CIRCUIT CELLAR ONLINE

Technically Speaking

James Antonakos

An Introduction to VHDL

Designing Hardware with Software

This month, James looks at designing a digital circuit using software, namely VHDL. You'll see the benefit of using a simulated model in designing. And, with his detailed look into the program, you'll have no problem applying what you've learned.

This month, let's take a look at a different method of designing a digital circuit. Instead of connecting integrated circuits together with wires on a breadboard, I'll describe (using plain text) and simulate the circuit via software. This design method uses very-high-speed integrated circuit (VHSIC) hardware description language (VHDL).

WHAT IS VHDL?

VHDL is a technique for describing the hardware associated with a digital system. The VHDL design (also called a model) is similar to the structure of an ordinary program, such as a C program. The C program is compiled to make an executable file, whereas the VHDL design is simulated to test the validity of the hardware design. These processes are illustrated in Figure 1.

Why bother learning how to write a VHDL specification for a digital circuit if you are already familiar with the techniques of digital design? Consider the benefits. As in Electronics Workbench, the hardware in a VHDL model is simulated, which means it is not necessary to build the circuit from actual components to see if it works. Also, after the VHDL model has been created, it is easy to transfer it to other environments. For example, two designers working in different cities can e-mail the VHDL model of their circuit to each other, a faster and easier method than sending a bulky hardware prototype through the mail. In addition, the standardization of VHDL by the IEEE (standards 1076-1987 and 1076-1993, also called VHDL-87 and VHDL-93) guarantees that a VHDL model is portable, so the same model can be processed by any VHDL-87/93-compliant package.

LEVELS OF DESIGN

VHDL designs may contain one or more levels, as indicated by the 4-bit ripple adder structure shown in Figure 2. The top level (system) contains the least amount of detail regarding the actual hardware implementation. The bottom level (gates) contains the actual logic gates used to implement the system.

The design method of starting with the topmost level and adding new levels of increasing detail is called top-down design. Let's start with the design of a 4-bit ripple adder, begin-

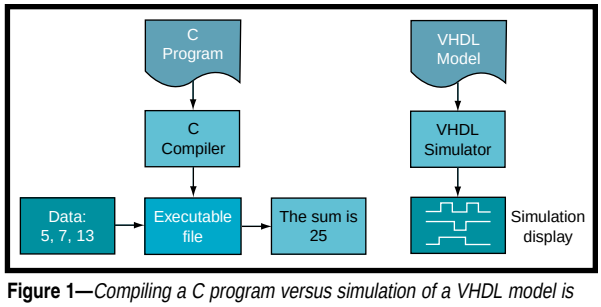

Figure 1—*Compiling a C program versus simulation of a VHDL model is demonstrated here.*

ning with the topmost level.

THE INTERFACE

Design entities, which contain an interface and a body, are the objects used to specify what hardware components are used and how they are connected. The interface describes the number and type of signals used by the entity, and the body describes how the signals are connected and related.

The interface for the 4-bit ripple adder is shown in Figure 3. The name of the entity is RIP4. The 4-bit ripple adder adds two groups of four bits together (plus a carry input bit, CI) and generates a 4-bit sum and a carry output bit (CO). Inputs and outputs to the entity are identified by the port keyword. A port can be an input, output, or bidirectional signal. The A and B inputs to RIP4 are specified as 4-bit vectors, groups of bits that share common properties. The "3 downto 0" portion of the statement indicates that four input bits are needed for A and B, numbered 3, 2, 1, and 0, from left to right. Bit 3 is the MSB and bit 0 is the LSB. In other areas of the VHDL specification, these bits are referred to as A(0) through A(3) and B(0) through B(3). Note that there is nothing in the entity interface to indicate how the ripple adder does its job. This is left for the second part of the design entity, the body portion.

THE BODY

The body portion of a design entity describes how the function of the entity is performed. In Figure 2, you can see how the first expansion of the ripple adder indicates that it contains four full adder (FA) blocks. These are the components of the ripple adder.

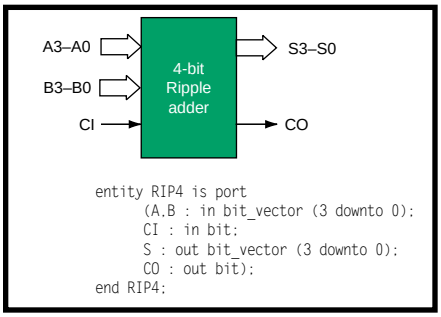

Figure 3—*The ripple adder design entity interface and diagram can be seen here.*

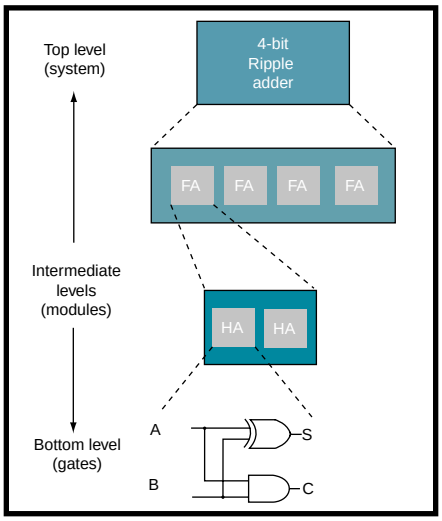

Figure 2—*A multilevel design hierarchy detailing components of a full adder is used in VHDL designs.*

The body portion of the RIP4 design entity is responsible for specifying the number and type of components used by the entity (see Listing 1).

Refer to Figure 4 and the body statements as I discuss the individual pieces of RIPADD. The component portion of RIPADD indicates that the *Full_Adder* design entity will be used to implement the ripple adder. The port information of the *Full_Adder* entity is required to make the necessary connections indicated by Figure 4.

The signal keyword is used to define internal signals that allow the four full adders to be cascaded by connecting the carry output signals (CYO) to the carry inputs (CYI).

As expected in an object-oriented environment, you are able to reuse the single *Full_Adder* component by instantiating four copies of it (FA0 through FA3). Each copy is instantiated differently in the four port-mapping statements, with actual signal names being substituted for the desired input/output connections.

FULL_ADDER

As previously mentioned, more detail is added to the design as you proceed to the lower levels. Now that you have an idea of how the full adders are used inside the ripple adder, it's time to work on the *Full_Adder* entity construction (see Listing 2). As usual, the entity interface indicates only the associated input and output signals. The implementation details are left for the body statements.

In Figure 5 you can see the internal structure of the full adder. Two half adders (HA) and an OR gate are required to implement a full adder. Three internal signals complete the wiring scheme. Compare the diagram in Figure 5 with the corresponding *Full_Adder* body statements.

The body of the *Full_Adder* entity can be seen in Listing 3. There are many similarities to the body statements of the ripple adder. However, notice the last statement before the end of the code snippet, where the CYO output is generated. This is called a signal assignment statement. Here only a single logic gate is needed to combine signals, so a simple Boolean expression can be used.

The logical operations available in VHDL are NOT, AND, NAND, OR, NOR, XOR, and XNOR (VHDL-93 only). All operations have the same precedence except NOT, which has the highest precedence. This means that parenthesis must be used to enforce a particular order of operations.

HALF_ADDER

The last piece of the design is the

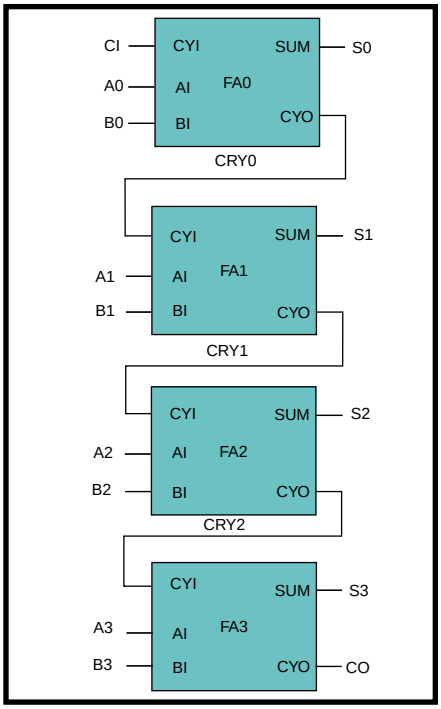

Figure 4—*Here you can see the full adder connections in the ripple adder.*

Half_Adder entity (see Figure 6), which is instantiated twice in each *Full_Adder* entity, for a total of eight times. The interface and body of the half adder are shown in Listing 4. Again, signal assignment statements are used to generate the required logic function.

Bear in mind that the focus of this design is to implement a 4-bit ripple adder using VHDL. You may have many questions about how the half adder works, or why the OR gate is required in the full adder, or why I did not start with the half adder and work my way up. The first two questions can be answered by reviewing your digital textbook. The third question almost answers itself during a design. With experience, you will discover that adding layer after layer of detail is a natural process, after the big picture is known.

PUTTING IT ALL TOGETHER

All of the design entities for the 4-bit ripple adder are shown in Listing 5. At this point, I should pause to review what has been accomplished. The design began with a high-level specification of a 4-bit ripple adder, the RIP4 entity. Then, in line with the nature of top-down design, the next level of the design was detailed, showing how the *Full_Adder* entity (used as a component in RIP4) was designed. Then the *Half_Adder* component of the *Full_Adder* entity was described as well, completing the design.

IDENTIFIERS, DATA TYPES, AND OPERATORS

All three items discussed in this section can be found in the 4-bit ripple adder design. Identifiers are names that are chosen for the various signals and entities in the design. Identifiers have the following properties:

- any length, however, avoid long identifiers
- not case sensitive, so upper or lowercase may be used (unless some other style rules are in effect)
- legal symbols are A–Z, a–z, 0–9, and the underscore
- the first symbol must be a letter
- the last symbol can not be an under-

Listing 1—*Here you can see the body of the RIP4 entity.*

```
architecture RIPADD of RIP4 is

-- Specify the Full Adder subcircuit
component Full_Adder
port
      (AI,BI,CYI: in bit;
      SUM,CYO : out bit);
end component;

-- Reserve a few signals for internal connections
signal CRY0, CRY1, CRY2 : bit;

begin
      FA0: Full_Adder port map (A(0), B(0), CI, S(0), CRY0);
      FA1: Full_Adder port map (A(1), B(1), CRY0, S(1), CRY1);
      FA2: Full_Adder port map (A(2), B(2), CRY1, S(2), CRY2);
      FA3: Full_Adder port map (A(3), B(3), CRY2, S(3), CO);
end RIPADD;
```

Listing 2—*The interface for the Full_Adder entity can be seen here.*

```
entity Full_Adder is
port
      (AI,BI,CYI: in bit;
      SUM,CYO : out bit);
end Full_Adder;
```

Listing 3—*These statements form the body of the Full_Adder entity.*

```
entity DECODE is
port
         (A,B,ENbar : in bit;
         Y : out bit_vector(3 downto 0));
end DECODE;

architecture LGATES of DECODE is
begin
         Y(0) <= ENbar or (not A nand not B);
         Y(1) <= ENbar or (A nand not B);
         Y(2) <= ENbar or (not A nand B);
         Y(3) <= ENbar or (A nand B);
end LGATES;
```

Listing 4—*These statements describe the internal operation of the Half_Adder.*

```
entity Half_Adder is
port
      (W,X: in bit;
      Y,Z : out bit);
end Half_Adder;

architecture HA of Half_Adder is
begin
      Y <= W xor X;  -- This is the sum bit
      Z <= W and X;  -- This is the carry bit
end HA;
```

score

The data types used in the ripple adder were *bit* and *bit_vector*. Several other data types are available in VHDL, as indicated in Table 1.

Several operators are also used in the ripple adder, such as <= *(assignment),* — *(comment),* and three logical operators: AND, OR, and XOR. Additional operators are arithmetic (+, -, *, /, mod, rem, abs, **), relational (>, >=, <, <=, =, and /=), and shift (sla, sra, sll, srl, rol, and ror).

You may have noticed that the <= operator has two different meanings. In one case, it means assignment and in the other it means less-than-or-equal-to. The VHDL compiler determines which interpretation is valid based on the context of where it is used. VHDL allows for other operators, and even certain identifiers, to be overloaded as well.

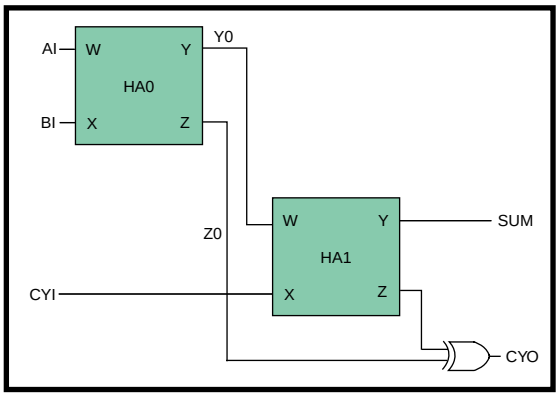

Figure 5—*The full adder implementation can be seen here.*

EXAMPLES

Let's take a look at several examples of applying VHDL to digital design. The first example involves a 2-bit greater-than comparator (see Figure 7). The interface for the comparator can be seen in Listing 6.

One way to implement the 2-bit greater-than comparator is to define several internal signals, write equations for the output of each AND gate, and a final equation for the OR gate. The design entity body for this method is shown in Listing 7.

Compare this implementation style with another design entity body that does the same thing (see Listing 8). It makes you wonder if there's any advantage to the first architecture body, doesn't it?

THE FIVE-INPUT AND GATE

By definition, the logical operators in VHDL are limited to two inputs (with the exception of the inverter). What do you do if we need a five input AND gate? First, let's specify the interface (see Listing 9).

Here you can see the five inputs (A through E). How should they be reduced to a single output? The first technique can be seen in Listing 10.

Take a moment to sketch the schematic of the AND5 circuit. Does it look like it will work? When you are finished, do the same for this architecture body (see Listing 11).

Can you think of any advantages or disadvantages of one specification over the other? Is there any reason why you should stay away from the following third solution (see Listing 12)?

THE 2:4 DECODER

The design entity for a 2:4 decoder is shown in Listing 13. Note the use of an enable input (*ENbar*), which is set up as an active-low enable. When ENbar is low, the selected output of the decoder will go low. If ENbar is high, all outputs remain high.

USING PACKAGES

Packages are holding places for design entities that are frequently used. For example, after the 4-bit ripple adder is working, you may want to place it in a package so that it can be easily accessed by the VHDL compiler at a later time. Packages are stored in libraries, which are then imported into your design using a simple statement, *use MYLIB.RIP4.all;*. Here, the RIP4 package within the MYLIB library is made available to all the entities in the design.

There are several packages that contain useful data types, input/output operations, logic definitions, and many other entities. These packages are called STANDARD, STD_LOGIC, and TEXTIO. One package, called STD_LOGIC_1164 (located in the IEEE library), contains a definition for a nine-value logic system popular with most designers. The nine values are U (uninitialized), X (forcing unknown), 0 (forcing zero), 1 (forcing one), Z (high impedance), W (weak unknown), L (weak low), H (weak high), and – (don't care).

TIMING EXAMPLES

As always, time can be your friend or your enemy in a digital circuit. For applications that have critical time requirements or for those that wish to simulate a digital circuit down to the femtosecond, VHDL provides the necessary timing features. One way to specify the delay of a logic operation is to indicate its time directly in the body (see Listing 14).

Here, the delays of the XOR gate and the AND gate are fixed at 5 ns. Figure 8 shows a simplified timing diagram illustrating the effect of gate delay in a two-input gate. Table 2 shows the units of time available in VHDL.

VHDL provides an inertial delay model that essentially eliminates the effects of short-duration logic level changes. For example, a gate with a delay of 8 ns may experience a high-level pulse for 3 ns. The inertial delay model ignores the 3 ns pulse because it is less than the propagation delay.

Getting back to the five-input AND gate design entity covered previously, which design do you prefer now?

OTHER METHODS

Typing in a VHDL source file for a large design would be tedious. Most design tools now offer several ways to

Data type	Range of values
Boolean	True, False
Bit	0, 1
Bit_vector	Array of bits
Character	Single ASCII symbol
String	Array of characters
Integer	Depends on software
Real	Depends on software
Natural	Zero to the maximum integer value
Positive	One to the maximum integer value
Time	Depends on software

Table 1—*There are several VHDL data types that may be used in a design.*

input a design. Graphical design entry allows you to draw a schematic of the circuit or specify its operation through a state diagram. It is even possible to specify the operation of the digital circuit by entering its input and output waveforms and letting the design tool determine the logic required.

AND THE POINT IS?

After going through the trouble of entering your design, what is next after you successfully compile and simulate it? The nice part is that when you're finished, the same software that helped you develop your design in a virtual environment will control your real-world PLD programmer and program your design into a real piece of hardware. Here are some of the device types supported by VHDL compilers:

- Altera—APEX 20K, FLEX 6000/8000/10K, MAX 5000/7000/9000
- Xilinx—XC4000 FPGA, XC9500 CPLD

Several thousand gates are common in many of these devices, with some offering over 100,000 gates. Once again, a little effort searching the web will reward you with a long list of devices, manufacturers, and support.

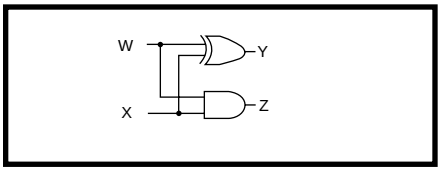

Figure 6—*An XOR gate and an AND gate are used to make a half adder.*

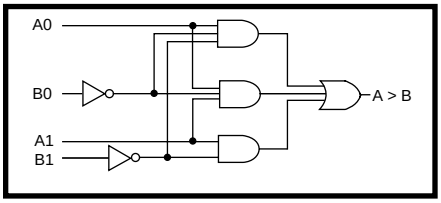

Figure 7—*In this 2-bit greater-than comparator, the output is high if A > B.*

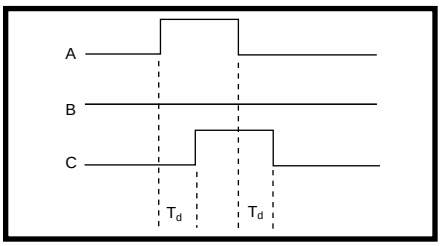

Figure 8—*The gate delay between the input and output can be seen here.*

Unit	Definition
fs	femtosecond (10^{-15} s)
ps	picosecond (10^{-12} s)
ns	nanosecond (10^{-9} s)
us	microsecond (10^{-6} s)
ms	millisecond (10^{-3} s)
sec	seconds
min	minutes
hr	hours

Table 2—*Here you can see what the units of time in VHDL are.*

James Antonakos is a professor in the Department of Electrical Engineering Technology at Broome Community College, with over 25 years of experience designing digital and analog circuitry and developing software. He is also the author of numerous textbooks on microprocessors, programming, and microcomputer systems. You may reach him at antonakos_j@sunybroome.edu or visit his web site at www.sunybroome.edu/~antonakos_j.

RESOURCES

Xilinx Foundation Series
Xilinx, Inc.
(408) 559-7778
Fax: (408) 559-7114
www.xilinx.com

MAX+plus II
Altera Corp.
(408) 544-7000
www.altera.com

Compiler for Programmable Logic and FPGA Design (CUPL)
Logical Devices, Inc.
(303) 456-2060
44 1970 621041
Fax: (303) 456-2404
Fax: 44 1970 621040
www.logicaldevices.com

Listing 5—*Here you can see the entire VHDL specification for the 4-bit ripple adder.*

```
entity RIP4 is
port
     (A,B : in bit_vector(3 downto 0);
     CI: in bit;
     S : out bit_vector(3 downto 0);
     CO : out bit);
end RIP4;

architecture RIPADD of RIP4 is
-- Specify the Full Adder subcircuit
component Full_Adder
port
     (AI,BI,CYI: in bit;
     SUM,CYO : out bit);
end component;
-- Reserve a few signals for internal connections
signal CY0, CY1, CY2 : bit;
begin
     FA0: Full_Adder port map (A(0), B(0), CI, S(0), CY0);
     FA1: Full_Adder port map (A(1), B(1), CY0, S(1), CY1);
     FA2: Full_Adder port map (A(2), B(2), CY1, S(2), CY2);
     FA3: Full_Adder port map (A(3), B(3), CY2, S(3), CO);
end RIPADD;

entity Full_Adder is
port
     (AI,BI,CYI: in bit;
     SUM,CYO : out bit);
end Full_Adder;

architecture FA of Full_Adder is
component Half_Adder
port
     (W,X: in bit;
     Y,Z : out bit);
end component;
signal Y0,Z0,Z1 : bit;
begin
     HA0: Half_Adder port map (AI, BI, Y0, Z0);
     HA1: Half_Adder port map (CYI, Y0, SUM, Z1);
     CYO <= Z0 or Z1;
end FA;

entity Half_Adder is
port
     (W,X: in bit;
     Y,Z : out bit);
end Half_Adder;

architecture HA of Half_Adder is
begin
     Y <= W xor X;  -- This is the sum bit
     Z <= W and X;  -- This is the carry bit
end HA;
```

Listing 6—*The interface for the comparator is shown here.*

```
entity GTCOMP is
port
        (A,B : in bit_vector(1 downto 0);
        AGTB : out bit);
end GTCOMP;
```

Listing 7—*This is one way to specify the comparator design.*

```
architecture LGATES of GTCOMP is
signal A1, A2, A3 : bit;
begin
        A1 <= A(0) and not B(0) and not B(1);
        A2 <= A(0) and not B(0) and A(1);
        A3 <= A(1) and not B(1);
        AGTB <= A1 or A2 or A3;
end LGATES;
```

Listing 8—*This is a second way of specifying the comparator design.*

```
architecture LGATES of GTCOMP is
begin
        AGTB <= (A(0) and not B(0) and not B(1))
                or (A(0) and not B(0) and A(1))
                or(A(1) and not B(1));
end LGATES;
```

Listing 9—*This snippet of code shows the VHDL interface for the five-input AND gate.*

```
entity AND5 is
port
        (A,B,C,D,E : in bit;
        F : out bit);
end AND5;
```

Listing 10—*This code is used to implement a five-input AND gate using four two-input AND gates.*

```
architecture LGATES of AND5 is
signal A1, A2, A3;
begin
        A1 <= A and B;
        A2 <= A1 and C;
        A3 <= A2 and D;
        F <= A3 and E;
end LGATES;
```

Listing 11—*Another way to design the five-input AND gate can be seen here.*

```
architecture LGATES of AND5 is
signal A1, A2, A3;
begin
         A1 <= A and B;
         A2 <= C and D;
         A3 <= A1 and A2;
         F <= A3 and E;
end LGATES;
```

Listing 12—*Here is one more method of specifying the five-input AND gate.*

```
architecture LGATES of AND5 is
begin
      F <= A and B and C and D and E;
end LGATES;
```

Listing 13—*The specification for the 2:4 decoder with an active-low enable input can be seen here.*

```
entity DECODE is
port
         (A,B,ENbar : in bit;
         Y : out bit_vector(3 downto 0));
end DECODE;

architecture LGATES of DECODE is
begin
         Y(0) <= ENbar or (not A nand not B);
         Y(1) <= ENbar or (A nand not B);
         Y(2) <= ENbar or (not A nand B);
         Y(3) <= ENbar or (A nand B);
end LGATES;
```

Listing 14—*Use this code for specifying gate delay in a VHDL design.*

```
architecture HA of Half_Adder is
begin
      Y <= W xor X after 5 ns;
      Z <= W and X after 5 ns;
end HA;
```