Verification Beyond the Chip

Richard Munden

Siemens Medical Solutions
Ultrasound Division
1230 Shorebird Way
Mountain View, CA 94043
Presented at Mentor Users Group 2004

Portions of this paper are excerpted from an upcoming book to be published by Morgan Kaufmann Publishers

Abstract

Have you ever spent weeks or months verifying an ASIC or FPGA only to discover an interface problem after it was on the board?

This paper discusses a methodology used at Siemens Ultrasound that allows a chip to be verified at the board level. It shows how RTL code, either VHDL or Verilog, can be instantiated in a VITAL wrapper with propagation delays and timing checks, and simulated at the board level using publicly available component models and a VHDL netlist from ConceptHDL.

1.0 Introduction

These days, boards and systems are designed using schematics while chips, ASICs and FPGAs are designed using HDLs, usually VHDL or Verilog. From the schematic, a VHDL netlist can be generated. VHDL behavioral models of the off-the-shelf components can be written based on the vendors' datasheets or downloaded from the Free Model Foundry (FMF) website at: www.eda.org/fmf/. If you started your chip design by writing a behavioral model, that could also be included and you could perform early simulation to test the correctness of your design specification. However, most engineers get their design specs from a text document. They start their chip designs by writing and simulating RTL code. The RTL code could be used in the board-level simulation but, without timing, it might not behave in a manner indicative of the hardware you intend to build. This paper demonstrates a technique for adding timing to your RTL code that is compatible with the other models in your simulation.

2.0 Using VITAL to Simulate Your RTL

Unless you have written a behavioral model of your FPGA, your only choices for board-level verification are gate-level and RTL. Gate-level is the most accurate. But, you can not get a gate-level model until you have completed the RTL model sufficiently to synthesize and run the vendor's place and route tool. Then, you will find that gate-level simulation is also the slowest and most memory intensive way to verify an ASIC or FPGA.

RTL is much faster and uses much less memory than gate-level. However, the RTL model has no timing information. Fortunately, you can embed the RTL in a VITAL wrapper and have the best of both worlds, RTL speed plus full timing. You can even start your simulations with the timing constraints you intend to use for synthesis to verify you are not over- or under-constraining the design. After place and route, you can substitute timing values from the vendor's timing analyzer to reverify with realized timing.

There are other advantages to performing board-level simulation using the RTL model of your chip design:

The interfaces between your design and the other components on the board become better understood.
The chip design is verified earlier and without spending time on synthesis and place and route.
The board design is verified earlier and can be released to layout sooner and with a higher degree of confidence.
On-board diagnostics can be developed and verified earlier in the design cycle.

Although writing a VITAL wrapper for a complex ASIC or FPGA design is non-trivial, it can to some extent be automated. The bilingual capabilities of today's simulators allow this strategy to work whether the RTL code is written in VHDL or Verilog.

2.1 The Basic Wrapper

The RTL code used for synthesis describes the behavior of the component being designed but it does not describe the timing. The timing is necessary to ensure the chip interfaces correctly with the rest of the system. We can attach propagation delays and timing constraint checks to the RTL without changing it by instantiating it in a higher level component. We call this higher level component a wrapper because it does not change the functionality of the RTL model, it just wraps it in a set of path delay and timing check procedures.

As an example of how to create a timing wrapper, let's begin with a fictitious design named FPGA299. This design is much smaller than a real FPGA but will illustrate most of the concepts. We will first look at a VHDL version of the design. The entity is:

```
ENTITY fpga299 IS
    PORT (
        CLR_L           : IN    std_logic;
        OE1_L           : IN    std_logic;
        OE2_L           : IN    std_logic;
        S0              : IN    std_logic;
        S1              : IN    std_logic;
        CLK             : IN    std_logic;
        IO              : INOUT std_logic_vector(7 downto 0)
        Q0              : OUT   std_logic;
        Q7              : OUT   std_logic;
        SL              : IN    std_logic;
        SR              : IN    std_logic
    );
END fpga299;
```

In order to avoid naming conflicts, at the board-level (in the schematic), we will call this component chip299.

One begins by writing an entity that will be compatible with the schematic symbol that represents our FPGA design at the board-level. The wrapper will be a VITAL model (of sorts) so the file begins with LIBRARY and USE clauses which include the VITAL packages:

```
-------------------------------------------------------------------------------
--  File Name: chip299.vhd
-------------------------------------------------------------------------------
--  Description: Timing wrapper for fpga299
-------------------------------------------------------------------------------

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;

-------------------------------------------------------------------------------
-- ENTITY DECLARATION
-------------------------------------------------------------------------------
ENTITY chip299 IS
    GENERIC (
        -- tipd delays: interconnect path delays
        tipd_CLRNeg             : VitalDelayType01 := VitalZeroDelay01;
        tipd_OE1Neg             : VitalDelayType01 := VitalZeroDelay01;
        tipd_OE2Neg             : VitalDelayType01 := VitalZeroDelay01;
        tipd_S0                 : VitalDelayType01 := VitalZeroDelay01;
        tipd_S1                 : VitalDelayType01 := VitalZeroDelay01;
        tipd_CLK                : VitalDelayType01 := VitalZeroDelay01;
        tipd_IO0                : VitalDelayType01 := VitalZeroDelay01;
        tipd_IO1                : VitalDelayType01 := VitalZeroDelay01;
        tipd_IO2                : VitalDelayType01 := VitalZeroDelay01;
        tipd_IO3                : VitalDelayType01 := VitalZeroDelay01;
        tipd_IO4                : VitalDelayType01 := VitalZeroDelay01;
        tipd_IO5                : VitalDelayType01 := VitalZeroDelay01;
        tipd_IO6                : VitalDelayType01 := VitalZeroDelay01;
        tipd_IO7                : VitalDelayType01 := VitalZeroDelay01;
        tipd_SL                 : VitalDelayType01 := VitalZeroDelay01;
        tipd_SR                 : VitalDelayType01 := VitalZeroDelay01;
```

The GENERIC list starts with the tipd generics for the interconnect delays. This allows us to backannotate the PCB wire delays. These delays are particularly important for designs that incorporate an internal phase locked loop with a feedback path that is on the board.

Next come the rest of the timing generics for pin to pin delays, setup, hold, pulsewidth, and any other timing constraints along with the control parameters:

```
-- tpd delays
        tpd_CLRNeg_IO0          : VitalDelayType01 := UnitDelay01;
        tpd_CLRNeg_Q0           : VitalDelayType01 := UnitDelay01;
        tpd_OE1Neg_IO0          : VitalDelayType01Z := UnitDelay01Z;
        tpd_S0_IO0              : VitalDelayType01Z := UnitDelay01Z;
        tpd_CLK_IO0             : VitalDelayType01 := UnitDelay01;
        tpd_CLK_Q0              : VitalDelayType01 := UnitDelay01;
        -- tsetup values: setup times
        tsetup_S0_CLK           : VitalDelayType := UnitDelay;
        tsetup_SL_CLK           : VitalDelayType := UnitDelay;
        tsetup_IO0_CLK          : VitalDelayType := UnitDelay;
        -- thold values: hold times
        thold_S0_CLK            : VitalDelayType := UnitDelay;
        thold_SL_CLK            : VitalDelayType := UnitDelay;
        thold_IO0_CLK           : VitalDelayType := UnitDelay;
        -- trecovery values: release times
        trecovery_CLRNeg_CLK    : VitalDelayType := UnitDelay;
        -- tpw values: pulse widths
        tpw_CLK_posedge         : VitalDelayType := UnitDelay;
        tpw_CLK_negedge         : VitalDelayType := UnitDelay;
        tpw_CLRNeg_negedge      : VitalDelayType := UnitDelay;
        -- tperiod_min: minimum clock period = 1/max freq
        tperiod_CLK_posedge     : VitalDelayType := UnitDelay;
        -- generic control parameters
        InstancePath        : STRING    := DefaultInstancePath;
        TimingChecksOn      : BOOLEAN   := DefaultTimingChecks;
        MsgOn               : BOOLEAN   := DefaultMsgOn;
        XOn                 : BOOLEAN   := DefaultXon;
        -- For FMF SDF technology file usage
        TimingModel         : STRING    := DefaultTimingModel
    );
```

The generics and port list, indeed the entity, are written as if this was a model of an off-the-shelf component.

```
PORT (
        CLRNeg          : IN    std_ulogic := 'U';
        OE1Neg          : IN    std_ulogic := 'U';
        OE2Neg          : IN    std_ulogic := 'U';
        S0              : IN    std_ulogic := 'U';
        S1              : IN    std_ulogic := 'U';
        CLK             : IN    std_ulogic := 'U';
        IO0             : INOUT std_ulogic := 'U';
        IO1             : INOUT std_ulogic := 'U';
        IO2             : INOUT std_ulogic := 'U';
        IO3             : INOUT std_ulogic := 'U';
        IO4             : INOUT std_ulogic := 'U';
        IO5             : INOUT std_ulogic := 'U';
        IO6             : INOUT std_ulogic := 'U';
        IO7             : INOUT std_ulogic := 'U';
        Q0              : OUT   std_ulogic := 'U';
        Q7              : OUT   std_ulogic := 'U';
        SL              : IN    std_ulogic := 'U';
        SR              : IN    std_ulogic := 'U'
    );
    ATTRIBUTE VITAL_LEVEL0 of chip299 : ENTITY IS TRUE;
END chip299;
```

The entity has a VITAL_LEVEL0 attribute set to TRUE.

Note that the FPGA has eight pins of mode INOUT. These present a special challenge. They are not a problem in a component model because the control logic that determines whether they are acting as drivers or receivers is in the model. But in a wrapper, that control logic is in the instantiated RTL and is not visible to the wrapper. A method for overcoming this obstacle of unknown directionality, uses the std_logic_1164 resolution function and is explained as we progress through the code.

The architecture of the wrapper is not VITAL compliant. Therefore the VITAL attribute is omitted.

```
-------------------------------------------------------------------------------
-- ARCHITECTURE DECLARATION
-------------------------------------------------------------------------------
ARCHITECTURE vhdl_behavioral of chip299 IS

    CONSTANT partID            : STRING := "chip299";

COMPONENT fpga299
    PORT (
        CLR_L           : IN    std_logic;
        OE1_L           : IN    std_logic;
        OE2_L           : IN    std_logic;
        S0              : IN    std_logic;
        S1              : IN    std_logic;
        CLK             : IN    std_logic;
        IO              : INOUT std_logic_vector(7 downto 0);
        Q0              : OUT   std_logic;
        Q7              : OUT   std_logic;
        SL              : IN    std_logic;
        SR              : IN    std_logic
    );
END COMPONENT;
```

The RTL model is declared as a component that will be instantiated further down in the wrapper. Then signals are declared.

```
SIGNAL CLRNeg_ipd      : std_ulogic := 'U';
    SIGNAL OE1Neg_ipd      : std_ulogic := 'U';
    SIGNAL OE2Neg_ipd      : std_ulogic := 'U';
    SIGNAL S0_ipd          : std_ulogic := 'U';
    SIGNAL S1_ipd          : std_ulogic := 'U';
    SIGNAL CLK_ipd         : std_ulogic := 'U';
    SIGNAL IO0_ipd         : std_ulogic := 'U';
    SIGNAL IO1_ipd         : std_ulogic := 'U';
    SIGNAL IO2_ipd         : std_ulogic := 'U';
    SIGNAL IO3_ipd         : std_ulogic := 'U';
    SIGNAL IO4_ipd         : std_ulogic := 'U';
    SIGNAL IO5_ipd         : std_ulogic := 'U';
    SIGNAL IO6_ipd         : std_ulogic := 'U';
    SIGNAL IO7_ipd         : std_ulogic := 'U';
    SIGNAL SL_ipd          : std_ulogic := 'U';
    SIGNAL SR_ipd          : std_ulogic := 'U';
    SIGNAL IO_w            : std_logic_vector(7 downto 0) := (others => 'Z');
```

A configuration specification is required and supplied:

```
FOR ALL : fpga299 USE ENTITY work.fpga299(rtl);
```

The standard wire delay block is employed to apply interconnect delays to the input pins.

```
BEGIN

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

        w_1 : VitalWireDelay (CLRNeg_ipd, CLRNeg, tipd_CLRNeg);
        w_2 : VitalWireDelay (OE1Neg_ipd, OE1Neg, tipd_OE1Neg);
        w_3 : VitalWireDelay (OE2Neg_ipd, OE2Neg, tipd_OE2Neg);
        w_4 : VitalWireDelay (S0_ipd, S0, tipd_S0);
        w_5 : VitalWireDelay (S1_ipd, S1, tipd_S1);
        w_6 : VitalWireDelay (CLK_ipd, CLK, tipd_CLK);
```

```
        w_8 : VitalWireDelay (IO0_ipd, IO0, tipd_IO0);
        w_9 : VitalWireDelay (IO1_ipd, IO1, tipd_IO1);
        w_10 : VitalWireDelay (IO2_ipd, IO2, tipd_IO2);
        w_11 : VitalWireDelay (IO3_ipd, IO3, tipd_IO3);
        w_12 : VitalWireDelay (IO4_ipd, IO4, tipd_IO4);
        w_13 : VitalWireDelay (IO5_ipd, IO5, tipd_IO5);
        w_14 : VitalWireDelay (IO6_ipd, IO6, tipd_IO6);
        w_15 : VitalWireDelay (IO7_ipd, IO7, tipd_IO7);
        w_18 : VitalWireDelay (SL_ipd, SL, tipd_SL);
        w_19 : VitalWireDelay (SR_ipd, SR, tipd_SR);

    END BLOCK;
```

Getting back to the unknown directionality problem of the INOUT ports, we need a way to tell whether the port is being driven from the RTL model or an external device. We know the RTL model will only output strong signals, ‘0’, ‘1’, and ‘Z’. So the next group of assignments takes the external inputs and converts them to weak signals, ‘L’, ‘H’, and ‘Z’.

```
IO_w(0) <= To_UXLHZ(IO0_ipd);
    IO_w(1) <= To_UXLHZ(IO1_ipd);
    IO_w(2) <= To_UXLHZ(IO2_ipd);
    IO_w(3) <= To_UXLHZ(IO3_ipd);
    IO_w(4) <= To_UXLHZ(IO4_ipd);
    IO_w(5) <= To_UXLHZ(IO5_ipd);
    IO_w(6) <= To_UXLHZ(IO6_ipd);
    IO_w(7) <= To_UXLHZ(IO7_ipd);
```

The assignments use a function found in the FMF.gen_utils package named To_UXLHZ. These function calls result in the assignment of weak signal values to each bit of the signal IO_w.

The RTL model has a vectored port. Most FPGA and ASIC models will have several vectored ports. However, at the board-level, the wrapper has scalar ports. A block statement is used to make the scalar to vector conversion that enables the advantageous use of vectored signals.

```
-------------------------------------------------------------------------------
    -- Main Behavior Block
    -------------------------------------------------------------------------------
    Behavior : BLOCK

    PORT(
        CLRIn    : IN    std_logic;
        OE1In    : IN    std_logic;
        OE2In    : IN    std_logic;
        S0In     : IN    std_logic;
        S1In     : IN    std_logic;
        CLkIn    : IN    std_logic;
        IOIn     : IN    std_logic_vector(7 downto 0);
        IOOut    : OUT   std_logic_vector(7 downto 0);
        Q0Out    : OUT   std_logic;
        Q7Out    : OUT   std_logic;
        SLIn     : IN    std_logic;
        SRIn     : IN    std_logic
    );
    PORT MAP (
        CLRIn => CLRNeg_ipd,
        OE1In => OE1Neg_ipd,
        OE2In => OE2Neg_ipd,
        S0In => S0_ipd,
        S1In => S1_ipd,
        CLKIn => CLK_ipd,
        IOIn => IO_w,
        IOOut(0) => IO0,
        IOOut(1) => IO1,
        IOOut(2) => IO2,
        IOOut(3) => IO3,
```

```
        IOOut(4) => IO4,
        IOOut(5) => IO5,
        IOOut(6) => IO6,
        IOOut(7) => IO7,
        Q0Out => Q0,
        Q7Out => Q7,
        SLIn => SL_ipd,
        SRIn => SR_ipd
    );
```

Within the block, all INOUT ports are split into separate IN and OUT ports. This provides some simplification.

The zero delay signals are declared:

```
SIGNAL IO_zd    : std_logic_vector(7 downto 0);
    SIGNAL Q0_zd    : std_ulogic;
    SIGNAL Q7_zd    : std_ulogic;
```

Because they are used both inside and outside the timing process, they are declared as signals in a wrapper instead of as variables as would usually be the case in a component model.

Once all the declarations are completed the RTL model is instantiated:

```
BEGIN
        -- connect VHDL RTL model
        fpga299_1 : fpga299
        PORT MAP(
            CLR_L   => CLRIn,
            OE1_L   => OE1In,
            OE2_L   => OE2In,
            S0      => S0In,
            S1      => S1In,
            CLK     => CLKIn,
            IO      => IO_w,
            Q0      => Q0_zd,
            Q7      => Q7_zd,
            SL      => SLIn,
            SR      => SRIn
        );
```

Input ports are mapped to the block IN ports and outputs are mapped to the block OUT ports. The IO port, which is of mode INOUT in the RTL model is mapped to the IO_w signal that was declared earlier. Values being read in from the outside on this signal will always be weak while values being driven from the RTL will always be strong. Therefore, any value originating from the RTL model will override the value coming from outside unless the RTL model is driving ‘Z’. This is an important characteristic and it will be exploited further down the code.

Now comes the main process. In a model this would be the behavioral process. In a wrapper, all the behavior is described in the RTL model so there is little left for this process other than timing constraints and path delays.

As always, the process begins with a sensitivity list and variable declarations.

```
-------------------------------------------------------------------------
    -- Main Behavior Process
    -------------------------------------------------------------------------
    TIMING : PROCESS (CLKIn, CLRIn, OE1In, OE2In, S0In, S1In, IO_w, SLIn, SRIn)

        -- Timing Check Variables
        VARIABLE Tviol_SL_CLK       : X01 := '0';
        VARIABLE TD_SL_CLK          : VitalTimingDataType;

        VARIABLE Tviol_SR_CLK       : X01 := '0';
        VARIABLE TD_SR_CLK          : VitalTimingDataType;

        VARIABLE Tviol_S0_CLK       : X01 := '0';
        VARIABLE TD_S0_CLK          : VitalTimingDataType;
```

```
        VARIABLE Tviol_S1_CLK       : X01 := '0';
        VARIABLE TD_S1_CLK          : VitalTimingDataType;

        VARIABLE Tviol_IO_CLK       : X01 := '0';
        VARIABLE TD_IO_CLK          : VitalTimingDataType;

        VARIABLE Rviol_CLRNeg_CLK   : X01 := '0';
        VARIABLE TD_CLRNeg_CLK      : VitalTimingDataType;
        VARIABLE PD_CLK         : VitalPeriodDataType := VitalPeriodDataInit;
        VARIABLE Pviol_CLK      : X01 := '0';

        VARIABLE PD_CLRNeg      : VitalPeriodDataType := VitalPeriodDataInit;
        VARIABLE Pviol_CLRNeg   : X01 := '0';

        VARIABLE Violation      : X01 := '0';

        -- Output Glitch Detection Variables
        VARIABLE Q0_GlitchData   : VitalGlitchDataType;
        VARIABLE Q7_GlitchData   : VitalGlitchDataType;
```

They are followed by the timing check section with its constraint procedure calls:

```
BEGIN

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

            VitalSetupHoldCheck (
                TestSignal      => S0In,
                TestSignalName  => "S0",
                RefSignal       => CLKIn,
                RefSignalName   => "CLK",
                SetupHigh       => tsetup_S0_CLK,
                SetupLow        => tsetup_S0_CLK,
                HoldHigh        => thold_S0_CLK,
                HoldLow         => thold_S0_CLK,
                CheckEnabled    => true,
                RefTransition   => '/',
                HeaderMsg       => InstancePath & partID,
                TimingData      => TD_S0_CLK,
                XOn             => XOn,
                MsgOn           => MsgOn,
                Violation       => Tviol_S0_CLK
            );

            VitalSetupHoldCheck (
                TestSignal      => S1In,
                TestSignalName  => "S1",
                RefSignal       => CLKIn,
                RefSignalName   => "CLK",
                SetupHigh       => tsetup_S0_CLK,
                SetupLow        => tsetup_S0_CLK,
                HoldHigh        => thold_S0_CLK,
                HoldLow         => thold_S0_CLK,
                CheckEnabled    => true,
                RefTransition   => '/',
                HeaderMsg       => InstancePath & partID,
                TimingData      => TD_S1_CLK,
                XOn             => XOn,
                MsgOn           => MsgOn,
                Violation       => Tviol_S1_CLK
```

```
);

VitalSetupHoldCheck (
    TestSignal      => SLIn,
    TestSignalName  => "SL",
    RefSignal       => CLKIn,
    RefSignalName   => "CLK",
    SetupHigh       => tsetup_SL_CLK,
    SetupLow        => tsetup_SL_CLK,
    HoldHigh        => thold_SL_CLK,
    HoldLow         => thold_SL_CLK,
    CheckEnabled    => true,
    RefTransition   => '/',
    HeaderMsg       => InstancePath & partID,
    TimingData      => TD_SL_CLK,
    XOn             => XOn,
    MsgOn           => MsgOn,
    Violation       => Tviol_SL_CLK
);

VitalSetupHoldCheck (
    TestSignal      => SRIn,
    TestSignalName  => "SR",
    RefSignal       => CLKIn,
    RefSignalName   => "CLK",
    SetupHigh       => tsetup_SL_CLK,
    SetupLow        => tsetup_SL_CLK,
    HoldHigh        => thold_SL_CLK,
    HoldLow         => thold_SL_CLK,
    CheckEnabled    => true,
    RefTransition   => '/',
    HeaderMsg       => InstancePath & partID,
    TimingData      => TD_SR_CLK,
    XOn             => XOn,
    MsgOn           => MsgOn,
    Violation       => Tviol_SR_CLK
);

VitalSetupHoldCheck (
    TestSignal      => IO_w,
    TestSignalName  => "IO",
    RefSignal       => CLKIn,
    RefSignalName   => "CLK",
    SetupHigh       => tsetup_IO0_CLK,
    SetupLow        => tsetup_IO0_CLK,
    HoldHigh        => thold_IO0_CLK,
    HoldLow         => thold_IO0_CLK,
    CheckEnabled    => true,
    RefTransition   => '/',
    HeaderMsg       => InstancePath & partID,
    TimingData      => TD_IO_CLK,
    XOn             => XOn,
    MsgOn           => MsgOn,
    Violation       => Tviol_IO_CLK
);

VitalRecoveryRemovalCheck (
    TestSignal      => CLRIn,
    TestSignalName  => "CLRNeg",
    RefSignal       => CLKIn,
    RefSignalName   => "CLK",
    Recovery        => trecovery_CLRNeg_CLK,
```

```
            ActiveLow       => TRUE,
            CheckEnabled    => TRUE,
            RefTransition   => '/',
            HeaderMsg       => InstancePath & partID,
            TimingData      => TD_CLRNeg_CLK,
            XOn             => XOn,
            MsgOn           => MsgOn,
            Violation       => Rviol_CLRNeg_CLK
        );

        VitalPeriodPulseCheck (
            TestSignal      => CLKIn,
            TestSignalName  => "CLK_ipd",
            Period          => tperiod_CLK_posedge,
            PulseWidthHigh  => tpw_CLK_posedge,
            PulseWidthLow   => tpw_CLK_negedge,
            CheckEnabled    => TRUE,
            HeaderMsg       => InstancePath & partID,
            PeriodData      => PD_CLK,
            XOn             => XOn,
            MsgOn           => MsgOn,
            Violation       => Pviol_CLK
        );

        VitalPeriodPulseCheck (
            TestSignal      => CLRIn,
            TestSignalName  => "CLRNeg",
            PulseWidthLow   => tpw_CLRNeg_negedge,
            CheckEnabled    => TRUE,
            HeaderMsg       => InstancePath & partID,
            PeriodData      => PD_CLRNeg,
            XOn             => XOn,
            MsgOn           => MsgOn,
            Violation       => Pviol_CLRNeg
        );

    END IF;
```

Since IO_w was used in its vector form in its setup/hold check only one procedure call was required rather than eight.

The final element in supporting the IO bus is the loop:

```
FOR i IN IO_w'range LOOP
        IF IO_w(i) = '1' OR IO_w(i) = '0' THEN
            IO_zd(i) <= IO_w(i);
        ELSE
            IO_zd(i) <= 'Z';
        END IF;
    END LOOP;
```

In the loop the strength of each bit in IO_w is tested. If the bit has a strong value. it must be coming from the RTL model so it is assigned to IO_zd. If it has a weak value, the RTL model is driving 'Z', so IO_zd gets 'Z'.

What remains of the wrapper is the path delays. The two scalar outputs have their path delay procedure calls within the timing process:

```
-------------------------------------------------------------------------
    -- Path Delay Section
    -------------------------------------------------------------------------
    VitalPathDelay01 (
        OutSignal       =>  Q0Out,
        OutSignalName   =>  "Q0",
        OutTemp         =>  Q0_zd,
        Paths           => (
            0 => (InputChangeTime   => CLRIn'LAST_EVENT,
                  PathDelay         => tpd_CLRNeg_Q0,
```

```
                    PathCondition     => true),
                1 => (InputChangeTime   => CLKIn'LAST_EVENT,
                    PathDelay         => tpd_CLK_Q0,
                    PathCondition     => true ) ),
            GlitchData      => Q0_GlitchData );

        VitalPathDelay01 (
            OutSignal       =>  Q7Out,
            OutSignalName   =>  "Q7",
            OutTemp         =>  Q7_zd,
            Paths           => (
                0 => (InputChangeTime   => CLRIn'LAST_EVENT,
                    PathDelay         => tpd_CLRNeg_Q0,
                    PathCondition     => true),
                1 => (InputChangeTime   => CLKIn'LAST_EVENT,
                    PathDelay         => tpd_CLK_Q0,
                    PathCondition     => true ) ),
            GlitchData      => Q7_GlitchData );

        END PROCESS;
```

In order to reduce the code size, the IO bus has its path delay in a generate statement outside the process:

```
IO_OUT : FOR i IN IO_zd'range GENERATE
        PROCESS(IO_zd(i))
            VARIABLE IO_GlitchData   : VitalGlitchDataType;

        BEGIN
        VitalPathDelay01Z (
            OutSignal       =>  IOOut(i),
            OutSignalName   =>  "IO",
            OutTemp         =>  IO_zd(i),
            Paths           => (
                0 => (InputChangeTime   => CLRIn'LAST_EVENT,
                    PathDelay         =>
                                VitalExtendToFillDelay(tpd_CLRNeg_IO0),
                    PathCondition     => true),
                1 => (InputChangeTime   => CLKIn'LAST_EVENT,
                    PathDelay         =>
                               VitalExtendToFillDelay(tpd_CLK_IO0),
                    PathCondition     => true ),
                2 => (InputChangeTime   => OE1In'LAST_EVENT,
                    PathDelay         => tpd_OE1Neg_IO0,
                    PathCondition     => true ),
                3 => (InputChangeTime   => OE2In'LAST_EVENT,
                    PathDelay         => tpd_OE1Neg_IO0,
                    PathCondition     => true ),
                4 => (InputChangeTime   => S0In'LAST_EVENT,
                    PathDelay         => tpd_S0_IO0,
                    PathCondition     => true ),
                5 => (InputChangeTime   => S1In'LAST_EVENT,
                    PathDelay         => tpd_S0_IO0,
                    PathCondition     => true ) ),
            GlitchData      => IO_GlitchData );
        END PROCESS;
   END GENERATE IO_OUT;

    END BLOCK;

END vhdl_behavioral;
```

This is a relatively simple wrapper for a very small FPGA. Still, it came out to 480 lines. Do not despair, most of it can be generated by a perl script or cut and pasted from other models.

2.2 A Wrapper for Verilog RTL

If you work in California, there is a good chance your RTL model is written in Verilog. If so, the exact same wrapper will work. As long as you have a bilingual simulator (like ModelSim), and you make the port names in the component declaration match those in your Verilog module, there is no difference. Just make sure to compile the Verilog into the same directory as the wrapper.

ModelSim has a utility to make the job easier. It is called "vgencomp". It will read your compiled Verilog model and generate a matching VHDL component declaration.

2.3 Modeling Delays in Designs with Internal Clocks

In the example wrapper, all path delays were timed to a transition on a port. This will not work for many modern ASIC/FPGA designs. Many designs today take advantage of internal DLLs, PLLs or clock multipliers. If an output is clocked by an internally generated signal another strategy is needed.

In most cases, the FPGA timing tool will report the delay of a signal relative to some input. However, that input may not be the signal that actually clocks the output. The example below is from the tco section of a .tao file from an Altera design.

```
Path Number        : 175
Slack              : 0.008 ns
Required tco       : 4.000 ns
Actual tco         : 3.992 ns
Source Name        : mDFB:DFB|mDRAM80:mDFBMem|mDRAM80IO:mDRAM80IO|cke
Destination Name   : dfb_cke
Source Clock Name  : acqdet_clkin
```

Although source clock is said to be `acqdet_clkin, in this design, it is really a PLL generated clock based on acqdet_clkin but running at twice the frequency.`

Because a RTL model has no internal delays, that delay value can be applied to the signal referencing only itself. The example shows a path delay statement that references a single signal.

```
VitalPathDelay01 (
            OutSignal       => DFBCKE0,
            OutSignalName   => "DFBCKE0",
            OutTemp         => DFBCKE0_zd,
            GlitchData      => DFBCKE0_GlitchData,
            XOn             => XOn,
            MsgOn           => MsgOn,
            Paths           => (
                0 => (InputChangeTime   => DFBCKE0_zd'LAST_EVENT,
                      PathDelay         => tpd_ACQDETCLKIN_DFBCKE0,
                      PathCondition     => TRUE)
            )
        );
```

In the above procedure call, DFBCKE0_zd is the input signal and the event on which the delay will be based. `If the` InputChangeTime was specified as "ACQDETCLKIN'LAST_EVENT" incorrect outputs would result.

The generic tpd_ACQDETCLKIN_DFBCKE0 is backannotated through SDF with a value of 3.992 ns.

It is not possible to use internal states to control path selection in a wrapper.

2.4 Caveats

The implementation of timing constraints in a wrapper is subject to some restrictions. The first of which is that internal state can not be used to enable/disable timing checks. Likewise internal signals are not accessible for use in timing checks.

In many models it is possible to use the violation flags to control some aspect of a model's behavior, such as corrupting memory if there is a violation during a write cycle. Similar opportunities may not exist in a wrapper.

In ASIC/FPGA design, it is common that differential inputs and/or outputs are handled by I/O cell selection. The differential signals do not appear as ports in the RTL model. However, the differential I/O will be included in the schematic. In such cases, the wrapper must serve as mediator between the schematic and the RTL model.

For differential outputs the solution is as simple as generating the compliment through inversion:

```
DFBCK1N_zd <= not(DFBCK1int);
DFBCK1_zd <= DFBCK1int;
DFBCK0N_zd <= not(DFBCK0int);
DFBCK0_zd <= DFBCK0int;
```

For differential inputs the solutions range from just passing through one of the two signals to more complex schemes that check that both signals are active and compliment each other. The best solution depends on the design and verification requirements.

3.0 Case Study

We were designing a new board at Siemens. It was a large board that was captured with hierarchical schematics using ConceptHDL. It was decided to simulate one block of the design. This block included two large Altera FPGAs, 90Mb of QDR SRAM, 2.25Gb of DDR DRAM, and a small amount of miscellaneous logic.

3.1 Integration with ConceptHDL

All components and their FMF VHDL models are kept in our libraries except the FPGAs. The FPGAs are local to the design. They are

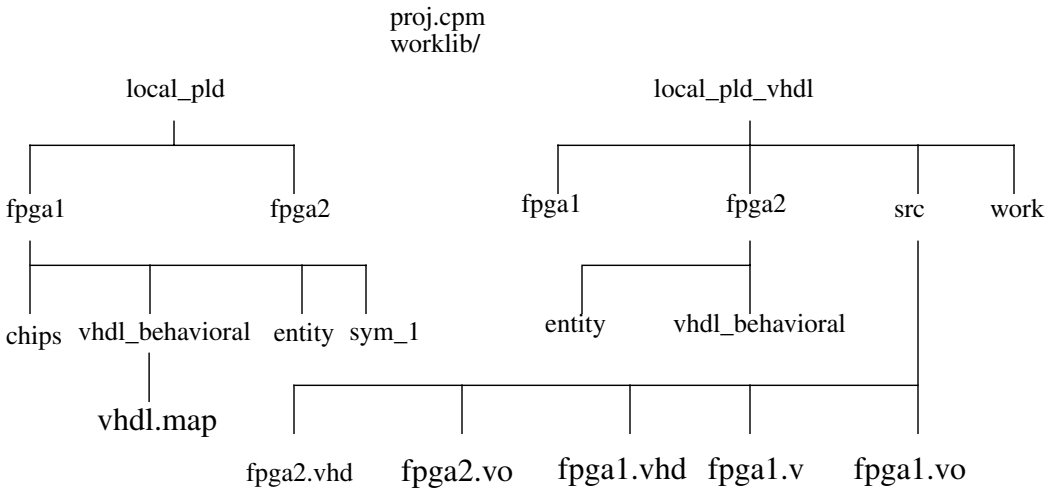

FIGURE 1. Project directory structure

placed in a directory named “local_pld” as shown in Figure 1. The file structure of local_pld is the same as for any component library. Likewise, because it references VHDL models, a directory named “local_pld_vhdl” is required by ConceptHDL. Each directory has a subdirectory for each FPGA and each of those has subdirectories named entity and vhdl_behavioral. The contents of those bottom sub-directories is quite different. In local_pld, the entity directory contains a number of files written by ConceptHDL when the symbol is written. In local_pld_vhdl, the entity directory contains a link named vhdl.vhd that points to the VHDL entity or model of the FPGA. In local_pld, the vhdl_behavioral directory contains a file named vhdl.map. This file maps the ConceptHDL pin names to the VHDL port names used in the model. In the local_pld_vhdl directory, the vhdl_behavioral directory contains a link named vhdl.vhd that points to the VHDL model of the FPGA.

Adding a level of complication is the fact that fpga2 was designed using Altera’s AHDL language. This effectively prevents simulation until after place and route. As a result, fpga2 uses a gate-level Verilog netlist (fpga2.vo) for simulation. The file fpga2.vhd is a VHDL entity of the component with exactly the same port names as the Verilog model. It is used only by the Cadence VHDL netlister and is not compiled. The other files in src are compiled into the work directory. Vmap is run in the simulation directory so ModelSim will be able to find these design units.

Fpga1 is simulated using either a Verilog RTL model (fpga1.v) or a Verilog gate-level netlist fpga1.vo). The file fpga1.vhd is a VHDL wrapper. It has an entity and two architectures. The first architecture instantiates the RTL model and adds a complete set of VITAL timing constraint checks and path delays. This allows the Verilog RTL model to be simulated in a VHDL netlist with realistic delays and the constraint checks of the gate-level model but with the performance of an RTL model.

3.2 Preparation for Netlisting

Some parts in the design are best left out of the simulation. De-coupling capacitors, for example, will add nothing to the functionality. They will just make the netlist longer. Series termination resistors also will not help your simulation but the pullup and pulldown resistors should stay.

ConceptHDL has an attribute that can help. The attribute name is “REMOVE”. There are four possible values. The two we have used are “LINK” and “EXCLUDE”. If a body has “REMOVE = EXCLUDE” attached to it, the netlister will remove that component from the netlist all together. This is good for all those de-coupling caps, voltage regulators and test points. If a body has “REMOVE = LINK” the two nets attached will be aliased, in effect shorting them together.This is good for series terminating resistors.

3.3 Netlisting

The ConceptHDL netlister is run to generate a VHDL netlist. Errors will most likely be found during netlisting but after several tries a netlist will be creeated. When the netlist is compiled with vcom, another set of errors may be discovered which could require either updating the schematic or the files in the library.

3.4 SDF Generation

Each FMF VHDL model has an associated timing file that describes the internal delays of a component along with any required timing constraints.

SDF generation produces a file in Standard Delay Format that may be used by the simulator to provide accurate timing for the simulation. Initial simulation runs may not require timing and can skip this step. Without SDF annotation, FMF models default to unit delays. However, some models such as certain memories, may give misleading results when run with unit delays.

Gate-level FPGA models will have their own SDF files produced by the FPGA place and route tool.

The SDF tool is called *mk_sdf* and may be obtained as a perl script from the Free Model Foundry (at no cost). It uses a command file named *mk_sdf.cmd* which should reside in the working directory.

```
SET sdffile_suffix .sdf
SET use_global_timing_dir false
SET timingfile_dir TimingModels
SET timingfile_suffix .ftm
SET time_scale 1ns
SET local_path .
SET diagnostics off
SET vhdl_file vhdllink.vhd
SET lwb off
```

FIGURE 2. Sample mk_sdf.cmd file

Some of the lines in the command file that may require some explanation are:

- SET sdffile_suffix .sdf
  - The SDF file will have the same name as the VHDL netlist with this suffix appended.
- SET use_global_timing_dir false
  - Set this to true if you will have all of your timing files in one local directory. Otherwise, mk_sdf will read your modelsim.ini file for the location of your libraries and assume the timing file directories are adjacent.
- SET timingfile_dir TimingModels
  - The name of the directory(ies) that contain the timing files.
- SET timingfile_suffix .ftm
  - Timing files use the same name as their associated models but with this extension instead of .vhd.

•SET diagnostics off
- If you are having difficulties getting mk_sdf to work, turning diagnostics on will result in some additional output files that may (or may not) help locate the problem.

•SET vhdl_file design_name.vhd
- The name of the netlist. This will typically be the design_name.vhd. It may also be supplied on the command line.

•SET lwb off
- mk_sdf understands the old Cadence lwb file structure. Turn this on only if you are running Logic Work Bench.

Any line in the command file may be overridden from the command line. Normal execution is accomplished by simply entering *mk_sdf* without any arguments.

More detailed instructions for mk_sdf can be downloaded with the script from http://www.eda.org/fmf/fmf_public_models/tools/.

3.5 Results

Using the previously described environment, the design was simulated. The testbench was derived from actual diagnostic code that will be run on the board when it is built. A number of problems were discovered during simulation. A few were simple schematic errors. Most were more complex.

Errors were found in the way the fpga1 initialized it DDR SDRAM. The SDRAM requires 200 microseconds to initialize. The FPGA was not waiting long enough. A more significant benefit was it made debugging the FPGA code easier because all the supporting components had models and were netlisted by the schematic. Otherwise, the testbenches would have had to include all that functionality and probably would not have had correct timing.

The biggest win was from being able to start writing and testing diagnostic code prior to the availability of hardware. Diagnostics were ready for board bring up when the boards came in from assembly instead of eight weeks later.

4.0 Summary

The complexity of the interfaces of today's ASICs and FPGAs make board level simulation desirable if not a necessity. Gate-level simulation is too slow and occurs too late in the design to fill this need. In this paper, a method has been presented that allows timing constraints and propagation delays to be added to the RTL mode of ASIC/FPGA so it may be used in board-level verification before chip-level place and route has been performed.

The method closely resembles the modeling of an off-the-shelf component except instead of coding the functionality, the RTL model is instantiated. Special care must be taken in dealing with ports of mode INOUT and with outputs clocked by internally generated signals.

Mentor's ModelSim supports board-level simulation including designs with embedded FPGAs. As always, capability and usability are strongly affected by library architecture. A good architecture combined with freely available VHDL component models facilitates the early verification of FPGA designs.