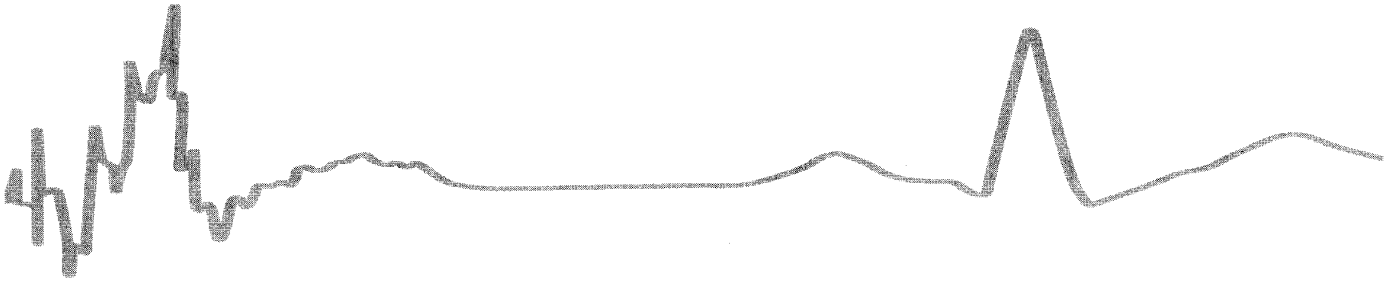
FEATURE



Designing a Custom DSP Circuit Using VHDL

Apart from providing a DSP designer with a wide range of modeling capabilities in one standard language, VHDL assists in the complete top-down design process (algorithm to application). Here, we describe a conversion circuit in a succession of high-level to bit-level models. We suggest using structured programming practices when developing complex VHDL descriptions.

Krishna A. Kumar
Intermetrics, Inc.

Brian Petrasko
University of Central Florida

With digital signal processing systems growing in size and complexity, designers require increasingly sophisticated computer-aided design methods throughout the design and development process. Traditional computer-aided engineering tools, which primarily support the development phase, do not perform effectively in the design of algorithms and architectures. In addition, they often lack facilities for the exchange of designs and the insertion of new technologies.

VHDL addresses these issues and offers the DSP designer a wide range of modeling capabilities for examining algorithms, architecture, and technologies. VHDL (VHSIC, or Very High-Speed IC, Hardware Description Language) is the IEEE standard language for the description of electronic circuits. It is the language of choice for the majority of ASIC (application-specific IC) designers, and most of the major CAD vendors have integrated VHDL tools into their product lines.[1] These tools include simulators, debuggers, graphics interfaces, and synthesizers.[2]

Here, we describe the application of VHDL in the design and development of a custom DSP complex-to-magnitude circuit. This circuit produces the magnitude of a complex number using an approximation algorithm. The target implementation is a bit-serial architecture using a set of primitive operators for DSP design. We investigate the algorithm and architecture at the word level using the real number type and apply the results to the bit level. We drew this example from Denyer and Renshaw's case study of the design of a 16-point FFT (fast Fourier transform) machine.[3]

A design environment

Many DSP applications demand high throughput and real-time response, performance constraints that often dictate unique architectures with high levels of concurrency. DSP designers need the capability to manipulate and evaluate complex algorithms to extract the necessary level of concurrency. Performance constraints can also be addressed by applying alternative technologies. A change at the implementation level of design by the insertion of a new technology can often make viable an existing marginal algorithm or architecture.

0272-1732/90/1000-0046$01.00 © 1990 IEEE

The VHDL language supports these modeling needs at the algorithm or behavioral level, and at the implementation or structural level. It provides a versatile set of description facilities to model DSP circuits from the system level to the gate level. Recently, we have also noticed efforts to include circuit-level modeling in VHDL.

At the system level we can build behavioral models to describe algorithms and architectures. We would use concurrent processes with constructs common to many high-level languages, such as if, case, loop, wait, and assert statements. VHDL also includes user-defined types, functions, procedures, and packages.[4] In many respects VHDL is a very powerful, high-level, concurrent programming language.

At the implementation level we can build structural models using component instantiation statements that connect and invoke subcomponents. The VHDL generate statement provides ease of block replication and control. A dataflow level of description offers a combination of the behavioral and structural levels of description. VHDL lets us use all three levels to describe a single component.

Most importantly, the standardization of VHDL has spurred the development of model libraries and design and development tools at every level of abstraction. VHDL, as a consensus description language and design environment, offers design tool portability, easy technical exchange, and technology insertion.

VHDL: The language

An entity declaration, or entity, combined with an architecture or body constitutes a VHDL model. VHDL calls the entity-architecture pair a design entity. By describing alternative architectures for an entity, we can configure a VHDL model for a specific level of investigation. The entity contains the interface description common to the alternative architectures. It communicates with other entities and the environment through ports and generics. Generic information particularizes an entity by specifying environment constants such as register size or delay value. For example,

```
entity A is
    port (x,y: in real ; z: out real);
    generic (delay :time);
end A ;
```

The architecture contains declarative and statement sections. Declarations form the region before the reserved word *begin* and can declare local elements such as signals and components. Statements appear after *begin* and can contain concurrent statements. For instance,

```
architecture B of A is
    component M
        port ( j : in real ; k : out real);
    end component;
        :
    signal a,b,c : real := 0.0;
        :
begin
    "concurrent statements"
end B;
```

The variety of concurrent statement types gives VHDL the descriptive power to create and combine models at the structural, dataflow, and behavioral levels into one simulation model.

The structural type of description makes use of component instantiation statements to invoke models described elsewhere. After declaring components, we use them in the component instantiation statement, assigning ports to local signals or other ports and giving values to generics.

```
invert: M port map ( j => a ; k => c);
```

We can then bind the components to other design entities through configuration specifications in VHDL's architecture declarative section or through separate configuration declarations.

The dataflow style makes wide use of a number of types of concurrent signal assignment statements, which associate a target signal with an expression and a delay. The list of signals appearing in the expression is the sensitivity list; the expression must be evaluated for any change on any of these signals. The target signals obtain new values after the delay specified in the signal assignment statement. If no delay is specified, the signal assignment occurs during the next simulation cycle:

```
c <= a + b after delay;
```

VHDL also includes conditional and selected signal assignment statements. It uses block statements to group signal assignment statements and makes them synchronous with a guarded condition. Block statements can also contain ports and generics to provide more modularity in the descriptions.

We commonly use concurrent process statements when we wish to describe hardware at the behavioral level of abstraction. The process statement consists of declarations and procedural types of statements that make up the sequential program. Wait and assert statements add to the descriptive power of the process statements for modeling concurrent actions:

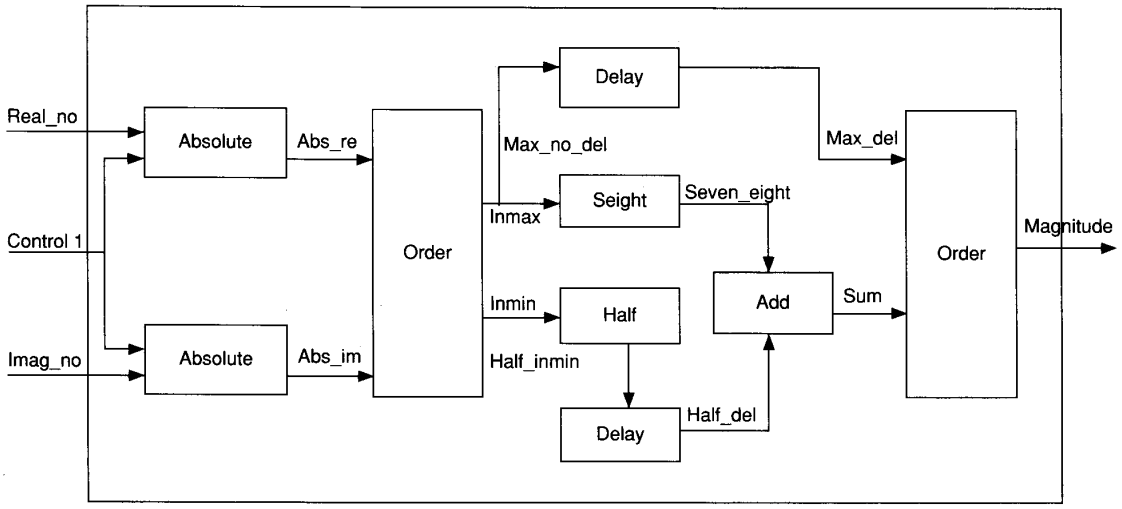


Figure 1. Dataflow diagram for a complex-to-magnitude algorithm.

```
process
    variable i : real := 1.0;
begin
    wait on a;
    i = b * 3.0 ;
    c <= i after delay ;
end process ;
```

Other concurrent statements include the concurrent assertion statement, concurrent procedure call, and generate statement. Packages are design units that permit types and objects to be shared. Lipsett, Schaefer, and Ussery provide further VHDL language information.[5]

Algorithm to architecture

The four-region approximation algorithm for converting a complex number $I + jQ$ to a magnitude number M, as presented by Denyer and Renshaw, is:

$$M = \text{Max} \begin{cases} |I| \\ 7/8|I| + 1/2|Q| \\ 1/2|I| + 7/8|Q| \\ |Q| \end{cases}$$

This form of the algorithm implies a maximum operator that produces the largest of four numbers. The variation shown next uses an Order operator to find the greater G and lesser L of the absolute values of the real component I and the imaginary component Q of the complex number.

$$G,L = \text{Order} \{ |I|,|Q| \}$$

$$M = \text{Max} \begin{cases} G \\ 7/8G + 1/2L \end{cases}$$

Figure 1 is a dataflow diagram of an architecture for this algorithm. All of the operators except Seight (seven-eights) come from a primitive DSP library, which includes operator latency or delay. We added delay operators to accommodate the latency of the various primitive operators.

We initially expressed the ctom algorithm behaviorally as a single process statement (Figure 2). We embedded statements for stimulus and control in the process model. From an external file we read inputs to stimulate the model and included an assert statement to end the simulation once all of the inputs were read. The use of an external file offered the flexibility to change the inputs and rerun the model with relative ease. We used this simple process model to develop test data, which we used throughout the design.

Figure 3 shows a behavioral model with an interface description and timing information. We included ports in the entity declaration and a signal assignment statement with a delay expression in the architecture. A package construct, which is shown in Figure 4, is an alternative to the generic construct and provides the delay constants. We calculated the timing information using library information, but it can also be derived from the simulation of a more detailed model.

We simulated ctom in Figure 3 using a design entity test_bench, which is shown in Figure 5 on page 50. We used a structural description for the test bench and

```
entity ctom_algo is end;

use std. textio.all;
architecture behav of ctom_algo is
  type ip_value_type is array (0 to 1) of real;
  signal real_no, imag_no, magnitude : real :=
  0.0000 ;
begin
    main:process (real_no, imag_no)
    file inputs:text is in "test_ip.dat";
    variable next_line : line ;
    constant half : real := 1.0/2.0 ;
    constant seven_by_eight : real := 7.0/8.0 ;
    variable result, greater, lesser, var_1 : real ;
    variable temp_ip : ip_value_type;
  begin
    readline(inputs,next_line);
    read (next_line,temp_ip(0));
    read (next_line,temp_ip(1));
    real_no <= temp_ip(0);
    mag_no <= temp_ip(1);
    if abs(real_no) < abs(imag_no) then
      greater := abs(imag_no) ;
      lesser := abs(real_no) ;
    else
      greater := abs(real_no) ;
      lesser := abs(imag_no) ;
    end if ;
    var_1 := (seven_by_eight * greater) + (half *
      lesser) ;
    if greater > var_1 then
      result := greater ;
    else
      result := var_1 ;
    end if ;
    Magnitude <= result ;
    assert not(endfile(inputs))
      report "Simulation Completed"
      severity error ;
  end process ;
end behav ;
```

Figure 2. Initial behavioral description with embedded simulation stimulus and control statements.

included front_end and clock components. The front-end component consists of statements similar to those found in Figure 2 for stimulus and control. The use of a test bench for testing individual components as well as the final design is an important part of the model development process.

Figure 6 on page 50 depicts a dataflow architecture for ctom. This level of description provides a convenient way of encoding a dataflow diagram such as that found in Figure 1. Note that a signal assignment statement describes the behavior and timing of each block of

```
entity ctom is
port ( real_no, imag_no : in real ; control_1 : in bit;
  magnitude : out real ) ;
end ctom;

use work.ctom_constants.all;
architecture behav of ctom is
begin
  process
    variable result, greater, lesser : real ;
    constant seven_by_eight ; real := 7.0/8.0 ;
    constant half : real := 1.0/2.0 ;
  begin
    wait until (not control_1'stable and (control_1
      = '1'));
    if abs(real_no) < abs(imag_no) then
      greater := abs(imag_no) ;
      lesser := abs(real_no) ;
    else
      greater := abs(real_no) ;
      lesser := abs(imag_no) ;
    end if ;
    if greater >  ((seven_by_eight * greater) +
      (half * lesser)) then
      result := greater ;
    else
      result := (seven_by_eight * greater) + (half *
      lesser) ;
    end if ;
    magnitude <= transport result after
      clk_period*
    ((system_word_length * 3) + 17 ) :
  end process ;
end behav ;
```

Figure 3. A behavioral model with an interface declaration and timing.

```
package ctom_constants is
  constant half_period : time := 20ns ;
  constant clk_period : time := 2 * half_period ;
  constant system_word_length : integer := 32 ;
  subtype my_real is real range -1000.0 to 1000.0;
-- THESE ARE FOR THE DATA-FLOW ARCHITECTURE
  constant abs_delay : integer := system_word_
    length + 3 ;
  constant ord_delay : integer := system_word_
    length + 3 ;
  constant dshift_delay : integer := 4 ;
  constant seveneights_delay : integer := 7 ;
    constant add_delay : integer := 1 ;
end ctom_constants ;
```

Figure 4. A package for system-level constants.

```
entity test_bench is end;

architecture arch of test_bench is

  component clock
    port(clk : inout bit) ;
  end component ;

  component front_end
    port (clk_in : in bit ;
      c_1 : out bit ;
      real_ip, imag_ip : out real ) ;
  end component ;

  component ctom
    port (real_no : in real ;
      imag_no : in real ;
      control_1 : in bit ;
      magnitude : out real ) ;
  end component ;

  signal clk_conn, c_1_conn : bit := '0';
  signal real_op, imag_op, mag_op : real : = 0.0;

begin

  system_clock : clock port map
          (clk => clk_conn) ;

  front_end_stage : front_end port map
          (clk_in => clk_conn ,
          c_1 => c_1_conn ,
          real_ip => real_op ,
          imag_ip => imag_op ) ;

  main : ctom port map
    control_1 => c_1_conn ,
     real_no => real_op ,
     imag_no => imag-op ,
     magnitude => mag_op) ;

end arch ;
```

Figure 5. A test bench for the ctom, complex-to-magnitude design entity.

the diagram. These statements are embedded in a block statement, and the guarded statements are active only when the guarded expression is true. This guard is true at the leading edge of the port signal control_1.

The structural architecture seen in Figure 7 uses component declarations and component instantiation statements. We validated this body using the same test bench we used to validate the behavioral and dataflow architectures. Figure 8 shows the bit-level dataflow diagram. A right-shift-by-one operation implements the Half operator in Figure 1, the word-level dataflow

```
use work.ctom_constants.all;
architecture data_flow of ctom is
-- local signal and constant declarations
begin
  main:block(control_1 = '1' and not
  control_1'stable)
  begin
    abs_re <= guarded transport abs (real_no) after
      abs_delay * clk period ;
    abs_im <= guarded transport abs(imag-no)
      after abs_delay * clk_period ;
    inmax <= transport abs_re after ord_delay .*
      clk_period when abs_re > abs_im else
      abs_im after ord_delay * clk period ;
    inmin <= transport abs_re after ord_delay .*
      clk_period when abs_re > abs_im else
      abs_re after ord_delay * clk_period
    half_inmin <= transport inmin * half after
      dshift_delay * clk_period ;
    seven_eight <= transport inmax *
      seven_by_eight after seveneights delay *
      clk_period ;
    max_no_del <= transport half inmin after 3 *
      clk_period ;
    half_del <= transport max_no_del after
      clk_period ;
    max_del <= transport max_no_del after
      clk_period ;
    sum <= transport half_del + seven_eight after
      add_delay * clk_period ;
    magnitude <= transport sum after ord_delay *
      clk _period when sum > max_del else
      max_del after ord_delay * clk_period ;
  end block main ;
end data_flow ;
```

Figure 6. A dataflow architecture.

diagram. This is the DSP primitive Dshift_1. In Figure 8, subtracting one eight of inmax from inmax implements the Seight operator shown in Figure 1. One eight of inmax is formed by a right-shift-by-three (Dshift_3). An external control signal generator controls the bit-level primitives.

Figure 9 shows the bit-level description of the absolute module called entity abs_module and the gate_str architecture. We used a generate construct to describe the register stage, which stores the incoming bit stream. We then specified the register size with a global constant called system_word_length, which is declared in package ctom_constants.

We constructed two general bit-level test benches for the lower level modules: one for modules with two inputs and one for modules with one input. Both of the test benchs used similar front ends to supply the stimulating test vectors, which were obtained from external files.

```
use work.ctom_constants.all;
architecture stru of ctom is
-- component declarations:  absolute, order, dshift,
   delay seveneights and adder
-- local signal declarations
begin
   absolute_re : absolute
      generic map (delay_in_bits =>
      system_word_length + 3)
      port map (input => real_no , control =>
      control_1 , output => abs_re ) ;
   absolute_im : absolute
      generic map (delay_in_bits =>
      system_word_length + 3)
      port map (input => imag_no , control =>
      control_1 , output => abs_im ) ;
   order_1 : order
      generic map (delay_in_bits =>
      system_word_length + 3)
      port map (input_1 => abs_re , input_2 =>
      abs_im , maximum => inmax , minimum =>
           inmin ) ;
   divide_half : dshift
      generic map (delay_in_bits => 4)
      port map (input => inmin, output => half-
      inmin ) ;
   delay_by_three : delay
      generic map (number => 3 )
      port map (input => half_inmin, output =>
      half_del ) ;
   seven_by_eight : seveneighths
      generic map (delay_in_bits => 7)
      port map (input => inmax , divide =>
      seven_eight , actual => max_no_del ) ;
   delay_by_one : delay
      generic map (number => 1)
      port map (input => max_no_del , output =>
      max del ) ;
   adder_stage : adder
      generic map (delay_in_bits => 1)
      port map (input_1 => half_del , input_2 =>
      seven-eight , output => sum ) ;
   order_II : order
      generic map (delay_in_bits => sum , input_2
      => max_del , maximum => magnitude ,
           minimum => open) ;
end stru ;
```

Figure 7. A structural architecture.

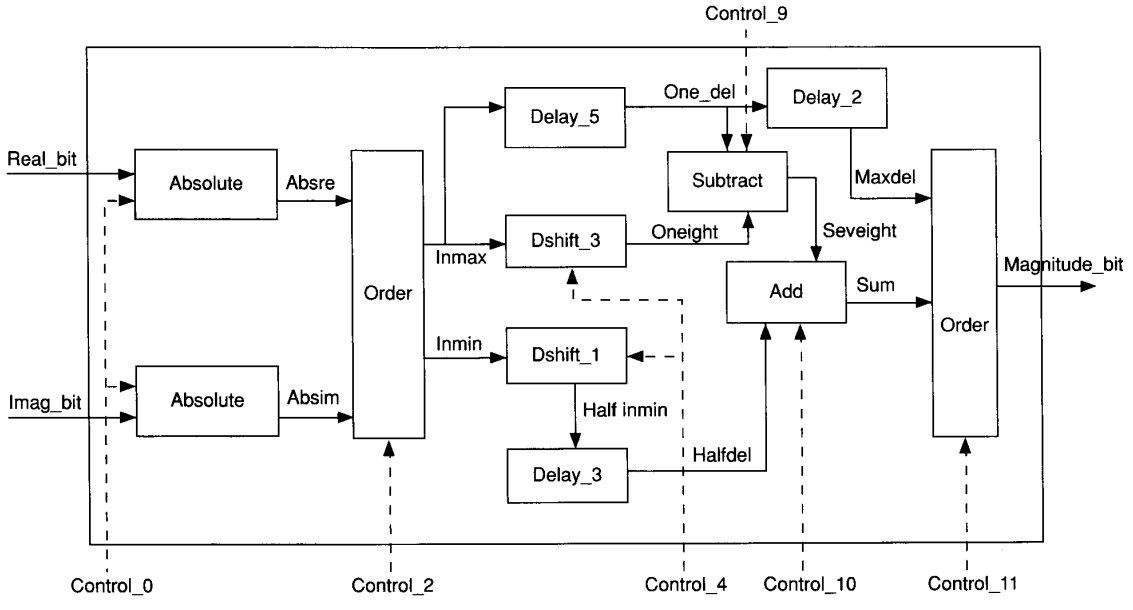


Figure 8. A bit-level dataflow diagram.

To validate the bit-level design, we modified and reused the previous word-level test bench, finding that most of the changes occurred in the front-end and back-end modules. We stored the external test vectors in arrays and used functions to fetch them. The front-end component of the test bench entity performed the real-

```
entity abs_module is
port (clk_in, control, input_bit : in bit ; output_bit :
  out bit ) ;
end abs_module ;

use work.common_constraints.all ;
architecture gate_str of abs_module is
--component dff, xor_gate declarations
    subtype number_of_bits is bit_vector(0 to
    system_word_length) ;
    signal sig : number_of_bits := (others =>'0' );
    signal msb, msb_del, op_skew : bit := '0' ;
begin
  ip_stage : dff
    port map (clk => clk_in , input => input_bit ,
    output => sig(0) ) ;
  msb_storage : dff
    port map (clk => control , input => sig(0) ,
    output => msb ) ;
  msb_delay : dff
    port map (clk => clk_in , input => msb , output
    => msb_del ) ;
  register_stage : for i in 1 to system_word_length
    generate char_i : dff
    port map (clk => clk_in , input => sig(i-1) ,
    output => sig(i) ) ;
  end generate ;
  xor_control : xor_gate
    port map (input_1 =>
sig(system_word_length) , input_2 => msb_del ,
    output =>op_skew ) ;
  op_stage : dff
    port map (clk => clk_in , input => op_skew ,
    output => output_bit ;
end gate_str ;
```

Figure 9. A bit-level description of the Abs operator.

number-to-bit conversion and produced a bit-serial input with a control signal identifying the LSB. We used the back end to accumulate the output from the module under test and reconstruct the real number. Using type-conversion functions simplifies a word-level to bit-serial transformation.

VHDL modeling suggestions

VHDL clearly offers the DSP designer a flexible and powerful modeling environment.[6] Although this flexibility is extremely useful, it can also lead to problems with portability and to poor programming practices. We developed our design using the traditional logic signal values of 0 and 1. Many libraries and simulation systems presently being developed use multivalue logic systems that provide additional levels of detail in the description of logic circuits. The conversion between logic value systems can present portability problems that should be addressed early in the modeling process.[7]

VHDL is strongly typed with strict scope and visibility checking. We found that structured programming practices are very useful in developing complex VHDL descriptions.[8] These practices include the:

- use of naming conventions to help in code maintenance and debugging;
- use of packages for the centralization of type and subtype declarations;
- restricted use of packages with the .all option to assist in debugging;
- use of a set of standard generics and attributes for the design if back annotation is used to insert application environment-dependent data such as delay or latency back into a model;
- use of stand-alone configuration specifications as opposed to specifications in the declarative section of the architecture; and
- use of general, reusable test benches for testing individual components.

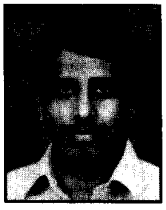
We found that VHDL provided strong support and clear documentation for each level of abstraction in the design of our complex-to-magnitude circuit. At the behavioral level, VHDL can be used to investigate complex DSP algorithms. The easy-to-use structural level facilities help in creating hierarchical descriptions and array structures, while configuration control provides the flexibility to analyze and evaluate alternative designs. Current and planned enhancements to the easy-to-use features of the commercial tools will give more control to the designer, offer transparent views of the model, and provide easy-to-understand graphical representations.

VHDL provides a quick turnaround time for the DSP design environment. In addition, high-level synthesis tools for VHDL behavioral descriptions are beginning to come onto the market, promising even further acceleration of the design process. Presently, the use of VHDL for the design and description of DSP systems offers a facility to experiment, analyze, and evaluate different implementations of algorithms. It also provides a standard description that can be used with other CAD tools and a documentation of the design process that is helpful in debugging and maintenance.

References

1. R. Harr, "VHDL Comes of Age for Designers," *ASIC Technology and News*, Vol. 2, No. 2, June 1990, pp. 20-24.

2. S. Weber, "Here's a Synthesizer that Supports VHDL," *Electronics*, Vol. 13, No. 4, Apr. 1990, pp. 59-60.
3. P. Denyer and D. Renshaw, *VLSI Signal Processing: A Bit-Serial Approach*, Addison-Wesley Publishing Company, Inc., Reading, Mass., 1985.
4. M. Shahdad, "VHDL Hardware Description Language," *Computer*, Vol. 18, No. 2, Feb. 1985, pp. 94-103.
5. R. Lipsett, C. Schaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Boston, 1987.
6. *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1987, IEEE, New York, 1988.
7. J.R. Armstrong, "Tuning VHDL for Multivalue Logic Modeling," *IEEE Design & Test of Computers*, Vol. 7, No. 3, June 1990, pp. 8-10.
8. K.A. Kumar, *A Comparison of FIRSTSIM and VHDL for the Description and Simulation of Concurrent Systems*, thesis report, University of Central Florida, Orlando, 1989.



Krishna A. Kumar works in the VHDL Products Group at Intermetrics, Inc. His technical interests include digital signal processing, design automation, and modeling. He obtained his BE degree in electronics and communication from P.S.G. College of Technology, Coimbatore, India, and his MS in electrical engineering from the University of Central Florida.

Brian Petrasko is an associate professor in the Department of Computer Engineering at the University of Central Florida. His research interests include concurrent architectures and distributed simulation.

Petrasko holds BS, MS, and PhD degrees in electrical engineering from the University of Detroit. He is a member of the IEEE Computer Society and currently serves on the Area Activities Board and acts as Area Committee Chair of Region 3.

Address questions concerning this article to Krishna A. Kumar, Intermetrics, Inc., 4733 Bethesda Avenue, Suite 415, Bethesda, MD 20814.