

# Adopting VHDL for PLD design and simulation

TROY SCOTT, ORCAD INC

As your designs become more complex, it becomes more efficient to use tools that allow you to design at a higher level of abstraction. Hardware-description languages, such as VHDL, are a natural next step for tackling large designs.

Time-to-market, vendor-flexibility, and design-complexity requirements have caused electrical engineers to adopt VHDL. One benefit of using VHDL for hardware design is design abstraction. Logic design using an HDL such as VHDL improves productivity by allowing you to work with logic operations and behavior rather than with the traditional approach of drawing circuit diagrams of logic gates and wires.

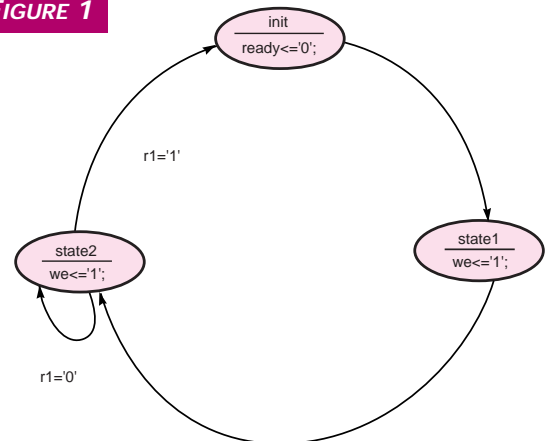
Using VHDL also results in improved documentation and quality. VHDL design encourages simulation. In addition to the obvious benefits of logic debugging, VHDL-testbench source code helps document design behavior. VHDL design also describes design inputs and expected outputs, making it easier to maintain the design after completion.

Another benefit of using VHDL is design portability. Perhaps the most compelling reason to adopt a standard design language is that it lets you easily migrate a design source from one PLD vendor to another. Engineers typically

describe schematic-based designs with PLD-vendor-specific logic symbols. These schematics require major overhauls and component substitution if you redo the design using another vendor's PLDs.

Simulation is another benefit of using VHDL. Unlike traditional Boolean languages, VHDL is a naturally simulatable language. You can debug your design intent early in the design and continually verify that the design does what you

FIGURE 1



```

-- state machine process
process (current_state,r1) begin
  we<='0';
  case current_state is
    when init => ready<='0';
                    next_state<=state1;
    when state1 => we<='0';
                    next_state<=state2;
    when state2 => we<='1';
                    if r1='0' then
                        next_state<=state2;
                    else
                        next_state<=init;
                    end if;
    when others =>
                        next_state<=init;
  end case;
end process;
  
```

## LISTING 1—GENERAL SYNTAX OF A VHDL CASE STRUCTURE

```

case expression is
  -- branch #1
  when choices => sequential-statements
  -- branch #2
  when choices => sequential-statements
  -- last branch
  [ when others => sequential-statements ]
end case;
  
```

This simple state diagram represents a three-state FSM. The accompanying VHDL code describes the FSM's operation.

## VHDL FOR PLDs

intend it to do as it evolves from equations to gates. In addition, VHDL is a powerful way to model almost any aspect of a digital system.

A state-machine description is one of the most appropriate applications of VHDL for hardware design. The schematic equivalent of a state machine is awkward to implement and difficult to interpret because of the irregular Boolean logic required to decode next-state logic. Finite-state-machine (FSM) logic has both input stimuli and output responses, but it also has several bits of internal memory to keep track of the most recent events. A state machine uses a clock to advance one step at a time. VHDL state machines are coded with a combinatorial block that decodes and controls current states and next states. A VHDL-coded state machine also includes a register block to synchronize the machine with the system clock and to apply other controls, such as preset or reset. Each active-clock transition causes a change from the current state to the next state.

### Combinatorial logic

To implement the combinatorial logic of the FSM, you use a VHDL case structure. VHDL case/when uses the same semantics to control program branching in most high-level programming languages. **Listing 1** shows the general syntax for a VHDL case structure. Leveraging this expression for an FSM design, the case statement selects one of the states (branch) to be executed based on the current value of the state bits (expression). **Figure 1** is a state diagram for a simple three-state FSM, along with the VHDL description of the FSM.

### Sequential logic

To implement the register block of the FSM, you use a VHDL if-then structure, according to the style recommended by your synthesis tool for flip-flop inference (**Listing 2**).

A key consideration of FSM design is how many registered elements you need to portray the machine's current state.

### LISTING 2—A VHDL IF-THEN STRUCTURE

```
-- register block
process (ck,reset,next_state)
begin
  if reset='1' then
    current_state<=init;
  elsif ck='1' and ck'event then
    current_state<=next_state;
  end if;
end process;
```

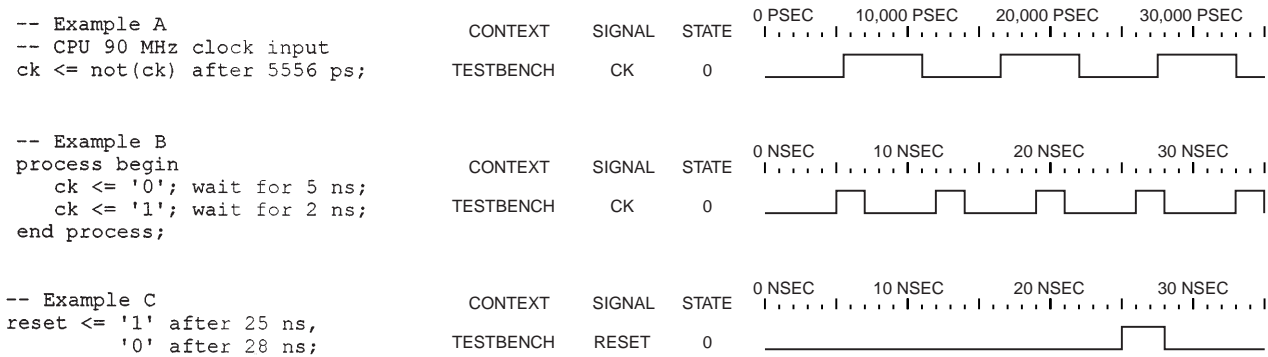
### LISTING 3—TWO SIGNALS FOR DESCRIBING THE FSM'S STATE

```
architecture behave of fsm is
  type state_type is (init,state1,state2);
  signal current_state,
         next_state : state_type;
begin
```

The traditional approach to encoding an FSM minimizes the number of flip-flops you need to represent  $n$  states. However, given the large number of registered elements in modern FPGAs, dedicating one flip-flop per state is the preferred approach to encoding large state machines. The VHDL style guides that accompany programmable-logic-vendor place-and-route software kits document this technique, "one-hot encoding" (OHE).

Designers prefer the OHE method because it requires min-

**FIGURE 2**



These example VHDL statements model a 50% duty cycle, 90-MHz clock (a); a variable-duty-cycle clock (b); and a reset pulse (c).

**LISTING 4—A VHDL LOOP STRUCTURE**

```

architecture arch_name of testbench is
  signal nibble : std_logic_vector(2 downto 0);
begin
  process begin
    for i in 0 to 7 loop
      nibble <= std_logic_vector(to_unsigned(i,3));
      wait for 100 ns;
    end loop;
    wait;
  end process;
end arch_name;

```

Signals = time testbench.nibble

Radix = nsec Hex

Type I/O

Width 3

0	0
100	1
200	2
300	3
400	4
500	5
600	6
700	7

imal decoding to arrive at the next state of the machine. OHE increases speed and reduces signal fan-in to each FPGA hardware-logic block.

Most logic-synthesis tools allow you to control the encoding scheme that the software implements. Typically, you define the state labels as a custom VHDL data type. You then use two signal sets to portray each state. In [Listing 3](#), VHDL declares and defines *state\_type* to represent all state labels used in the FSM of [Figure 1](#): *init*, *state1*, and *state2*. The code declares two signals, *current\_state* and *next\_state*, of type *state\_type* to hold the FSM state signals.

These VHDL declarations give no explicit encoding. Instead, the system interface to the software (switches, dialogue boxes, and command-line scripts) controls the encoding scheme at compilation time. All logic-synthesis tools document how to control state encoding.

The power of a VHDL testbench

VHDL is a powerful test-vector-generation and output-verification language. Most published technical articles regarding VHDL focus on synthesis topics and ignore the benefits gained by designers who adopt the language purely as a vehicle to create input stimuli for simulation. Even if you intend to use only schematics to design your programmable-

**LISTING 5—PLD BUS-INTERFACE MODEL**

```

architecture arch_name of testbench is
  type table_type is array(0 to 7) of
    std_logic_vector(2 downto 0);
  constant input_vector : table_type :=
    (x"0", x"1", x"2", x"3",
     x"4", x"5", x"6", x"7");
  signal nibble : std_logic_vector(2 downto 0);
begin
  process begin
    for i in 0 to 7 loop
      nibble <= input_vector(i);
      wait for 100 ns;
    end loop;
    wait;
  end process;
end arch_name;

```

Signals = time testbench.nibble

Radix = nsec Hex

Type I/O

Width 3

0	0
100	1
200	2
300	3
400	4
500	5
600	6
700	7

logic parts, VHDL can help debug and confirm component behavior before you program these parts.

VHDL describes the behavior of digital hardware. This attribute provides a powerful way to describe input-signal patterns for your programmable part. The synthesis tool never interprets the testbench, which surrounds the design under test. This attribute frees you to use VHDL's expressive power, regardless of synthesis-style guidelines.

Generating clocks

Consider an example of a VHDL expression to produce a 90-MHz clock with a 50% duty cycle ([Figure 2a](#)) and the accompanying source-code sample. Assume that signal *ck* initializes to 0. When the simulator encounters the expression in [Figure 2a](#), *ck* toggles after the specified period. This VHDL concurrent expression executes indefinitely, producing a repeating pattern of zeros and ones. If you need a variable duty cycle, you use a VHDL process to portray the difference between on- and off-times ([Figure 2b](#)). You model a

## VHDL FOR PLDs

**LISTING 6—A VHDL ENTITY/ARCHITECTURE PAIR**

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity entity_name is
  port(port1_name : in std_logic;
        port2_name : out std_logic);
end entity_name;

architecture arch_name of entity_name is
-- insert declarations here
begin
-- insert statements here
end arch_name;

```

sample reset pulse by the VHDL concurrent expression of [Figure 2c](#).

You use expressions and tables to model the bus interface to the PLD under test. To quickly generate a pattern of input vectors for a bus, you use the VHDL loop statement. VHDL loop uses the same semantics to control a looping situation that you have in most high-level programming languages. [Listing 4](#) uses a simple VHDL loop structure to generate values 0 to 7. The sequential statement within the loop converts the loop index to a `std_logic_vector` data type that is compatible with the interface of the PLD under test.

An alternative approach to using a VHDL expression to compute the input stimulus data is to store the data as a table either within the testbench or in a file. [Listing 5](#) uses an array of `std_logic_vector` to store the bus data. You use a loop to index and apply each member of the array to the bus interface. This sample mimics the example in [Listing 4](#), but it allows for a non-sequential pattern of input data as you customize the table.

If you've drawn a schematic, you're already familiar with "structural modeling" in VHDL. An annotated part in a schematic is equivalent to a VHDL component instance, a local wire on the schematic is a VHDL signal, the pinout of a library port is the VHDL entity interface, and so on. The terminology changes in VHDL, but the concepts are the same. The structural-modeling chapter of a VHDL textbook

should be familiar ground for most electrical engineers.

Block diagrams, library parts, and VHDL components are a natural way to partition a design. However, what is the best approach to connecting them? The basic VHDL design unit is an entity/architecture pair ([Listing 6](#)). The entity section defines the interface to the hardware model and is equivalent to the pinout of a schematic-library component: signal name, port mode (such as input and output), and data type. It's easy to see the correlation between the name and type declarations of the VHDL model to a PLD schematic-library component ([Table 1](#)). In schematic editors, the most common pin types are input, output, and bidirectional. These pin types are equivalent to the VHDL port modes: in, out, and inout, respectively.

As an example, consider the FDCE flip-flop defined by the Xilinx ([www.xilinx.com](http://www.xilinx.com)) XC4000E schematic library and an equivalent VHDL entity with an equivalent interface ([Figure 3](#)). The port data type, `std_logic`, is an extra aspect of the model that does not appear as part of the schematic part definition. The data type defines the potential signal states that the port may represent. The data type `std_logic` is part of the VHDL `std_logic_1164` package, which most VHDL-simulation and -synthesis tools support. This database defines VHDL data types and functions for digital logic. `Std_logic` is a nine-state system that represents the most common digital-circuit conditions ([Table 2](#)). Each of these nine states may appear on an `std_logic` signal or port.

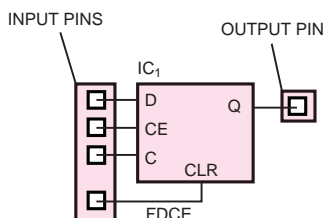
The most common logic levels are logic true/false

**TABLE 1—PIN DEFINITION FOR AN FDCE-LIBRARY FLIP-FLOP**

Pin name	Pin type
D	Input
CE	Input
C	Input
CLR	Input
Q	Output

**TABLE 2—LIST OF LOGIC STATES THAT CAN APPEAR ON AN STD\_LOGIC SIGNAL OR PORT**

Data state	Description
U	Uninitialized
X	Forcing unknown
0	Forcing 0
1	Forcing 1
Z	High impedance
W	Weak unknown
L	Weak 0
H	Weak 1
."	Don't care

**FIGURE 3**

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity FDCE is
  port(D, CE, C, CLR : in std_logic;
        Q : out std_logic);
end FDCE;

architecture BEHAVIOR of FDCE is
-- insert declarations here
begin
-- insert statements here
end BEHAVIOR;

```

This VHDL code models an FDCE flip-flop from the Xilinx XC4000E family.



## VHDL FOR PLDs

U1:AND2 to pass through the circuit.

- 400 nsec: The testbench releases the output-enable signal. The circuit responds by generating a high-impedance output.

In a paper written by engineers at Hughes Aircraft Co (Westchester, CA), the authors describe the design method used by a team to develop the winning VHDL-based application of the VIUF (VHDL International Users' Forum) '97 design contest ([Reference 1](#)). The process used by Hughes has evolved over the past five years and the company has successfully employed it in a variety of programs from a multiASIC space-based surveillance system to an FPGA-based video-processing subsystem.

Hughes divides its VHDL system design into three main tasks ([Figure 5](#)). The company splits the design-synthesis phase into four subtasks covering VHDL coding through synthesis-results analysis ([Figure 6](#)). The primary goals of development efforts at Hughes are to reduce design-cycle

time and to minimize design cost. Given VHDL as the vehicle for describing systems, Hughes establishes four design phases to help reduce design-cycle time: design reuse, top-down design entry, "real-world" testbenches, and design error containment.

To maximize VHDL reuse, you must keep reuse in mind throughout the design process. You must thoroughly comment the design unit, give meaningful names to signals and ports, and create a testbench to document design functionality. Other techniques that ease VHDL-design-unit portability include using standard data types, operators, and attribute-free source code.

### LISTING 7—VHDL BIDIRECTIONAL-PORT CIRCUIT MODEL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity bi_buf is
  port(A,B,C,Out_Enable: in std_logic;
        Bus0:             inout std_logic;
        O0:               out std_logic);
end;

architecture behavior of bi_buf is
  signal fb : std_logic;
begin
  -- tri-state process
  process (A,B,Out_Enable)
  begin
    if Out_Enable = '1' then
      Bus0 <= A and B;
    else
      Bus0 <= 'Z';
    end if;
  end process;
  -- feedback buffer
  fb <= Bus0;
  -- output logic
  O0 <= fb or C;

end;
```

### LISTING 8—SAMPLE VHDL TESTBENCH

```
-- Test bench sample for bidirectional
-- macrocell circuit
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity test_bi_buf is end test_bi_buf;

architecture testbench of test_bi_buf is
  -- input ports
  signal a,b,c : STD_LOGIC:= '0';
  signal out_enable : STD_LOGIC:= '0';
  -- bidirectional and output ports
  signal bus0 : STD_LOGIC;
  signal o0 : STD_LOGIC;

begin
  a      <= '1' after 100 ns;
  b      <= '1' after 100 ns;
  out_enable <= '1' after 50 ns,
           '0' after 100 ns,
           '1' after 350 ns,
           '0' after 400 ns;
  bus0   <= '0' after 150 ns,
           '1' after 200 ns,
           '0' after 250 ns,
           'Z' after 300 ns;

  dut : bi_buf port map (
    out_enable => out_enable,
    bus0 => bus0,
    o0 => o0, a => a, b => b, c => c);
end testbench;
```

## VHDL FOR PLDs

With top-down design, the design should flow naturally from requirements and decompose into hierarchical subunits. This flow ensures that the design meets all design requirements and that you can trace it.

A robust testbench makes a design simulation that is as close to the operating environment as possible. The testbench represents the design's interface to other digital system components. A robust interface model serves as a natural functional specification of the design unit because it may document software test vectors: input stimuli and expected outputs. The model also qualifies the data type and ranges allowed for information that flows in and around the design unit. You should also be aware of the delays that the circuit introduces, which are created by synthesis or timing analysis of the placed and routed circuit. A robust testbench anticipates the signal latency caused by potentially hundreds of internal logic levels. The testbench supplies data at a rate such that it holds data stable before clocking and that output checks anticipate output signal skew.

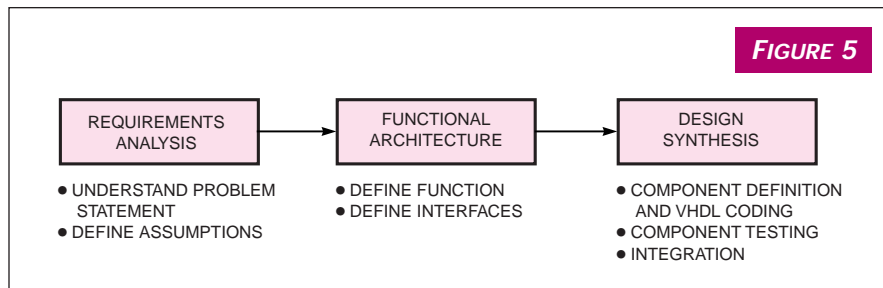
In general, when trying to contain design errors, the least expensive fix for design problems occurs early in the design's development.

**Reference 1** documents a high-level methodology and justification for hardware development at Hughes Aircraft with VHDL. The authors base their advice on experience that other designers can use to improve productivity at their own companies. Another key design-productivity trick is to record errors during the design. This record of experiences allows engineering managers to better forecast the schedule of development efforts and to back up their projections with data.

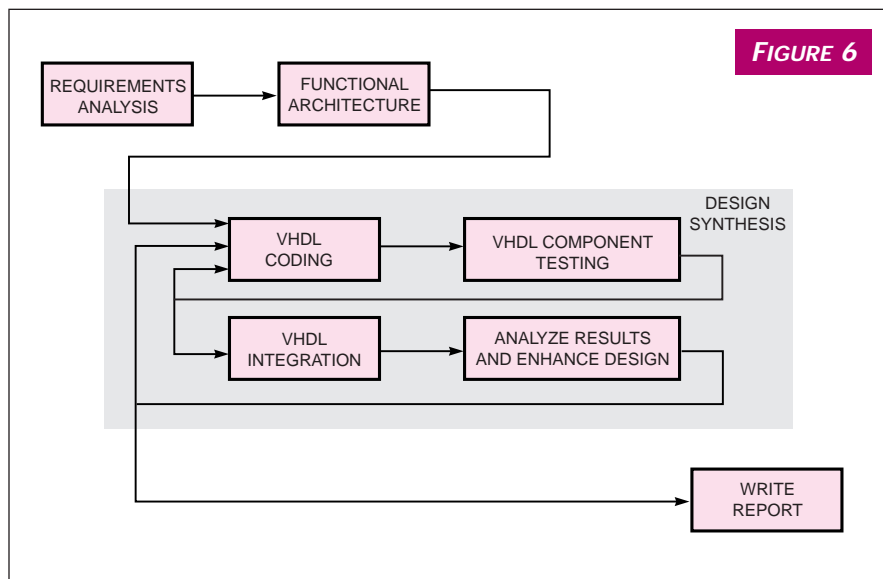
EDN

## References

1. McCoullough, Mike, and W Lee, "Processes, Metrics, and VHDL," VIUF Spring 1997 Conference Notebook of Sessions, pg 87.
2. Rushton, Andrew, *VHDL for Logic Synthesis*, McGraw-Hill Book Co Europe, Berkshire, UK, 1995.
3. Cohen, Ben, *VHDL Answers to Frequently Asked Questions*, Kluwer Academic Publishers, Norwell, MA, 1996.
4. Bhasker, J, *A VHDL Primer*, Prentice Hall Series in Innovative Technology, Englewood Cliffs, NJ, 1995.
5. *OrCAD Express for Windows, VHDL Style Guide*, online help system, [www.orcad.com](http://www.orcad.com).



Hughes divides its VHDL design process into three major phases, encompassing design-requirements analysis through synthesis.



Hughes splits its design-synthesis task into VHDL coding, integration, and test, and results analysis.

## Author biography



*Troy Scott is the technical marketing manager for programmable-logic solutions at OrCAD (Beaverton, OR). He received his BSCE degree with a technical communication option from the Oregon Institute of Technology (Klamath Falls, OR). During his five years at OrCAD, Scott has worked in technical services, documentation, testing, marketing, and software engineering. His current job concerns high-level design methodologies for programmable logic using EDA technology.*

## VOTE

Please use the Information Retrieval Service card to rate this article (circle one):

High Interest	Medium Interest	Low Interest
598	599	600