Translating the Instructional Processor from VHDL to Verilog

Dr. Ronald J. Hayne, The Citadel

Ronald J. Hayne is an Associate Professor in the Department of Electrical and Computer Engineering at The Citadel. He received his B.S. in Computer Science from the United States Military Academy, his M.S. in Electrical Engineering from the University of Arizona, and his Ph.D. in Electrical Engineering from the University of Virginia. Dr. Hayne's professional areas of interest include digital systems design and hardware description languages. He is a retired Army Colonel with experience in academics and Defense laboratories.

Translating the Instructional Processor from VHDL to Verilog

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 (FPGA). 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 been successfully 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.

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] - [3]. 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 an FPGA.

VHDL and Verilog are IEEE standard languages used for the development, verification, synthesis, and testing of hardware designs [4], [5]. 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 [6], [7]. 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 [8].

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 [9], [10]. 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.

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) [11]. 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 [2]. The complete data path and memory map are shown in Figure 1.

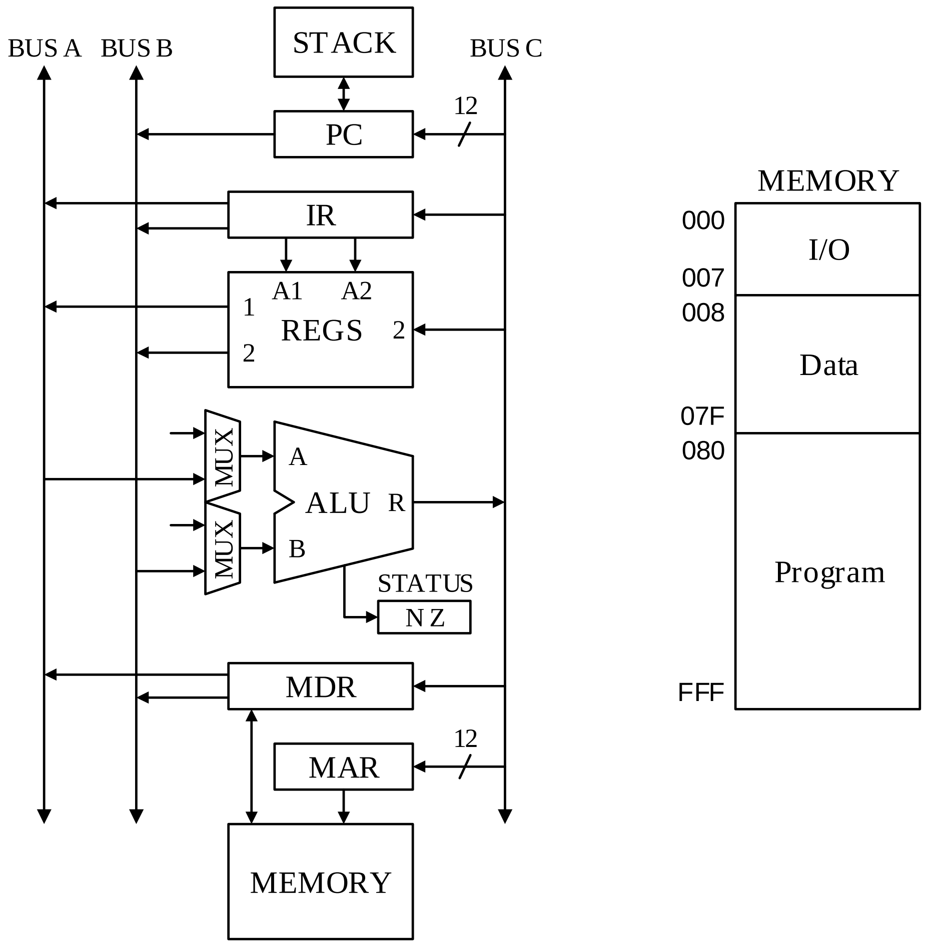

Figure 1. Data Path and Memory Map for the Instructional Processor.

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 Figure 2.

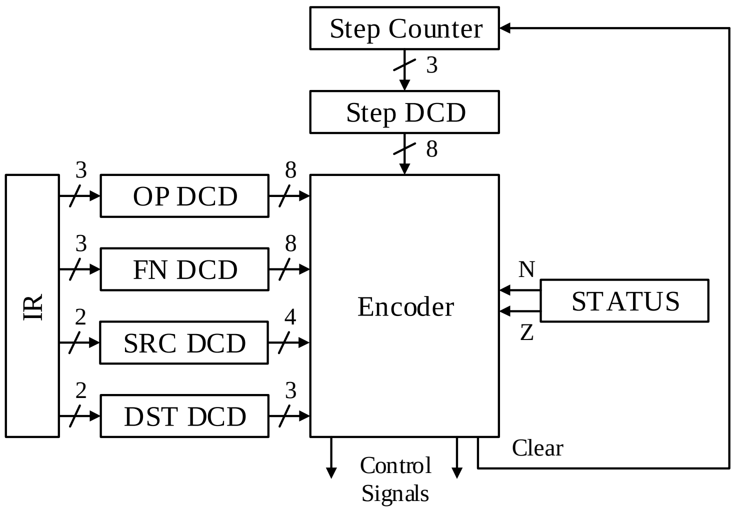

Figure 2. Control Unit Organization for the Instructional Processor.

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 [12]. 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 4 x 16 Register File shown in Figure 3. The first part of the VHDL model is the **entity**, which describes the input/output interface, shown in Figure 4. The equivalent Verilog **module** is shown in Figure 5. Each model defines both the size and direction of all external signals.

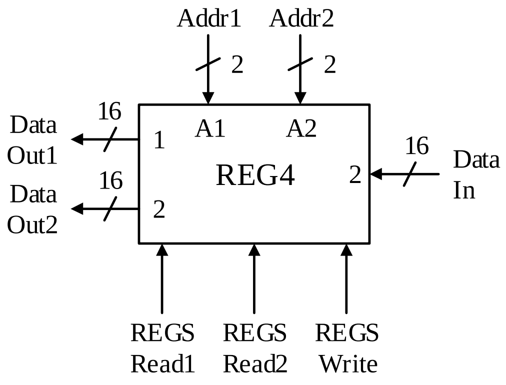

Figure 3. REG4: 4 x 16 Register File.

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

Figure 4. VHDL Entity for REG4.

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

Figure 5. Verilog Module for REG4.

The internal function of the VHDL model is specified using the **architecture** shown in Figure 6. It defines the internal register array and the synchronous and asynchronous behavior of the signals. The equivalent Verilog model is shown in Figure 7. It also models the timing behavior of the signals as well as the use of tri-state buffers, indicated by high-impedance **Z**.

```
architecture Behave of REG4 is
  type RAM4 is array (0 to 3) of std_logic_vector(15 downto 0);
  signal REG4: RAM4 := (others => X"0000");
begin
  process(REGS_Read1, REGS_Read2, Addr1, Addr2) -- async read
  begin
    if REGS_Read1 = '1' then
      Data_Out1 <= REG4(conv_integer(Addr1));
    else
      Data_Out1 <= (others => 'Z'); -- high impedance
    end if;
    if REGS_Read2 = '1' then
      Data_Out2 <= REG4(conv_integer(Addr2));
    else
      Data_Out2 <= (others => 'Z'); -- high impedance
    end if;
  end process;
```

```
    process(CLK)
    begin
      if rising_edge(CLK) then -- synchronous write
        if REGS_Write = '1' then
          REG4(conv_integer(Addr2)) <= Data_In;
        end if;
      end if;
    end process;
  end Behave;
```

Figure 6. VHDL Behavioral Architecture for REG4.

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

Figure 7. Verilog Behavioral Module for REG4.

For the design of the data path, the REG4 component is mapped to the data and control signals using a structural model for both VHDL and Verilog. These very similar constructs are shown in Figures 8 and 9. Of special note is the use of the **wire** type in Verilog for modeling combinational logic connections to the output ports of the register file.

```
  signal IR : std_logic_vector(15 downto 0) := X"0000"; -- Instruction Reg
  signal REGS_Read1  : std_logic := '0'; -- Register File
  signal REGS_Read2  : std_logic := '0';
  signal REGS_Write  : std_logic := '0';
  signal BUS_A : std_logic_vector(15 downto 0) := (others => 'Z'); -- Buses
  signal BUS_B : std_logic_vector(15 downto 0) := (others => 'Z');
  signal BUS_C : std_logic_vector(15 downto 0) := (others => 'Z');
  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);
```

Figure 8. VHDL Structural Model for Register File.

```
  reg [15:0] IR;  // Instruction Register
  reg REGS_Read1;  // Register File
  reg REGS_Read2;
  wire [15:0] Data_Out1;
  wire [15:0] Data_Out2;
  reg REGS_Write;
  reg [15:0] BUS_A;  // Buses
  reg [15:0] BUS_B;
  wire [15:0] BUS_C;
  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);
```

Figure 9. Verilog Structural Model for Register File.

As a final example, the encoder from the control unit in Figure 2 is implemented using nested **case** statements to model the various decoders. 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 Figures 10 and 11.

```
-- Control Unit
Control : process(STEP, IR, STATUS, PC) -- Control Signal Encoder
begin
  case OP is  -- Execute
    when MOVE | INV | SHL | ASHR => -- 1-Operand
      case SRC_MODE is -- Addressing Modes
        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;
```

Figure 10. VHDL Behavioral Model for Control Signal Encoder.

```
// Control Unit
always @(*)  // Control Signal Encoder
begin
  case (OP)  // Execute
    MOVE, INV, SHL, ASHR:  // 1-Operand
      case (SRC_MODE)  // Addressing Modes
        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
```

Figure 11. Verilog Behavioral Model for Control Signal Encoder.

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 Figure 9. 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, a memory **module** was initialized from a binary file using the following function:

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

The correct contents and performance of the memory were verified via simulation. 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 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 with the following simple initialization at the beginning of the control process:

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

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 encoder. Due to the multiple nested **case** statements, several iterations were required to find the correct placement that would be recognized by the synthesis tool.

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

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.

Results and Conclusions

The VHDL and Verilog models were compiled using the Xilinx ISE design tools and behavioral simulations were performed using Xilinx iSim [13]. 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 [8]. 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 are summarized in Table 1.

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

Table 1. Xilinx XST Synthesis Results.

The synthesis results show that device utilization is virtually identical. Timing results are very consistent and much improved once the multiplexer versus tri-state buffer problem was resolved. The slight difference (5.4%) can be attributed to varying order of placement and routing of components produced by the VHDL and Verilog versions of the synthesis tools. Both the designs meet the timing requirements to run on the FPGA prototype board with a 50 MHz clock source.

The Instructional Processor has been successfully 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 hardware synthesized by the Xilinx tools 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 [1], [2], now available in two languages [3], [14].

References

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

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

[3] 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.

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

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

[6] 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.

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

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

[9] L. Dolittle, *vhd2vl*. [Online]. Available: http://doolittle.icarus.com/~larry/vhd2vl/, Accessed: March 7, 2018.

[10] Synapticad, Inc., *VHDL2VeriLog*. [Online]. Available: http://www.syncad.com/verilog_vhdl_translator.htm, Accessed: March 7, 2018.

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

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

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

[14] 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.