Tricks and techniques for writing VITAL 3.0 compliant ECL models

Russ Vreeland
Sr. Consulting Engineer
IKOS Systems
1921 Palomar Way, Suite 205, Carlsbad, CA 92008

(619) 603-3005

russv@ikos.com

Introduction:

The VITAL 3.0 standard was written with at least some consideration for providing model writers with enough tools to effectively do board level component modeling - not just ASIC and FPGA primitives. When board level models are very complex, or have unusual requirements not typically encountered by ASIC model writers, strict VITAL compliance often is abandoned. That is, the “Level 1” compliance level is abandoned - the “Level 0” compliance can and should be maintained for backannotation purposes in even the most complex models.

ECL models apparently must fall into the category of models “that have unusual requirements not typically encountered by ASIC model writers”. This paper will show why that is so, and will discuss several tricks and techniques for “moving the line” dividing models simple enough to write completely compliant to VITAL Level 1, and models too complex to do so farther towards the complex models. This greater adherence to the standard should prove beneficial in general since more models will be portable, readily readable to VITAL-knowledgable persons, and acceleratable by the simulators that take advantage of VITAL.

Specific areas investigated by this paper include: single-ended versus differential inputs and how to do them using VITAL, modeling ECL signal strengths, VITAL-compliant tables which effectively model ECL differential inputs and clocks, the case for a VitalSignalAssign primitive, and the use of VITAL processes in series - why series processes are the best way to do some ECL models and how to pass signals between processes.

Single-ended and differential signals in ECL parts.

ECL models are high-speed and often utilize differential signal lines - electrical signals consisting of a pair of logically opposed signal lines running generally side-by-side. The logic level of the

differential signal is measured by the difference in voltage of the two signal lines, thus, noise induced on the signal line pair is cancelled out by the measurement of the differential voltage level. An ECL part with a differential input can be converted to single-ended by connecting one of the input pairs to V_{BB} - a reference voltage signal level that is usually about halfway between ECL logic '0' and logic '1'. (Of course, the noise immunity of the circuit is reduced by a factor of 2 due to the decreased logic transition voltage and more due to the loss of the common mode noise rejection of the differential inputs.) If the negative input of the differential input pair is connected to V_{BB}, the input recognizes single-ended ("normal") logic signals on the positive input. If the positive input of the differential input pair is connected to V_{BB}, the input inverts the single-ended signal connected to the negative input.

In modeling ECL parts with differential signals, outputs represent a case that hardly needs discussion; the output signal is generated for one of the two opposite polarity pins, and the other is inverted.

With ECL differential inputs, the modeling is more complicated. There are two cases that we've isolated for our modeling at TRW, although more could be defined if greater accuracy were required. Both cases involve the treatment of input skew, or the delta time between the edge of one differential signal pair and the other during a logic transition. Skew can enter a simulation by backannotating PCB route delays onto the differential signal nets, for example. Both cases use the assignment of an internal signal or variable to represent as a scalar the result of an evaluation of the differential signal pair.

One case is defined for inputs where the value of the differential signal can be assigned std_logic value 'X' during any skew time. This is close to the real-world expectation of an ECL part's behavior since, when both halfs of a differential signal pair are at the same logic value (another way of defining skew) the noise immunity of the input is practically nothing. While some hysteresis may prevent a logic transition given little or no noise or overshoot on the the signal line, unknown behavior is a prudent assumption. An example of this case is the "D" differential inputs of an ECL flip flop. Since the "D" input is only relevant at the clock edge, and need be stable only for the setup and hold times measured from the clock edge, the state of the "D" input as seen by the model (or the internal state) can be accurately assigned 'X' during a skew interval when the logic value of "D" is changing.

Another case is defined for inputs where skew time between the positive and negative differential inputs is relatively short, but the value 'X' cannot be assigned to the internal signal or variable representing the differential signal pair during the skew time because 'X' should not propagate through the model. Instead, the model should wait until the lagging half of the differential signal pair "completes the transition", and make the change in logic value. In the real world, the circuit ought to have been designed so the skew time is minimal so that any noise-induced false transitions during skew time are less than the glitch sensitivity of the part (at this point, this paper cops a plea that the subject of high speed design is not the area of expertise of the writer and is not the subject of this paper). An ECL clock input is an example of this case; if during a rising edge event, the positive differential input transitions from logic '0' to '1' a few hundred

picoseconds skew time before the negative differential input transitions from logic ‘1’ to ‘0’, the internal value of the clock input within the model should stay logic ‘0’ until the negative differential input changes, then go to ‘1’.

Std_logic signal strengths.

In many ECL parts families, the output drivers are a ‘Wired-OR’ technology where outputs don’t drive the low logic value strongly and can be wired together and the resultant net resolved to a logic value ‘1’ if any of the outputs are at that value. In std_logic terms, the output drives are the weak low value ‘L’ when logic 0, and the strong high value ‘1’ when logic 1.

VITAL primitives provide an input parameter called ResultMap, of type VitalResultMap, which performs the mapping onto the default VITAL primitive output values (‘U’, ‘X’, ‘0’, ‘1’) of a user-defined set. To model ECL Wired-OR parts accurately, a ResultMap of the form (‘U’, ‘X’, ‘L’, ‘1’) maps the weak low value ‘L’ onto an output where ‘0’ would normally occur. We do not use the five-valued form of the ResultMap to map a different value, such as ‘W’, onto the default output value ‘Z’ because we don’t see a need for it, and we wish to reserve the use of ‘W’ (as described in a subsequent section).

If an ECL part is being driven with a weak signal value, a VITAL-compliant model can do one of two things. Either it can accept that value without modification, and VITAL procedures and functions will treat it like a strong value of the same logic type; or the model can somehow test the value to determine the strength and proceed accordingly - most likely treating the ‘L’ value on the input the same as a ‘Z’ or ‘X’ value.

Since calls to functions and procedures other than VITAL are not allowed in VITAL architectures, nor are control structures such as “IFs”, “CASEs”, or “LOOPs” allowed in VITAL processes, the model is somewhat limited in what it can do to test input strengths that are not tested in anything supplied by VITAL. Certainly the following is not allowed:

```
IF ( input_signal = ‘0’ or input_signal = ‘1’) THEN
     internal_input_variable := input_signal;
ELSE
     internal_input_variable := ‘X’;
END IF;
```

Fortunately, constants in VITAL architectures are not restricted, so a model can be constructed with static, constant lookup maps and tables to perform just about any conceivable function on std_logic values. (This paper uses the naming convention ‘map’ for a one-dimensional array and table for a 2-dimensional array). The only limitation is that their use must be restricted to variable assignments within processes, since VITAL explicitly outlaws signal assignments except in VITAL concurrent procedure calls, or in the VitalPathDelay procedure within a process.

An example of a lookup map for doing the above assignment of internal_input_variable consists of two parts. First a type declaration and a constant declaration are put into the architecture declaration region of the model.

```
TYPE stdlogic_map IS ARRAY(std_ulogic) OF std_ulogic;
CONSTANT ECL_no_weak_inputs: stdlogic_map :=
     ------------------------------------------------
     -- U .. X .. 0 .. 1 .. Z .. W .. L .. H .. - ...
     ------------------------------------------------
     ( 'X', 'X', '0', '1', 'X', 'X', 'X', 'X', 'X' );
```

The std_ulogic type inside the array guarantees that when the variable assignment from the table is performed, the variable will be of type std_ulogic which is required of VITAL process variables (types BOOLEAN or TIME are legal too but are not applicable here).

Then the variable assignment is performed in a process.

```
-- assume we're in a process ...
internal_input_variable := ECL_no_weak_inputs(input_signal);
```

The model uses internal_input_variable to determine the model behavior. If the input_signal has not been pulled-down, the model will propagate the 'X' value.

Utilizing Std_logic 'W' in ECL models to distinguish VBB.

Thus far, this paper has discussed std_logic strengths in regard to modeling ECL Wired-OR outputs and how to distinguish them at ECL model inputs. The std_logic value 'W' can only occur in a circuit using VITALmodels if it is explicitly driven from outside the models, if a model uses it in a ResultMap, or if two signals, one of value 'L' and the other of value 'H', are resolved.

If we rule out the use of 'W' and 'H' in any user-defined ResultMap (which is fine in the ECL Wired-OR environment - but might conceivably be a problem in an environment mixing, say, ECL Wired-OR parts and TTL open collector parts - but even there the two kinds of parts should never be wired directly together, should they?), and we take care not to drive a 'W' from a test bench or a script, 'W' can be used for other purposes without concern it might appear at the input to a model unintentionally.

We propose and use 'W' to indicate that an input to an ECL model is connected to VBB. Also, VBB outputs from ECL models are driven to 'W' by the model.

Then, techniques which build on the technique of using lookup maps and tables can be used within the model to determine whether the usage of the model is single-ended or differential, and process accordingly.

ECL differential “D” type inputs.

The case of an ECL flip flop with differential “D” inputs is handled using a lookup table to convert the 2 input port signals into an internal variable which behaves just like a single-ended “D” input. If one of the differential “D” inputs is std_logic ‘W’, it indicates the designer has used the ECL “D” flip flop in single-ended mode, and the other “D” input is a single-ended signal.

The type declaration and constant declaration for this lookup table are as follows.

```
TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;
CONSTANT ECL_s_or_d_inputs_tab: stdlogic_table := (
    ------------------------------------------------------
    -- U .. X .. 0 .. 1 .. Z .. W .. L .. H .. -...------
    ------------------------------------------------------
    ( ‘U’, ‘U’, ‘U’, ‘U’, ‘U’, ‘U’, ‘U’, ‘U’, ‘U’), -- U --
    ( ‘U’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’), -- X --
    ( ‘U’, ‘X’, ‘X’, ‘0’, ‘X’, ‘0’, ‘X’, ‘0’, ‘X’), -- 0 --
    ( ‘U’, ‘X’, ‘1’, ‘X’, ‘X’, ‘1’, ‘1’, ‘X’, ‘X’), -- 1 --
    ( ‘U’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’), -- Z --
    ( ‘U’, ‘X’, ‘1’, ‘0’, ‘X’, ‘X’, ‘1’, ‘0’, ‘X’), -- W --
    ( ‘U’, ‘X’, ‘X’, ‘0’, ‘X’, ‘0’, ‘X’, ‘0’, ‘X’), -- L --
    ( ‘U’, ‘X’, ‘1’, ‘X’, ‘X’, ‘1’, ‘1’, ‘X’, ‘X’), -- H --
    ( ‘U’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’, ‘X’) --- - --
    );
```

In a VITAL model, the port signals D and D_N, the differential inputs, propagated through a VitalWireDelay procedure become D_ipd and D_N_ipd respectively. The internal variable Dint_zd is assigned in a process as shown in the statement below. Note that the order of the array subscripts is important: if D_ipd has value ‘W’, the table inverts the value of D_N_ipd.

Dint_zd := ECL_s_or_d_inputs_tab(D_ipd, D_N_ipd);

The variable Dint_zd is used like a single-ended input to determine the behavior of the D flip flop. If that functionality is in the same process, this table lookup must precede it since processes are executed sequentially and variables are assigned immediately in VHDL.The ‘X’ values assigned to Dint_zd whenever skew in the transitions of D_ipd and D_N_ipd occur do not affect the model since Dint_zd should be stable when the clock edge occurs. The VITAL model ought to be written with Timing Checks for Setup and Hold if the data for these is available.

ECL Differential Clock inputs.

Differential clocks on ECL parts are modeled with two lookup tables. One is used to determine if the std_logic value ‘W’ is present on either CLK or CLK_N, indicating single-ended mode. The other is used to assign an internal scalar clock variable based on the transitions of the differential signals and the result of the first lookup.

In the architecture delclaration region the following TYPE and two CONSTANTs are declared.

```
TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;
CONSTANT ECL_diff_mode_tab: stdlogic_table := (
    ---------------------------------------------------
    -- U .. X .. 0 .. 1 .. Z .. W .. L .. H .. -...---------
    ---------------------------------------------------
    ( 'X', 'X', 'X', 'X', 'X', '0', 'X', 'X', 'X'), -- U --
    ( 'X', 'X', 'X', 'X', 'X', '0', 'X', 'X', 'X'), -- X --
    ( 'X', 'X', 'X', 'X', 'X', '0', 'X', 'X', 'X'), -- 0 --
    ( 'X', 'X', 'X', 'X', 'X', '0', 'X', 'X', 'X'), -- 1 --
    ( 'X', 'X', 'X', 'X', 'X', '0', 'X', 'X', 'X'), -- Z --
    ( '1', '1', '1', '1', '1', 'X', '1', '1', '1'), -- W --
    ( 'X', 'X', 'X', 'X', 'X', '0', 'X', 'X', 'X'), -- L --
    ( 'X', 'X', 'X', 'X', 'X', '0', 'X', 'X', 'X'), -- H --
    ( 'X', 'X', 'X', 'X', 'X', '0', 'X', 'X', 'X'), -- - --
    );
CONSTANT ECL_clk_tab : VitalStateTableType := (
    -- --INPUTS-------|--PREV--|-OUTPUT----
    -- CLK CLK_N Mode | CLKint | CLKint' --
    ---------------------------------------
    -- Below are single-ended clock cases
    ( '-', 'X', '1', '-', 'X' ), -- single-ended, Vbb on CLK
    ( '-', '0', '1', '-', '1' ), -- single-ended, Vbb on CLK
    ( '-', '1', '1', '-', '0' ), -- single-ended, Vbb on CLK
    ( 'X', '-', '0', '-', 'X' ), -- single-ended, Vbb on CLK_N
    ( '0', '-', '0', '-', '0' ), -- single-ended, Vbb on CLK_N
    ( '1', '-', '0', '-', '1' ), -- single-ended, Vbb on CLK_N
    -- Below are differential clock cases
    ( 'X', '-', 'X', '-', 'X' ), -- CLK unknown
    ( '-', 'X', 'X', '-', 'X' ), -- CLK unknown
    ( '1', '-', 'X', 'X', '1' ), -- Recover from 'X'
    ( '0', '-', 'X', 'X', '0' ), -- Recover from 'X'
    ( '/', '0', 'X', '0', '1' ), -- valid ECL rising edge
    ( '1', '\', 'X', '0', '1' ), -- valid ECL rising edge
    ( '\', '1', 'X', '1', '0' ), -- valid ECL falling edge
    ( '0', '/', 'X', '1', '0' ), -- valid ECL falling edge
    ( '-', '-', '-', '-', 'S' ) --- default
    ); -- End of VitalStateTableType definition
```

These lookup tables are used in a VITAL process by first declaring a variable Mode, of type X01, a subtype of std_logic. Mode is assigned by lookup from the ECL_diff_mode_tab table using the differential clock signals as indices. Mode has the value '1' if the differential clock input is single-ended, V_{BB} on CLK, the value '0' if single-ended, V_{BB} on CLK_N, or the value 'X' if it the clock is diffenential, no V_{BB} seen on either pin.

With Mode detemined, the second table, a VitalStateTable, is used to assign an internal variable CLKint_zd to represent the logical state of the clock. The VitalStateTable has one state to keep track of the previous value of CLKint_zd which is important for distinguishing the various cases (the full line-by-line study of the table is left as an exercise for the reader !). The full process that we use over and over again in our ECL parts is shown below. Notice that the process assigns a signal, CLKint, so that other processes may utilize it to implement the functionality of the model. ECLclock just does ECL clocks.

```
-------------------------------------------------------------------------
-- ECL CLock Process
-------------------------------------------------------------------------
ECLclock: PROCESS (CLK_ipd, CLK_N_ipd)

    -- Functionality Results Variables
    VARIABLE Mode     : X01 := 'X';
    VARIABLE CLKint_zd: std_ulogic := 'X';
    VARIABLE PrevData : std_logic_vector(1 to 3) := (OTHERS => 'X');

    -- Output Glitch Detection Variables
    VARIABLE CLK_GlitchData    : ValidGlitchDataType;

BEGIN

    -------------------------------------------------------------------------
    -- Funtionality Section
    -------------------------------------------------------------------------
    Mode := ECL_diff_mode_tab(CLK_ipd, CLK_N_ipd);

    VitalStateTable (
        StateTable        => ECL_clk_tab,
        DataIn            => (CLK_ipd, CLK_N_ipd, Mode),
        Result            => CLKint_zd,
        PreviousDataIn    => PrevData );

    -------------------------------------------------------------------------
    -- (Dummy) Path Delay Section
    -------------------------------------------------------------------------
    VitalPathDelay (
        OutSignal         => CLKint,
        OutSignalName     => "CLKint",
        OutTemp           => CLKint_zd,
        Paths             => ( 0 => UseDefaultDelay),
        GlitchData        => CLK_GlitchData );

END PROCESS;
```

Multiple processes and signal assignments.

The signal assignment in the last process example provides an internal clock signal that functions in exactly the same manner as a single-ended “normal” clock would if it were in the port list of the VHDL model. For example, our model for a differential “D” flip flop, the Motorola ECLinPS-Lite 100EL52, has 3 processes, of which the ECLclock process is one. A second “Dinputs” process converts the differential “D” inputs into an internal variable Dint_zd - using the same table lookup discussed earlier - and assigns a signal Dint with zero delay just like ECLclock does to CLKint. The third process does timing checks and implements a VitalStateTable for the simple “D” flip flop using Dint and CLKint instead of the differential input signals. VITAL timing checks are performed using Dint as the test signal and CLKint as the reference, alleviating the need for four times as many of them - one for each differential clock and “D” input.

One consideration to take into account in “splitting” functionality into multiple processes is a speed-up of the model that occurs because of the narrowed sensitivities of the separate processes. If a single, large process has all the model’s functionality, and an all-encompassing sensitivity list to boot, then it is activated from front to end every time any of those sensitivity list signals has an event. Smaller, separate processes have smaller sensitivity lists so that an event on a particular signal need not force the entire functionality of the model to be run. In this respect, the separate processes isolate and filter the activity of the model.

More about this “dummy” use of VitalPathDelay: one of the intended consequences of VITAL is to restrict the ways signals can be assigned within VHDL models, so that simulation vendors can have a very limited set of cases for event scheduling to deal with when they go about the task of accelerating their simulators. That is a layman’s way of describing it, but I think it’s close. To that end, in a process, VitalPathDelay is the ONLY legal way to assign a signal. Since VITAL3.0 explicitly allows multiple processes, it is inevitable that, for those models where processes “ in series” are a natural, efficient, and/or readable way to describe a part, VitalPathDelay is going to be used as I’ve used it here: as a dummy signal assignment statement.

By the way, “UseDefaultDelay” is just a constant which passes a null VitalPathType into the procedure. That parameter and the GlitchData parameter have to be passed into dummy VitalPathDelay because there is no default value declared in the procedure.

```
CONSTANT UseDefaultDelay : VitalPathType ( InputChangeTime => 0 ps,
                                           PathDelay => VitalZeroDelay,
                                           PathCondition => FALSE );
```

It is my hope that the VITAL TAG will put a new procedure into VITAL which ought to be called VitalSignalAssign. This procedure would only be allowed in processes, only be allowed once per signal (all operations would be done on the variable, not the signal), and would not unduly burden the efforts of simulation vendors seeking the highest possible performance. They could even treat VitalSignalAssign as an alias for the VitalPathDelay procedure, but with the zero-delay parameters automatically passed. My wish is to not have to write code that declares a GlitchData

variable, and passes a null VitialPathType when all I want to do is assign a zero-delayed signal. If implemented, the process signal assignment would look something like this.

```
VitalSignalAssign (SignalOut => CLKint, VarIn => CLKint_zd);
```

Other modeling consideration: input clamps, skew timing checks.

There are a couple of other topics for ECL modeling this paper will mention briefly, but not describe in detail. Input clamp circuits are used in ECL parts to provide a logic 0 when both of the differential inputs are left opened (by being pulled down to VEE or lower than the clamp voltage which is less than the logic 0 voltage). Parts that have input pulldown resistors are not sensitive to std_logic strengths, they can be modeled simply by assigning a default value of '0' in the input port list (takes care of the 'not connected' case) and relying on the VITAL procedures to convert weak strengths to strong. If both ports of a clamped differential signal are open, then the two '0's defaulted to on the signals may not decode to a logic 0 in the model. In the case of the differential "D" input lookup table discussed earlier, a 'X' would result. Perhaps the best answer for modeling of this detail might be to use another value of the std_logic system to indicate "not connected" and use a lookup table to test for pull-down and clamping conditions on the inputs. The value '-' is a good candidate to indicate an input port is pulled down to VEE with an input clamp circuit present. Of course, there is the consideration that the IEEE.std_logic_1164 package is a standard and ought not to be twiddled with in this fashion (though we have twiddled with the value 'W' already). We have chosen for the present to not model input clamp circuits.

Skew has been discussed with respect to inputs where skew time can be safely modeled as 'X', and clocks where the model ought to wait until the other differential signal "completes the transition". Models for some ECL parts might need values for maximum skew time allowed. If extreme thoroughness in the model is desired, one of the Vital Timing Check procedures can be cleverly used to check for maximum skew time exceeded. If the violation occurs, the model can set a violation variable and the ECL_clk_tab VitalStateTable can be enhanced to have a column for the violation input.

Conclusions.

It is possible to write models for board-level ECL parts, using VITAL, that contain highly accurate functionality for those parts down to the nitty-gritty of differential inputs, testing of strengths, or handling of skew to whatever detail desired, to mention a few. The techniques and tricks covered in this paper are proven to work, being the basis for our newer ECL libraries at TRW.

In the Cadence environment, Leapfrog supplies the simula - TOR (i.e., the person simulating) with one of its better products.The VHDL models are moderately easy to integrate into a mixed simulation environment using Opensim - we do this all the time with existing RapidSim library parts. Users wishing to implement a simulation environment based on standards, easily portable to other vendors' tools, and guaranteed to be valid for a long time (no pun intended), might want to seriously consider writing models in VITAL.