z_out: OUT BIT); END COMPONENT;
FOR machine : detect USE ENTITY WORK.detector(mealy);
FOR encoded : detect USE ENTITY WORK.detector(mealy_encoded);
FOR onehot : detect USE ENTITY WORK.detector(mealy_onehot);
SIGNAL x, c, z_machine, z_encoded, z_onehot : BIT := '0';
-BEGIN
c <= NOT c AFTER (scale/2)*1 NS; WHEN NOW <= scale*8 NS ELSE c;
x <= '1' AFTER scale*05NS, '0' AFTER scale*15NS, '1' AFTER scale*25NS,
'0' AFTER scale*41NS, '1' AFTER scale*44NS, '0' AFTER scale*48NS,
'1' AFTER scale*55NS, '0' AFTER scale*56NS, '1' AFTER scale*57NS,
'0' AFTER scale*67NS;
machine : detect PORT MAP (x, c, z_machine);
encoded : detect PORT MAP (x, c, z_encoded);
onehot : detect PORT MAP (x, c, z_onehot);
-END mealy;

```

Figure 11. Test Bench for the Sequence Detector

A detailed simulation is provided in Figure 12. This Figure is a tabular list of signal values for times during which one of the signals change value. All numbers on the left hand side of the tables are time values in nanoseconds. The '+' signs in the time column, after a time value, indicate events that occur within that time in the specified order. For example two events occur at 500 Nano Second (NS). The first event changes *c* from 0 to 1, and the next event occurs on *z_machine* causing it to change from 1 to 0.

4.4 SIMULATION ANALYSIS

Analyzing the results of Figure 12 reveals that the general pattern of the outputs at the behavioral level (*mealy*) and the structural levels (*mealy_encoded*, *mealy_onehot*) are the same. Upon careful inspection of Figure 12, however, timing related differences between each of the implementations are observed. These differences are due to *hazards* and *delays*.

The first noticeable difference is a 6 NS glitch in *z_encoded*, which occurs at 222 and 722 NS. This glitch results from a hazard in the encoded states and delays in the output logic. The hazard develops during the transition from state b (01) to state c (10), at which time the logic forces a transition to state a (00) before continuing on to stable state c (10). Glitches caused by hazards in the architecture of *mealy_encoded* do not occur in the behavioral model, *mealy*, or the structural model, *mealy_onehot*. Furthermore, a glitch which does occur in the *moore* model at time 550+1 NS does not occur in the *moore_encoded* model due to delays in the output logic which exceed the 10 NS transition time of the *x_in* input. Since the output logic is reduced, resulting in a smaller delay in the *mealy_onehot* model, this glitch does occur in the output of *z_onehot* at a 14 NS delay time of 564 NS.

TIME (NS)	SIGNAL NAMES					TIME (NS)	SIGNAL NAMES				
	C	X	Z	Z	Z		C	X	Z	Z	Z
			\bar{M}	\bar{E}	\bar{O}				\bar{M}	\bar{E}	\bar{O}
			A	N	N				A	N	N
			C	C	E				C	C	E
			H	O	H				H	O	H
			I	D	O				I	D	O
			N	E	T				N	E	T
			E	D					E	D	
0	'0'	'0'	'0'	'0'	'0'	480	...	'0'
40	'1'	+1	'1'
45	'0'	499	'1'
50	...	'1'	500
100	+1	'1'
+1	'1'	+3	'0'
105	'0'	505	'0'
150	...	'0'	506	'1'	...
200	523	'0'
+1	'1'	528	'0'	...
205	'0'	550	...	'1'
222	'1'	...	+1	...	'1'
228	'0'	...	560	...	'0'
250	...	'1'	+1	'0'
+1	'1'	564	'1'	...
264	'1'	...	570	...	'1'	'0'
271	'1'	...	+1	'1'
300	584	'1'
+1	'1'	591	'1'	...
+3	'0'	600
305	'0'	+1	'1'
318	'0'	...	+3	'0'
323	'0'	605	'0'
400	618	'0'	...
+1	'1'	623	'0'
405	'0'	670	...	'0'
410	...	'0'	700
+1	'1'	+1	'1'
429	'1'	705	'0'
436	'1'	...	722	'1'	...
440	...	'1'	728	'0'	...
+1	'0'						
453	'0'						
458	'0'	...						

Figure 12. Detailed Mealy Simulation Run

The *mealy_encoded* and *mealy_onehot* models do not provide consistent delays. The difference in delays depends upon the state and its respective logic. This simulation run indicates that just by looking at the output, the exact delay is not predictable. Therefore, the best that can be done is to use the worst case or average delays for the output transitions. Obtaining the values of worst case delays is tedious, and may require exhaustive simulation of the gate level models. Since in most cases only the primary outputs of a circuit are available, back annotating each state output is not possible. It is only possible to back annotate an average delay to the primary output of the behavioral model, *mealy* [5].

5. BACK ANNOTATION IN MOORE MACHINES

In the following sub-sections, two structural models of the *moore_sequence_detector* will be presented, simulation results of these models will be compared, and the back annotation of the Moore behavioral model will be discussed.

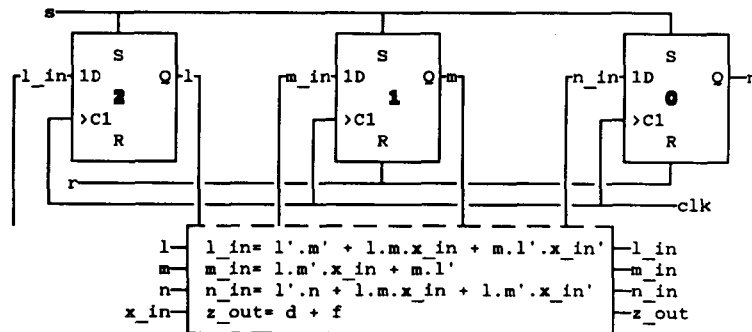


Figure 13. Encoded State Assignment Implementation

5.1. ENCODED STATE ASSIGNMENT MODEL

A simple gate level implementation of the *moore_sequence_detector* circuit can be obtained by using standard logic design methods. The encoded state assignment 000, 001, 010, 011, 100, and 101 is used for states *a, b, c, d, e, and f* respectively. This design, shown in Figure 13, consists of three rising-edge D-type flip-flops with set/reset capabilities. AND gates and OR gates are used for implementing the input equations of the flip-flops, and the logic for the output circuit.

The VHDL description of the circuit of Figure 13 can be written by wiring the components that have been shown in Figure 6. For each of the components of Figure 13, an instantiation statement is used in Figure 14.

```

ARCHITECTURE moore_encoded OF detector IS
-- ... same as Figure 8 ... --
SIGNAL node_a,node_b,node_c,node_d,node_e,node_f,node_g,node_h,node_i : BIT;
SIGNAL node_j,node_k,node_l,node_m,node_n,node_o,node_p,node_q,node_r : BIT;
SIGNAL node_s,node_t,node_u,node_v,node_w,node_x,node_y,node_z : BIT;
SIGNAL s, r : BIT := '0';
BEGIN
d1: dff PORT MAP (node_y, s, r, clk, node_l);
d2: dff PORT MAP (node_b, r, s, clk, node_m);
d3: dff PORT MAP (node_k, r, s, clk, node_n);
i1: inv PORT MAP (x_in, node_o);
i2: inv PORT MAP (node_l, node_p);
i3: inv PORT MAP (node_m, node_q);
i4: inv PORT MAP (node_n, node_r);
a1: and2 PORT MAP (node_p, node_q, node_s);--d1 input logic
a2: and2 PORT MAP (node_w, x_in, node_t);
a3: and2 PORT MAP (node_l, node_m, node_u);
o1: or2 PORT MAP (node_u, node_s, node_w);
a4: and2 PORT MAP (node_m, node_p, node_v);
a5: and2 PORT MAP (node_v, node_o, node_x);
o2: or2 PORT MAP (node_x, node_t, node_y);
a6: and2 PORT MAP (node_l, node_q, node_z);--d2 input logic
a7: and2 PORT MAP (node_z, x_in, node_a);
o3: or2 PORT MAP (node_a, node_v, node_b);
a8: and2 PORT MAP (node_z, node_o, node_c);--d3 input logic
a12: and2 PORT MAP (node_p, node_n, node_d);
o4: or2 PORT MAP (node_u, node_d, node_i);
a13: and2 PORT MAP (node_i, x_in, node_j);
o7: or2 PORT MAP (node_j, node_c, node_k);
o5: or2 PORT MAP (node_d, node_c, node_e);--d4 input logic
a9: and2 PORT MAP (node_m, node_r, node_f);
a10: and2 PORT MAP (node_q, node_n, node_g);
o6: or2 PORT MAP (node_f, node_g, node_h);--output logic
a11: and2 PORT MAP (node_h, node_l, z_out);
END moore_encoded;

```

Figure 14. Structural Description of the Moore Encoded Detector

5.2 ONE-HOT STATE ASSIGNMENT MODEL

For the *one-hot* state assignment implementation, the state assignment 000001, 000010, 000100, 001000, 010000, and 100000 is used for states *a, b, c, d, e* and *f* respectively. This design implements one flip-flop per state for a total of six D-type set/reset flip-flops. As in the *moore_encoded* AND gates and OR gates are used for implementing the Boolean equations of the states and output of *moore_onehot*. The VHDL architecture specification, *moore_onehot* of the *detector*, is shown in Figure 16.

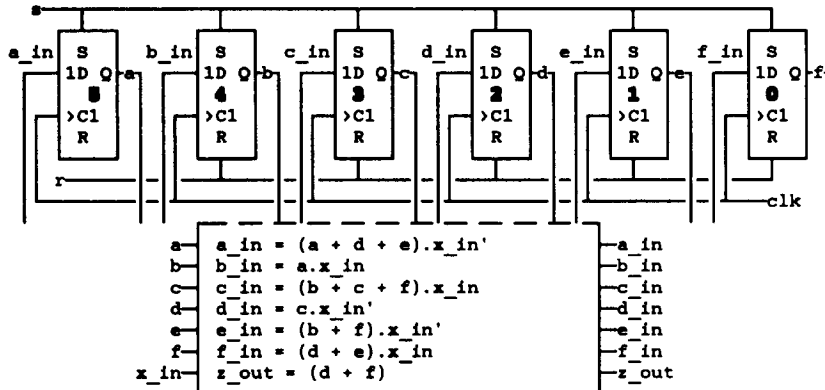


Figure 15. One-Hot State Assignment Implementation

```

ARCHITECTURE moore_onehot OF detector IS
-- ... same as Figure 8 ... --
SIGNAL s : BIT := '1';
SIGNAL r : BIT := '0';
SIGNAL node_a,node_b,node_c,node_d,node_e,node_f : BIT;
SIGNAL node_o,node_p,node_q,node_r,node_s,node_t,node_u : BIT;
SIGNAL node_l,node_m,node_n,node_v,node_w,node_x : BIT;
BEGIN
s <= '0' AFTER 25 NS;
d1: dff PORT MAP (node_n, s, r, clk, node_a);
d2: dff PORT MAP (node_o, r, s, clk, node_b);
d3: dff PORT MAP (node_r, r, s, clk, node_c);
d4: dff PORT MAP (node_s, r, s, clk, node_d);
d5: dff PORT MAP (node_u, r, s, clk, node_e);
d6: dff PORT MAP (node_w, r, s, clk, node_f);
i1: inv PORT MAP (x_in, node_p);
o1: or2 PORT MAP (node_d, node_e, node_l);--d1 input logic
o2: or2 PORT MAP (node_l, node_a, node_m);
a1: and2 PORT MAP (node_m, node_p, node_n);
a2: and2 PORT MAP (node_a, x_in, node_o);--d2 input logic
o3: or2 PORT MAP (node_b, node_c, node_q);--d3 input logic
o7: or2 PORT MAP (node_f, node_q, node_x);
a3: and2 PORT MAP (node_x, x_in, node_r);
a4: and2 PORT MAP (node_c, node_p, node_s);--d4 input logic
o4: or2 PORT MAP (node_b, node_f, node_t);
a5: and2 PORT MAP (node_t, node_p, node_u);--d5 input logic
o5: or2 PORT MAP (node_d, node_e, node_v);
a6: and2 PORT MAP (node_v, x_in, node_w);--d6 input logic
o6: or2 PORT MAP (node_d, node_f, z_out);--z_out output logic
END moore_onehot;

```

Figure 16. Structural Description of the Moore One-Hot Detector

5.3 TEST BENCH AND SIMULATION IN VHDL

The *test_bench* for the three Moore machine implementations is similar to that used for the Mealy machine in section 5.3, with the exception that *mealy* is replaced with *moore* in the configuration specifications. This *test_bench* provided the simulation run as shown in Figure 17.

TIME (NS)	SIGNAL NAMES					TIME (NS)	SIGNAL NAMES				
	C	X	Z	Z	Z		C	X	Z	Z	Z
			\bar{M}	\bar{E}	\bar{O}				\bar{M}	\bar{E}	\bar{O}
			A	N	N				A	N	N
			C	C	E				C	C	E
			H	O	H				H	O	H
			I	D	O				I	D	O
			N	E	T				N	E	T
			E	D					E	D	
0	'0'	'0'	'0'	'0'	'0'	440	...	'1'
40	'1'	480	...	'0'
45	'0'	500
50	...	'1'	+1	'1'
100	+3	'1'
+1	'1'	505	'0'
105	'0'	522	'1'	'1'
150	...	'0'	550	...	'1'
200	560	...	'0'
+1	'1'	570	...	'1'
205	'0'	600
250	...	'1'	+1	'1'
300	605	'0'
+1	'1'	628	'0'	...
+3	'1'	639	'1'	...
305	'0'	670	...	'0'
322	'1'	'1'	700
400	+1	'1'
+1	'1'	+3	'0'
+3	'0'	705	'0'
405	'0'	718	'0'	'0'
410	...	'0'
418	'0'	'0'	

Figure 17. Detailed Moore Simulation Run

5.4 SIMULATION ANALYSIS

As in the Mealy simulation run, the general pattern of the outputs of the Moore simulation run, seen in Figure 17, at the behavioral level (*moore*) and the structural levels (*moore_encoded*, *moore_onehot*) are the same. Varying from the inconsistencies

of the Mealy machine implementations, the output event delays are consistent for both the *mealy_encoded* and *mealy_onehot* implementations. When the output makes a transition from '0' to '1' as seen in Figure 17 at 300 NS, a delay of 22 NS is viewed. Whereas a delay of 18 NS is viewed at time 400 NS, when the output makes a transitions from '1' to '0'. These delays are dependent upon the flip-flop delay (rising=15 NS, falling=13 NS) and the ORing logic (rising=7 NS, falling=5 NS) for the output, *z_out*.

The hazard issue results from the encoding of states, which is a major difference in simulation between *moore_encoded* and *moore_onehot*. This hazard causes an 11 NS glitch in the output, *z_encoded*, at time 628 NS. With the encoded implementation, there is always the possibility of hazards, which cause glitches to occur when more than one bit changes in the state. In this case, the glitch occurs when the encoded state assignment makes the transition from state d (011) to state b (001) before continuing on to the stable state of f (101).

In conclusion, the *moore_encoded* and *moore_onehot* models provide consistent delays, which are only dependent upon the state change and not upon input changes. The *moore_onehot* implementation is preferred due to the removal of hazard issues. This simulation run indicates that by inspecting the output the exact delay can be predicted, and therefore back annotated to the behavioral *moore* model.

5.5 BACK ANNOTATING *moore_sequence_detector* MODEL

For back annotating behavioral descriptions with information from gate level simulation, the issues that will be dealt with are *delays* and *hazards*. Hazards can be disregarded, when limiting the hardware implementation to one flip-flop per state. Therefore, the *moore* description of the *sequence_detector* example will be back-annotated with the timing information from the *moore_onehot* model. For this purpose the discrepancies found between *z_machine* and *z_onehot* in the simulation run of Figure 17 section will be used.

Due to the flip-flop and gate delays, the *z_onehot* output lags behind the *z_machine* by 18 NS to 22 NS. The amount of delay depends on the rising and falling output delays only, and these can easily be back annotated to the *z_machine* output of the *moore* model. The code necessary for this purpose is shown in Figure 18. The addition of AFTER clauses to the *z_out* concurrent assignment causes delays of specified times on this output. For general state machines with more output lines the same VHDL constructs can be used.

```

ARCHITECTURE new_moore OF detector IS
  -- ... same as Figure 5 ... --
BEGIN
  machine : BLOCK (clk = '1' AND NOT clk'STABLE)
  BEGIN
    -- ... case statement ... --
    -- ... same as Figure 5 ... --
    z_out <= '1' AFTER 22 NS WHEN ( current = d OR current = f ) ELSE '0' AFTER 18 NS;
  END_BLOCK machine;
END new_moore;

```

Figure 18. Back-Annotated Behavioral Description of the Moore Sequence Detector

Back-annotating behavioral level model of a Moore state machine makes the new model more useful for simulation purposes. The simulation of the new behavioral model shown in Figure 18 was obtained using the same *test_bench* and *scale* value as previous simulations, and the results are shown in Figure 19. Simulation results of Figure 19 indicate that the *new_moore* model functions exactly like the actual hardware model.

TIME (NS)	-----SIGNAL NAMES-----					TIME (NS)	-----SIGNAL NAMES-----				
	C	X	Z	Z	Z		C	X	Z	Z	Z
			\bar{M}	\bar{E}	\bar{O}				\bar{M}	\bar{E}	\bar{O}
			A	N	N				A	N	N
			C	C	E				C	C	E
			H	O	H				H	O	H
			I	D	O				I	D	O
			N	E	T				N	E	T
			E	D					E	D	
0	'0'	'0'	'0'	'0'	'0'	440	...	'1'
40	'1'	480	...	'0'
45	'0'	500					
50	...	'1'	+1	'1'
100						+3
+1	'1'	505	'0'
105	'0'	522	'1'	'1'	'1'
150	...	'0'	550	...	'1'
200						560	...	'0'
+1	'1'	570	...	'1'
205	'0'	600					
250	...	'1'	+1	'1'
300						605	'0'
+1	'1'	628	'0'	...
+3	639	'1'	...
305	'0'	670	...	'0'
322	'1'	'1'	'1'	700					
400						+1	'1'
+1	'1'	+3
+3	705	'0'
405	'0'	718	'0'	'0'	'0'
410	...	'0'						
418	'0'	'0'	'0'						

Figure 19. Simulation Run of Back-Annotated Description

6. CONCLUSIONS

We have shown how VHDL can be used to model Mealy and Moore state machines. The discrepancies between the results of simulating first the mealy machine, and second the Moore machine were analyzed and back annotation of the behavioral model of the Moore machine with information obtained from the gate level was presented. It was observed that back annotation can be done with the Mealy machine using average or worst case timing, but this method is not desirable when exactness is an issue. Upon review of the Moore machine simulation run, the conclusion made is that exact structural timing can be inserted into the behavioral model provided a *onehot* state assignment is used. This restriction is due to hazards in the architecture when *encoded* state assignments are used. In the case of hazards, programming them into the behavioral is complex, and requires exhaustive simulation run for finding all hazards.

Within the restrictions, we inserted timing information into the behavioral description and compared the simulation results with the results obtained from simulating the original behavioral model. The back annotated model was as accurate as the gate level model. These results show that back annotation of behavioral models can be provided, and the exact timing can be back annotated, when FSMs are restricted to Moore models, implemented with *onehot* state assignments.

REFERENCES

- [1] "IEEE Standard VHDL Language Reference Manual", **IEEE Standard 1076-1987**, The Institute of Electrical and Electronic Engineers, Inc., 1988.
- [2] Lipsett, L., Carl Schaefer, and Cary Ussery, **VHDL: Hardware Description and Design**, Kluwer Academic Publishing, Boston, 1988.
- [3] Zainalabedin Navabi, and John Spillane, **Templates For Synthesis From VHDL**, Proceedings of the 1990 ASIC Seminar and Exposition, September 1990.
- [4] Steve Carlson, **Introduction to HDL-Based Design Using VHDL**, Synopsys, Inc. Mountain View CA, 1990.
- [5] Z. Navabi and M. Massoumi "Investigating simulation of hardware at various levels of abstraction and timing back-annotation of dataflow descriptions,"**The Journal of Simulation**, The Society for Computer Simulation, November 1991, pp. 321-332.