

## FEATURE ARTICLE

Ingo Cyliax

# Designing with VHDL

After something becomes mainstream, you better think about jumping on board. Such is the case with hardware description languages (HDL) for designing FPGAs and CPLDs. Even if it's new to you, Ingo explains that Xilinx's WebPack makes digital design understandable. The benefits far outweigh the chore of having to learn the language. And, because it's at no cost to you, what more could you ask for?



was going to write about using ASMs to design digital logic for FPGAs, however, I thought I would digress a little and write about using VHDL for digital design. What prompted me was Xilinx's release of a free version of its design environment for download from the company web site. It's called WebPack and is a version of Xilinx's Foundation 3.2i software environment, but it is limited to only a few architectures. Why am I talking about VHDL? Well, it seems like hardware description languages (HDL) such as VHDL and Verilog are becoming the mainstream design entry mechanism for designing FPGAs and CPLDs. The release of WebPack somewhat illustrates

this because it's becoming obvious that you have to use HDLs.

Most of the comments I make here really apply to both Verilog and VHDL, but I will use VHDL as my main example to illustrate some of the techniques. VHDL is perhaps the harder of the two languages to learn and is not as popular in many U.S. companies as Verilog. But, it's widely used in Europe and is the preferred choice by the DOD and NASA. To many programmers, VHDL looks arcane, and Verilog looks more C-like. However, both are just as adequate at expressing digital logic, and most logic compilers will understand both languages. They convert both VHDL and Verilog into an internal representation first and then go from there.

### THE BENEFITS

What are the benefits of using HDLs? Most designers have been shying away from HDLs because they feel like they have to give up control over how functions are actually laid out. HDLs do offer a level of abstraction over schematics, but it is possible to have absolute control over the structure as well. The arguments concerning schematics and HDLs are much like those about C language and assembly language (i.e., schematics usually are targeted to specific hardware libraries, but it's possible to express logic in equations in HDL). However, you can also drop down and instantiate library components in HDLs. This is akin to using the *asm()* directive in C. It makes for more efficient logic because you can target specific architectural fea-

Want	Package	Type to use
Signed	<i>std_logic_arith</i>	Signed
Signed	<i>std_logic_signed</i>	<i>std_logic_vector</i>
Unsigned	<i>std_logic_arith</i>	Unsigned
Unsigned	<i>std_logic_signed</i>	<i>std_logic_vector</i>

Table 1—Here is a matrix on how you can select whether a specific signal is signed or unsigned.

tures, but it makes your code less portable. An example of instantiating versus describing the function can be seen in Listings 1a and b.

The analogy can also be carried a bit further (to some abstraction). The output of an HDL compiler is a net list of components like LUTs and flip-flops and how they're wired together. This is the same net list that you get from schematics. So, net lists are the assembly language for FPGA/CPLDs, and schematics editors are simply net list editors.

Just like C compilers, HDL logic compilers handle optimizing for specific targets with more sophistication. It used to be true that logic compilers would synthesize everything down to logic elements like 4:1 muxes or NAND gates. Now, even the free compilers included in WebPack will analyze your code and, based on optimization parameters you can select, map your logic to specific FPGA/CPLD structures that are available in specific chip families. For example, if the HDL compiler detects something that looks like an adder and a particular chip supports an efficient fast adder using internal carry chains, the compiler will generate this structure in the net list it outputs. Many HDL compilers also know how to detect multipliers and generate appropriate logic to implement them. It can even detect if one of the multiplicands is a power of two and implement it as a shift register.

Although HDL compilers are good at optimizing for specific architectures, they do need some help. Some of the expertise of using HDL compilers comes from the fact that you have to use specific code styles in order for the compiler to figure out what you intend to do. Even though it's possible to express yourself like any programming language in HDL, it will not be able to synthesize efficient hardware from just any HDL inputs. It needs some help recognizing what signals are registers or latches and if you mean to implement structures like muxes, tristate buses, and so on. Also, if you want to implement state machines, you need to follow certain coding styles so the compiler can recognize traditional

**Listings 1a**—You can specify an adder by simply adding two signals together. **b**—By using the component structure in VHDL, you can instantiate library symbols writing VHDL.

**a)**

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
entity adder is
  port(A,B : in std_logic_vector(7 downto 0);
        SUM : out std_logic_vector(7 downto 0));
end adder;

architecture archi of adder is
begin
  SUM <= A + B;
end archi;
```

**b)**

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
entity adder is
  port(A,B : in std_logic_vector(7 downto 0);
        SUM : out std_logic_vector(7 downto 0));
end adder;

architecture archi of adder is

  component fadd8
    port ( a,b: in std_logic_vector(7 downto 0);
           o: out std_logic_vector(7 downto 0)
          )
  end component

begin

  add1: fadd8 port map (A,B,SUM);

end archi;
```

state machines styles and make intelligent implementations from them.

The benefit comes when you try to port your code to another FPGA/CPLD architecture. After the compiler knows what you're trying to do, it has no problem finding an implementation that will be the most efficient for a particular chip family. It may recognize your adder, and if a chip has no carry chain, it will try to optimize the adder in different ways (speed or area).

## LEARNING THE LANGUAGE

The best way to learn a specific HDL compiler's style is to refer to the HDL compiler's programmer's manual in which there are examples of specific coding styles to use to optimize. In fact, most of the examples I use come straight from the XST User Guide, which is the compiler that comes with WebPack. There are perhaps two things that are hard to get used to for

new HDL programmers, especially if you have had previous programming experience in languages like C, Pascal, or Visual Basic.

The most important thing is that in HDL, everything usually happens in parallel and asynchronously. You have to specify when you want something to happen in sequence. Listing 2a shows a sample of VHDL code. In this example, you can think of it as two combinatorial circuits that will arrive at an answer in parallel. x and y will be computed at the same time.

Listing 2b shows some sequential code. In this example, an event is expressed in a process block. The process block specifies which signals are monitored for a change before the computation in the block will happen. The equation is guarded by an *if* statement specifying that the result should only change when the signal latch is high.

You can use the same style if you

**Listings 2a**—HDLs will implement structure in parallel. Both results (X and Y) are computed at the same time, unlike traditional programming languages. **b**—To force sequential assignments in VHDL, you use the process block to define a sensitivity list of signal and an if statement to guard expressions. In this example, you use a level sensitive latch. **c**—This is an example of an edge sensitive clock.

**a)**

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
entity func1 is
  port(A,B,C,D : in std_logic;
        X,Y : out std_logic;
end func1;

architecture archi of func1 is
begin
  x = ( a and b ) or ( not a and not b );
  y = ( c and not d ) or ( not c and d );
end archi;
```

**b)**

```
library ieee;
use ieee.std_logic_1164.all;
entity latch is
  port(L, D : in std_logic;
        Q : out std_logic);
end latch;

architecture archi of latch is
begin
  process (L)
  begin
    if (L='1') then
      Q <= D;
    end if;
  end process;
end archi;
```

**c)**

```
library ieee;
use ieee.std_logic_1164.all;
entity flop is
  port(C, D : in std_logic;
        Q : out std_logic);
end flop;

architecture archi of flop is
begin
  process (C)
  begin
    if (C'event and C='1') then
      Q <= D;
    end if;
  end process;
end archi;
```

want to express clocked circuits. In this case, the code will look similar to Listing 2c. The event on the clock signal indicates that you want to look at the signal after the event happens. If the signal is high, then it was a rising edge, and on a falling edge, the signal would be low after an event. You can also use the functions *rising\_edge(ck)* or *falling\_edge(ck)* in many VHDL compilers. Extending this, you can also

implement a register; in this case, the signals have to be defined as *std\_logic\_vectors*, but the code is the same.

## SIGNAL TYPES

VHDL has several signal types. This is necessary because you want to be able to differentiate between signed and unsigned arithmetic. The type of signals available depends on the type

of package you include. Table 1 shows what types you need to use with specific packages to use unsigned or signed arithmetic.

Of course, normal signals of *std\_logic* are only 1 bit long and don't really need types.

Inferring muxes can also seem like a black art. In FPGAs, there are many kinds of muxes you can use. Listings 3 a, b, c, and d show the different styles. The first style seen in Listing 3a is implemented using *if/elseif* statements. This style of mux is sometimes hard to read, and realize that the compiler will actually infer a mux from it. A better method is to use the code in Listing 3b. Here the mux is implemented using a case statement. In Listing 3c, the mux is implemented using tristate buffers. This style of mux is efficient in FPGAs that support internal tristate buffers, and the number of mux states is big. Finally, in Listing 3d, you can see that it's similar to the style in Listing 3a, but in this case, no mux is inferred because there is no final *else* clause.

## MATHEMATICAL OPERATIONS

One area where HDLs really make for compact code is in doing mathematical operations. I showed you earlier that HDL compilers know how to infer adders. Remember that Listing 1a will infer adders. The type of adder depends on whether the signal type is signed or unsigned.

Some other examples of math you can do can be seen in Listing 4, which shows various functions. Of course, implementing dividers and multipliers on smaller parts can be fruitless because these structures tend to be big unless they are constant power of two functions. I've included some comparison examples as well.

Listing 5 shows a simple counter. More complex counters are easy to implement by simply changing the next state equation. Also, note that this counter uses an asynchronous clear signal. It's implemented by adding the clear signal to the process block sensitivity list and using it outside the clock-guarded *if* statement block. In a sense, the clear signal is a second clock-like signal, which is level-sensitive.

I've just scratched the surface with this article, but you should be able to get a feel for the type of programming that can be done with HDLs. The corresponding Verilog examples are similar and can be found in the XST manual as well.

HDL may be new to you, but you will eventually have to take the plunge and learn it if you want to stay fluid while programming FPGA/CPLDs. It's not really all that hard. I'd start with some sample designs and modify them to see what happens and then start with small HDL modules of your own by referring to the style guides and other examples. A good place to start is with a state machine that can be a component in a large design that may still be a schematic. After you familiarize yourself with smaller examples, you can try a complete design in an HDL. WebPack is certainly an excellent place to start, especially because the price is right.

*Ingo Cyliax is a computer and electrical engineer (BSCEE) and the founder of EZComm Consulting, which specializes in embedded systems and FPGA design services as well as troubleshooting. Ingo has been writing about various topics ranging from real-time operating systems to nuts-and-bolts hardware issues for several years. He can be reached at cyliax@ezcomm.com.*

## SOFTWARE

The software is available for downloading on the *Circuit Cellar* web site.

## REFERENCE

Xilinx, *Xilinx Synthesis Technology (XST) User Guide*, [www.toolbox.xilinx.com/docsan/3-li/data/fise/xst/xst.htm](http://www.toolbox.xilinx.com/docsan/3-li/data/fise/xst/xst.htm).

## SOURCE

### WebPack

Xilinx, Inc.  
 (408) 559-7778  
 Fax: (408) 559-7114  
[www.xilinx.com](http://www.xilinx.com)

**Listings 3a**—Here you can see a mux implemented using if/elseif structures. **b**—Using a CASE statement makes it clear that this is a mux **c**—If the hardware supports it, you can also specify to use tristate buffers to implement muxes. The "Z" is the value of a tristate signal when it's in the "off" state. **d**—Take care with this structure. It looks like Listing 3a, but it's not implemented as a mux because it is missing the else clause.

```

a)
library ieee;
use ieee.std_logic_1164.all;
entity mux is
  port (a, b, c, d : in std_logic;
        s : in std_logic_vector (1 downto 0);
        o : out std_logic);
end mux;
architecture archi of mux is
begin
  process (a, b, c, d, s)
  begin
    if (s = "00") then o <= a;
    elsif (s = "01") then o <= b;
    elsif (s = "10") then o <= c;
    else o <= d;
    end if;
  end process;
end archi;

b)
library ieee;
use ieee.std_logic_1164.all;
entity mux is
  port (a, b, c, d : in std_logic;
        s : in std_logic_vector (1 downto 0);
        o : out std_logic);
end mux;
architecture archi of mux is
begin
  process (a, b, c, d, s)
  begin
    case s is
      when "00" => o <= a;
      when "01" => o <= b;
      when "10" => o <= c;
      when others => o <= d;
    end case;
  end process;
end archi;

c)
library ieee;
use ieee.std_logic_1164.all;
entity mux is
  port (a, b, c, d : in std_logic;
        s : in std_logic_vector (3 downto 0);
        o : out std_logic);
end mux;
architecture archi of mux is
begin
  o <= a when (s(0)='0') else 'Z';
  o <= b when (s(1)='0') else 'Z';
  o <= c when (s(2)='0') else 'Z';
  o <= d when (s(3)='0') else 'Z';
end archi;

d)
library ieee;
use ieee.std_logic_1164.all;
entity mux is
  port (a, b, c, d : in std_logic;
        s : in std_logic_vector (1 downto 0);
        o : out std_logic);

```

(continued)

**Listing 3—continued**

```
end mux;
architecture archi of mux is
begin
  process (a, b, c, d, s)
  begin
    if      (s = "00") then o <= a;
    elsif  (s = "01") then o <= b;
    elsif  (s = "10") then o <= c;
    end if;
  end process;
end archi
```

**Listing 4—Here you can see some math functions. Most hardware compilers do not implement dividing by arbitrary numbers. However, dividing by a power of two is handy because it's just a shift.**

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity math is
port(
  A: in std_logic_vector(7 downto 0);
  B: in std_logic_vector(3 downto 0);
  C: in std_logic_vector(7 downto 0);
  D: in std_logic_vector(7 downto 0);
  M0: out std_logic_vector(11 downto 0);
  D0: out std_logic_vector(7 downto 0);
  GTE: out std_logic;
  LTE: out std_logic);
end math;

architecture archi of math is
begin
  M0 <= A * B;
  D0 <= D / 2;

  GTE <= '1' when C >= D else '0';
  LTE <= '1' when C <= D else '0';
end archi;
```

**Listing 5—Here you can see a synchronous counter with an asynchronous clear. Sequential counters are automatically implemented with efficient adder/subtractors architectural features if available. Other counter sequences can be implemented by using a CASE statement as a mux and explicit sequencing the values.**

```
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity counter is
  port(C, CLR : in std_logic;
        Q : out std_logic_vector(3 downto 0));
end counter;
architecture archi of counter is
  signal tmp: std_logic_vector(3 downto 0);
begin
  process (C, CLR)
  begin
    if (CLR='1') then
      tmp <= "0000";
    elsif (C'event and C='1') then
      tmp <= tmp + 1;
    end if;
  end process;
  Q <= tmp;
end archi;
```

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, [subscribe@circuitcellar.com](mailto:subscribe@circuitcellar.com) or [www.circuitcellar.com/subscribe.htm](http://www.circuitcellar.com/subscribe.htm).