any articles have been written about the advantages of using a hardware description language such as VHDL or Verilog to design hardware. First, example VHDL code is provided to describe the operation of circuits. Second, the software description of the circuit is simulated and proven to work inside the simulator. Finally, the author states that the software description can be transformed into the actual hardware. But where is this hardware, and how is the process completed?

In this article, I’ll present you with a hands-on tutorial that you can use to create an actual arithmetic logic unit (ALU) chip. I’ll explain how to design the ALU with VHDL and show you that it simulates correctly. Then, I’ll demonstrate how the VHDL description of the ALU is used to create the actual hardware. Finally, you’ll see that the ALU hardware chip actually runs according to the initial design.

To prove that the chip can be easily reprogrammed with another design, I’ll add two other components that interface with the ALU. The first component is a BCD-to-seven-segment display driver that connects to the output of the ALU. With these two additional components, testing the ALU is simplified.

USING SOFTWARE

The world of computers is changing rapidly. Moore’s Law states that the densities of transistors on a chip doubles every 18 months. One thing that has helped the computer industry to achieve this goal is its ability to use software to design and build hardware-integrated circuit chips. This technology requires three components: the ability to describe the behavior of a digital logic circuit using software, the capability to translate the software description of the circuit into a description of how the circuit is connected, and the ability to implement the circuit in an IC.

VHDL is a computer software language that’s used for describing the behavior of digital logic circuits. Jointly developed by the U.S. Department of Defense and IEEE in the mid-1980s, VHDL was standardized by the IEEE in 1987 and extended in 1993. In many ways, the syntax is similar to high-level programming languages, as you’ll see later.

However, just being able to formally describe a hardware circuit with a software language is neither sufficient nor particularly useful. The analogy is like writing a computer program in C: if you don’t have a C compiler (or
can power up and then see lights flash. What you need is an actual hardware chip that is programmed with the given netlist. The field programmable gate array [FPGA] chip provides this important link, giving you the ability to create hardware with software automatically.

An FPGA contains many generic logic cells that can be programmed to connect in any fashion. Hence, these logic cells can be programmed to connect according to the given netlist. What you end up with is an actual chip that has been programmed to operate according to your original software description. The end result is hardware that’s been created using software.

Similar to flash memories, the connections made inside an FPGA are nonvolatile, although they can be erased and reprogrammed. Newer FPGAs can contain several hundreds of thousands of basic logic gates. Therefore, a fairly complex circuit can fit inside one FPGA, or you could put several independent circuits inside the same FPGA to reduce the chip count.

**ALU DESIGN**

Now, I’ll describe the process of building hardware with software by designing an ALU. I purposely kept the following example to a bare minimum. This will allow you to concentrate on the overall process of making the hardware with software rather than focusing on design details.

The ALU is responsible for all of the arithmetic and logic operations inside a CPU. The arithmetic operations include addition, subtraction, increment, and decrement. Multiplication and division are usually performed in another module, but there is no reason why you couldn’t put it in the ALU too. Because you’re building your own chip, you can do whatever you want to.

Logic operations include AND, OR, NOT, XOR, and so on. The ALU has two main inputs for the two operands. A set of select lines is used to select the operation to perform on the two operands. The number of required select lines is dependent on the number of operations you want to implement.
ment in the ALU. The ALU has one output, which outputs the result of the operation on the two operands.

Using behavioral VHDL code to design an ALU is extremely simple. You have to choose the width of the two input operands and the operations to implement in the ALU. For simplicity, I implemented the eight operations shown in Table 1. With eight operations, three select lines were needed ($S_2$, $S_1$, and $S_0$). The variables $A$ and $B$ are the two input operands using four bits each.

The behavioral VHDL code that describes the ALU is shown in Listing 1. The ENTITY section defines the ALU’s input and output signals. The select-line signal $S$ is declared as a 3-bit input vector. $A$ and $B$ are the two input operands declared as 4-bit vectors, and $F$ is the 4-bit output result from the operation.

In the ARCHITECTURE section, behavioral code is used to describe the ALU’s actual operations. Specifically, a CASE statement is implemented to select a particular operation, depending on the three select lines. In each of the cases, a signal assignment statement is used to assign the value from the result of the intended operation to the output $F$.

For example, if $S = "000"$, then the $A$ operand is passed to the output $F$. If $S = "001"$, then you want to add $A$ and $B$ to get the statement $F <= A + B$.

As you can see, the behavioral coding is straightforward. You need to know the operations of the ALU without having to know the details of the required logic gates and how they should be connected together to realize the circuit.

**SYNTHESIS**

Now that you know how to write VHDL code for the ALU, you’re ready to learn how to synthesize it. Recall that synthesizing the code will translate the high-level description of the circuit into the netlist that creates the circuit. The synthesis tool that I’ll describe is Altera’s MAX+PLUS II. You may download a free student edition of the software from the company’s web site.

The complete development system with the MAX+PLUS II GUI environment on the screen is shown in Photo 1. The VHDL code for the ALU is entered or copied into the Text Editor window (see Listing 1). To start the synthesis process, click on the Start button in the Compiler window. A red progress line in the Compiler window shows that the synthesizer is working.

At the end of the synthesis process, a message window opens to show you that the process has been completed.

---

**Table 1**—Each ALU operation is given a unique binary code as specified by the three select lines $S_2$, $S_1$, and $S_0$. For example, when the select lines are given the value 001, the logical AND operation is selected. The ALU will perform the AND operation on the two input operands $A$ and $B$, and pass the result to the output $F$.

<table>
<thead>
<tr>
<th>Select lines</th>
<th>Operation name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_2$</td>
<td>$S_1$</td>
<td>$S_0$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Listing 1—The ENTITY section of the VHDL code describes the interface to the module, which consists of the two input operands $A$ and $B$, the 3-bit select line $S$ (for selecting what ALU operation to perform), and the ALU output $F$ for outputting the result of the operation. The actual coding of the circuit is in the ARCHITECTURE section. The CASE statement uses $S$ to select one of the eight operations to perform according to Table 1. For each case, the associated operation is performed, and the result is assigned to the output $F$.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;

ENTITY alu IS
    PORT (S: IN STD_LOGIC_VECTOR(2 downto 0); //Select for operations
          A, B: IN STD_LOGIC_VECTOR(3 downto 0); //Input operands
          F: OUT STD_LOGIC_VECTOR(3 downto 0)); //Output
END alu;

ARCHITECTURE Behavior OF alu IS
BEGIN
    PROCESS(S, A, B)
    BEGIN
        CASE S IS
            WHEN "000" => //Pass
                F <= A;
            WHEN "001" => //AND
                F <= A AND B;
            WHEN "010" => //OR
                F <= A OR B;
            WHEN "011" => //Not A
                F <= NOT A;
            WHEN "100" => //Add
                F <= A + B;
            WHEN "101" => //Subtract
                F <= A - B;
            WHEN "110" => //Increment
                F <= A + 1;
            WHEN OTHERS => //Decrement
                F <= A - 1;
        END CASE;
    END PROCESS;
END Behavior;
```
without errors (hopefully). It’s not necessary to know all of the details concerning the netlist that’s generated from the synthesis process; however, if you’re curious, you can take a look.

**FUNCTIONAL SIMULATION**

After completing the synthesis process, I performed a functional simulation of the ALU code. First, I needed to bring up the Waveform Editor window and specify the signals that I wanted to observe in the simulation trace. For the ALU, I had the select lines, two operands (A and B), and output (F) that I wanted to include in the simulation trace (see Photo 2). Next, I had to assign values for the S, A, and B input signals. I wanted to test all eight of the ALU’s functions, so I specified the eight possible values for S. I arbitrarily set the two 4-bit operands A and B to three and six respectively.

Issuing the Start Simulation command begins the actual simulation. The simulator generates the trace for all of the output signals [i.e., the trace for the output F in this case]. Looking at the F signal trace in Photo 2 and referring back to the ALU definitions in Table 1, you’ll see the results of the ALU operations. For example, when S = 000, A is passed through to F, which is 0011. When S = 001, F = 0011 AND 0110 = 0010. When S = 101, the operation is 3 – 6. The result, –3, is shown as the signed value 1101 in two’s complement representation.

**PROGRAMMING THE FPGA**

Look at the close-up of Altera’s UP1 FPGA development board in Photo 3. This board contains two FPGAs. Located on the left, the MAX7000S chip has a gate capacity of about 2500 logic gates. On the right, the FLEX10K has a gate capacity of about 20,000 logic gates. These two FPGAs are independent of each other and programmed individually. The remaining components on the board are mostly switches and LEDs that are used for testing the I/O signals to and from the FPGAs.

The ALU example I’m describing can fit on the MAX7000S chip, so I’ll show you how to program this chip. To perform the synthesis, a target chip must be specified so that the synthesizer produces a netlist and also fits the netlist onto the specific FPGA. Furthermore, you can specify which I/O pins on the chip are used for ALU I/O signals. To do this, the Floorplan Editor window should be brought up for a graphical view of the chip with all of its pins. I/O pins that are currently assigned to a signal are colored blue with the signal name next to it. The unused I/O pins are white. A signal can be repositioned by dragging it from one I/O pin to another. If an I/O

---

**Listing 2**—In the ARCHITECTURE section of the VHDL code, the 3-bit binary counter (cnt) is defined as a 3-bit register (variable). At each rising edge of the clock, cnt is incremented by one. The current value of the count is then assigned to the output (Q).

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
ENTITY counter IS
  PORT (clk : IN BIT; 
       Q : OUT STD_LOGIC_VECTOR(2 downto 0));
END counter;
ARCHITECTURE Behavior OF counter IS
BEGIN
  PROCESS (clk)
  VARIABLE cnt : STD_LOGIC_VECTOR(2 downto 0);
  BEGIN
    IF (clk'EVENT AND clk = '1') THEN
      cnt := cnt + 1;
    END IF;
    Q <= cnt;
  END PROCESS;
END Behavior;
```

---

**Photo 5**—Take a look at the results of the same eight ALU operations on the two operands 3 and 6 automatically cycling through and displayed as a hexadecimal digit: a—pass A; b—A and B; c—A or B; d—not A; e—A + B; f—A – B; g—increment A; h—decrement A. In contrast to Photo 4, you don’t need the 3-bit select switches for S, because you’re using the 3-bit counter to provide for that input. Also, you’re using the binary-to-seven-segment display decoder to convert the 4-bit result to a hexadecimal digit for display on the seven-segment LED.
pin is reassigned, the design must be synthesized again so that it will refit the signals to the newly assigned pins.

To actually program the chip, connect the JTAG cable between the board and parallel port of the computer, connect a 9-V power source to the board, bring up the Programmer window, and click on the Program button. In just a few seconds, the ALU design will be programmed onto the FPGA chip. Finally, an actual ALU chip!

POWER THE CHIP

The next step is to test the ALU on the chip. First, the DIP switches for the inputs must be connected to the $S$, $A$, and $B$ input signals. Because the switches are on the board, the jumper wires are connected to the appropriate pins on the chip according to the floor plan.

The output signal $F$ is connected to four LEDs. By setting the DIP switches for the input operands $A$ and $B$ and the operation select $S$, it’s easy to see the result of the operation from the four LEDs.

Photo 4 shows eight operations performing in sequence ($S = 0$ to 7) with the two operands 3 and 6. For the DIP switches, the down position is zero and up is one. The four leftmost DIP switches relate to the operand $A$. The next group of four DIP switches is for $B$. The rightmost group of three DIP switches is for $S$. The four red LEDs on the left show the binary result of the operation $F$ with the top LED being the most significant bit. The results for the eight operations are in the following order: 0011, 0010, 0111, 1100, 1001, 1101, 0100, and 0010, which matches the $F$ signal trace in Photo 2.

Listing 3—The VHDL code for the BCD to seven-segment decoder contains a `CASE` statement that’s used to determine the input BCD value. Depending on the BCD value, the corresponding LED is turned on or off by assigning a one or zero to it. For example, to show a zero on the seven-segment LCD all of the LEDs are turned on except the one in the center. Hence, you assign a string of six 1 bits followed by a 0 bit. On the UPI board, the LEDs are actually turned on with a zero, not a one; therefore, you need to invert the zeroes and ones at the end for the actual output.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY BCD IS
PORT (I :IN STD_LOGIC_VECTOR(3 DOWNTO 0);
SegmentsI :OUT STD_LOGIC_VECTOR(1 TO 7) );
END BCD;
ARCHITECTURE Behavioral OF BCD IS
SIGNAL Segs: STD_LOGIC_VECTOR(1 to 7);
BEGIN
PROCESS(I)
BEGIN
CASE I IS
WHEN "0000" => Segs <= "1111110"; -- 0
WHEN "0001" => Segs <= "0110000"; -- 1
WHEN "0010" => Segs <= "1101101"; -- 2
WHEN "0011" => Segs <= "1111001"; -- 3
WHEN "0100" => Segs <= "0110011"; -- 4
WHEN "0101" => Segs <= "1011011"; -- 5
WHEN "0110" => Segs <= "1110000"; -- 6
WHEN "0111" => Segs <= "1110011"; -- 7
WHEN "1000" => Segs <= "1111111"; -- 8
WHEN "1001" => Segs <= "1110000"; -- 9
WHEN "1010" => Segs <= "1111111"; -- A
WHEN "1011" => Segs <= "0011111"; -- b
WHEN "1100" => Segs <= "1101111"; -- C
WHEN "1101" => Segs <= "0111111"; -- d
WHEN "1110" => Segs <= "1001111"; -- E
WHEN "1111" => Segs <= "1000111"; -- F
WHEN OTHERS => Segs <= "0000000"; -- all off
END CASE;
SegmentsI <= not Segs; -- invert the 0's and 1's
END PROCESS;
END Behavioral;
```
CHANGING DESIGNS

To show you how easy it is to change the design and reprogram the FPGA chip, I’ll explain how to add a 3-bit binary counter and BCD to the seven-segment decoder on the original ALU design. These two components make testing the ALU a little easier and the output more user-friendly. The output of the 3-bit binary counter is used to drive the three select lines on the ALU, and the BCD-to-seven-segment decoder is used to display the ALU output signal as a hexadecimal digit on the seven-segment LED.

The behavioral VHDL code for the 3-bit binary counter is shown in Listing 2, and the BCD-to-seven-segment decoder is shown in Listing 3.

With the three components defined, the next step is to connect them. To connect two or more together in VHDL, it’s important to define a high-level entity with the needed components inside it, as you can see in Listing 4.

The components are connected using the structural-level method. Signals declared in the ARCHITECTURE section are used to connect the component I/O lines together. The component declaration declares the components needed in the top-level design.

The top-level design includes only the three components defined previously. The PORT MAP statement produces an instance of that particular component. The signals listed in the PORT MAP statements define how the I/O signals for the components are actually connected together. For example, the output of the binary counter must be connected to the ALU select input; therefore, the 3-bit signal name S is used for both the counter PORT MAP output and the ALU PORT MAP select input.

It turns out that this design, with the three components, cannot fit on the smaller MAX7000S chip. So, the larger FLEX10K chip is programmed instead.

On the development board, the DIP switches and seven-segment LEDs are connected directly to the FLEX10K chip so that no jumpers are required. However, to use these switches and LEDs, it’s important to use the appropriate I/O pins on the FLEX10K chip that are connected to them.

Again, the Floorplan Editor specifies the pins needed to assign to the design’s I/Os. In addition, an external square-wave generator can provide the clock signal for the counter. After setting the DIP switches to three and six for the two operands [A and B] and starting the clock, you can see the ALU’s results displayed as a hexadecimal digit on the seven-segment LED and automatically cycling through the eight select operations (see Photo 5).

YOUR TIME AND MONEY

As you now know, it’s easy to build hardware with software. Using a hardware description language, you can write the functional description for complex circuits, synthesize it using a synthesis tool, and then program an FPGA chip that realizes the circuit. Moreover, it’s easy to modify or add to your design and then reprogram the FPGA. This capability saves development time and money because you don’t have to manufacture a custom chip.

Enoch Hwang has a Ph.D. in computer science. Currently, he is an associate professor at La Sierra University and a lecturer at the University of California, Riverside. Enoch’s technical interests include embedded systems, software/hardware partitioning, and power reduction. You may reach him at ehwang@cs.lasierra.edu.

Library ieee;
use ieee.std_logic_1164.all;

entity top is
  port (clk: in bit;
    A, B : in STD_LOGIC_VECTOR(3 downto 0); //input operands
    Segments: out STD_LOGIC_VECTOR(1 to 7)); //7-segment output
  end top;

architecture Structure of top is
  signal S: STD_LOGIC_VECTOR(2 downto 0);
  signal ALUout: STD_LOGIC_VECTOR(3 downto 0);
  component ALU port (S: in STD_LOGIC_VECTOR(2 downto 0);
    A: in STD_LOGIC_VECTOR(3 downto 0);
    B: in STD_LOGIC_VECTOR(3 downto 0);
    F: out STD_LOGIC_VECTOR(3 downto 0));
    end component;
  component counter port (clk: in bit;
    q: out STD_LOGIC_VECTOR(2 downto 0));
    end component;
  component BCD port (I: in STD_LOGIC_VECTOR(3 downto 0);
    Segments: out STD_LOGIC_VECTOR(1 to 7));
    end component;
begin
  u1: counter port map (clk, S);
  u2: ALU port map (S, A, B, ALUout);
  u3: BCD port map (ALUout, Segments);
end Structure;