A CPLD VHDL Introduction

XAPP 105 January12, 1998 (Version 1.0) Application Note

Summary

This introduction covers the basics of VHDL as applied to Complex Programmable Logic Devices. Specifically included are those design practices that translate well to CPLDs, permitting designers to use the best features of this powerful language to extract the best performance from CPLD designs.

Family

XC9500

Introduction

VHDL, is an extremely versatile tool developed to aid in many aspects of IC design. The language allows a user to express circuits in many levels of detail. Unfortunately, this versatility also makes the VHDL synthesis tools job a lot more difficult. There is room for interpretation depending on the VHDL coding style. A particular synthesis tool may implement the same code very differently from another. To achieve the best results using VHDL, the designer should work at the Register Transfer Level (RTL).

Although working at the RTL for designs may be more tedious, all of the major synthesis tools on the market are capable of generating a reasonable implementation of designs for CPLDs when specified this way. Using higher levels of abstraction may give adequate results, but tend to be less efficient. Additionally, by expressing designs simply, the designer also gains the ability to port VHDL designs from one synthesis tool to another with minimal edits.

Overview

VHDL looks very similar to any other kind of programming language. [Note VHDL is not case sensitive.] Let's look at the composition of a typical design entered in VHDL. The first thing you will notice are library statements. Libraries allow a designer to use very common functions and declarations without having to redefine them every time they need to be used. Next, each design requires an ENTITY declaration. This defines the designs I/O signals. Following the entity is an ARCHITECTURE description detailing the entity behavior. Within the architecture, the designer declares internal signals, components, and processes to describe the designs detailed behavior.

Figure 1 shows a complete VHDL design with a single D-type Flip Flop.

```
library ieee;
use ieee.std_logic_1164.all;
--Comments are denoted by two - signs.
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
--Two very useful libraries. Use these if
--you use any kind of math function, such as
--building counters.
entity DFLOP is
   port
   (
   my_clk     : in   std_logic;
   D_input    : in   std_logic;
   Q_output   : out  1std_logic,
   );
end DFLOP;

architecture behavior of DFLOP is
signal my_useless signal:     std_logic;
--Additional internal signals and constants
--would be added here.
begin
   process
   begin
     wait until my_clk'event and my_clk = '1';
     Q_output <= D_input;
   end process;
end behavior;
```

Figure 1: Simple VHDL Design for a D Flip Flop

Entity

The ENTITY declaration is similar to that of a C header file, or a schematic symbol block. The ENTITY declares all of the input and output ports of a particular component. The port directions are defined as input, output, inout, or buffers. Inputs are for declaring dedicated inputs, and outputs declare dedicated outputs. Inouts define bi-directional signals, and a buffer is used when a dedicated output must also be used as an internal signal. The type of signals

associated with these declarations should be of the std_logic type. The std_logic types are defined in the library STD_LOGIC.1164. Table 1 shows different port declarations.

Table 1: Port Declaration Types

Signal	Type	Output
my_clock	:in	std_logic;
my_input	:in	std_logic;
my_address bus	:in	std_logic_vector (15 downto 0)
my_data_bus	:inout	std_logic_vector (31 downto 0);
my_output	:out	std_logic;
my_counter	:buffer	std_logic vector (7 downto 0);

Architecture

The ARCHITECTURE statement in Figure 1 allows the designer to define entity behavior. Multiple architectures can be defined for the same entity. This allows a user to assign different designs to the same entity block. Within the architecture block, the designer can declare signals to be used internally, additional components that the design will require, and different processes to describe how the design will respond to external signals.

Signals

Signal declaration is similar to the entity declarations. Signals are generally declared as bits, or vectors. See Table 2 for examples of proper syntax. The designer should also be aware that these signals may or may not actually be used after synthesis, depending on the level of optimization that occurs.

Table 2: Signal Declaration

Signal Declaration	Output
signal internal_clock	:std_logic;
signal my_signal	:std_logic;
signal my_register	:std_logic_vector (15 downto 0);

Operators

Assignment

The most basic operator in VHDL is the assignment operator, <=. For example:

```
A <= B;
```

means ‘A’ is assigned the value of ‘B’. Note that ‘A’ and ‘B’ must have the same number of elements, or bits, in order for assignment to be valid, otherwise an error will be flagged at compile time. Additionally, we can assign bit values.

```
Data_out <= "00001111";
```

And if Data_out was declared as an inout, we can also assign the tristate value as well.

```
Data_out <= "ZZZZZZZ";
```

Logical Operators

In the previous examples, you may have noted that we use the word "**and**" to perform a logical “**and**” function. Unlike many other HDLs, VHDL uses the following text to perform the logical functions: **and**, **or**, **xor**, **nor**, **nand** and **not**. Relational operators are simply: **=**, **>**, **>=**, **<**, **<=, /=**.

Arithmetic Operators

Binary arithmetic in VHDL must be defined. Luckily, there are predefined libraries for the most common arithmetic functions. The two IEEE libraries:

```
USE ieee.std_logic_arith.all,
USE ieee.std_logic_unsigned.all,
```

defines the binary add, subtract, multiply, and divide functions. The operators are simply: +, -, * ,and / .

Process

The process statement allows designers to define a set of procedures the design must follow when a certain signal changes. A behavioral description may contain several different processes to achieve the desired functionality. The most distinguishing factor of any HDL compared to a standard programming language is the idea of concurrent evaluation, or parallel execution. Keeping this in mind, a designer new to VHDL must remember that all processes run in parallel. In order to process VHDL code, however, a compiler must still decide when to evaluate the process in order to successfully simulate and synthesize the code. The concept of a sensitivity list is used where a process is only considered when the state of a signal in the sensitivity

list changes. In the following example, myoutput will be assigned myinput whenever myinput changes. This is shown in Figure 2.

Synchronous logic can employ an implied sensitivity by using the WAIT UNTIL statement. This allows a process to idle until the function specified becomes true. This is convenient for defining rising edge triggered flip-flops. Any design utilizing the flip flops of the XC9500 can be coded as in Figure 3.

In this example, a D-type flip flop is generated where the input comes from myinput, the output attaches to myoutput, and the clock signal is tied to myclock.

Conditionals

Conditional statements in VHDL are handled primarily by two statements, the "if-then-else", and "case" statements.

IF-THEN-ELSE Statements

You must completely specify the default conditions when synthesizing designs. For example compare Figure 4 to Figure 5.

The code shown in Figure 4 will implement a one-shot circuit and set chip_sel when the address_bus is greater than the bit pattern “00001111”. The code in Figure 5, however, will perform the proper address decode.

```
process (myinput)
   begin
   myoutput <= myinput;
end process;
```

Figure 2: Process Statement

```
process
   begin
   Wait until myclock'event and myclock = '1';
   myoutput <= myinput;
end process;
```

Figure 3: Flip Flop

```
if my_address_bus > "00001111" then chip_sel <= '1';
end if;
```

Figure 4: One-shot Circuit

```
if my_address_bus > “00001111” then chip_sel <= '1';
   else chip_sel <= ‘0’;
end if;
```

Figure 5: Correct If-Then-Else Decoder

CASE Statements

For case statements, make sure there is a **when others** to specify the default condition, even if it's a null statement. Figure 6 shows how to code a simple 4 to 1 mux.

```
library ieee;
use ieee.std_logic_1164.all;

entity mux is
   port
   (
   Ain : in std_logic_vector (15 downto 0);
   Bin: instd_logic_vector (15 downto 0);
   Cin: instd_logic_vector (15 downto 0);
   Din: instd_logic_vector (15 downto 0);
   Muxout: outstd_logic_vector (15 downto 0);
   sel: instd_logic_vector (1 downto 0)
   );
end mux;

Architecture behavior of mux is:
begin
process (Ain, Bin, Cin, Din, sel)

begin
   case sel is
   when "00" => Muxout <= Ain;
   when "01" => Muxout <= Bin;
   when "10" => Muxout <= Cin;
   when "11" => Muxout <= Din;
   when others => null;
   end case;
end process;

end behavior of mux
```

Figure 6: Case Statement Example

State Machines

Writing an efficient state machine in VHDL can be a tricky task. The easiest way to write a concise state machine with a clean implementation is to use the CASE statement. Maintaining your state machine code this way allows any other user (or even yourself a week later) to review your code and understand it quickly. The following example describes the operation of a generic traffic light controller.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity traffic is
   port
   (
   My_clk: instd_logic;
   Red_light: bufferstd_logic;
   Yellow_light: bufferstd_logic;
   Green_light: bufferstd_logic
   );
end traffic;
architecture behavior of traffic is
signal my_counter : std_logic_vector (4 downto 0);
signal my_state   : std_logic_vector (2 downto 0);
constant RED_STATE : std_logic_vector (2 downto 0) := "100";
constant YELLOW_STATE : std_logic_vector (2 downto 0) := "010";
constant GREEN_STATE : std_logic_vector (2 downto 0) := "001";

begin
Red_light <= my_state(2);
Yellow_light <= my_state(1);
Green_light <= my_state(0);

   process
   begin
   wait until My_clk'Event and My_clk = '1';
   case my_state is

   when GREEN_STATE =>
     my_counter <= my_counter + 1;
     if my_counter = "11110" then
       my_state <= YELLOW_STATE;
     end if;

   when YELLOW_STATE =>
     my_counter <= my_counter + 1;
     if my_counter = "11111" then
       my_state <= GREEN_STATE;
     end if;

   when RED_STATE =>
     if my_counter < "11110" then
       my_counter <= my_counter + 1;
     else
       my_state <= GREEN_STATE;
       my_counter <= "00000";
     end if;

   when others => my_state <= RED_STATE;
     my_counter <= "00000";
   end case;
   end process;

end behavior;
```

Figure 7: State Machine Example

Coding Techniques

Efficient synthesis of VHDL often depends on how the design was coded. As different synthesis engines will produce different results, leaving as little as possible to chance will increase the speed and improve the density of your design. This, however, often trades off some of the advantages of using higher level constructs and libraries.

Compare functions:

Address decodes often require a decode of a range of address locations. It is inevitable to use the greater than or less than test. Wait state generation, however, often waits a known number of clock cycles. Consider this VHDL code.

```
when wait_state =>
   if wait_counter < wait_time then
     wait_counter <= wait_counter + 1;
     my_state <= wait_state;
   else
      my_state <= next_state;
   end if;
```

This generates extensive logic to implement the compare. A more efficient implementation would be to branch on the equality condition.

```
when wait_state =>
   if wait_counter = wait_time then
     my_state <= next_state;
   else
     wait_counter <= wait_counter + 1;
     my_state <= wait_state;
   end if;
```

Don't Care Conditions

When writing VHDL, it is easy to forget about signals that are of no concern in the specific piece of code handling a certain function. For example, you may be generating a memory address for a memory controller that is important when your address strobes are active, however, these outputs are essentially don't care conditions otherwise. Don't forget to assign them as such, otherwise, the VHDL synthesizer will assume that it should hold the output at the last known value.

```
when RAS_ADDRESS =>
   memory_address <= bus_address[31 downto 16]
     RAS <= '0';
     CAS <= '1';
     my_state <= CAS_ADDRESS

when CAS_ADDRESS =>
   memory_address <= bus_address[15 downto 0]
     RAS <= '0';
     CAS <= '0';
     wait_count <= zero;
     my_state <= WAIT_1

when WAIT_1 =>
     RAS <= '0';
     CAS <= '1';
   if wait_count = wait_length then
     my_state <= NEXT_ADDRESS;
   else
     wait_count <= wait_count + 1;
   end if;
```

This design can be implemented much more efficiently if coded as:

```
when RAS_ADDRESS =>
   memory_address <= bus_address[31 downto 16]
     RAS <= '0';
     CAS <= '1';
     wait_count <= "XXXX";
     my_state <= CAS_ADDRESS

when CAS_ADDRESS =>
   memory_address <= bus_address[15 downto 0]
     RAS <= '0';
     CAS <= '0';
     wait_count <= "0000";
     my_state <= WAIT_1

when WAIT_1 =>
   memory_address <= "XXXXXXXXXXXXXXXX";
     RAS <= '0';
     CAS <= '1';
   if wait_count = wait_length then
     my_state <= NEXT_ADDRESS;
   else
     wait_count <= wait_count + 1;
   end if;
```

Note that we add don't care for the wait_count register in the RAS_ADDRESS state as well as adding a don't care assignment for the memory address register in the WAIT_1 state. By specifying these don't cares, the final optimized implementation will be improved.

Using Specific Assignments

There is a temptation with VHDL to use language tricks to compact code. For, example, using a counter to increment a state variable. While this allows you to write quicker and visually appealing code, the result is the synthesis tool generates more complex logic to implement the adder and try to optimize the logic that is unused later. It is generally better to simply assign your desired bit pattern directly. This generates logic that is quicker to collapse during the subsequent fitter process.

Modularity

A designer can “rubber stamp” a design by instantiating multiple instances of an existing design entity. Component instantiation is basically the same as applying schematic macros. First, apply the COMPONENT declaration to define what the input and output ports are. The component declaration must match the actual entity declaration of the component. For example, if we reused the flip-flop from our previous example from Figure 1 in another design, we can declare it with:

```
component DFLOP port
    (
    my_clk     :in     std_logic;
    D_input    :in     std_logic;
    Q_output   :out    std_logic
    );
```

The component can than be instantiated with the signals necessary to connect the component to the rest of the design. The signals can be mapped positionally or explicitly. Positional mapping is quicker to enter, but forbids omitting unneeded logic. For example, if you had an 8 bit loadable counter that was never reloaded, explicit mapping allows you to omit signal assignment to the input ports and still use the same 8 bit counter definition.

To instantiate the component requires a unique label. Then we state the component name being instantiated followed by the positional signal assignments we are attaching to the component. An example of position signal mapping would be:

```
my_flop: DFF port map(clk, my_input, my_output);
```

The same flop mapped explicitly is shown below. Note: that the order can be altered when mapping explicitly.

```
my_second_flop: DFLOP port map (my_clk => clk,
Q_output => my_other_output,
D_input => my_other_input);
```

Multiple Instances

Generating many instances of the same component can be done with the looping construct. The for loop in this case generates our index for vectored signals. In this example, we first define how a latch will operate.

```
library ieee;
use ieee.std_logic_1164.all;
entity latch is
   port
   (
   Din: instd_logic;
   enable: instd_logic;
   Qout: inout std_logic
   );
end latch;

architecture behavior of latch is:

begin
   process (Din, enable)
   begin
   if enable = '1' then Qout <= Din;
   end if;
   end process;
end behavior;
```

In this example, we first define how a latch will operate. We will use this latch as a component in our loadable counter.

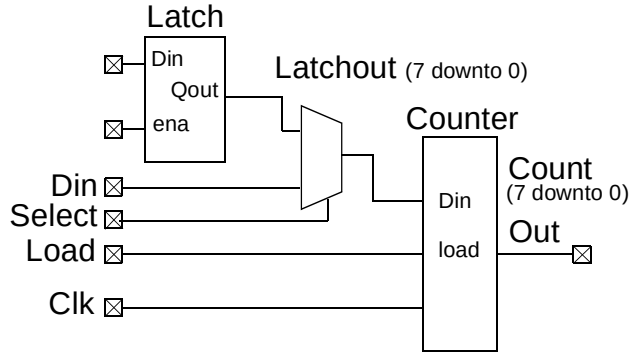

Figure 8: Loadable Counter

The value it loads will either be from the latched inputs (Din), or a set of direct inputs (Syncin). This will be selected by the sel signal. See Figure 9 next page.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use WORK.ALL;
entity counter is:
   port
   (
   Din :instd_logic_vector (7 downto 0);
   Syncin:instd_logic_vector (7 downto 0);
   load:instd_logic;
   sel :instd_logic;
   clk :instd_logic;
   enable:instd_logic;
   Count_out:bufferstd_logic_vector (7 downto 0)
   );
end counter;
architecture behavior of counter is:

signalcounter:std_logic_vector (7 downto 0);
signallatchout:std_logic_vector (7 downto 0);
component latch port (Din, enable:in_ std_logic; Qout:out_ std_logic);
end component;

begin
Count_out <= counter(7 downto 0);
--Here, we map 8 instances of our latch with a FOR
--loop.
--Note: The label sync. EachFOR loop must have a
--unique label.

sync: for i in 0 to 7 GENERATE
   latch_load: latch port map (Din(i), enable, latchout(i));
end GENERATE sync;

   process
   begin
     wait until clk'Event and clk = '1';
     if (load = '1' and sel = '0') then counter
       <= latchout;
     elsif (load = '1' and sel = '1') then counter
       <= Syncin;
     else counter <= counter + 1;
     end if;
   end process;
end behavior;
```

Figure 9: Counter Example

Bi-Directional Ports

To implement bi-directional ports (Figure 10), we must first define the port to be of type inout. Then, we must define when the port is driving, and when it is in a high-Z mode. The example in Figure 11 implements this structure.

```
library ieee;
use ieee.std_logic_1164.all;
entity bidi is
   port
   (
   Data:inoutstd_logic_vector (7 downto 0);
   direction in std_logic;
   clk:in std_logic
   );
end bidi;

architecture behavior of bidi is:

signal my_register : std_logic_vector (7 downto
0);

begin
     process (direction, my_register)
     begin
       if (direction = '1') then
       Data <= "ZZZZZZZZ";
       else
       Data <= my_register;
       end if;
     end process;
     process
     begin
       wait until clk'event and clk = '1';
       my_register <= Data;
     end process;
end behavior;
```

Figure 10: Bi-directional Port Definition

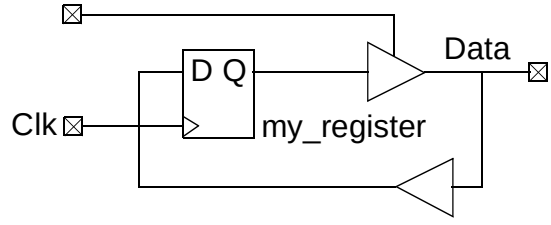

Figure 11: Bidirectional Port

Summary

The basic structure of a VHDL design has been shown, as well as numerous examples of basic building blocks. The examples provided are independent of specific third party synthesis tools. By using these basic constructs, you can implement your own designs and generate a netlist for the Xilinx M1 software.