CPEN311 Homework 3

Name: _____ ID: _____

Instructions: This homework assignment is **individual**. You can use a computer except where handwritten answers are specifically requested. You may use either SystemVerilog or VHDL for your answers, according to your preference.

Background

In class we learned about the start-finish protocol. We learned that it allowed two state machines, let's call them fsm_a and fsm_b, to "call" another state machine, let's call it fsm_c. One disadvantage of the method described in class is that it required that fsm_a and fsm_b coordinate with each other to avoid trying to "call" fsm_c at the same time.

In order to fix this, we can write a state machine that is inserted between fsm_a and fsm_b, on the one hand, and fsm_c, on the other hand. That state machine, that we will call "shared_access_to_one_state_machine", will "trick" fsm_a and fsm_b and make them think that they are the only ones talking to fsm_c. A schematic of the system is shown on the next page. Note that the clocks "clk" and "sm_clk" are all connected to the same clock, but those connections are omitted for simplicity in the diagram.

The system works as follows. The "trap_edge" modules trap the rising edge of the "start" signal, with the output in trapped_edge signifying that a "start" request has been received.

The state machine "shared_access_to_one_state_machine" monitors the start requests that are conveyed to it by the trap_edge modules. When a start request is received, "shared_access_to_one_state_machine" first waits for fsm_c to finish (if it is in the middle of completing a "call"), and (once fsm_c is free) it then calls fsm_c using the start-finish protocol (it resets the trapped start edge when doing so to make room for the next start request).

Arguments to fsm_c are conveyed either from fsm_a or fsm_b according which is currently calling fsm_c. When fsm_c finishes, results from fsm_c are registered and passed back from fsm_c to fsm_a or fsm_b according to which of fsm_a or fsm_b called fsm_c.

From the point of view of fsm_a and fsm_b, they have no idea that between them and fsm_c there is another state machine. The only way they might be able to figure that out is if they measure how much time it takes between when they assert "start" to when they get "finish"; if they do that, they may realize it is taking longer than expected.

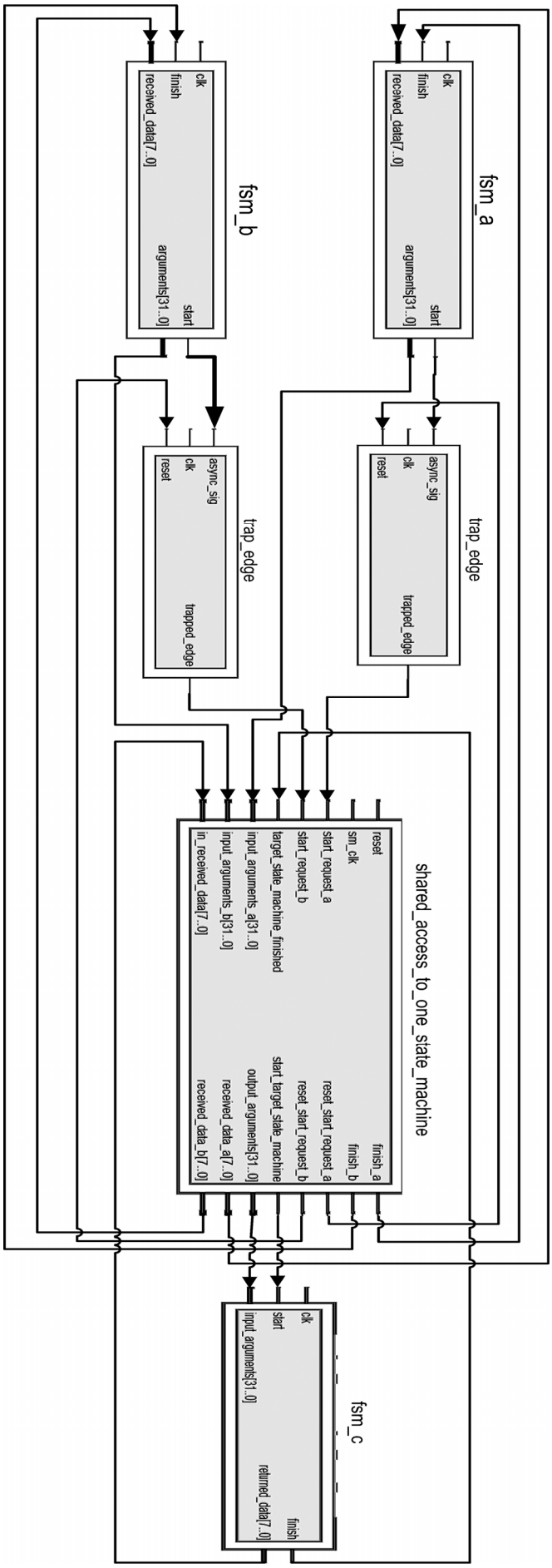

Figure 1 - Usage of the "shared_access_to_one_state_machine" FSM

The following flow chart shows the algorithm of the state machine "shared_access_to_one_state_machine".

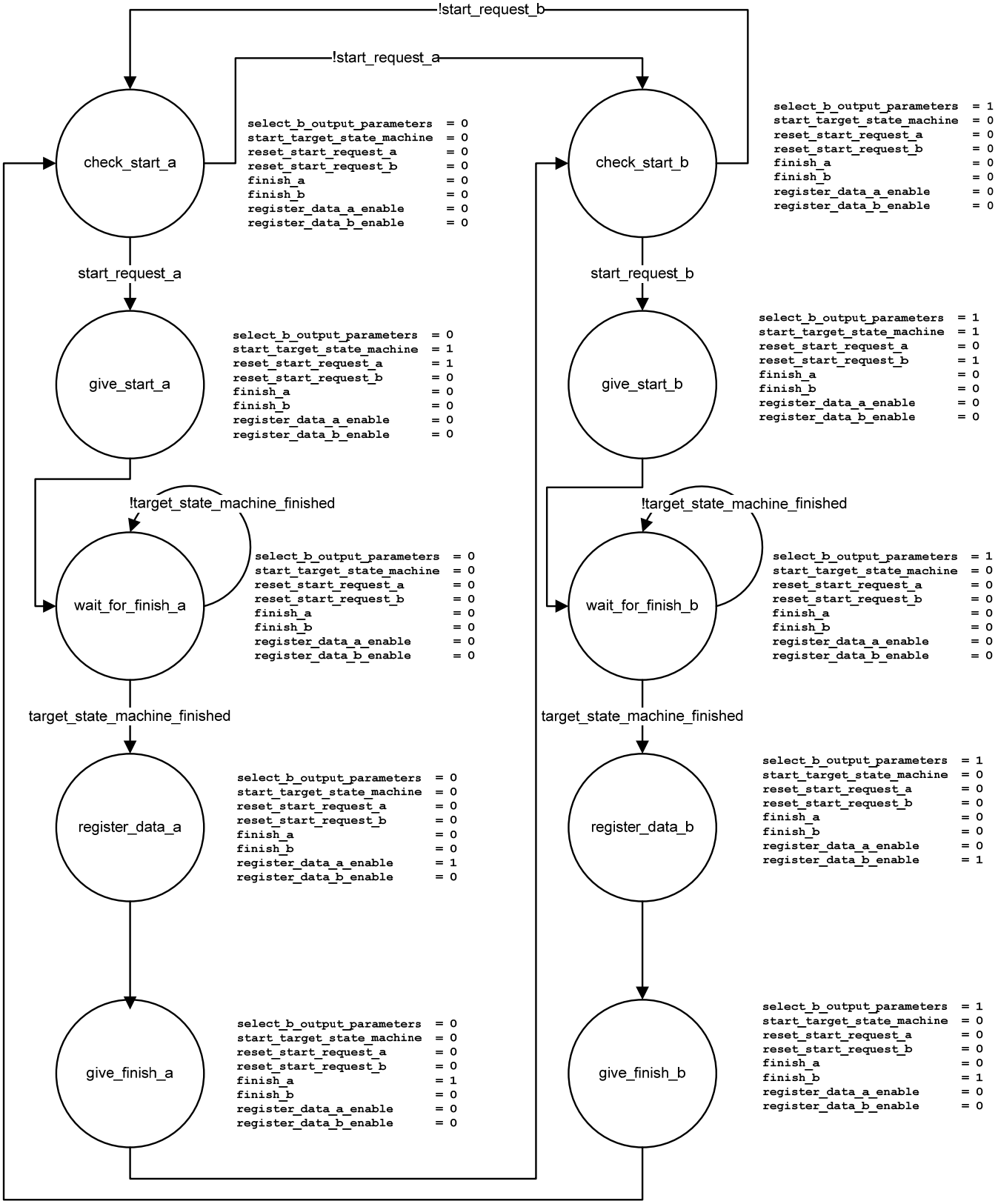

Figure 2 - Flow Diagram of shared_access_to_one_state_machine

In Figure 2, the outputs of the state machine are also the outputs of the module, except for:

select_b_output_parameters: the function of this state machine output is (in pseudo-code):

```
if (select_b_output_parameters) then
    output_arguments = input_arguments_b
else
    output_arguments = input_arguments_a
```

register_data_a_enable: this is a clock-enable signal to a register. When high, it enables registering of the input "in_received_data" into a register that goes to the output "received_data_a"

Similarly, register_data_b_enable is a clock-enable signal to a register. When high, it enables registering of the input "in_received_data" into a register that goes to the output "received_data_b" (outputs and inputs of the module are defined below)

(10%) (a) We will first design the module trap_edge. Write SystemVerilog or VHDL code to design a module trap_edge as follows:

Assume async_sig is asynchronous to the clock "clk". The module trap_edge should trap the rising edge of async_sig, properly and safely synchronize it to clk, and the result should be output in the output trapped_edge. Trapped edge should remain high until the asynchronous active-high reset "reset" is asserted.

Hand in your code and a handwritten schematic of the module trap_edge.

(40%) (b) The following is the header of the module shared_access_to_one_state_machine. Complete the module using the state machine design method taught in class. Provide an asynchronous active-high reset input to the state machine named "reset". Of course, Make sure the state machine is glitch-free.

```
module shared_access_to_one_state_machine
#(
 parameter N = 32,
 parameter M = 8
)
(
 output reg [(N-1):0] output_arguments,
 output start_target_state_machine,
 input target_state_machine_finished,
 input sm_clk,
 input  logic start_request_a,
 input  logic start_request_b,
 output logic finish_a,
 output logic finish_b,
 output logic reset_start_request_a,
```

```
 output logic reset_start_request_b,
 input [(N-1):0] input_arguments_a,
 input [(N-1):0] input_arguments_b,
 output  reg [(M-1):0] received_data_a,
 output  reg [(M-1):0] received_data_b,
 input reset,
 input [M-1:0] in_received_data
);
```

(c) (30%) Draw a schematic, using handwriting, of the module shared_access_to_one_state_machine. For clarity, use state names (as defined in the flowchart) in your schematic (i.e. do not write the numerical values of the state encoding, write the state name instead)

(d) (20%) Simulate the module shared_access_to_one_state_machine to prove that indeed it does what it is supposed to do. Hand in annotated simulation results, where your annotations on the simulation waveforms clearly show that the state machine shared_access_to_one_state_machine is working properly and that you understand what the simulation waveforms show.

(e) (20%) Bonus: Using parameters, "generate" statements, "for" statements, and possibly multidimensional arrays, write an advanced version of shared_access_to_one_state_machine that will allow for any number of state machines (not just two) to call a target fsm without coordinating between them. The number of calling state machines should be given as a parameter. Hand in:

1. The code of your state machine

2. An annotated simulation that proves that your state machine works.