Network Modeling in VHDL

Stephen A. Bailey
Vantage Analysis Systems
Suite 200
42808 Christy Street
Fremont, CA 94538
uunet!vntage!sab

1. Abstract

This paper discusses modeling techniques that can be employed by designers to model network behavior. The techniques can be used to implement a wide range of network behaviors ranging from pre-layout signal delays to complex system-level queuing models.

2. Modeling Pre-layout Timing

The need to model timing easily, flexibly and accurately in VHDL is well recognized. In order to assess the possibilities, two modeling techniques were used to implement the default timing-estimation algorithm employed by RapidSim.[1] The following sections contain a short description of the RapidSim default timing-estimation algorithm and two network modeling techniques. Each of the modeling techniques use the same simple design model shown in Figure 2-1.

2.1. The RapidSim Default Timing-Estimation Algorithm

RapidSim is a gate level simulator developed by Valid Logic Systems and is now the property of Cadence via their recent merger. RapidSim allows users to define their own timing-estimation algorithm in C. A default timing-estimation algorithm is also provided. This algorithm was originally developed for RapidSim's predecessor -- ValidSim.[2,3]

The RapidSim default timing-estimation algorithm works by summing the capacitance loads at all the "stops" on a net. Stops are pins connected to the net -- input, output or bidirectional. The capacitance of the driving pin is NOT included. The net's capacitance is determined through a table lookup (provided by the user). The total number of stops on the net is used as the index into the net load table. The total net loading (net load plus all non-driving stop loads) determine the network delay. The following equation completely expresses this algorithm:

$$D_{net} = DF * (L_{ip} + L_{op} + L_w)$$

1 RapidSim is a registered trademark of Cadence Design Systems

2 ValidSim is a registered trademark of Cadence Design Systems

3 **Customizing the Timing Processor**, June 15, 1990, Cadence Design Systems

Where:

D_{net} : Total delay of the network
DF : Drive factor of the output pin
L_{ip} : Σ (input pin load factors)
L_{op} : [Σ (output pin load factors)] - Driving pin load factor
L_w : Load of the wire

The above algorithm is fully implemented in one of the examples below. Another example uses an approximation of the algorithm. The details of the approximation are explained in the appropriate section.

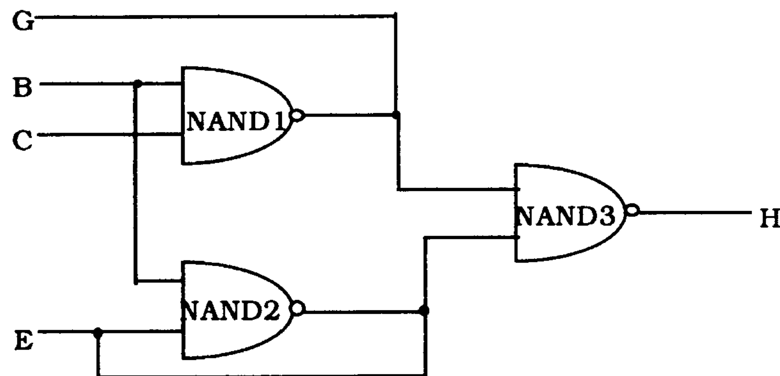

Figure 2-1. Demonstration Model for Delay-Estimation Techniques.

2.2. Modified RapidSim Default Timing-Estimation Algorithm

This approach to pre-layout timing estimation uses resolution functions. The timing-estimation algorithm is the default RapidSim algorithm with one modification. Since it is difficult to determine within the resolution function which driver was active (resulting in the activation of the resolution function), we simply subtract out the smallest of all the driving load capacitances and use the corresponding drive factor.

The key features of this technique are:

a) The resolution function is noticeably complex.

b) All stops on the network are drivers (in the VHDL-sense) of the net. This helps to get each network stops' load factor into the resolution function.

c) The type of the signal contains 4 fields -- driving/effective value, load factor, drivers (now used to determine whether the stop is a real driver or not) and the network delay as calculated at any resolved point in the signal's hierarchy.

d) Driven devices are more complex. Specifically, driven devices have local signals which serve as the mechanism for applying the network delay.

Example 2-1 presents the signal type and its associated resolution function.

```
package NET_PACK2 is
    type LOGIC_REC is record
       Value : STD_ULOGIC;
       Drivers : Natural;
        Load   : Real; -- Capacitance is in normalized units
        Drive : Real; -- Drive factor. Not used if not driving
    end record;
    type MY_ULOGIC is record
        Value_Data : Logic_Rec;
       Net_delay : Time;
    end record;
    ...
      type MY_ULOGIC_VECTOR is array(Natural range <>) of MY_ULOGIC;
      function Wired_Or( Drivers : in MY_ULOGIC_VECTOR ) return MY_ULOGIC;
      subtype RESOR_MY_ULOGIC is Wired_Or MY_ULOGIC;
end NET_PACK2;
package body NET_PACK2 is
    ...
  function Wired_Or( Drivers : in MY_ULOGIC_VECTOR ) return MY_ULOGIC is
   variable Effective_Value : MY_ULOGIC := (('0', 0, 0.0), 0 ns);
   variable Output_Pin_Load : Real := 0.0;
   variable Input_Pins_Load : Real := 0.0;
   variable Min_Load : Real := 0.0;
   variable Drive_Factor : Real := 0.1;
 begin
   for I in Drivers'Range loop
     if Drivers(I).Value_Data.Drivers /= 0 then
        Effective_Value.Value_Data.Value := Effective_Value.Value_Data.Value OR
       Drivers(I).Value_Data.Value;
   end if;
    if Drivers(I).Value_Data.Value /= 'Z' then
        Effective_Value.Value_Data.Drivers :=   Effective_Value.Value_Data.Drivers +
       Drivers(I).Value_Data.Drivers;
   end if;
     if Drivers(I).Value_Data.Drivers > 0 then
      if Output_Pin_Load = 0.0 then
        Min_Load := Drivers(I).Value_Data.Load;
        Drive_Factor := Drivers(I).Value_Data.Drive;
      elsif Drivers(I).Value_Data.Load < Min_Load then
         Min_Load := Drivers(I).Value_Data.Load;
         Drive_Factor := Drivers(I).Value_Data.Drive;
    end if;
       Output_Pin_Load := Drivers(I).Value_Data.Load;
   else
        Input_Pins_Load := Input_Pins_Load + Drivers(I).Value_Data.Load;
   end if;
  end loop;
    Effective_Value.Value_Data.Load := Input_Pins_Load + Output_Pin_Load -
       Min_Load + Net_Load_Lookup(Drivers'Length);
    Effective_Value.Value_Data.Drive := Drive_Factor;
   Effective_Value.Net_Delay := Drive_Factor * (Effective_Value.Value_Data.Load * 1 ns);
  if Effective_Value.Net_Delay < 0 ns then
    Effective_Value.Net_Delay := 0 ns;
  end if;
   return Effective_Value;
 end Wired_Or;    end NET_PACK2;
```

Example 2-1. RapidSim Default Algorithm in Resolution Function

The details of the driven devices are exemplified here.

```
entity NAND_GATE is
  generic ( IN1_LF : Real := 0.5;
        IN2_LF : Real := 0.5;   OUTPUT_LF : Real := 1.0;
          Drive : Real := 5.0;  BODY_DELAY : TIME := 5 ns );
   port ( IN1 : inout RESOR_MY_ULOGIC := MU_U;
        IN2 : inout RESOR_MY_ULOGIC := MU_U;
        OUTPUT: out MY_ULOGIC := MU_U );
end NAND_GATE;
architecture PRIMITIVE of NAND_GATE is
        signal DELAYED_IN1 : STD_ULOGIC;
        signal DELAYED_IN2 : STD_ULOGIC;
begin
     MAIN : process   begin
        IN1.Value_Data.Value <= 'X'; -- value ignored
        IN1.Value_Data.Drivers <= 0; -- because Drivers is zero
        IN1.Value_Data.Load <= IN1_LF;
          IN1.Value_Data.Drive <= Drive;
        IN1.Net_Delay <= 0 ns; -- Need a driver for all subelems
        IN2.Value_Data.Value <= 'X';
        IN2.Value_Data.Drivers <= 0;
        IN2.Value_Data.Load <= IN2_LF;
          IN2.Value_Data.Drive <= Drive;
        IN2.Net_Delay <= 0 ns;
         OUTPUT.Value_Data.Drivers <= 1;
          OUTPUT.Value_Data.Load <= OUTPUT_LF;
     loop
           DELAYED_IN1 <= IN1.Value_Data.Value after IN1.Net_Delay;
           DELAYED_IN2 <= IN2.Value_Data.Value after IN2.Net_Delay;
          wait on IN1.Value_Data.Value, IN2.Value_Data.Value;
      end loop;
      end process MAIN;
          OUTPUT.Value_Data.Value <= (DELAYED_IN1 nand DELAYED_IN2) after BODY_DELAY;
end PRIMITIVE;
```

Example 2-2. Driven Devices for RapidSim-algorithm, Resolution Function Models.

The pros and cons of this timing-estimation modeling approach are:

CON: The WYSIWYG benefits of modeling in VHDL are partially lost now that IN mode ports must be changed to INOUT mode to provide load factors to the resolution function.

More signals are required, impacting performance. N signals are required where N is the number of inputs to driving devices (times the number of subelements) plus 5x the number of normal signals.

Modeling driven devices is slightly more complex.

Finally, only an approximation of the algorithm is possible since it is difficult to tell which driver(s) are active when the resolution function is called.

PRO: This approach is a close approximation of the default RapidSim timing-estimation algorithm.

The algorithm implementation is performed without extra VHDL components (see next section).

2.3. RapidSim Default Algorithm Implemented through Components

This approach to implementing a timing-estimation algorithm is radically differentiated from the first one. The key to this approach is the use of component instantiations of a network model that is connected to both the driver and the reader stops on the network. This network component performs both multiple driver resolution and network delay calculation.

The key features of this technique are:

a. No resolution function is required.

b. A new Network model entity/architecture is required.

c. The network delay is applied within the network model (not previously possible with the resolution function approach) and not within the driven devices.

d. The signal data structure is average in complexity relative to the previous example.

e. Additional signals are required to provide non-driving stops' load factors to the network component.

Example 2-3 contains a network package (NET_PACK3). The resolution function is replaced by a component declaration for the network behavior. (Note, the RESOLVE resolution function is for internal use of the network component only.) The signal type declaration contains three subelements -- driving/effective value, drivers and cumulative load. Here there is a need for additional signals to provide load factors to the network. These are reflected in the first declared port of the component.

```
package NET_PACK3 is
    type MY_ULOGIC is record
        Value : STD_ULOGIC;
        Drivers : Natural;
         Load   : Real; -- Capacitance is in normalized units
         Drive : Real;  -- Drive factor. Not used if not driving
   end record;
        ...
     type Load_Vector is array(Positive range <>) of Real;
      type MY_ULOGIC_VECTOR is array(Natural range <>) of MY_ULOGIC;
      function Resolve( Inputs : MY_ULOGIC_VECTOR ) return MY_ULOGIC;
     subtype MY_LOGIC is Resolve MY_ULOGIC;
    component WIRED_OR
      generic ( Non_Driving_Stops : Positive := 1;
           Num_Drivers : Positive := 1;     WIRE_LOAD : Real := 1.0 );
        port ( Non_Driving_Loads : Load_Vector(1 to Non_Driving_Stops);
           DRIVERS : in MY_ULOGIC_VECTOR(1 to Num_Drivers);
         EFFECTIVE : out MY_ULOGIC );
    end component;
end  NET_PACK3;
```

Example 2-3. Network Package for Component Modeling Approach

Example 2-6 provides the VHDL that implements the network component. Points of interest in the model include the generate that creates a process that is sensitive to each network driver. These processes accurately calculate the network delay. The generate results in multiple drivers for the internal resolved value of the net requiring a locally resolved signal.

```
entity Network is
  generic ( Non_Driving_Stops : Positive := 1;
       Num_Drivers : Positive := 1; WIRE_LOAD : Real := 1.0 );
  port ( Non_Driving_Loads : Load_Vector( 1 to Non_Driving_Stops);
       Drivers : in MY_ULOGIC_VECTOR(1 to Num_Drivers);
      Effective : out MY_ULOGIC := MU_U );
end Network;

architecture Wired_Or of Network is
  signal Resolved_Effective : MY_LOGIC;
  procedure Calc_Effective_Value( Effective_Value : inout MY_ULOGIC;
              Active_Driver : in Positive )
 is begin
  for I in Drivers'Range loop
    Effective_Value.Value := Effective_Value.Value OR  Drivers(I).Value;
   if Drivers(I).Value /= 'Z' then
       Effective_Value.Drivers := Effective_Value.Drivers + Drivers(I).Drivers;
    if I /= Active_Driver then
       Effective_Value.Load := Effective_Value.Load + Drivers(I).Load;
   else
        Effective_Value.Drive := Drivers(I).Drive;
   end if;
  end if;
  end loop;
   for I in Non_Driving_Loads'Range loop
     Effective_Value.Load := Effective_Value.Load + Non_Driving_Loads(I);
  end loop;
   Effective_Value.Load := Effective_Value.Load + WIRE_LOAD;
  end Calc_Effective_Value;
begin
  FORGEN:   for I in 1 to Num_Drivers generate
   process( Drivers(I) )
    variable Effective_Value : MY_ULOGIC := ('0', 0, 0.0);
   begin
    Effective_Value := ('0', 0, 0.0, 0.0);
     Calc_Effective_Value( Effective_Value, I );
     Resolved_Effective <= Effective_Value *
         (Effective_Value after Effective_Value.Load * 1 ns);
   end process;
  end generate;
   Effective <= Resolved_Effective;
end Wired_Or;
```

Example 2-4 Network Component Implementation.

Finally, example 2-5 shows a simpler implementation of the nand gate.

```
entity NAND_GATE is
  generic ( IN1_LF : Real := 0.5;  IN2_LF : Real := 0.5;
      OUTPUT_LF : Real := 1.0;  Drive : Real := 5.0;
      BODY_DELAY : TIME := 5 ns );
  port ( IN1 : in MY_ULOGIC := MU_U;
      IN1_Load : out REAL := IN1_LF;
      IN2 : in MY_ULOGIC := MU_U;
      IN2_Load : out REAL := IN2_LF;
      OUTPUT: out MY_ULOGIC := MU_U );
end  NAND_GATE;

architecture  PRIMITIVE  of  NAND_GATE  is
begin
      OUTPUT <= (Value => (IN1.Value nand IN2.Value),
               Drivers => 1, Load => OUTPUT_LF, Drive => Drive)
             after Body_Delay;
end  PRIMITIVE;
```

Example 2-5. Nand Gate for Network Component Approach.

Here are the pros and cons of this modeling approach:

CON: The first approach used the resolution function capability. With resolution functions, hierarchy support comes virtually free. This approach requires closer attention to design hierarchy and connectivity.

Additional signals are required to provide non-driving load data to the network component.

The generate in the network component as well as the additional signal network activity impacts performance.

PRO: Components can be modeled without the need to apply network delays or with much concern whatsoever on how network delay calculation is performed or applied.

Simpler than the 1st approach.

A truly accurate implementation of the algorithm. The active driver is known (ignoring the situation when more than one driver is active at the same time).

3. System- Level Network Modeling

First, since the term system is highly context dependent, for the purposes of this section the following informal definition of a system is used.

System: A functionally complete set of operations organized to serve some purpose. It is not limited to an IC, ASIC or PCB. It may be as simple as a low-density PCB or as complex as a steel mill.

Specifically, this section is concerned with high-level modeling of the entire system. The example used for this section is a bank teller operation. The reason for using this system is to exemplify queues where those queues have certain entry, exit and service distributions. Where customer ENTRY is new people arriving in the queue to be

SERVICEd by the teller(s) or who may EXIT the queue without being SERVICEd for a variety of reasons. It is the fact that customers may EXIT the queue without being serviced that makes this system interesting.

The basic approach is to use the technique employed in the 2nd timing-estimation approach above. That is, the queue of customers is implemented as a network component (model). Customer entry is implemented as a stimulus model (stimulus is read in from a file -- empirical data -- or provided by a statistical stimulus generation function). The customer servicing function is modeled by a fluctuating number (demand-based) of tellers who work at a variety of efficiency levels and service customers in an amount of time dependent upon the transaction being conducted. Examples 3-1 and 3-2 depict the customer entry, queue and teller-service models.

```
entity bank_entrance is
  generic ( number_of_days : Positive := 1;
     stim_input : String := "in_queue.txt" );
  port ( New_Customer : out Customer_Record );
end bank_entrance;
```

```
entity Teller is
  generic ( Ability : Proficiency := Typ;
       Teller_Number : Positive := 1 );
  port ( Next_Customer : in Dispatched_Customer;
       Ready_For_Next_Cust : out Res_Bool := True );
end Teller;
```

```
entity Cust_Q is
  generic ( Max_Q_Size : Positive := 50;    Num_Tellers : Positive := 1 );
  port ( New_Cust : in Customer_Record;
       Next_Ready_Teller : inout Boolean_Vector( 1 to Num_Tellers ) := (others => True);
       Cust_Dispatched : out Disp_Cust_Array( 1 to Num_Tellers ));
end Cust_Q;
```

Example 3-1. Bank Entrance,Teller Entities and Customer Queue.

The pros and cons of using VHDL to model at this level are:

CON: The implementation results in more signals than what is necessary for the model. The type of the signals are records. However, the components are sensitive on any change to the signal and not on changes in subelements.

VHDL-87 does not provide global variables which would be convenient for implementing statistical distribution of arrivals and exits to/from the queue. VHDL-92 will probably address this issue.

The VHDL signal semantics can get in the way. The array of "ready tellers" had to be resolved since there were multiple drivers. However, the model only cared about the last value assigned.

PRO: It is straight-forward and relatively simple to implement high-level models in VHDL.

```
architecture Queue of Cust_Q is begin
 Q : process
     ...
   procedure Dispatch_Cust(...)
     ...
 begin
    wait on New_Cust, Next_Ready_Teller;
     Local_Ready_Teller := Next_Ready_Teller;
   -- Service Q before checking for new arrivals.
     while Q_Head /= Null and Local_Ready_Teller /= No_Teller_Available loop
       Dispatch_Cust( Cust_Dispatched, Q_Head.Customer, Local_Ready_Teller );
     Tmp_Ptr := Q_Head;  Q_Head := Q_Head.next_cust;
    if Q_Head /= Null then
     Q_Head.Prev_Cust := Null;
   end if;
    Deallocate( Tmp_Ptr );
   end loop;
    if New_Cust.Cust_No'Active then
    -- Check to see if the Q is empty and if a teller is available
      if Q_Head = Null and Local_Ready_Teller /= No_Teller_Available then
        Dispatch_Cust( Cust_Dispatched, New_Cust, Local_Ready_Teller );
    elsif Q_Head /= Null then
      -- Add new customer to Q then dispatch as many as possible.
     if Q_Count < Max_Q_Size then
       Tmp_Ptr := new Queue_Node'(New_Cust, Null, Null);
         Q_Tail.Next_Cust := Tmp_Ptr; Tmp_Ptr.Prev_Cust := Q_Tail;
      Q_Tail := Tmp_Ptr;    Q_Count := Q_Count + 1;
    end if;
        while Q_Head /= Null and Local_Ready_Teller /= No_Teller_Available loop
          Dispatch_Cust( Cust_Dispatched, Q_Head.Customer, Local_Ready_Teller );
      Tmp_Ptr := Q_Head;    Q_Head := Q_Head.next_cust;
      Q_Head.Prev_Cust := Null;   Deallocate( Tmp_Ptr );
    end loop;
      elsif Q_Head = Null and Local_Ready_Teller = No_Teller_Available then
     -- Empty Q; but all tellers are busy.
      Tmp_Ptr := new Queue_Node'(New_Cust, Null, Null);
     Q_Tail := Tmp_Ptr; Q_Head := Q_Tail;  Q_Count := 1;
   end if;
  end if;
    -- Finally simulate a pseudo-random exit-without-service distribution of
   -- customers leaving the Q.
   if Now > Last_Time then
    -- Only one customer at a time exits the Q.
    aNumber := abs( Integer( Now / 0.5 sec));
     aNumber := aNumber mod Max_Q_Size;
    if aNumber <= Q_Count and J > 2 then
     J := 1;Tmp_Ptr := Q_Head;
     while J /= aNumber loop
       Tmp_Ptr := Tmp_Ptr.Next_Cust; J := J + 1;
     end loop;
        Tmp_Ptr.Prev_Cust.Next_Cust := Tmp_Ptr.Next_Cust;
        Tmp_Ptr.Next_Cust.Prev_Cust := Tmp_Ptr.Prev_Cust;
   end if;
  end if;
 end process;
end Queue;
```

Example 3-2. The Bank Customer Queue Implementation.

4. Conclusions

While VHDL is powerful enough to model most things of interest to digital hardware and system designers, the designer is forced to "jump-through-hoops" for some problems. The greatest frustration for today's designer is the inability to gracefully specify a timing algorithm. Most non-V HDL digital simulators either build-in a timing estimation algorithm or provide some mechanism for the user to specify the algorithm in a single place. Built-in algorithms are contrary to the open, flexible modeling philosophy promoted by VHDL. However, language support for specifying timing algorithms that can be applied globally or locally in a design would be consistent with the VHDL philosophy.

Although VHDL is not yet being used extensively at higher system modeling levels, this will surely change in the near future. Language capabilities that will assist this level of modeling are:

a) Composite signals (a change to any subelement is a single event on the entire signal). That is, there is one signal with many fields.

b) Generalized data structures (variant records) or, better yet, object classes.

c) Global variables (which will probably be in VHDL '92).

d) A more generalized signal model. At the gate-level, signals are NOT hardware-oriented enough. At the system-level, signals are TOO hardware-oriented.