

Modeling Techniques for Simulation, Test and Documentation of Digital Components

Zinalabedin Navabi
Electrical and Computer Engineering Department
Campus# 2, University of Tehran
North Kargar Avenue
14399 Tehran, IRAN
navabi_z@irearn.bitnet

Maghsoud Abbaspour, Farzin Karimi, Mohammad Serjoui, Mirsalam Teimouri
ECE Department, University of Tehran

Massoud Eghtesad
Viewlogic Systems Inc.

Ladan Aminzadeh
Parstel Telecommunication Inc.

ABSTRACT

In this paper we will present techniques for simulation, test and documentation of digital circuits. The techniques present VHDL coding styles for modeling characteristics of digital components for the purpose of simulation, test, verification and documentation. The paper consists of five sub-sections, each describing an application and a VHDL coding style and the modeling technique that suits the application. The techniques presented here are for 1) BDD representation and lookup, 2) Reducing simulation time of flip-flops by use of alternative clocking schemes, 3) Detail logic simulation by considering rate of change of signals and a threshold value, 4) Developing a generic unconstrained state machine model, and 5) Modeling for critical path tracing for test and fault simulation applications.

1. BDD representation and lookup

BY: Farzin Karimi and Mirsalam Teimouri

A logic circuit can be represented by a Binary Decision Diagram (BDD). BDDs can be used for simulation, synthesis and testing of Boolean circuits. Input variables of a logic function are used to label nodes of a BDD. In an ordered BDD a consistent ordering exists in the way nodes of the diagram are labeled. OBDD's have received the most attention in design verification. A Boolean circuit can be translated to its equivalent OBDD. The size of a BDD determines its feasibility for use in synthesis, where time to generate it determines how well it is suited for verification

purposes. Because a BDD presents a compact tabular representation of a Boolean circuit, it can be generated to replace a large part of a combinational circuit for simulation purposes.

In VHDL, a netlist of gates with interconnecting signals can be replaced by a corresponding OBDD. In a fanout free circuit, adding all path delays and associating them with inputs of a circuit, produces the same timing accuracy as the circuit with distributed delays. Therefore, annotating a BDD with such delay values generates a circuit model that is as accurate as the original circuit.

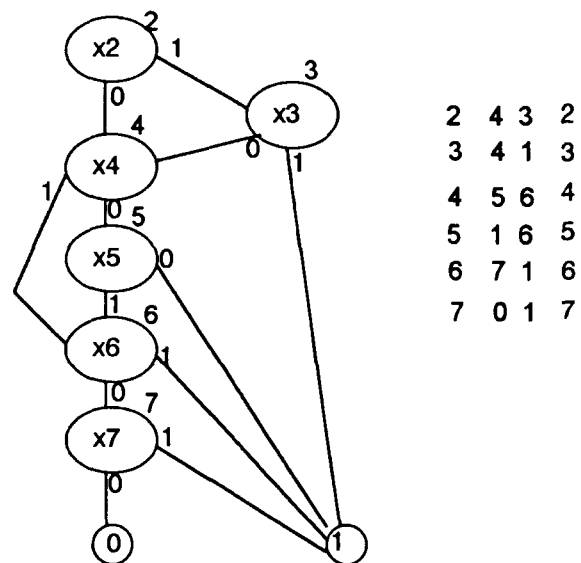


Figure 1.1 A BDD and its tabular representation

1.1 Tabular representation

We will show a method of representing a general BDD in VHDL. This representation is simulatable and can easily be generated from gate level netlist of the circuit. In our representation, every node of a BDD can be represented by an *index*, 0 and 1 transition indices, and an *input_index*. The *index* identifies the node, the transition indices specify nodes to branch to when the input is 0 or 1, and the *input_index* represents the input variable. Figure 1.1 shows an example BDD and its corresponding tabular representation.

1.2 VHDL Implementation

Figure 1.2 shows a package declaration and body for representation of BDDs in VHDL. In this implementation, only consistency with the logical operation of the original circuit is considered and timing values are not considered.

```
PACKAGE bdd_definitions IS
  TYPE bit_indexed_integers IS ARRAY (BIT) OF INTEGER;
  TYPE bdd_entry IS RECORD
    transitions : bit_indexed_integers;
    input_index : INTEGER;
  END RECORD;
  TYPE bdd_table IS ARRAY
    (NATURAL RANGE <>) OF bdd_entry;
  TYPE integer_indexed_bit IS ARRAY (0 TO 1) OF BIT;
  CONSTANT int2bit : integer_indexed_bit := ('0', '1');
  PROCEDURE lookup_bdd
    (f_bdd : bdd_table; in_vals : BIT_VECTOR;
     SIGNAL z : OUT BIT; delay : TIME);
END bdd_definitions;
--
PACKAGE BODY bdd_definitions IS
  PROCEDURE lookup_bdd
    (f_bdd : bdd_table; in_vals : BIT_VECTOR;
     SIGNAL z : OUT BIT; delay : TIME) IS
    VARIABLE t : INTEGER;
  BEGIN
    t := 2;
    WHILE t >= 2 LOOP
      t := f_bdd(t).transitions(in_vals((f_bdd(t).input_index)-2));
    END LOOP;
    z <= int2bit (t) AFTER delay;
  END lookup_bdd;
END bdd_definitions;
```

Figure 1.2 VHDL package for BDD implementation

Each BDD entry is a record of transition indices and the input index. The BDD table is an unconstrained array of such records. The first index of the BDD table is 2. This leaves index values 0 and 1 to correspond to logic values 0 and 1. A lookup procedure searches its input BDD table until it reaches values 0 or 1. These are the logical values of the function for the given input values and are assigned to the output signal of the lookup procedure. The input

values are associated with the *in_vals* parameter of the procedure, and are ordered according to the index of the inputs in the BDD table.

1.3 Modeling Logical circuits

Figure 1.3 shows the VHDL description for the BDD of Figure 1.1. A constant table defines the structure of the corresponding BDD. The indices of this table are the node labels, the first two entries specify the 0 and 1 branches, and the last entry is the index of the input variable corresponding to the node. The concurrent procedure call in the statement part of the architecture of Figure 1.3 traces the table and assigns a BIT type value to the output of the circuit.

```
USE WORK.bdd_definitions.ALL;
ENTITY fig7bdd IS
  -- Fig 7 IEEE Trans on Comp, Vol 42, No. 2 Feb 93, S.
  Chakravarty
  PORT (x1, x2, x3, x4, x5, x6 : IN BIT; z : OUT BIT);
END fig7bdd;
--
ARCHITECTURE bdd OF fig7bdd IS
  CONSTANT fig7_bdd :
    bdd_table (2 TO 7) := (
      2 => ((4, 3), 2),
      3 => ((4, 1), 3),
      4 => ((5, 6), 4),
      5 => ((1, 6), 5),
      6 => ((7, 1), 6),
      7 => ((0, 1), 7)
    );
  BEGIN
    lookup_bdd (fig7_bdd, x1 & x2 & x3 & x4 & x5 & x6, z, 5 NS);
  END bdd;
```

Figure 1.3 VHDL model for BDD of Figure 1.1

The technique illustrated here also applies to circuits with several outputs or circuits with fanout. In such cases, a separate BDD constant table must be defined for each part of the circuit. BDD tables can use signals internal to an architecture or the ports of the architecture. The constant table of Figure 1.3 has been automatically generated from a VHDL gate level netlist. Simulation of the description of Figure 1.3 is about 30 times faster than its corresponding gate level netlist.

2. Flip-flop models for simulation performance

BY: Mirsalam Teimouri

Clocked sequential circuits are the most common type of sequential circuits used in the design of digital systems. In most cases, a periodic signal is used for clocking purposes. This signal is distributed to most parts of a circuit and triggers flip-flops of the circuit.

When triggered, flip-flops read their input data values, and load their outputs accordingly. The following observations apply to a large class of digital circuits.

1. Although a clock passes through the clock distribution logic, the same signal values arrive at all flip-flops. Therefore, simulating the clock distribution logic may be unnecessary.
2. In most circuits data variations are less frequent than those of the clock. Therefore, triggering a flip-flop by the clock causes the flip-flop to unnecessarily load its output with a data that may have not changed for many clock cycles.
3. In most circuits the clock signal is directly connected to all flip-flops, and enabling is done by the flip-flop itself. Therefore, the time of triggering, and not the actual triggering signal, is what is needed for a flip-flop to trigger.

Based on the above observations, the following strategies can be used for improving the simulation performance of flip-flop circuits.

1. Prioritizing data over clock. This scheme makes a flip-flop look at the clock only when it is certain that data has changed.
2. Suppressing the clock signal and using fixed clock timing. This scheme eliminates the unnecessary simulation of clock distribution logic, and removes the clock signal as an activating input of the flip-flops.

We have developed VHDL flip-flop models for prioritizing data and for clock suppression. Simulation of these models and comparing the simulation times with standard VHDL flip-flop models will be presented in the sections that follow. For comparisons, we will use the ISCAS-85 benchmark circuits.

2.1 Flip-flop models

Figure 2.1 shows four architectures for a clocked D-type flip-flop. The main difference in the four models is in the way each handles the clock. To be able to compare simulation performances based on the clocking scheme, timing details and timing checks are not included in the flip-flop models.

The *guarding* and *assigning* architectures use the clock as the main triggering signal. The *guarding* architecture uses a guarded signal assignment for assigning data values to the output of the flip-flop,

where the *assigning* architecture uses a signal assignment for this purpose.

The *prioritizing* architecture of the D-type flip-flop is sensitive to the events on the data input. When an event occurs on the data input, the edge of the clock is searched for. The data will be transferred into the flip-flop on the edge of the clock only after data changes. The *guarding*, *assigning*, and *prioritizing* architectures of the flip-flop use the same entity declaration.

```
ENTITY dff IS PORT (d, clk : IN BIT; q : OUT BIT); END dff;
```

```
ARCHITECTURE guarding OF dff IS
BEGIN
  clocking: BLOCK (clk = '1' AND NOT clk'STABLE)
  BEGIN
    q <= GUARDED d;
  END BLOCK;
END guarding;
```

```
ARCHITECTURE assigning OF dff IS
  SIGNAL temp:BIT;
BEGIN
  temp <= d
    WHEN (clk='1' AND (NOT clk'STABLE)) ELSE temp;
  q <= temp;
END assigning;
```

```
ARCHITECTURE prioritizing OF dff IS
BEGIN
  PROCESS
  BEGIN
    WAIT ON d;
    WAIT UNTIL clk = '1';
    q <= d;
  END PROCESS;
END prioritizing;
```

```
ENTITY dff IS
  GENERIC (period: TIME);
  PORT (d: IN BIT; q: OUT BIT)
END dff;
ARCHITECTURE suppressing OF dff IS
BEGIN
  PROCESS
  BEGIN
    WAIT ON d;
    WAIT FOR (NOW / period) * period + period - NOW;
    q <= d;
  END PROCESS;
END suppressing;
```

Figure 2.1 Architectures for a clocked D-type flip-flop

The *suppressing* architecture of the flip-flop does not use a clock signal. Instead, a generic parameter defines the period of the clock signal. As in the *prioritizing* architecture, this architecture is also sensitive to the data input. However, instead of waiting for an event of the clock signal, this

architecture waits for the completion of the clock period before it assigns data to its output. This waiting is based on the time data changes and fixed intervals of the clock period. The entity declaration of this architecture is different from that of the other models. The clock signal in this entity is replaced by a generic period.

Figure 2.2 shows simulation time in Seconds for 8 ISCAS benchmark circuits. Each circuit is identified by an *s* number. The number of primary inputs, primary outputs, D-type flip-flops, inverters and general logic gates of each circuit are shown in Figure 2.2. It is shown that the *suppressing* architecture has the best simulation run time. Model performance depends on the ratio of data input activities to clock activities. The examples shown in the table of Figure 2.2 were run with test inputs with a relatively high data input activity. For circuits with large number of flip-flops and low data input activity, the difference in performance of the *suppressing* architecture and *prioritizing* architecture becomes more significant. For some data, simulation of *s444* or *s526* using the *suppressing* flip-flop model runs up to ten times faster than when the *prioritizing* model is used.

Ckt	PI	PO	Dff	Inv	Gate	guar	assig	prior	supp
s1196	14	14	18	141	388	900	571	283	276
s27	4	1	3	2	8	366	274	199	109
s298	4	6	14	44	75	418	318	226	195
s444	3	6	21	62	119	75	60	43	40
s510	19	7	6	32	179	1475	391	187	139
s526	3	6	21	52	141	632	458	306	277
s713	35	23	19	254	139	174	90	16	13
s820	18	19	5	33	256	152	93	44	39

Figure 2.2 ISCAS-85 benchmarks: Simulation time of flip-flop models

Ckt	PI	PO	Dff	Inv	Gate	guar	assig	prior	supp
s1196	14	14	18	141	388	100	63	32	31
s27	4	1	3	2	8	100	75	54	30
s298	4	6	14	44	75	100	76	54	46
s444	3	6	21	62	119	100	80	57	53
s510	19	7	6	32	179	100	26	13	10
s526	3	6	21	52	141	100	72	48	44
s713	35	23	19	254	139	100	52	9	7
s820	18	19	5	33	256	100	61	29	26

Figure 2.3 Simulation time of models of Fig. 2.1

In Figure 2.3 simulation run times are shown with respect to the slowest model, the *guarding* architecture. In all cases the *suppressing* architecture has the best performance. In circuits with large number of flip-flops at least a 5X improvement is

obtained by using the *prioritizing* or the *suppressing* architectures.

3. Generic state-machines

BY: Ladan Aminzadeh

State machines constitute an important part of a digital system. Often a controller circuit is described by a state machine, and outputs of the controller state machine control the movement of data in the data unit. Correct description and operation of a system model heavily depends on the description of its controller. State machines can easily be described tabularly.

A tabular representation of a state machine is useful for synthesis and verification. In addition, such a representation can automatically be generated or extracted from other forms of circuit representation. This section shows an unconstrained generic state machine description in VHDL. The VHDL code of the model uses a constant table which defines the state transitions and output values. The only necessary customization is in defining the constant table.

```

ENTITY detector_m IS PORT (x, clk : IN BIT; z : OUT BIT);
END detector_m;
--
ARCHITECTURE multiple_moore_machine_1 OF detector_m IS
FUNCTION oring( drivers : BIT_VECTOR) RETURN BIT IS
  VARIABLE accumulate : BIT := '0';
BEGIN
  FOR i IN drivers'RANGE LOOP
    accumulate := accumulate OR drivers(i);
  END LOOP;
  RETURN accumulate;
END oring;
SUBTYPE ored_bit IS oring BIT;
TYPE ored_bit_vector IS
  ARRAY (NATURAL RANGE <>) OF ored_bit;
--
-- Fill in next_val, out_val, and s arrays
--
SIGNAL o : ored_bit REGISTER;
BEGIN
clocking : BLOCK (clk = '1' AND (NOT clk'STABLE))
BEGIN
  g: FOR i IN s'RANGE GENERATE
    si: BLOCK (s(i) = '1' AND GUARD)
    BEGIN
      s(next_val(i,'0')) <= GUARDED '1' WHEN x='0' ELSE '0';
      s(next_val(i,'1')) <= GUARDED '1' WHEN x='1' ELSE '0';
      o <= GUARDED out_val(i, x);
    END BLOCK si;
    s (i) <= GUARDED '0';
  END GENERATE;
END BLOCK clocking;
z <= o;
END multiple_moore_machine_1;

```

Figure 3.1 General state machine VHDL description

3.1 VHDL modeling

Figure 3.1 shows VHDL description for a Moore state machine. This description can be used for any number of states or transitions. The coding allows any number of the states of the machine to be active simultaneously. Although this description has a one bit output, it can easily be modified for an unconstrained output vector.

The states of the machine are defined by a resolved array using the *oring* resolution function. Assignments to the states of the machine are done within block statements, one of which is generated for every state of the machine. The part of the code that is generated for every state of the state machine is highlighted in Figure 3.1.

Since state signals are guarded of REGISTER kind, states stay active between clock pulses. In order for states not activated by other states to reset to non-active mode, a '0' is assigned to all state signals with every clock edge. This value causes the resolution function to return '0' for non-active states. The placement of '0' on the driver of a signal of an active state keeps its value unchanged.

Figure 3.2 shows next state and output tables for defining a particular state machine to be described by the partial code of Figure 3.1. These tables can be defined in a package or in the declarative part of the architecture of Figure 3.1.

```

TYPE next_table IS ARRAY (1 TO 6, BIT) OF INTEGER;
TYPE out_table IS ARRAY (1 TO 6, BIT) OF BIT;
-----
--These tables program the generic Moore description--
-----
--
-- Next States: ----- x=0, x=1 --
CONSTANT next_val : next_table := (
  (1 , 2), --S1: -> S1, S2 --
  (1 , 3), --S2: -> S1, S3 --
  (1 , 4), --S3: -> S1, S4 --
  (1 , 1), --S4: -> S1, S1 --
  (5 , 6), --S5: -> S5, S6 --
  (5 , 6) ); --S6: -> S5, S6 --
--
-- Output Values: ----- x=0, x=1 --
CONSTANT out_val : out_table := (
  ('0' , '0'), --S1: == z=0, 0 --
  ('0' , '0'), --S2: == z=0, 0 --
  ('0' , '0'), --S3: == z=0, 0 --
  ('0' , '0'), --S4: == z=1, 1 --
  ('0' , '0'), --S5: == z=0, 0 --
  ('1' , '1') ); --S6: == z=1, 1 --
--
-- Initial Active States: --
SIGNAL s : ored_bit_vector (1 TO 6) REGISTER := "100010";
--

```

Figure 3.2 Tables for defining a state machine

4. Linear waveform dependency

BY: Massoud Eghtesad

For a more accurate simulation of logical circuits, gate models that are dependent on their input waveform should be used. Input dependency can be modeled in several ways. We have chosen a simple linear method for modeling gates, in which timing of a gate is dependent on the logic level and rate of change of its inputs.

Each gate model has its fixed pull-up and pull-down resistance values. The total load capacitance at the output of the gate is calculated by a resolution function and is reported to the capacitance field of the INOUT output of the gate. Circuit nodes can take one of eleven values between 0 and A. A fixed threshold value determines the logical response of a gate to its inputs. Input values above the threshold have positive effect on the output, and values below the threshold have a negative effect on the output. The speed of the input waveform influences that of the output in reaching its final value.

The models use a package of types and subprograms. Using these utilities, a general structure for modeling gate level components will be demonstrated. We will use a two-input NAND gate for demonstrating linear modeling techniques.

```

PACKAGE linear_logic IS
TYPE discrete IS ('0','1','2','3','4','5','6','7','8','9','A');
TYPE discrete_vector IS
  ARRAY(INTEGER RANGE <>)OF discrete;
TYPE cases IS (oo,ou,uo,uu);
--o: Over threshold; u: Under threshold
TYPE direction IS (up,down,nop); -- output signal direction
..
FUNCTION
  equivalent (sources : capacitance_vector)RETURN capacitance;
SUBTYPE equivalent_cap IS equivalent capacitance;
FUNCTION wiring (drivers:discrete_vector)RETURN discrete;
SUBTYPE wired_discrete IS wiring discrete;
TYPE node IS RECORD
  logic : wired_discrete;
  cap : equivalent_cap;
END RECORD;
END linear_logic;

```

Figure 4.1 Package declaration for linear logic models

4.1 Modeling Principles

Figure 4.1 shows the declaration of the package used by the linear gate models. All circuit nodes are of type *node*. This type has a field for logical values which is a resolved type of *discrete*, and another field for capacitance values which is a resolved type of

capacitance. A *wiring* resolution function is used for the logical field, and an *equivalent* resolution function is used for the load capacitance field of a signal. The logic field of a *node* takes one of eleven values between '0' and 'A'.

```

PACKAGE BODY linear_logic IS
...
FUNCTION input_status (i, i_last, th : discrete) RETURN cases IS
  VARIABLE temp : cases := oo;
BEGIN
  IF discrete'POS(i_last) >= discrete'POS(th)
    AND discrete'POS(i) < discrete'POS(th) THEN temp := ou;
  ELSIF discrete'POS(i_last) >= discrete'POS(th)
    AND discrete'POS(i) > discrete'POS(th) THEN temp :=oo;
  ELSIF discrete'POS(i_last) <= discrete'POS(th)
    AND discrete'POS(i) < discrete'POS(th) THEN temp :=uu;
  ELSIF discrete'POS(i_last) <= discrete'POS(th)
    AND discrete'POS(i) > discrete'POS(th) THEN temp :=uo;
  END IF;
  RETURN temp;
END input_status;

FUNCTION nand_output_direction (i1, i2 : cases)
  RETURN direction IS
  CONSTANT cases_direction_table : cases_2d :=
    --oo --ou --uo --uu
    (oo => ( nop, up, down, up),
     ou => ( up, up, up, up),
     uo => (down, up, down, up),
     uu => ( up, up, up, nop));
BEGIN
  RETURN cases_direction_table(i1,i2);
END nand_output_direction;
END linear_logic;

```

Figure 4.2 Package body for linear logic models

Figure 4.2 shows the body of the *linear_logic* package. Based on a preset threshold value and the present and previous values of a signal, the *input_status* function determines the status of an input with respect to the threshold. The values returned by this function can be any of the values of the *cases* enumeration type of Figure 4.1. A function that determines the direction of the output of a logical function is developed for each logical operator. Figure 4.2 shows the *nand_output_direction*. Based on the status of the operands of NAND operation, the direction of the NAND output is determined by this function.

Figure 4.3 shows the VHDL description of a NAND gate. The pull-up, pull-down, and the threshold values are assumed to be fixed. For load dependent timing calculations, each port of the NAND gate reports its capacitance value to its *cap* field. The value read on the *cap* field of the output port includes the total capacitance at this node.

The *outing* process in the architecture of the two-input NAND gate, is sensitive to the logic field of the input signals. In this process, the direction of output is calculated based on the status of the inputs. Depending on this direction, ascending or descending values of the *discrete* type will be scheduled on the output of the NAND. The values are separated in time by the delay that is calculated by the multiplication of pull resistance values and the capacitance seen on the output.

```

ENTITY nand2 IS
  PORT (i1, i2 : INOUT node; o1: INOUT node);
  CONSTANT threshold : discrete :='5';
  CONSTANT rpd : resistance := 15 k_o;
  CONSTANT rpu : resistance := 10 k_o;
END nand2;
--
ARCHITECTURE linear OF nand2 IS
  SIGNAL out_val : discrete :='5';
BEGIN
  outing: PROCESS (i1.logic, i2.logic)
    VARIABLE dir : direction;
    VARIABLE rise_delay, fall_delay : TIME;
  BEGIN
    dir := nand_output_direction
      (input_status (i1.logic, i1.logic'LAST_VALUE, threshold),
       input_status (i2.logic, i2.logic'LAST_VALUE, threshold));
    IF (dir = up) THEN
      rise_delay := rpu * o1.cap * 3;
      FOR i IN discrete'POS (out_val) TO 10 LOOP
        out_val <= TRANSPORT discrete'VAL (i)
          AFTER rise_delay * (i-(discrete'POS(out_val)));
      END LOOP;
    ELSIF (dir=down) THEN
      fall_delay := rpd * o1.cap * 3;
      FOR i IN discrete'POS (out_val)-1 DOWNTO 0 LOOP
        out_val <= TRANSPORT discrete'VAL (i)
          AFTER fall_delay * ((discrete'POS(out_val))-i);
      END LOOP;
    END IF;
  END PROCESS outing;
  o1 <= (out_val, 50 ffr);
  i1 <= ('0', 160 ffr);
  i2 <= ('0', 160 ffr);
END linear;

```

Figure 4.3 Linear modeling of a 2-input NAND gate

Figure 4.4 Shows a sample output of the NAND gate when the inputs vary below and above the gate threshold. At 10000 PS the *i1* input crosses the threshold, and causes the output to change direction. The output changes linearly until it reaches at its final value after 22500 FS. At 20000 PS when the input changes again, the output starts going towards 'A', this trend is interrupted at 20009 PS when the same input crosses the threshold and causes the output to eventually go to '0' at 20022.5 PS. The value list in

this figure shows the dependency of the output on discrete input values and input direction.

fs	delta	i1	i2	o1
00000000	+0	(0,000)	(0,000)	(0,00)
00000000	+1	(3,160)	(8,160)	(5,50)
00001500	+1	(3,160)	(8,160)	(6,50)
00003000	+1	(3,160)	(8,160)	(7,50)
00004500	+1	(3,160)	(8,160)	(8,50)
00006000	+1	(3,160)	(8,160)	(9,50)
00007500	+1	(3,160)	(8,160)	(A,50)
10000000	+0	(8,160)	(8,160)	(A,50)
10002250	+1	(8,160)	(8,160)	(9,50)
10004500	+1	(8,160)	(8,160)	(8,50)
...				
10020250	+1	(8,160)	(8,160)	(1,50)
10022500	+1	(8,160)	(8,160)	(0,50)
20000000	+0	(1,160)	(8,160)	(0,50)
20001500	+1	(1,160)	(8,160)	(1,50)
20003000	+1	(1,160)	(8,160)	(2,50)
20004500	+1	(1,160)	(8,160)	(3,50)
20006000	+1	(1,160)	(8,160)	(4,50)
20007500	+1	(1,160)	(8,160)	(5,50)
20009000	+0	(9,160)	(8,160)	(5,50)
20009000	+1	(9,160)	(8,160)	(6,50)
20010500	+1	(9,160)	(8,160)	(7,50)
...				
20022500	+1	(9,160)	(8,160)	(0,50)

Figure 4.4 NAND output changes with the threshold

5. Modeling for critical path tracing

BY:Maghsoud Abbaspour and Mohammad Serjooei

This section presents a modeling style for Critical Path Tracing (CPT). CPT is a method of finding faults detected by a test vector in a combinational circuit, and can be used for fault simulation or test generation. For finding critical paths in a circuit, a test vector is applied to the primary inputs of the circuit. If changing value on line *l* from *v1* to *v2* toggles the value of a primary output, the line is said to be on a critical path. In this case, the input vector detects the stuck at *v2* fault on line *l*.

The major issues in critical path analysis are behavior of circuit gates for determining critical values at their inputs, and the behavior of fanouts for determining if their stems are critical. We have developed utilities and modeling style for modeling logical gates for critical path tracing. The simple modeling style presented here deals with gate models deciding on critical values of their inputs. Determination of critical value of a fanout stem has been simplified in our models. Our models are intended to demonstrate capabilities of VHDL for dealing with the CPT and other test related problems.

5.1 VHDL modeling

Figure 5.1 shows a fanout free circuit that we will use to demonstrate the main tasks of a gate CPT model. Initially the 10111 sequence is applied to the primary input of the circuit. After this test vector propagates to the output node, analysis of the circuit begins from the primary output towards the primary inputs. In this backward analysis each gate examines each of its inputs and decides if an input is critical. A critical line at the output of a gate, e.g., value '1' at the output of the AND gate, causes that gate to examine its inputs for critical values. This process continues until critical values reach primary inputs.

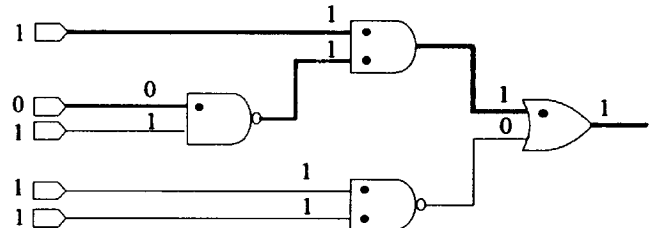


Figure 5.1 An example circuit for CPT

Based on its output and input values as well as its logical function, a VHDL gate model determines its critical inputs. The gate model will then write this information to an external file. All backward calculation are done in zero real time during several delta cycles that depend on the depth of the circuit. In order for critical path information written to an external file to be sequenced, and be identified as being associated with a certain gate, each gate model waits for an amount time that is related to its identification number before writing its critical information. This way, gate number *n* will write its information before gate *n+1*.

5.1 CPT package

Figure 5.2 shows the package used by the VHDL gate models for critical path tracing. The *logic_p_state* type, declared in this package is the type of all gate input and output ports. The resolved subtype of this type is used for interconnection of gate ports. Each signal value has three fields. The *p_state* field specifies if a circuit line of critical or non critical, i.e., *NCP* or *CP*. The *fanout* field lets the gate outputs know if they are connected to more than one gate input.

For every gate type a function determines critical values on the inputs of the gate. Such a function has been developed for two or three input AND, NAND, OR, and NOR gates. The function for

2-input AND and NAND gates is *AndNand2_Cp_val*. Figure 5.3 shows the table lookup performed by this function. If output of an AND gate is critical and its inputs are 11, then both inputs will have to be assigned CP values, indicating they are on a critical path.

```

PACKAGE Cpt_package IS
TYPE P_stateType IS (NCP, CP);
TYPE Ulogic_p_state IS RECORD
  Logic : BIT;
  P_state : P_stateType;
  Fanout : BOOLEAN;
END RECORD;
TYPE Ulogic_p_state_array
  IS ARRAY (natural range<>)of Ulogic_p_state;
TYPE Cp_pair IS RECORD
  in1 : P_stateType;
  in2 : P_stateType;
END RECORD;
FUNCTION AndNand2_Cp_val(i1, i2 : IN BIT) return Cp_pair;
...
FUNCTION OrNor2_Cp_val(i1, i2 : IN BIT) return Cp_pair;
...
FUNCTION one_of(drivers : Ulogic_p_state_array)
  RETURN Ulogic_p_state;
SUBTYPE Logic_p_state IS One_of Ulogic_p_state;
TYPE Logic_p_state_array IS
  ARRAY (NATURAL RANGE <>) OF Logic_p_state;
PROCEDURE Init(CONSTANT str : STRING);
PROCEDURE P_O_report( ... );
PROCEDURE Append ( ... );
...
END Cpt_package;

```

Figure 5.2 VHDL CPT package

logic values		critical values	
i1	i2	i1	i2
1	1	CP	CP
0	0	NCP	NCP
1	0	NCP	CP
0	1	CP	NCP

Figure 5.3 Critical values of AND/NAND

Other subprograms of Figure 5.2 are used for generating the report of the critical path tracing analysis in an external file.

5.3 VHDL AND model

Figure 5.4 shows the partial code of a two-input AND gate. The AND2 gate model waits for changing of the *P_state* field of its output signal. When this occurs, if output is *NCP* then the gate model assigns *NCP* to all its inputs. Otherwise, the *AndNand2_Cp_val* function is called to calculate critical values for the inputs of the AND2 gate.

If a gate output has a fanout of more than one, the stem will be given the CP value. This simple

method may cause a stem at a re-convergent fanout with different inversion parities to be marked critical even when it is not on a critical path.

In the backward direction, all signal values are calculated and assigned to the signals at zero time. Since concurrent reporting to a file is not possible, each gate reports related faults in times proportional to its *gate_id*.

```

ENTITY And2_cp IS
  GENERIC( Gate_id : INTEGER );
  PORT (I1, I2, O : INOUT Ulogic_p_state );
END And2_cp;
--
ARCHITECTURE Cp_behav OF And2_cp IS
BEGIN
  PROCESS
    VARIABLE Temp : Cp_pair;
  BEGIN
    WAIT ON O.P_state;
    IF O.P_state = CP THEN
      Temp := AndNand2_Cp_val ( I1.logic,I2.logic );
      I1.p_state <= temp.in1; I2.p_state <= temp.in2;

      WAIT FOR Gate_id * 10 ns;
      -- Output Report
      IF ( O.Fanout = TRUE ) THEN
        -----REPORT STUCK AT VALUE OF STEM.
      END IF;
      -- Input Reports
      IF ( I1.P_state = CP ) THEN
        -----REPORT STUCK AT VALUE OF I1 INPUT.
      END IF;
      IF ( I2.P_state = CP ) THEN
        -----REPORT STUCK AT VALUE OF I2 INPUT.
      END IF;
    ELSE
      I1.p_state <= NCP; I2.p_state <= NCP;
    END IF;
  END PROCESS;
  O.logic <= I1.logic AND I2.logic;
END Cp_behav;

```

Figure 5.4 VHDL CPT AND gate model

The gate models are instantiated in a test bench that causes propagation of test vector values in the forward direction and evaluation of critical lines in the opposite direction. In the test bench, first a test vector is applied to inputs, so that logic values of all signals will be evaluated. After 1 FS, the *CP* value is assigned to the *P_state* of the primary output. This causes the output gate to start propagation of critical values in the direction of output to inputs of the circuit. While this propagation is going on, the report of faults detected by the initial test vector is being generated.

Figure 5.5 shows a sample input file used by a CPT test bench, and a partial report file generated by the CPT gate models. The report shows that application of 1011 results in the detection of stuck-at-

0 on the output of gate 3. It also shows that stuck-at-1 on this same line can be detected by application of 1100 test.

```

----- Input file -----
TEST_VECTORS
1011
1100
1000
1111
----- Output file -----
*GATE_NO.  PORT_NAME  SA_V
Test vector:1011
3          O          0
Test vector:1100
3          O          1
7          I2         1
8          I2         1
Test vector:1000
3          O          1
Test vector:1111
3          O          1
1          PI         0
2          PI         0
. . .

```

Figure 5.5 Sample input and output of CPT

6. Conclusions

This paper demonstrates that simple VHDL modeling techniques can be used for developing useful models for test, simulation, verification, synthesis, and documentation of digital systems. In some instances we showed complete models, but in the interest of saving space, in several cases we have shown partial VHDL codes. In most cases, however, the models shown here are simplified versions of more complete models. The examples of this paper only highlight the key modeling techniques used. In addition to the above modeling techniques, we have used VHDL for test generation, fault simulation, fault collapsing, power consumption calculation, and many other advanced CAD applications. We believe that there are simple VHDL solutions to many complex CAD applications. Details of the language should be learned in order to be able to take advantage of many powerful features of this modeling tool.

REFERENCES

Navabi, Z., "VHDL: Analysis and Modeling of Digital Systems," McGraw-Hill Publishing, Inc., New York, 1993.

Abramovici, M., M. A. Breuer, A.D. Friedman, "Digital System Testing and Testable Design," Computer Science Press, New York, 1990.

Navabi, Z. Hashemi, A., Eghtesad M., Vai, M., "Modeling Timing Behavior of Logic Circuits Using Piecewise Linear Models," Proceedings of 1993 CHDL Conference, Ottawa, Ontario, Canada.

Shadfar, M., Paymandoust, A., Navabi, Z., "New VHDL Based Critical Path Tracing Method for Fault Simulation," *Submitted for: Proceedings of 1995 ICEHDL Conference, Las Vegas Nevada.*