

# Spring 2008 VHDL Exam Solution Set

525.446.31: VHDL/FPGA Design

Revised 3/22/08

## Question 1

This question asked for synthesizable VHDL code to model the 4 simple logic components constructed only from CSAs. The four simple logic components were:

- a) Toggle flip flop with an asynchronous reset that operates on the rising edge of the clock. The output should be assigned to t. Please only use CSAs.
- b) 100 bit shift register. The input should use shift\_in, the output should be assigned to shift\_out, and the signal shift\_reg should be used for the shift register. Please only use CSAs.
- c) An 8-bit up-counter with an active-high enable that rolls at 40 (i.e., it counts 0, 1, 2, 3, ... 39, 0, 1, 2, ...). The signal cnt\_reg should hold the counter. The signal enable should be used as the enable. Please only use CSAs.
- d) A transparent D-latch with an active-high enable. Use d\_to\_latch as the input, enable as the enable, and latch as the output. Please only use CSAs.

*Answer:*

```
entity question1 is
  Port ( clock : in std_logic;
        reset : in  STD_LOGIC;
        enable : in  STD_LOGIC;
        shift_in : in  STD_LOGIC;
        d_to_latch : in  STD_LOGIC;
        toggle_out : out std_logic;
        shift_out : out  STD_LOGIC;
        latch : out  STD_LOGIC);
end question1;

architecture Behavioral of question1 is

  signal t : STD_LOGIC;
  signal cnt_reg, nxtCnt : unsigned(7 downto 0);
  signal shift_reg : std_logic_vector(99 downto 0);

begin
  --a) toggle
  t <= '0' when reset = '1'
      else not t when rising_edge(clock);
```

```

--this is just here to prevent XST from optimizing out
toggle_out <= t;

--b) shift reg
shift_reg <= shift_reg(98 downto 0) & shift_in when rising_edge(clock);
shift_out <= shift_reg(99);

--c) counter
nxtCnt <= (others => '0') when cnt_reg = 40 else cnt_reg + 1;
cnt_reg <= nxtCnt when rising_edge(clock) and enable = '1';

--this way doesn't work
--XST (Xilinx Synthesizer results)
--ERROR:Xst:827 - Signal cnt_reg cannot be synthesized,
--bad synchronous description.
cnt_reg <= cnt_reg + 1 when
    rising_edge(clock) and enable = '1' and cnt_reg < 40
    else (others => '0') when rising_edge(clock);

--this way doesn't work
--XST (Xilinx Synthesizer results)
--ERROR:HDLParasers:808 - Mod can not have such operands
--in this context.
cnt_reg <= (cnt_reg + 1) mod 2 when rising_edge(clock);

--d) transparent d-latch
latch <= d_to_latch when enable = '1';
end Behavioral;

```

## Question 2

- a) A fellow student comes to you and asks, “I’m confused. I know that the types signed, unsigned, and std\_logic\_vector are all defined in the same manner (that is, unconstrained arrays of std\_logic). What is the difference between them?” How do you answer this student (no, “post a question to vhdforum@echelonembedded.com” is not correct!)?

*Answer:* All three are defined the same, as follows:

```

=====
-- Numeric array type definitions
=====
type STD_LOGIC_VECTOR is array (NATURAL range <>) of STD_LOGIC;

=====
-- Numeric array type definitions
=====
type UNSIGNED is array (NATURAL range <>) of STD_LOGIC;
type SIGNED is array (NATURAL range <>) of STD_LOGIC;

```

It is clear from the type definitions that the types are defined exactly the same. This makes sense, because each type simply represents the concept of a grouping of bits in hardware. What is different however, is how we expect these bits to be interpreted. The difference therefore, is what functions are overloaded for these packages and how they operate.

For example, the “+” operator is not overloaded at all for STD\_LOGIC\_VECTOR because we don’t know what it means to add bits (should it be an and? should it be a signed add? should it be an unsigned add?). It is of course defined for the UNSIGNED and SIGNED types. For example, the following is all of overloaded versions of the “+” operator:

```

-----
function "+" (L,R: UNSIGNED ) return UNSIGNED;
function "+" ( L,R: SIGNED) return SIGNED;
function "+" ( L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "+" ( L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "+" ( L: INTEGER; R: SIGNED) return SIGNED;
function "+" ( L: SIGNED; R: INTEGER) return SIGNED;
-----

```

Also, as expected, functions behave differently based on the type that it is defined for. For example, the “resize” function sign-extends SIGNED types, and does not sign-extend UNSIGNED types.

- b) After hearing your answer, the student seems to be clear on the differences, but then asks: "I have been trying to get VHDL to directly assign the value from a signal which is signed to a std\_logic\_vector but it won't! How do I do that? I don't see any type conversion function in ieee.numeric\_std which handles that specific case!" Explain to the student how to do this, and also why it is required in VHDL.

*Answer:* VHDL is a strongly typed language and thus does not allow an assignment from one type to another. The student would just need to use the built-in conversion functions for closely related types. As an example:

```

signal a : std_logic_vector(7 downto 0);
signal b : unsigned(7 downto 0);
...
a <= std_logic_vector(b);
...
b <= unsigned(a);

```

### Question 3

Why does the VHDL language provide support for resolution functions? A good answer will explain what a resolution function is and will include a circuit diagram in the description of its usefulness. The entire page dedicated to this question does NOT imply your answer should be long – keep it concise!

*Answer:* VHDL is designed to simulate digital hardware. Therefore, the language must provide means to simulate the situation where multiple signals are connected to a single node. For instance, a simple example would be a tri-state buffer connected to a trace that is also connected through a pull-up resistor to VCC. In this case, the voltage on the wire is dependent on the state of the tri-state buffer (which could be a '1', a '0', or a 'Z') and the pull-up resistor (driving an 'H'). The resolution function allows VHDL to simulate such a case, by calculating resolved value of a signal given the all of it's input drivers.

## Question 4

- a) As discussed in class, Xilinx FPGAs use Static RAM (SRAM) to realize combinational logic functions. In order to explain how SRAM can be used to generate combinational logic, show the contents of a 16 position, 1 bit wide SRAM which would be used to implement the function. Ensure you illustrate how the SRAMs address is connected.

```
my_output <= (A AND B) XOR (C);
```

*Answer:* The distributed RAM located on each Configurable Logic Block (CLB) – there are 1164 CLB's in the 500K-gate Spartan 3E on our Nexys-2 board – is used to implement combinational logic functions. Each unit of distributed RAM is a 16x1 RAM – that is, there are 16 1-bit memory locations. Since this function only has 3 inputs (A, B, and C), we will only need to use half of the RAM. The table below shows the contents of the RAM, as well as how the address lines are connected.

	Address 3 (0)	Address 2 (C)	Address 1 (B)	Address0 (A)	Output (my_output)
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	0
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	1
6	1	1	1	0	1
7	1	1	1	1	0
8 - 15	-	-	-	-	-

- b) Describe the difference between Distributed RAM and Block RAM.

*Answer:* Block RAM are large (2048 bytes) synchronous read/write RAM resources located in a few discrete columns on the FPGA. Block RAM is appropriate for applications such as large buffers and coefficient look-up tables. In addition, each Block RAM on the Spartan3E has a dedicated multiplier available, making Block RAMs appropriately placed for digital signal processing operations.

Distributed RAM are small RAMs with an asynchronous read. They are located on the CLB's spread throughout the FPGA fabric. Distributed RAM is useful for implementing small FIFOs and data buffers.

## Question 5

The following is from a reference manual for a Spartan 3 FPGA Evaluation Board describing the switches that are connected to the FPGA:

When in the UP or ON position, a switch connects the FPGA pin to VCC0, a logic High. When DOWN or in the OFF position, the switch connects the FPGA pin to ground, a logic Low. The switches typically exhibit about 2 ms of mechanical bounce

and there is no active debouncing circuitry, although such circuitry could easily be added to the FPGA design programmed on the board. A 4.7K series resistor provides nominal input protection.

- a) What problems can arise if a slider switch is used directly as the input to a state machine implemented in an FPGA?

*Answer:* There are two main problems:

- (a) Since the switch is not debounced, the switch will appear as if it has been pressed multiple times to the finite state machine (FSM). This could lead to incorrect operation of the FSM. For example, if a transition on the switch was used to indicate an “add” for a calculator FSM, the switch bouncing would appear like multiple additions rather than just one.
  - (b) Since the switch is an asynchronous input to the state machine, it is possible for the switch to place the state machine into an illegal state. This can happen because the “state” of the FSM is held by multiple registers. If the switch changes simultaneously with the FSM clock, then some of the FSM registers may see that the switch has been actioned, and some may not. The resulting state of the registers may be incorrect or illegal.
- b) For an input clock rate of 1 MHz, construct a de-bounce circuit for the vector of pushbuttons. To assist you in constructing this architecture, a component is available that a co-worker has previously written. That component, given in the VHDL entity/architecture pair below, produces a 1 clock-wide pulse every “divideby” clocks.

You will note that there is a new keyword in the component definition below: `generic`. The `generic` keyword allows parameters to be passed to the VHDL entity/architecture that are used to parametrize the generation of hardware. For example, to instantiate the `clkdivider` above with a `generic`, you would use the following VHDL. This would create a one clock wide pulse every 10 clocks.

```
my_clk : clk_divider
  generic map (divideby <= 10)
  port map (clk, reset, mypulse);
```

*Answer:* To debounce these switches, we must sample them at a time interval longer than their bounce rate and also shorter than we expect our operator to press the buttons. Let’s sample the switches every 4 ms. To sample the switches every 4 ms, so the number of clocks in each 4 ms is  $.004s * 1e6s^{-1} = 4000$ .

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

entity toplevel is
port ( clk, reset : in std_logic;
       pushbuttons : in std_logic_vector(7 downto 0);
       pushbuttons_out : out std_logic_vector(7 downto 0)
);
end toplevel;

architecture rtl of toplevel is
  signal pushbuttons_debnc : std_logic_vector(7 downto 0);
  component clk_divider is
    generic (divideby : natural);
    port ( clk : in std_logic;
```

```

        reset : in std_logic;
        pulseout : out std_logic);
    end component;
    signal pulseout : std_logic;
begin
    my_clk : clk_divider
        generic map (divideby => 4000)
        port map (clk, reset, pulseout);

    pushbuttons_debnc <= pushbuttons when rising_edge(clk) and pulseout = '1';
    .
    .
    .
end architecture;
```

## Question 6

- a) Draw a timing diagram showing an example of this weird behavior that you might see in the simulator.

*Answer:* `bitnum` is a signal and set at the top of the process, so `bitnum` doesn't change until the process has finished executing.

- b) Explain why this behavior occurs in the simulator, and how you would fix it.

*Answer:* Add `bitnum` to the sensitivity list.

- c) How would the synthesizer deal with the original code?

*Answer:* The synthesizer would have no problems – it ignores the sensitivity list.

- d) After you have made your modification and you synthesize, you examine the synthesis results and find that 9 d-latches were created. For what signals were these latches created and why (please give a 2 or 3 sentence answer)

*Answer:* `zflag` and `bitnum` as both are not defined for all paths through the case statement.

- e) Pick one of the signals for which a latch was created, and sketch the latch, with the associated enable logic determining when the latch is transparent.

*Answer:* In VHDL (I know, not a drawing!)

```

zflag <= bbus(to_integer(abus(7 downto 5))) when alu_control = ALU_BTST;
result <= [case statement logic] when alu_control /= ALU_BTST;
```