Translating the Instructional Processor from VHDL to Verilog

Ronald J. Hayne, *Senior Member, IEEE*

***Abstract*— An Instructional Processor has been developed for use as a design example in an Advanced Digital Systems course. The system was originally modeled in VHDL and was simulated using Xilinx design tools to demonstrate operation of the processor. The design model can also be synthesized and implemented in hardware on a field programmable gate array. The goal of this project was to translate the Instructional Processor into the Verilog hardware description language, while maintaining the same operational characteristics. VHDL and Verilog are IEEE standard languages used for the development and testing of hardware designs. Used correctly, these languages describe hardware constructs, which can be implemented using computer aided design tools. These synthesis tools have their own design guidelines, which align modelling techniques with standard library modules such as multiplexers and registers. The process of translating the Instructional Processor from VHDL to Verilog has also resulted in several key insights and lessons learned. These range from correct use of signal types and library functions to important differences in simulation versus synthesis tools. The Instructional Processor has now been translated from its original VHDL to an equivalent Verilog model. By focusing on describing each hardware component, rather than just revising syntax, the design maintained its functional integrity. The hardware synthesized by the Xilinx tools was very consistent in both device utilization and system timing. The project was a success and the Instructional Processor continues to be a valuable instructional tool, now available in two languages.**

***Index Terms*—FPGA synthesis, logic simulation, Verilog, VHDL**

I. Introduction

Teaching digital design involves use of many examples including counters, registers, arithmetic logic units, and memory. The design of a computer processor combines these components into an integrated digital system. An Instructional Processor has been developed for use as a design example in an Advanced Digital Systems course at The Citadel [1] - [4]. The simple architecture provides sufficient complexity to demonstrate fundamental programming concepts. The entire system is modeled in VHDL and can be simulated to demonstrate operation of the processor. Memory-mapped I/O provides the external interfaces necessary to demonstrate an example microcontroller application, when synthesized to a field programmable gate array (FPGA).

R. J. Hayne is an Associate Professor with the Electrical and Computer Engineering Department, The Citadel, Charleston, SC 29409, USA (e-mail: ron.hayne@citadel.edu).

VHDL and Verilog are IEEE standard languages used for the development, verification, synthesis, and testing of hardware designs [5], [6]. While their language reference manuals specify the formal syntax used to model designs, they should not be mistaken for simple programming languages. Some language constructs should only be used for simulation, while others are only suitable for synthesis [7], [8]. Used correctly, these languages describe hardware constructs, which can be implemented using computer aided design tools. These synthesis tools have their own design guidelines, which align modelling techniques with standard library modules such as multiplexers, decoders, registers, and memory [9].

The goal of this project was to translate the Instructional Processor into the Verilog hardware description language, while maintaining the same operational characteristics. While there are language translation tools available, these mainly convert syntax between the languages and only support a subset of the overall language constructs [10], [11]. These tools still require significant human intervention to produce a functional result. The approach taken here was to focus on modelling hardware constructs, rather than simply looking at variations in syntax. The resulting design model replicates simulation results for a range of test programs, while also maintaining the same hardware timing constraints for the FPGA implementation.

II. Instructional Processor Architecture

The instruction set architecture of the example processor has been designed to illustrate multiple operations and basic addressing modes. It is based on a three-bus organization of a 16-bit data path with a four-word register file (REGS) [12]. Key registers include: program counter (PC), instruction register (IR), memory data register (MDR), and memory address register (MAR). The most recent update includes a subroutine STACK and a higher capacity, 4K word by 16-bit, MEMORY [3]. The complete data path is shown in Fig. 1.

The control unit for the Instructional Processor uses a step counter to generate a sequence of up to eight time steps. These time steps are used to determine the order of the control signals issued to the data path for the fetch and execute sequences. Decoding of the instruction is accomplished by four decoders (DCD) connected to specific fields of the IR. The organization of the control unit is shown in Fig. 2.

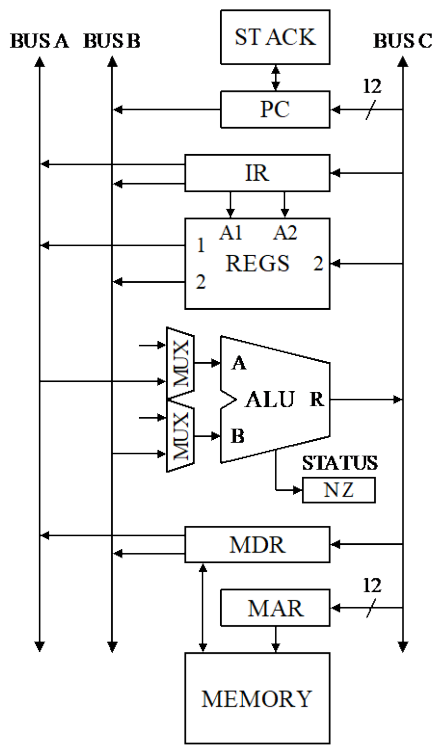

Fig.1. Data path for the Instructional Processor.

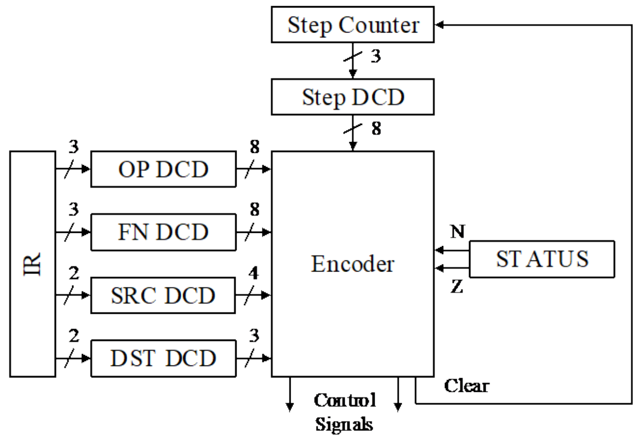

Fig. 2. Control unit organization for the Instructional Processor.

III. VHDL and Verilog Models

Keeping the focus on modelling hardware, rather than variations in syntax, VHDL and Verilog are more similar than different. Concurrent combinational logic, such as an arithmetic logic unit (ALU) or multiplexer (MUX), can be implemented using language specific signal assignment statements. Both languages can also model clock triggered sequential logic, such as a register or counter, using **process** or block statements. In addition, both VHDL and Verilog support design abstraction using behavioral or structural modelling constructs.

The Verilog version of the text that the Instructional Processor was designed to support contains a list of important guidelines to model and synthesize hardware [13]. Some of these include:

- If possible, use concurrent assignments (**assign**) to design combinational logic.
- It is possible to use procedural assignments (**always** blocks) to design either combinational logic or sequential logic.
- When procedural assignments (**always** blocks) are used for combinational logic, use blocking assignments (e.g., ‘=’).
- When procedural assignments (**always** blocks) are used for sequential logic, use non-blocking assignments (e.g., ‘<=’).
- Do not mix blocking and non-blocking statements in an **always** block.

As an initial example of translating a VHDL model into Verilog, consider the block diagram of a multi-port 4 x 16 Register File shown in Fig. 3. The first part of the VHDL model is the **entity**, which describes the input/output interface, shown in Fig. 4. The equivalent Verilog **module** is shown in Fig. 5. Each model defines both the size and direction of all external signals from the block diagram.

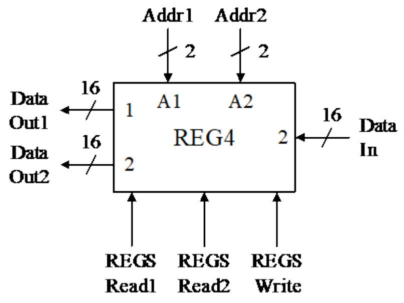

Fig. 3. Block diagram of REG4: 4 x 16 Register File.

```
entity REG4 is
  port(CLK: in std_logic;
    REGS_Read1: in std_logic;
    REGS_Read2: in std_logic;
    REGS_Write: in std_logic;
    Addr1: in std_logic_vector(1 downto 0);
    Addr2: in std_logic_vector(1 downto 0);
    Data_In: in std_logic_vector(15 downto 0);
    Data_Out1: out std_logic_vector(15 downto 0);
    Data_Out2: out std_logic_vector(15 downto 0));
end REG4;
```

Fig. 4. VHDL **entity** for REG4: 4 x 16 register file.

```
module REG4(CLK,
            REGS_Read1, REGS_Read2, REGS_Write,
            Addr1, Addr2, Data_In,
            Data_Out1, Data_Out2);
  input CLK;
  input REGS_Read1;
  input REGS_Read2;
  input REGS_Write;
  input [1:0] Addr1;
  input [1:0] Addr2;
  input [15:0] Data_In;
  output [15:0] Data_Out1;
  output [15:0] Data_Out2;
```

Fig. 5. Verilog **module** for REG4: 4 x 16 register file.

The internal function of the VHDL model for the 4 x 16 Register File is specified using the behavioral **architecture** shown in Fig. 6. It defines the internal register array (REG4 of type RAM4) as well as the synchronous and asynchronous behavior of the signals. The equivalent behavioral Verilog model is shown in Fig. 7. It also models the appropriate timing behavior of the signals as well as the use of tri-state buffers, indicated by high-impedance (Z). These tri-state outputs are used for connection to the buses in the data path.

```
architecture Behave of REG4 is
  type RAM4 is array (0 to 3) of
    std_logic_vector(15 downto 0);
  signal REG4: RAM4;
begin
  -- asynchronous read
  process(REGS_Read1, REGS_Read2, Addr1, Addr2)
  begin
    if REGS_Read1 = '1' then
      Data_Out1 <= REG4(conv_integer(Addr1));
    else
      -- high impedance
      Data_Out1 <= (others => 'Z');
    end if;
    if REGS_Read2 = '1' then
      Data_Out2 <= REG4(conv_integer(Addr2));
    else
      -- high-impedance
      Data_Out2 <= (others => 'Z');
    end if;
  end process;
  -- synchronous write
  process(CLK)
  begin
    if rising_edge(CLK) then
      if REGS_Write = '1' then
        REG4(conv_integer(Addr2)) <= Data_In;
      end if;
    end if;
  end process;
end Behave;
```

Fig. 6. VHDL behavioral architecture for REG4: 4 x 16 register file.

```
    // asynchronous read
    always @(REGS_Read1, REGS_Read2, Addr1, Addr2)
      begin
        if (REGS_Read1)
          Data_Out1 = REG4[Addr1];
        else
          // high-impedance
          Data_Out1 = 'bZ;
        if (REGS_Read2)
          Data_Out2 = REG4[Addr2];
        else
          // high-impedance
          Data_Out2 = 'bZ;
      end
    // synchronous write
    always @(posedge CLK)
    begin
      if (REGS_Write)
        REG4[Addr2] <= Data_In;
    end
  endmodule
```

Fig. 7. Verilog behavioral module for REG4: 4 x 16 register file.

For the implementation of the data path, the REG4 component is mapped to the data and control signals in the design hierarchy using a structural model for both VHDL and Verilog. Of special note is the use of the **reg** and **wire** types in Verilog. Usage of a signal in other parts of the model may dictate a specific signal type, which may be somewhat counter intuitive. This is a key reason that simple syntax translation often fails. Beyond the signal declarations, the structural models shown in Fig. 8 and Fig. 9 are actually very similar. They both map component ports to signal connections using positional association.

```
  -- Instruction Register
  signal IR: std_logic_vector(15 downto 0);
  -- Register File Control Signals
  signal REGS_Read1: std_logic;
  signal REGS_Read2: std_logic;
  signal REGS_Write: std_logic;
  -- Buses
  signal BUS_A: std_logic_vector(15 downto 0);
  signal BUS_B: std_logic_vector(15 downto 0);
  signal BUS_C: std_logic_vector(15 downto 0);
  -- Instruction Register Address Fields
  alias SRC_REG: std_logic_vector(1 downto 0)
    is IR(10 downto 9);
  alias DST_REG: std_logic_vector(1 downto 0)
    is IR(6 downto 5);
begin
-- Data Path
  -- Register File
  REGS: REG4 port map (CLK,
          REGS_Read1, REGS_Read2, REGS_Write,
          SRC_REG, DST_REG, BUS_C,
          BUS_A, BUS_B);
```

Fig. 8. VHDL structural model for register file integrated into data path.

```
  // Instruction Register
  reg [15:0] IR;
  // Register File Control Signals
  reg REGS_Read1;
  reg REGS_Read2;
  reg REGS_Write;
  // Register File Data Output Connections
  wire [15:0] Data_Out1;
  wire [15:0] Data_Out2;
  // Buses
  reg [15:0] BUS_A;
  reg [15:0] BUS_B;
  wire [15:0] BUS_C;
  // Instruction Register Address Fields
  wire [1:0] SRC_REG = IR[10:9];
  wire [1:0] DST_REG = IR[6:5];
// Data Path
  // Register File
  REG4 REGS (CLK,
         REGS_Read1, REGS_Read2, REGS_Write,
         SRC_REG, DST_REG, BUS_C,
         Data_Out1, Data_Out2);
// Control Unit
always @(*)  // Control Signal Encoder
begin
  BUS_A = Data_Out1;
  BUS_B = Data_Out2;
```

Fig. 9. Verilog structural model for register file integrated into data path.

As a final example, the control signal encoder from the control unit in Fig. 2 is implemented using nested **case** statements to model the various decoders connected to the specific fields of the IR. The appropriate control signals are asserted for each combination of opcode, source addressing mode, and destination addressing mode. Multiple time steps are used as required to correctly sequence the control signals. The VHDL and Verilog models for an example execution sequence are shown in Fig. 10 and Fig. 11.

```
-- Control Unit
  -- Control Signal Encoder
  Control : process(STEP, IR, STATUS, PC)
  begin
    -- Execute
    case OP is
      -- 1-Operand
      when MOVE | INV | SHL | ASHR =>
        -- Addressing Modes
        case SRC_MODE is
          when M0 =>
            case DST_MODE is
              when M0 =>
              -- OP Rs,Rd
                case STEP is
                  when T3 =>
                    REGS_Read1 <= '1';
                    ALU_OP <= OP;
                    Load_STATUS <= '1';
                    REGS_Write <= '1';
                    Clear <= '1';
                  when others =>
                    null;
                end case;
```

Fig. 10. VHDL behavioral model for control signal encoder.

```
// Control Unit
  // Control Signal Encoder
  always @(*)
  begin
    // Execute
    case (OP)
      // 1-Operand
      MOVE, INV, SHL, ASHR:
        // Addressing Modes
        case (SRC_MODE)
          M0:
            case (DST_MODE)
              M0:
              // OP Rs,Rd
                case (STEP)
                  T3: begin
                        REGS_Read1 = 1;
                        ALU_OP = OP;
                        Load_STATUS = 1;
                        REGS_Write = 1;
                        Clear = 1;
                      end
                  default:
                    ;
                endcase
```

Fig. 11. Verilog behavioral model for control signal encoder.

IV. Programming Environment

An assembly language program can be written as a simple text file, using the syntax specified by the instruction set architecture. A custom assembler is used to convert the program to a binary machine code file [3]. Standard library I/O functions are used to load the machine code into memory during compilation of the hardware description language models. An example program for computing the sum of an array, using a counter and a pointer, is shown in Fig. 12.

```
.data                 ;Begin Data Section
SUM                   ;Result goes here
N 3                   ;Number of Elements
X 7, -8, 10           ;Sample Array Data

.program              ;Begin Program Code
START: MOVE [N],R1    ;Init Counter
       MOVE X,R2      ;Init Pointer
       MOVE 0,R0      ;Init Sum
LOOP:  ADD  [R2],R0   ;Add using Pointer
       ADD  1,R2      ;Inc Pointer
       ADD  -1,R1     ;Dec Counter
       BNZ  LOOP      ;If not Zero then Loop
       MOVE R0,[SUM]  ;Store Result
STOP:  BRA  STOP      ;Done
```

Fig. 12. Example assembly language program.

The VHDL and Verilog models are compiled using the Xilinx ISE design tools and behavioral simulations are performed using Xilinx iSim [14]. Signal values can be traced in the simulations to verify correct operation of the data path and control unit as the test program is run.

V. Microcontroller Extension

Before synthesizing the models to hardware on an FPGA, an I/O interface is added. Input and output ports use the memory-mapped addresses shown in Fig. 13. The FPGA was chosen with sufficient block RAM resources for the 4K x 16-bit main memory. Integration of the processor, memory, and I/O onto a single chip is known as a microcontroller.

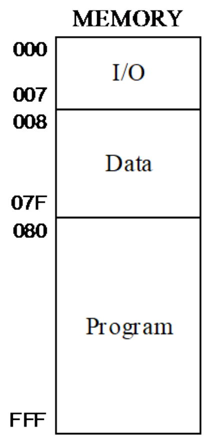

Fig. 13. Memory map for the Instructional Processor.

The Instructional Processor was synthesized to the target FPGA using Xilinx XST [9] and implemented on a Digilent BASYS 2 development board [15]. This board provides a large collection of built-in I/O devices, as well as ports for connecting additional peripheral modules (Pmods). Sample Pmods include A/D converter, keypad, and H-bridge motor driver.

VI. Key Insights and Lessons Learned

During the process of translating the Instructional Processor from VHDL to Verilog, several key insights became apparent along with lessons learned from refinement of the models. The first minor note is that all signal assignments in an **always** block must use the **reg** data type, even if modelling combinational logic. This often results in a confusing mix of **reg** and **wire** declarations like those shown in the example in Fig. 9. This is also an example of where simple syntax translation fails. Several iterations were required to ensure the correct signal types were used to model specific hardware.

The next lesson learned occurred using standard libraries. Verilog has a robust set of file I/O functions; however, these functions did not necessarily perform the same during different phases of the design process. For example, the main memory module was initialized from a binary file using the function shown in Fig. 14.

```
//Initialize Memory
initial $readmemb("program_pwm.bin", MEM4K);
```

Fig. 14. Memory initialization using standard file I/O function.

The correct contents and performance of the memory were verified via functional simulation. However, during synthesis of the model to an FPGA, an innocuous warning message reported that the 4K memory was only partially initialized and, therefore, initialization was ignored. The resulting failure of the hardware implementation was difficult to trace, but was readily corrected by adding thousands of zeros to the end of the binary file.

Finally, achieving the same hardware timing optimizations required very precise modelling techniques to force the synthesis tools to recognize specific design elements. For example, the bus connections (BUS_A, BUS_B) were intended to use tri-state buffers instead of multiplexers. This would use less FPGA resources and improve system timing. In VHDL the buses can be forced to tri-state buffers by using a simple initialization at the beginning of the control process, shown in Fig. 15.

```
-- Control Signal Encoder
Control : process(STEP, IR, STATUS, PC)
begin
  -- Synthesize tri-state buses
  BUS_A <= (others => 'Z');
  BUS_B <= (others => 'Z');
```

Fig. 15. Implementation of tri-state buses in VHDL.

In Verilog, however, a signal can be initialized to a one or a zero, but not high-impedance. Assigning this default value to the buses required use of a **default case** statement buried within the control signal encoder, as shown in Fig. 16. Due to the multiple nested case statements, several iterations were required to find an optimal placement that would be correctly recognized by the synthesis tool.

```
    // Synthesize tristate buses
    default:
      begin
        BUS_A = 'bZ;
        BUS_B = 'bZ;
      end
  endcase
```

Fig. 16. Implementation of tri-state buses in Verilog.

Once the correct modelling construct was found for the target hardware, the synthesis tool was able to replicate the desired bus structure and device utilization.

VII. Results and Conclusions

The VHDL and Verilog models were compiled using the Xilinx ISE design tools and behavioral simulations were performed using Xilinx iSim. Signal values were traced in the simulations to verify correct operation of the data path and control unit as test programs were run. Both the VHDL and Verilog models exactly replicate all register transfers and timing for multiple test sequences. From a simulation perspective, the results show that the two models are equivalent.

The VHDL and Verilog models were next synthesized to the target FPGA using Xilinx XST. Device utilization was characterized by the number of 4 input look-up tables (LUTs) used by the design. From a timing perspective, the worst-case propagation delay was used to determine the maximum clock frequency for the FPGA. The synthesis results were analyzed both with and without the tri-state buffer optimization.

The synthesis results, before resolution of the tri-state bus problem, are summarized in Table I. The Verilog design, with multiplexer routing for the buses, uses 40% more FPGA resources. This results in a significantly longer propagation delay and much slower clock frequency (18%). The synthesis results in Table II show that, once the tri-state buses are correctly implemented, device utilization is virtually identical. Timing results are very consistent and the slight difference (5%) can be attributed to the varying order of placement and routing of components produced by the VHDL and Verilog versions of the synthesis tools. Both the designs now meet the timing requirements to run on the BASYS 2 FPGA prototype board with a 50 MHz clock source.

TABLE I
Initial Synthesis Results (Verilog without tri-state buses)

	VHDL	Verilog	% Difference
Number of 4 input LUTs	724	1018	40.6%
Maximum clock frequency	60.6 MHz	46.9 MHz	18.2%

TABLE II
Final Synthesis Results

	VHDL	Verilog	% Difference
Number of 4 input LUTs	724	725	0.14%
Maximum clock frequency	60.6 MHz	57.3 MHz	5.4%

Various microcontroller applications were used to demonstrate functionality of the hardware implementations. These include DC motor speed control via pulse-width modulation (PWM) and time-multiplex scanning of a 4 x 4 hex input keypad. Serial I/O was also demonstrated using timing loops to send and receive serial data at 9600 baud. Both the VHDL and Verilog versions of the microcontroller were successfully able to execute the test programs and interface with the external hardware.

The Instructional Processor has now been translated from its original VHDL to an equivalent Verilog model. By focusing on describing each hardware component, rather than just revising syntax, the design maintained its functional integrity. Simulation results for both models exactly replicate all register transfers and timing for multiple test sequences. The synthesized hardware was also very consistent in both device utilization and maximum clock frequency. The project was a success and the Instructional Processor continues to achieve its goal as a valuable instructional tool [2], [3], now available in two languages [4], [16].

References

[1] R. J. Hayne, "Translating the Instructional Processor from VHDL to Verilog," *Proceedings ASEE Annual Conference and Exposition*, Salt Lake City, UT, June 2018.

[2] R. J. Hayne, "An Instructional Processor Design using VHDL and an FPGA," *Computers in Education Journal*, ASEE, Vol. 3 No. 2, April - June 2012.

[3] R. J. Hayne and J. I. Moore, "Evolution of the Instructional Processor," *Computers in Education Journal*, ASEE, Vol. 6 No. 4, October - December 2015.

[4] R. J. Hayne, "Design of an Instructional Processor," in C. Roth and L. John, *Digital Systems Design Using VHDL*, Third Edition, Cengage Learning, Boston, MA, 2018. [Online]. Available: http://academic.cengage.com/resource_uploads/downloads/1305635140_559956.pdf.

[5] *IEEE Standard VHDL Language Reference Manual*, IEEE Std 1076, 2000 Edition, IEEE, New York, NY, December 2000.

[6] *IEEE Standard for Verilog® Hardware Description Language*, IEEE Std 1364™-2005, IEEE, New York, NY, April 2006.

[7] R. Duckworth, "Embedded System Design with FPGAs using HDLs," *Proceedings of the 2005 IEEE International Conference on Microelectronic Systems Education*, IEEE, New York, NY, 2005.

[8] J. Schreiner, R. Findenig, and W. Ecker, "Design Centric Modeling of Digital Hardware," *Proceedings IEEE International High Level Design Validation and Test Workshop*, IEEE, New York, NY, 2016.

[9] *XST User Guide for Virtex-4, Virtex-5, Spartan-3, and Newer CPLD Devices*, UG627, v14.5, Xilinx, Inc., March 2013.

[10] L. Dolittle, *vhd2vl*. [Online]. Available: http://doolittle.icarus.com/~larry/vhd2vl/.

[11] Synapticad, Inc., *VHDL2VeriLog*. [Online]. Available: http://www.syncad.com/verilog_vhdl_translator.htm.

[12] R. J. Hayne, "VHDL Projects to Reinforce Computer Architecture Classroom Instruction," *Computers in Education Journal*, ASEE, Vol. XVIII No. 2, April - June 2008.

[13] C. Roth, L. John, and B. Lee, *Digital Systems Design Using Verilog*, First Edition, Cengage Learning, Boston, MA, 2016.

[14] *iSim User Guide*, UG660, v14.3, Xilinx, Inc., October 2012.

[15] Digilent, Inc., *Digilent Documentation*. [Online]. Available: https://reference.digilentinc.com/

[16] R. J. Hayne, "Design of an Instructional Processor," in C. Roth, L. John, and B. Lee, *Digital Systems Design Using Verilog*, First Edition, Cengage Learning, Boston, MA, 2016. [Online]. Available: http://academic.cengage.com/resource_uploads/downloads/1285051076_581158.pdf.

Ronald J. Hayne is an Associate Professor with the Electrical and Computer Engineering Department, The Citadel, Charleston, SC. He received the B.S. degree in computer science from the United States Military Academy, West Point, NY; the M.S. degree in electrical engineering from the University of Arizona, Tucson; and the Ph.D. degree in electrical engineering from the University of Virginia, Charlottesville.

Dr. Hayne's professional areas of interest include digital systems design, computer architecture, and hardware description languages. He is a retired Army Colonel with assignments at U.S. Army Space and Strategic Defense Command, Arlington, VA; Army Research Laboratory, Aberdeen Proving Ground, MD; and the National Security Space Architect, Fairfax, VA. His academic career also includes faculty positions at the United States Military Academy, West Point and George Mason University, Fairfax, VA.