

CSC258 Lab #7

Introduction

Learn why VHDL programmers get paid a lot of money.

Part 1: FSMs in VHDL

Open up a new project and call it Lab7. As always, make it of type VHDL, and create a new source for it called `seq_design`, of type VHDL Module. Once it's open, set its contents to the following:

```
-- import std_logic from the IEEE library
library IEEE;
use IEEE.std_logic_1164.all;
use work.all;

-----

entity seq_design is
port(  a:          in std_logic;
       clock:     in std_logic;
       reset:     in std_logic;
       x:         out std_logic
);
end seq_design;

-----

architecture FSM of seq_design is

    -- define the states of the FSM model
    type state_type is (S0, S1, S2, S3);
    signal next_state, current_state: state_type;

begin

    -- cocurrent process#1: state registers
    state_reg: process(clock, reset)
    begin

        if (reset='1') then
            current_state <= S0;
        elsif (clock'event and clock='1') then
            current_state <= next_state;
        end if;

    end process;

end process;
```

This code allows you to define an enumerated data type called `state_type`, which VHDL will translate into a flip-flop-based finite state machine for you. If you don't know what an enumerated data type is, ask the TA for the one-sentence explanation.

This architecture is going to have two processes. The first is shown above, which resets the state machine when the `reset` signal is high. The second is outlined in the code below. Add this code below the code that you've already written.

```
-- cocurrent process#2: combinational logic
comb_logic: process(current_state, a)
begin

    -- use case statement to perform the
    -- state transistion

    case current_state is

        when S0 => x <= '0';
            if a='0' then
                next_state <= S0;
            elsif a ='1' then
                next_state <= S1;
            end if;

        when others =>
            x <= 'X';
            next_state <= S0;

    end case;

end process;

end FSM;
```

There are a few things to note here. First, there is a case statement. It evaluates the value of `current_state`, and performs the block that corresponds to `current_state`'s value. Second, there are two processes here. This is fine, as long as they don't interfere with each other (e.g. write different things to the `x` line, for instance). Once you get this FSM running, try to make the `state_reg` process write the output as well, and see what happens. Third, the "others" case refers to any unmatched values of `current_state`. In this case, it actively writes X values to the output. Other possible values are Z, L and H (impedence, weak 0, weak 1).

Try to simulate this circuit using ModelSim, to see how the FSM behaves.

Oops, that's not very good, is it? Well, it only handles one case. Define the remaining cases so that this FSM gives a high output whenever the input has been high for the last three clock cycles. Verify that this is working correctly (along with the reset signal), and play with the output as well, just for fun.

Part 2: Testbenching

Testbenching is the concept of automatically testing your VHDL design, instead of simulating it with input waveforms all the time. Here's an example, as applied to your FSM design:

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

entity test_fsm is
end test_fsm;

-----

architecture behav of test_fsm is

component seq_design
port(  a:          in std_logic;
       clock:     in std_logic;
       reset:     in std_logic;
       x:         out std_logic
);
end component;

begin

    U_fsm: seq_design port map(T_a, T_clock, T_reset, T_x);

    process
    begin
        T_clock <= '1';          -- clock cycle 10 ns
        wait for 5 ns;
        T_clock <= '0';
        wait for 5 ns;
    end process;

    process

        variable err_cnt: integer :=0;

    begin

        -- case 1
        T_reset <= '1';
        wait for 20 ns;
        assert (T_x='0') report "Failed Case 1" severity error;
        if (T_x/='0') then
            err_cnt:=err_cnt+1;
        end if;

        wait;

    end process;

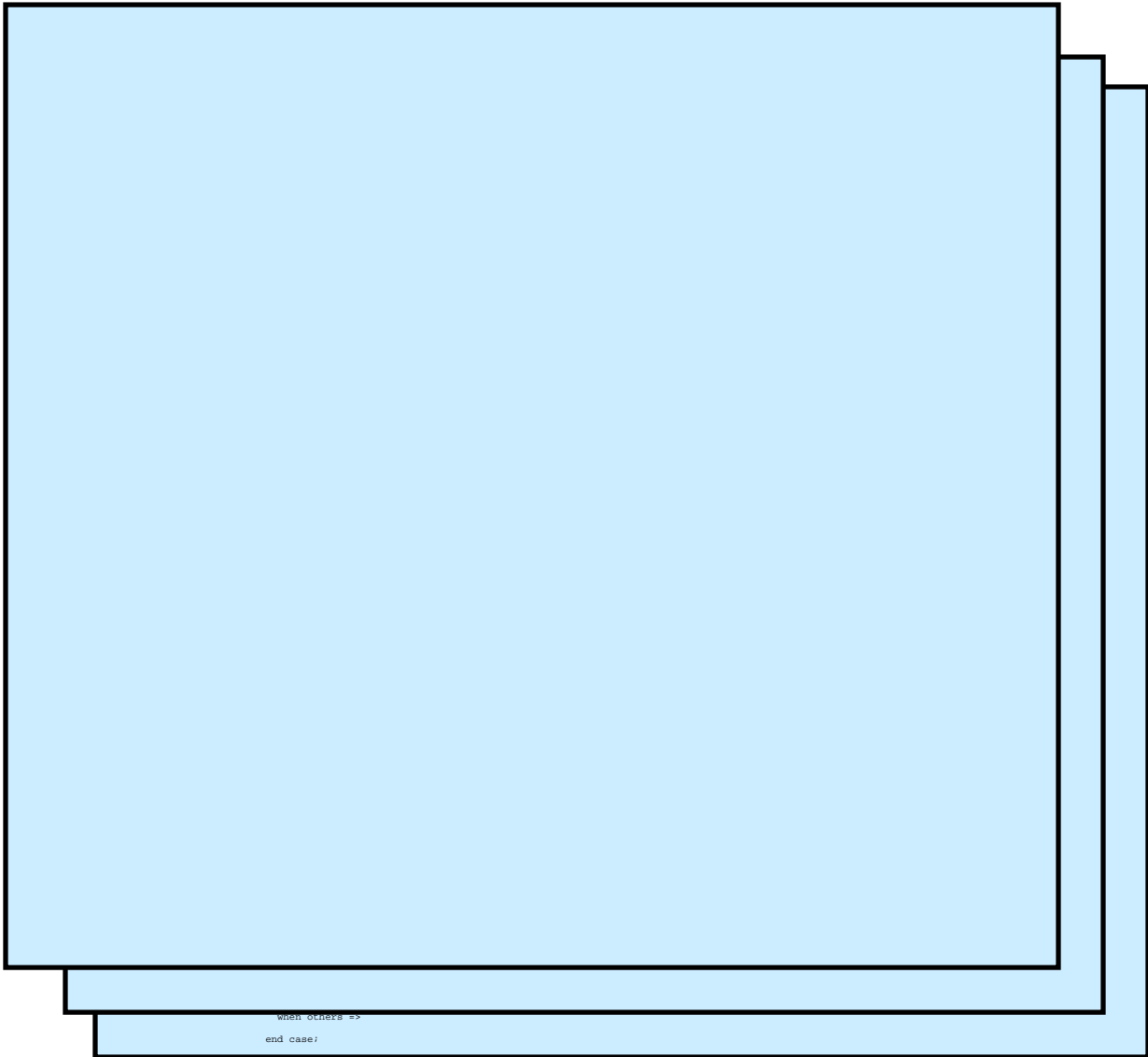
end behav;
```

The main new things to note here are the two processes (as seen earlier), how the `wait` command is used to simulate the clock in the first process, and how the `assert` and `report` statements are used to test if the output value is correct, given the input value (think JUnit). The port map keywords in the middle of the code show how the testbench signals `T_a`, `T_clock`, `T_reset` and `T_x` map over to the inputs of the component.

Make sure you understand the different components of this code. If there are any confusing parts, either ask or look up the concept that's confusing. Once you feel comfortable with it, add another test before running it.

Part 3: Food for Thought

Consider the VHDL code for a CPU instruction decoder:



That's all. You don't have to do anything. Just consider it 😊