Board-Level Simulation and the Free Model Foundry

by Richard Munden, Free Model Foundry

This article is the first in a series. It discusses some of the historical, economic, and technical issues that surround board-level simulation of digital electronic designs. It outlines the FMF modeling style for small components. Future articles will apply the FMF style to complex components in VHDL and Verilog.

Introduction

Over the years, IC designs have become denser, faster and more complex to the point that people now refer to some of them as systems on a chip. It is now considered unthinkable to try to design an IC of any size without simulation. The expense in both time and money is too great for the fab-debug-refab-until-it-works approach. Plus, how would someone debug a system-on-a-chip? You can't just throw it on the bench and hook a logic analyzer to the busses.

Yet, many companies are still trying to do board-level designs that way. Sure, I've done it myself. I just hooked the analyzer probes on to the wirewrap pins on the back of the board, or to the dip clip on top. Another probe went to the 10Mhz clock somewhere. No problem. Except few engineers get to design boards like that anymore. Now we have tiny surface mount components on multilayer PC boards. And the boards run at hundreds, sometimes thousands, of MegaHertz. And that system on chip turned out to be one of a half dozen subsystems on the board. And it is in a ball grid array package so you can't even see the pins much less lift one. Now it has become expensive in both time and money to reiterate a board design.

So, what is the difference between simulating a board and a chip? Why are ASICs always extensively simulated but boards are usually only partially simulated if at all? The most important difference is the models.

When an engineer starts an ASIC or FPGA design, all the simulation models are already sitting in a vendor supplied library. The ASIC or FPGA vendor understands that without the models, it might take several costly attempts to get the design working and ready for production. But in the board design world, most component vendors have not yet realized how important models are to the design process. As a result, many board designers spend more time in the lab, trying to debug complex designs, than they do creating them. Due to lot-to-lot and vendor-to-vendor component variations, even when one copy of the board works there is no guarantee the next will. And how do you know the problem isn't really in that big chip the guys down hall built just for this board?

What is an engineer to do?

Background

Back in the 1980s, when simulators started to become available to engineers, some component vendors considered supplying models. But back then, there were no standards. Every CAE vendor had their own proprietary simulator and models were not portable from one to another. Vendors had to decide how many and which tools to support. They quickly gave up. The CAE vendors tried to step in, they had to sell simulators, and for a short while, they provided models of the small and medium sized parts of the day. These models were open source because they could not be used with anyone else's simulator. But there was no way they could keep up with the proliferation of new and ever more complex parts. And they weren't making any money at it, they had to give the models away to sell the simulators.

Then some enterprising people came along with the idea of writing models in a vendor neutral format and translating them to each proprietary format. This had a couple of benefits; it reduced the total effort of writing models and, it made modeling someone's primary business. Someone now had clear financial incentives to generate and distribute simulation models of components. Being focused on modeling, they were able to cover many more components then the CAE vendors.

Unfortunately, it also created a couple of problems. First, it helped the IC vendors abdicate responsibility for modeling their parts. Second, because the business model was to charge the engineers for using the models, the

models had to be encrypted to control copying. There were also additional overhead expenses, such as licensing software and a sales force, that increased the cost, and hence the price of these models.

While encryption was a necessity for the model vendors, it was a disadvantage for the users. Engineers could no longer compare the models with the data sheets to improve their understanding of a part. If a model did not work as the engineer expected, he could not look at the model and determine whether the problem was with the model or his expectations. If the problem was traced to the model, a bug fix might have to wait until the next release of the library. Companies that needed to collaborate were unable to share models.

In 1987 VHDL became an IEEE standard. Verilog became a standard in 1993. Both of these languages facilitated board-level simulation using models that were portable from one CAE vendor to another.

In 1995, I was a CAE manager at TRW. My center was designing some very high speed boards for use in telecom systems. The parts on these boards were not widely used because they were small, hot, and expensive. We were running them about as fast as they would go and needed timing accurate models. They were not available from the vendor or the commercial model suppliers so we decided to create our own, as we usually wound up doing.

I began wondering how many different companies were modeling the same parts. None of us wanted to be in the modeling business. What if we shared the models we wrote while urging the IC vendors to take the responsibility themselves? A group of TRW engineers consisting of Russ Vreeland, Luis Garcia, and myself, decided to start an organization to do something about it. Later, we were joined by Robert Harrison and received technical assistance from Ray Steele, both also at TRW. We knew we wanted to distribute models as source code and to encourage companies to share their models so, we adopted a not-for-profit business model. We called it the Free Model Foundation in deference to Richard Stallman's Free Software Foundation that had produced so many of the software tools engineers take for granted. Our management agreed to allow us to contribute to the foundation, any models of commercially available, off the shelf components we created.

FMF was founded to solve a problem, the scarcity of simulation models for use in board design. There appeared to be two aspects to the problem a technical aspect and an economic or social aspect. Of the two, the economic aspect is the more important so I will cover it first.

The Economics of Modeling

There are an *awful lot of parts* to model. The end users can't do it. It would require that they expend more resources on component modeling than on designing their own products.

The CAE vendors can't do it. They tried back when there were a lot fewer parts. They couldn't keep up with the torrents of new parts and, it was a loss leader for them. Now that they all offer interoperable, standards based simulators, it makes more sense for someone else to do the modeling.

Modeling companies can't do it. They are trying, and they are doing a better job than the tool vendors did but, they can't keep up with flood of new parts either. Their business model also introduces costs and inefficiencies to the process. Modeling companies base their business on pay for use or, pay for access to models. To make any money and cover their cost of operations, they must enforce that model by restricting access to those that pay. It costs money to write models and they have to be paid for somehow. But restricting access also costs money. There are usually encryption schemes to prevent copying and licensing software to incorporate. Encrypting models, of course, reduces their value to the engineers because they cannot examine them or, if there are bugs, fix them. Even companies selling source level models have sales costs and restrictive licensing. All these problems aside, modeling companies still must decide which parts to model based modeling effort and anticipated revenues.

So who can provide engineers with the models they need to design new systems?

I believe the only reasonable answer is the component vendors. They are in the best position to solve the problem for several reasons. They have access to the all the information about the parts including the proprietary models used in the design process. They have a vested interest in their customers getting their designs to market sooner. Models provide them with a marketing advantage, a model is documentation for one of *their* products. Which means they have an interest in having models widely disseminated as opposed to restricting access. Most importantly, they have

the mechanism for cost recovery - increased product sales. This is the same method they use to recover the cost of data books (or web based data sheets).

All that remains is to make the component vendors understand that providing unrestricted models is in their best interest. It should be pointed out here, for the benefit of the paranoid, that the models engineers need do not contain any information beyond what they need in the data sheets. No proprietary, secret, or competitive data is involved. All designers want, is to get to volume production (meaning volume purchases of parts) sooner.

Many IC vendors do not have the in-house capability/capacity to model all their own parts. There are at least two ways of addressing this problem. Cooperate with second sources: If the data sheets are word for word copies, one model should be enough. Or, out source modeling: There are still all of those modeling companies, they will just make money from writing models rather than from distributing them.

Because the problem FMF set out to solve was model scarcity, all our models are licensed using the GNU General Public License (GPL). This makes them freely distributable. The only restrictions are: the copyright may not be removed; source code must be made available; and any derived works must be distributed under the same license. This open source form of distribution has been used for (and resulted in) much of the best and most widely used software in the world. I believe it is in everybody's best interest to continue the tradition.

Technical Aspects of Modeling

We knew we would need a standard method and style of modeling. This was about the time the first version of the VHDL Initiative Toward ASIC Libraries (VITAL) standard was being published. We examined it and saw that it would work well for modeling board-level components. It provided a supported means of writing models with generic timing parameters and then supplying the actual timing values at elaboration time by means of a Standard Delay Format (SDF) file. This meant the models could be technology independent. One model would be sufficient for a standard function and cover a variety of process technologies (LS, ALS, AS, FCT, ABT, etc.) and speed grades. VITAL also provides a standard, widely supported, means of backannotating interconnect delays from the physical design (the PCB). Many of today's high speed systems require this for timing accurate simulation.

FMF component models differ significantly from the RTL models that most users of VHDL typically write. They can be written at whatever level of abstraction is suitable to the problem. They must not be synthesizable (they document a commercial product not reverse engineer it). They have a very long life expectancy and are likely to be revised or updated several times. They are likely to be read and maintained by many engineers who were not involved with their original creation.

All of these attributes were considered in formulating the FMF modeling style. With them in mind we undertook to devise a rigid format for our models that would meet these requirements and so that we could write perl scripts to update the models to new versions of VITAL when they are released. Our ideas are described in the "FMF Style Guide" and may be found at http://vhdl.org/fmf/wwwpages/vhdl_papers.html along with other related documentation. Some of the suggestions in the FMF style guide may seem restrictive or arbitrary but they were all put in for good reasons. Please bear in mind that our experience at the time it was written was with relatively small models. Larger models have required exceptions or additions to the guide. Also, as the name implies, it is only a guide. The code police will not drag you away in the night if you do not follow it precisely.

Below is an example of the FMF modeling style for a simple component. A more complex component will be presented in the next article.

All FMF models start with a header that contains the copyright, a brief description of the model, and author and revision information. As with any software, it is important to know what version you are working with. All text is formatted to 80 columns wide to make printing easier.

```
--------------------------------------------------------------------------
--  File Name: std534.vhd
--------------------------------------------------------------------------
--  Copyright (C) 1998 Free Model Foundation
--
```

```
--  This program is free software; you can redistribute it and/or modify
--  it under the terms of the GNU General Public License version 2 as
--  published by the Free Software Foundation.
--
--  MODIFICATION HISTORY:
--
--  version: | author:  | mod date: | changes made:
--    V1.0   Hoi Nguyen   98 OCT 27   Conformed to style guide
--------------------------------------------------------------------------------
--  PART DESCRIPTION:
--
--  Library:     STD
--  Technology:  54/74XXXX
--  Part:        STD534
--
--  Description:  Flip-Flop With Inverted 3-State Output
--------------------------------------------------------------------------------
```

The style guide offers a precise tabular alignment for the code. This may seem restrictive (or retentive) but we felt it would be beneficial to the user if all models had the same “look and feel”. It has also made it easier to programmatically revise large numbers of models when the VITAL standard has changed (which it is about to do again!). Spaces are used instead of tabs because they will not change from one environment to another as tabs will.

All FMF models use a VITAL level 0 entity. This means timing may be backannotated to them through an SDF file. VITAL provides a number of standard generics. They are used for backannotating interconnect delays (tipd), pin to pin propagation delays (tpd), and numerous timing check generics (see the VITAL documentation). There are also some generics that control whether or not timing checks are performed and whether or not timing violations result in “X”s in the simulation output. In addition to these, FMF has defined a generic called “TimingModel”. The value of this generic for a particular instance, determines which set of timing values will be written into the SDF file for use in the simulation. The schematic capture packages tested to date all pass a value for TimingModel from the schematic to the netlist. A program FMF provides, called mk_sdf, reads the netlist and timing files, and writes an SDF file. (More on timing files later.)

```
--------------------------------------------------------------------------------

LIBRARY IEEE;    USE IEEE.std_logic_1164.ALL;
                 USE IEEE.VITAL_timing.ALL;
                 USE IEEE.VITAL_primitives.ALL;
LIBRARY FMF;     USE FMF.gen_utils.ALL;
                 USE FMF.ff_package.ALL;
--------------------------------------------------------------------------------
-- ENTITY DECLARATION
--------------------------------------------------------------------------------
ENTITY std534 IS
    GENERIC (
        -- tipd delays: interconnect path delays
        tipd_D              : VitalDelayType01 := VitalZeroDelay01;
        tipd_CLK            : VitalDelayType01 := VitalZeroDelay01;
        tipd_OENeg          : VitalDelayType01 := VitalZeroDelay01;
        -- tpd delays
        tpd_CLK_QNeg        : VitalDelayType01 := UnitDelay01;
        tpd_OENeg_QNeg      : VitalDelayType01Z := UnitDelay01Z;
        -- tsetup values: setup times
        tsetup_D_CLK        : VitalDelayType := UnitDelay;
        -- thold values: hold times
        thold_D_CLK         : VitalDelayType := UnitDelay;
```

```
        -- tpw values: pulse widths
        tpw_CLK_posedge     : VitalDelayType := UnitDelay;
        tpw_CLK_negedge     : VitalDelayType := UnitDelay;
        -- tperiod_min: minimum clock period = 1/max freq
         tperiod_CLK_posedge : VitalDelayType := UnitDelay;
        -- generic control parameters
        TimingChecksOn      : Boolean  := DefaultTimingChecks;
        MsgOn               : BOOLEAN := DefaultMsgOn;
        XOn                 : Boolean  := DefaultXOn;
        InstancePath        : STRING   := DefaultInstancePath;
        -- For FMF SDF technology file usage
        TimingModel         : STRING   := DefaultTimingModel
    );
    PORT (
        QNeg    : OUT   std_logic := 'U';
        D       : IN    std_logic := 'X';
        CLK     : IN    std_logic := 'X';
        OENeg   : IN    std_logic := 'X'
    );

    ATTRIBUTE VITAL_LEVEL0 of std534 : ENTITY IS TRUE;
END std534;
```

An unfortunate consequence of enabling SDF backannotation of interconnect delays is a requirement that all ports of modes IN and INOUT be scaler. This is not a problem for small models but, when modeling components that have multiple large buses it adds to the length and complexity of the model. We have been fortunate in having a team of experts at SEVA technologies work on this problem. They will be presenting their solutions in the next article.

The architecture may be either level 1 or level 0 depending on the complexity of the part. Level 1 is close to a gate level representation but is accelerated by the simulator. Small models will simulate faster if they are level 1. Large parts would run slower and be impractical to model at level 1.

```
-------------------------------------------------------------------------
-- ARCHITECTURE DECLARATION
-------------------------------------------------------------------------
ARCHITECTURE vhdl_behavioral of std534 IS
    ATTRIBUTE VITAL_LEVEL1 of vhdl_behavioral : ARCHITECTURE IS TRUE;
```

Next we declare the internal signals. These are the input signals after they have been delay by the PCB interconnect. The actual delay is applied in the wire delay block.

```
    SIGNAL D_ipd        : std_ulogic := 'X';
    SIGNAL CLK_ipd      : std_ulogic := 'X';
    SIGNAL OENeg_ipd    : std_ulogic := 'X';

BEGIN
    -------------------------------------------------------------------------
    -- Wire Delays
    -------------------------------------------------------------------------
    WireDelay : BLOCK
    BEGIN

        w_1: VitalWireDelay (D_ipd, D, tipd_D);
        w_2: VitalWireDelay (CLK_ipd, CLK, tipd_CLK);
        w_3: VitalWireDelay (OENeg_ipd, OENeg, tipd_OENeg);
```

```
    END BLOCK;
```

The behavior of the part is modeled inside a process. For small parts, it is desirable to capture all the behavior in a single process. This may not be possible for more complex parts.

```
    ---------------------------------------------------------------------
    -- Main Behavior Process
    ---------------------------------------------------------------------
    VitalBehavior : PROCESS (CLK_ipd, D_ipd, OENeg_ipd)
```

The process begins with the variable declarations and the timing checks. These always precede the functionality section because we want a failed timing check to be reflected on the device output pins.

```
         -- Timing Check Variables
        VARIABLE Tviol_D_CLK    : X01 := '0';
        VARIABLE TD_D_CLK       : VitalTimingDataType;

        VARIABLE Pviol_CLK      : X01 := '0';
        VARIABLE PD_CLK         : VitalPeriodDataType := VitalPeriodDataInit;

        VARIABLE Violation      : X01 := '0';

        -- Functionality Results Variables
        VARIABLE Qint           : std_ulogic;
        VARIABLE Q_zd           : std_ulogic;
        VARIABLE PrevData       : std_logic_vector(0 to 2);

        -- Output Glitch Detection Variables
        VARIABLE Q_GlitchData   : VitalGlitchDataType;

    BEGIN
        ---------------------------------------------------------------------
        -- Timing Check Section
        ---------------------------------------------------------------------
        IF (TimingChecksOn) THEN

            VitalSetupHoldCheck (
                TestSignal      => D_ipd,
                TestSignalName  => "D_ipd",
                RefSignal       => CLK_ipd,
                RefSignalName   => "CLK_ipd",
                SetupHigh       => tsetup_D_CLK,
                SetupLow        => tsetup_D_CLK,
                HoldHigh        => thold_D_CLK,
                HoldLow         => thold_D_CLK,
                CheckEnabled    => TRUE,
                RefTransition   => '/',
                HeaderMsg       => InstancePath & "/std534",
                TimingData      => TD_D_CLK,
                XOn             => XOn,
                MsgOn           => MsgOn,
                Violation       => Tviol_D_CLK
            );

            VitalPeriodPulseCheck (
                TestSignal      => CLK_ipd,
```

```
            TestSignalName  => "CLK_ipd",
            Period          => tperiod_CLK_posedge,
            PulseWidthHigh  => tpw_CLK_posedge,
            PulseWidthLow   => tpw_CLK_negedge,
            CheckEnabled    => TRUE,
            HeaderMsg       => InstancePath & "/std534",
            PeriodData      => PD_CLK,
            XOn             => XOn,
            MsgOn           => MsgOn,
            Violation       => Pviol_CLK
        );

    END IF;
```

After the timing checks we get to the functionality. You will note that this is the smallest section in the model. This model is 190 lines long. Only 8 lines are needed to describe the behavior of the part. The rest are concerned with timing. In this case, a part of the behavior is described by a VitalStateTable in the FMF ff_package. A VitalStateTable is similar to a Verilog UDP.

```
    -------------------------------------------------------------------------
    -- Functionality Section
    -------------------------------------------------------------------------
    Violation := Tviol_D_CLK OR Pviol_CLK;

    VitalStateTable (
        StateTable      => DFFQN_tab,
        DataIn          => (Violation, CLK_ipd, D_ipd),
        Result          => Qint,
        PreviousDataIn  => PrevData
    );

    Q_zd := VitalBUFIF0 (Data   => Qint,
                         Enable => OENeg_ipd);
```

At the end of the model is the path delay section. Here the internal delays that were backannotated through the tpd generics (and SDF) are applied to the output signals. The path delay procedure drives the outputs.

```
    -------------------------------------------------------------------------
    -- Path Delay Section
    -------------------------------------------------------------------------
    VitalPathDelay01Z (
        OutSignal       => QNeg,
        OutSignalName   => "QNeg",
        OutTemp         => Q_zd,
        GlitchData      => Q_GlitchData,
        XOn             => XOn,
        MsgOn           => MsgOn,
        Paths           => (
            0 => (InputChangeTime   => CLK_ipd'LAST_EVENT,
                PathDelay        => VitalExtendToFillDelay (tpd_CLK_QNeg),
                  PathCondition     => TRUE),
            1 => (InputChangeTime   => OENeg_ipd'LAST_EVENT,
                  PathDelay         => tpd_OENeg_QNeg,
                  PathCondition     => TRUE)
        )
    );
```

```
    END PROCESS;

END vhdl_behavioral;
```

This may seem like a lot of code to type for a simple component. However, in practice, the model writer is able to cut and paste nearly all of it from other models. Using the existing FMF libraries as a source of templates, new models can be written very quickly.

Timing Models

The timing models consist of sections of SDF code encapsulated in SGML (or XML). SGML is used because it is easily parsed by both humans and computers. It is a simple but powerful mark up language for which new uses are continually being found. The document type of a FMF timing model is called “FTML”. The first line of a timing file is always:

```
<!DOCTYPE FTML SYSTEM "ftml.dtd">
```

The rest of the file is divided into a head and a body. The head contains the title and the revision history of the model.

```
<FTML><HEAD><TITLE>FMF Timing for STD534 Parts</TITLE>
<REVISION.HISTORY>
version: |  author:  | mod date: | changes made:
  V1.0    Hoi Nguyen   98 OCT 27   Initial release
</REVISION.HISTORY>
</HEAD>
```

The body includes the SDF timescale, usually 1 ns. Next is the name of the model with which the timing data is to be used. Then comes the “FMFTIME” section. This segment first lists all the manufacturer’s part numbers for which the following timing section applies. For each part number the source of the data is cited. After the list of part numbers, comments may be added to describe the conditions for which the timing is valid and anything else the user may need to know. Then comes the “TIMING” section. This segment contains verbatim SDF code describing the timing of the above listed part numbers. Following the </TIMING> tag, as many more FMFTIME sections may be added as desired.

```
<BODY>
<TIMESCALE>1ns</TIMESCALE>
<MODEL>STD534
<FMFTIME>
SN74ABT534ADB<SOURCE>Texas Instruments SCBS187F-Revised Jan 1997</SOURCE>
SN74ABT534ADW<SOURCE>Texas Instruments SCBS187F-Revised Jan 1997</SOURCE>
SN74ABT534AN<SOURCE>Texas Instruments SCBS187F-Revised Jan 1997</SOURCE>
SN74ABT534APW<SOURCE>Texas Instruments SCBS187F-Revised Jan 1997</SOURCE>
<COMMENT>The Values listed are for VCC=5V, CL=50pF, Ta=+25 Celsius</COMMENT>
<TIMING>
(DELAY (ABSOLUTE
    (IOPATH CLK QNeg (2.6:4.5:5.9) (3.4:5.5:6.7))
    (IOPATH OENeg QNeg () () (1.0:3.4:4.2) (2.6:4.0:5.8) (2.4:4.7:6.6)
(2.3:3.8:5.8))
   ))
  (TIMINGCHECK
    (SETUP D CLK (1.6:1.6:1.6))
    (HOLD D CLK (2.0:2.0:2.0))
    (WIDTH (posedge CLK) (3.5:3.5:3.5))
    (WIDTH (negedge CLK) (3.5:3.5:3.5))
    (PERIOD (posedge CLK) (8.0:8.0:8.0))
```

```
  )
</TIMING></FMFTIME>
</BODY></FTML>
```

Some timing files have timing for more than 100 part numbers.

The mk_sdf program reads the VHDL netlist of the design and picks out the TimingModel generic for each instance. It then finds the timing file that goes with each model. The program searches the timing file for the FMFTIME section that includes the part number found in the TimingModel generic and copies the SDF code from that section into the new SDF file. The SDF file can be read by any VITAL compliant simulator. For those of the Verilog persuasion, the same timing files can also be used with Verilog models. This will be the subject of a future article.