CSEE W4840 Embedded System Design Lab 3

Stephen A. Edwards

Due February 17, 2005

Abstract

Reverse-engineer some synthesizable VHDL circuit models. Examine their source to draw a block diagram. Use a simulator to observe their behavior and draw timing diagrams. Explain what each circuit does.

1 Introduction

In this class, you'll be using VHDL (VHSIC Hardware Description Language) to describe hardware. It is a fairly verbose language, but fairly simple at its core. You will want to consult the *Writing VHDL for RTL Synthesis* handout available on the class webpage for examples of how to write VHDL. Unfortunately, the language is very big and large parts of it cannot be directly transated into hardware. As such, things like the language reference manual and the majority of VHDL books are useless because they are very complicated and it is difficult to determine when you may actually use what they teach to specify hardware.

Although the syntax of VHDL vaguely resembles that of an imperative language like C, do not be deceived: *VHDL is not a programming language*. In particular, the sort of imperative, algorithmic thinking that works well to solve problems in C *will not* work in VHDL. A C-like VHDL program will probably not compile, if it does compile, it probably will not work, and even if it does compile, you will not like the result.

VHDL is mostly a structural language. VHDL is useful mostly for defining how components connect. The main idea is that a system is composed of hierarchically-arranged blocks called entity/architecture pairs (everything in VHDL has a weird name). For each block, you define its interface (a list of wires that enter and leave it) and its guts, which may consist of instances of other blocks, dataflow expressions (e.g., a particular signal is the logical AND of two others), and processes that appear to contain imperative code.

2 An Example

Figure 1 shows a VHDL circuit model modeling a simple, and probably useless, ALU-like object. Defined in the entity, its inputs are a clock, reset and enable signals, and two eight-bit numbers. In response, it produces an eight-bit result. The architecture defines an eight-bit intermediate result called *w* and contains a single clock-triggered process that represents a block of combinational logic feeding a bank of eight edge-sensitive flip-flops (the *w* signal).

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

entity dumb_alu is
  port(clk     : in  std_logic;
       reset   : in  std_logic;
       enable  : in  std_logic;
       x, y    : in  std_logic_vector(7 downto 0);
       z       : out std_logic_vector(7 downto 0));
end dumb_alu;

architecture behavioral of dumb_alu is

  signal w : std_logic_vector(7 downto 0);

begin
  process (clk)
  begin
    if clk'event and clk='1' then
      if reset='1' then
        w <= "00000000";
      elsif enable='1' then
        w <= y - 4;
      else
        w <= x + y;
      end if;
    end if;
  end process;

  z <= w;

end behavioral;
```

Figure 1: A synthesizable VHDL entity/architecture.

The `clk'event and clk = '1'` idiom (a standard one) indicates that these are a positive-edge-triggered flip-flops. Inside this block is a series of if-then-else statements that assign to *w* depending on whether *reset* is true and whether *enable* is true. Together, these imply the steering (multiplexer) and arithmetic logic shown in Figure 3.

The semantics of VHDL are such that the inputs to this block are read just before the rising edge of the clock and the outputs are produced just after the rising edge, as shown in Figure 4. Note that the inputs may change at any time during a clock signal but that the output only changes on the rising edge.

Figure 2 shows a testbench for the block in Figure 1. Its job is

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

entity testbench is
end testbench;

architecture behavioral of testbench is

  signal clk : std_logic := '0';
  signal reset, enable : std_logic;
  signal x, y, z: std_logic_vector(7 downto 0);

  component dumb_alu
    port(
     clk    : in  std_logic;
     reset  : in  std_logic;
     enable : in  std_logic;
     x, y   : in  std_logic_vector(7 downto 0);
     z      : out std_logic_vector(7 downto 0));
  end component;

begin

  device_under_test: dumb_alu
    port map (
      clk => clk,
      reset => reset,
      enable => enable,
      x => x,
      y => y,
      z => z);

  clkgen : process
  begin
    wait for 10 ns;
    clk <= not clk;
  end process;

   reset <= '1',
            '0' after 60 ns;

   enable <= '0',
             '1' after 120 ns,
             '0' after 140 ns,
             '1' after 160 ns,
             '0' after 180 ns;

   datagen : process
   begin
     wait for 51 ns;
     x <= X"04"; y <= X"02"; wait for 20 ns;
     x <= X"03"; y <= X"00"; wait for 20 ns;
     x <= X"08"; y <= X"01"; wait for 20 ns;
     x <= X"07"; y <= X"04"; wait for 20 ns;
     x <= X"05"; y <= X"03"; wait for 20 ns;
     x <= X"06"; y <= X"05"; wait for 40 ns;
     x <= X"07"; y <= X"01";
     wait;  -- forever
   end process;

end behavioral;
```

Figure 2: A testbench for Figure 1.

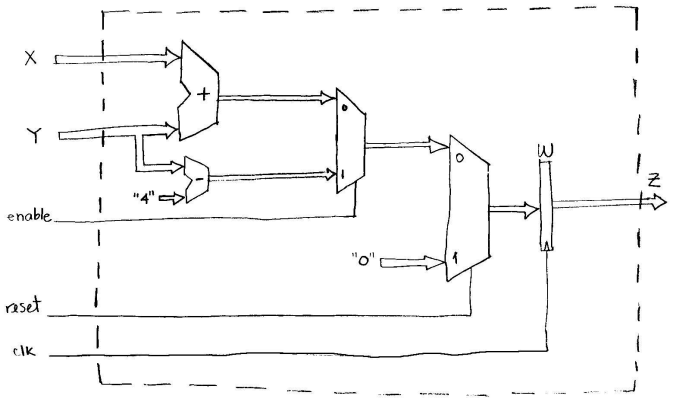

Figure 3: A block diagram for Figure 1.

to provide stimulus to the block so that a simulation does something interesting. This particular testbench applies the waveforms shown in Figure 4 to produce Figure 5, a screenshot from the Cadence waveform viewer that we will use to observe the output of these modules.

The testbench uses a different subset of the VHDL language that is not synthesizable, i.e., cannot automatically be turned into logic. For example, the clock is generated by the *clkgen* process in Figure 2 that toggles the signal every 10 ns, a behavior that cannot be implemented using only logic gates (you need a controlled oscillator).

A clock generator is probably the most common part of a testbench. Other parts include assignment statements using the *after* keyword, which provides control over the time at which the assignment will take place (c.f., the statements generating the *reset* and *enable* signals), and processes with delays for generating more complex waveforms. The *datagen* process is an example of this, which first waits until slightly after the second clock cycle before generating a sequence of assignments to the *x* and *y* vectors. Notice the use of a final, lone *wait* statement, which effectively terminates the process (it will otherwise repeat indefinitely).

I have created a Makefile that runs NCVHDL, an excellent VHDL simulator from Cadence Design Systems. Typing "make" will invoke ncvhdl, the simulator front-end; ncelab, sort of a linker; and ncsim, the actual simulator that has quite an elaborate GUI, as well as a text mode. You will get most of your errors from the front end, which will report on things such as undeclared names. The error messages are cryptic, but at least contain the line number of the offending VHDL code.

Once the simulator window comes up, click on the *navigator* button (upper-right corner) and then on the chip with a few dots near to it to see a list of signals in the design. Selecting these signals and then right-clicking will bring up a menu that, among others, allows you to display these signals on a waveform viewer. Run and stop the simulator by clicking on the big triangle in the upper-left corner and the waveform viewer will

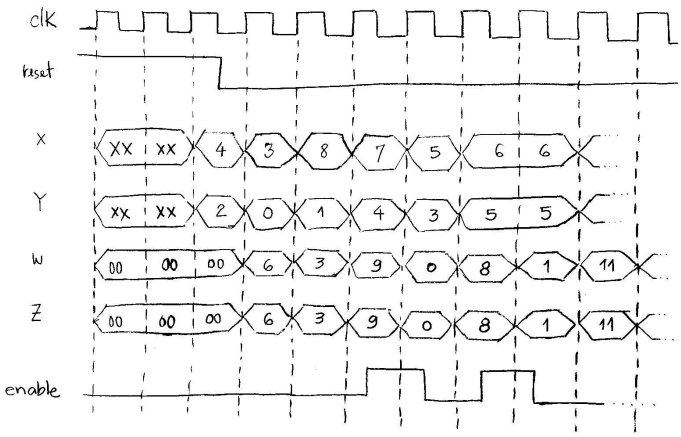

Figure 4: A timing diagram for Figure 1.

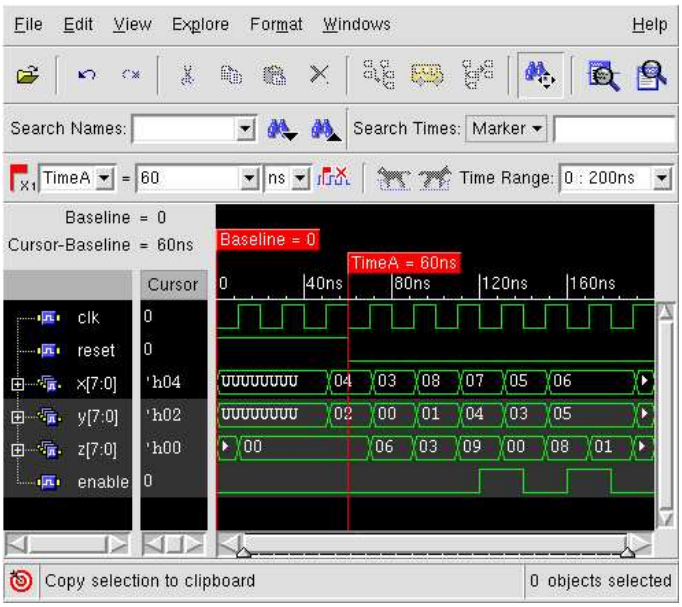

Figure 5: The waveform viewer output from the testbench in Figure 2.

display the results. Like all GUIs, I find it difficult to use, but it will deliver the goods in the end.

3 The Assignment

Draw block and timing diagrams for the three VHDL designs in the lab3 tarfile (sedwards/4840/lab3.tar.gz) and explain what they do. They are in the directories design1, design2, and design3. The lab-example directory contains the example and test bench in this lab handout.

Use the NCVHDL simulator to validate your analysis of the circuits. Create a testbench for each that puts the design through its paces, illustrating its operation.

Show the TA the waveforms on the screen (the windows are too ugly to print) and hand in your block and timing diagrams for each of these three designs.