Solutions to VHDL assignments

Linus Karlsson

November 14, 2017

Solutions

Exercise 1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity combinational is
    Port ( x : in STD_LOGIC_VECTOR(3 downto 0);
           z : out  STD_LOGIC;
           ge : out  STD_LOGIC;
           p : out  STD_LOGIC);
end combinational;

architecture Behavioral of combinational is
begin
        z <= '1' when x = "0000" else '0';
        ge <= '1' when x >= "0111" else '0';
        p <= x(3) xor x(2) xor x(1) xor x(0);
end Behavioral;
```

It is not a requirement to use `std_logic_vector` for the input `x`, but it reduces the amount of code. If separate signals are used, you could write `z` as, for example: (much like `p` in the code above)

```
z <= not(x3 or x2 or x1 or x0);
```

Exercise 2

The code for the full-adder.[1]

[1] Nothing strange here really. The expressions from the book are translated into VHDL. Note, however, that **and**, **xor**, and **or** have the same precedence in VHDL. Thus, parentheses must be used, otherwise you'll get subtle bugs that are really hard to find!

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fulladder is
    Port ( x : in  STD_LOGIC;
           y : in  STD_LOGIC;
           ci : in  STD_LOGIC;
           s : out  STD_LOGIC;
           co : out  STD_LOGIC);
end fulladder;

architecture Behavioral of fulladder is
begin
        s <= x xor y xor ci;
        co <= (x and y) or ((x xor y) and ci);
end Behavioral;
```

And now the code for the four-bit adder.[2]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fourbitadder is
    Port ( x : in  STD_LOGIC_VECTOR (3 downto 0);
           y : in  STD_LOGIC_VECTOR (3 downto 0);
           sub : in  STD_LOGIC;
           s : out  STD_LOGIC_VECTOR (3 downto 0);
           c4 : out  STD_LOGIC;
           ov : out  STD_LOGIC);
end fourbitadder;

architecture Behavioral of fourbitadder is
        component fulladder
                port (
                        x, y, ci: in std_logic;
                        s, co: out std_logic
                );
        end component;

        -- used to connect the carry between the full adders.
        signal ripple_carry: std_logic_vector(4 downto 1);

        -- if we subtract, we must XOR y with '1', requiring a new signal.
        signal fixed_y: std_logic_vector(3 downto 0);
begin
        fixed_y <= y xor (sub, sub, sub, sub); -- xor with four bit vector.

        fa0: fulladder port map (
                        x => x(0),
                        y => fixed_y(0),
                        ci => sub,
                        s => s(0),
                        co => ripple_carry(1)
                );
        fa1: fulladder port map (
                        x => x(1),
                        y => fixed_y(1),
                        ci => ripple_carry(1),
                        s => s(1),
                        co => ripple_carry(2)
                );
        fa2: fulladder port map (
                        x => x(2),
                        y => fixed_y(2),
                        ci => ripple_carry(2),
                        s => s(2),
                        co => ripple_carry(3)
                );
        fa3: fulladder port map (
                        x => x(3),
                        y => fixed_y(3),
                        ci => ripple_carry(3),
                        s => s(3),
                        co => ripple_carry(4)
                );
        -- if the last two carry bits are different, then overflow.
        ov <= ripple_carry(4) xor ripple_carry(3);
        c4 <= ripple_carry(4);
end Behavioral;
```

[2] Also, see page 179 of the course book. Some comments about this solution:

- Note that *components* are used to re-use our full-adder created in the previous step.
- Each component can be *instantiated* multiple times. In this example we instantiate our fulladder component four times.
- Each instantiation requires a port map, where we connect the inputs and outputs.
- To connect the carries between the full-adders, we must define a signal (here called `ripple_carry`). By using the signal name as an output in one full adder, and as an input in another full adder, a connection is made (think of it as a wire between them).

Exercise 3

Code for demux.[3]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity demux is
    Port ( D : in  STD_LOGIC;
           X : in  STD_LOGIC_VECTOR (1 downto 0);
           U : out  STD_LOGIC_VECTOR (3 downto 0));
end demux;

architecture Behavioral of demux is
begin
        process (D, X)
        begin
                U <= "0000";
                case X is
                        when "00"   => U(0) <= D;
                        when "01"   => U(1) <= D;
                        when "10"   => U(2) <= D;
                        when "11"   => U(3) <= D;
                        when others => null;
                end case;
        end process;
end Behavioral;
```

[3] Notes:

- Note that to use a case-statement, you must be inside a process.
- I have used a signal `U` with a default value, in which I only change a single bit depending on the input.
- An alternative solution is to do the signal assignment with concatenation instead. For example, for the row `when "00" =>`, the assignment could be: `U <= "000" & D;` The `&` operator concatenates values and creates a vector.

Exercise 4

Code for return maximum index with input 0.[4]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity maxindex is
    Port ( i0, i1, i2 : in  STD_LOGIC;
           U : out  STD_LOGIC_VECTOR (1 downto 0));
end maxindex;

architecture Behavioral of maxindex is
begin
        process (i0, i1, i2)
        begin
                if i2 = '0' then
                        U <= "10";
                elsif i1 = '0' then
                        U <= "01";
                elsif i0 = '0' then
                        U <= "00";
                else
                        U <= "11";
                end if;
        end process;
end Behavioral;
```

[4] Notes:

- Just like above we need to use a process when using if-statements.
- If $i_2 = 0$, we should return 10. If $i_2 = 1$, we proceed and check input 1, and so on.

Exercise 5

Code for D flip-flop.[5]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity dflipflop is
    Port ( clk : in  STD_LOGIC;
           D : in  STD_LOGIC;
           Q : out  STD_LOGIC);
end dflipflop;

architecture Behavioral of dflipflop is
begin
        process (clk)
        begin
                if rising_edge(clk) then
                        Q <= D;
                end if;
        end process;
end Behavioral;
```

[5] The D flip-flop copies the value of the D input upon the rising edge of the clock signal.

- Note that only the `clk` input need to be on the process sensitivity list, since we only want to update the output Q when there is a change in the clock signal.

Exercise 6

Code for parity sequential circuit.[6]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity parity is
    Port ( clk : in  STD_LOGIC;
           i : in  STD_LOGIC;
           p : out  STD_LOGIC);
end parity;

architecture Behavioral of parity is
        type state_type is (even, odd); -- our different states.
        signal current_state, next_state: state_type; -- state.
begin
        process (clk) -- process to update to next state, and output.
        begin
                if rising_edge(clk) then
                        current_state <= next_state;
                end if;
        end process;

        process (i, current_state) -- Calculate state and output.
        begin
                case current_state is
                        when even =>    if i = '0' then
                                                next_state <= even;
                                                p <= '0';
                                        else
                                                next_state <= odd;
                                                p <= '1';
                                        end if;

                        when odd =>     if i = '0' then
                                                next_state <= odd;
                                                p <= '1';
                                        else
                                                next_state <= even;
                                                p <= '0';
                                        end if;
                end case;
        end process;
end Behavioral;
```

[6] This is a sequential circuit, thus we need to maintain state.

- The state is stored in signals, which are of the type `state_type` which we define ourself. The type should contain all states, with logical names. In this way, we do not have to care about the state assignment, and can use the names instead.
- We use two processes, one which "drives" the machine forward by updating the state upon every clock cycle, and one which calculates the next state, and output upon a change in state or input value.

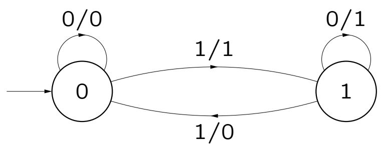

Figure 1: The parity check graph.

Exercise 7

Code for the implementation of our best friend – the lion cage.[7]

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity lioncage is
    Port ( clk : in  STD_LOGIC;
           detector : in  STD_LOGIC_VECTOR (1 downto 0);
           danger : out  STD_LOGIC);
end lioncage;

architecture Behavioral of lioncage is
        type state_type is (s0, s1, s1hat, s2);
        signal current_state, next_state: state_type;
begin
        process (clk)
        begin
                if rising_edge(clk) then
                        current_state <= next_state;
                end if;
        end process;

        process (current_state, detector) -- next state process
        begin
                next_state <= current_state; -- assume unchanged state.
                case current_state is
                        when s0 =>
                                case detector is
                                        when "01"   => next_state <= s1;
                                        when others => null;
                                end case;
                        when s1 =>
                                case detector is
                                        when "10"   => next_state <= s1hat;
                                        when "11"   => next_state <= s0;
                                        when others => null;
                                end case;
                        when s1hat =>
                                case detector is
                                        when "00"   => next_state <= s1;
                                        when "01"   => next_state <= s2;
                                        when others => null;
                                end case;
                        when s2 =>
                                case detector is
                                        when "11"   => next_state <= s1hat;
                                        when others => null;
                                end case;
                end case;
        end process;

        process (current_state, detector) -- output process
        begin
                danger <= '1'; -- assume danger, since that's common.
                if current_state = s0 and detector = "00" then
                        danger <= '0'; -- only case where danger is 0.
                end if;
        end process;
end Behavioral;
```

[7] The core idea of the implementation is just like the parity machine. There are some minor differences:

- The signals are assigned default values at the start of the process, since in most cases we should output danger = 1, and stay in the same state. In this way, we only have to change the signals in the few cases where they actually should be updated.
- We have created three separate processes:
  - The first process, just like in the previous assignment, updates the next state variable upon a rising edge on the clock.
  - The second process calculates the next state, which depends on the current state and the input signals.
  - The third process calculates the output signal. Since this is a Mealy-machine, the output depends on both the current state and the input signals. If we'd have a Moore machine, this process would only depend on the current state.

  This separation with different processes for next state and output is a good design, especially as your circuits grow larger and more complex.

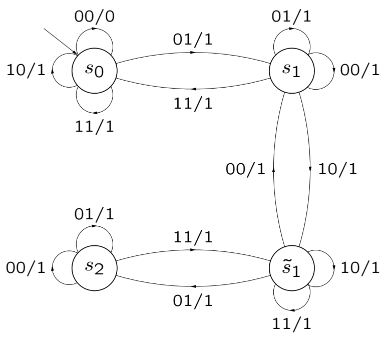

Figure 2: The lion cage.