Queen's University
Department of Electrical and Computer Engineering
ELEC 271 Digital Systems
Fall 2019

Lab 4:
Finite-State Machines

Objectives

This laboratory activity for *ELEC 271 Digital Systems* provides the opportunity for students to:

- acquire experience in applying the methodical design procedure for finite-state machines,
- show equivalence for gate-level and high-level VHDL descriptions for state machines,
- verify functionality in hardware and also through testbench simulation, and
- use code-converter logic to generate output on a seven-segment display of the DE0.

This activity involves the use of the worksheet included in the description. Each student must have a paper copy of the worksheet ready at the beginning of the laboratory session, and each student must complete it (with reasonable neatness) during the session, *otherwise no credit will be granted.*

Preparation BEFORE Your Scheduled Laboratory Session

- *You will individually prepare VHDL files and have them ready on your network storage or USB drive at the start of your scheduled laboratory session for use during the in-lab procedure. Each student will also have a separate worksheet ready at the start of the laboratory session.*

Preparation for Part 1

- In a file entitled `lab4.vhd`, prepare a template file as shown below.

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

entity lab4 is
  port (
        clk, reset_n : in  std_logic;
        ..., ...     : in  std_logic;    -- primary inputs to both
        ..., ...     : out std_logic;    -- gate-level fsm primary outputs
        ..., ...     : out std_logic;    -- gate-level fsm state-bit outputs
        ..., ...     : out std_logic     -- high-level fsm primary outputs
       );
end entity;

architecture combined of lab4 is

-- for gate-level specification:

-- define internal signals q0, q1, ... for individual state flip-flop outputs
signal ... : ...;

-- define internal signals d0, d1, ... for individual state flip-flop inputs
signal ... : ...;

-- for high-level specification:

-- define state type and signal for single-process high-level specification
type ... is (...);
signal ... : ...;

begin

-- for gate-level specification:

-- define combined process for behavior of individual state flip-flops
the_state_dffs: process (clk, reset_n)
begin
  .
  .
  .
end process;

-- signal assignment statements for next-state d0, d1, ... functions
... <= ...;

-- signal assignment statements for gate-level fsm primary output functions
... <= ...;

-- signal assignment statements for gate-level fsm state-bit outputs
... <= ...;
```

(continued on next page)

- The continuation of `lab4.vhd` is below. The contents pages 2 and 3 should be in a *single* file.

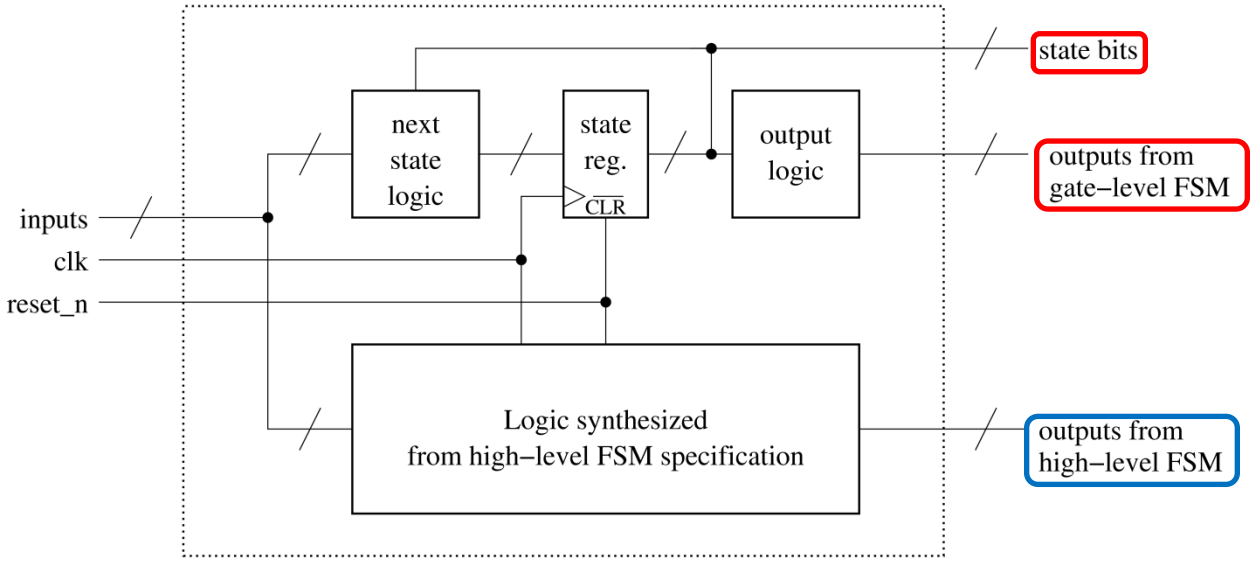
```
-- for high-level specification:

-- define the process that implements the state transitions on clock edges
the_fsm : process (clk, reset_n)
begin
   .
   .
   .
end process;

-- use signal assignment statements with when...else... syntax
-- to specify the behavior of high-level fsm outputs
... <= ...;

end architecture;
```

- The system diagram shown below reflects the intended implementation for the VHDL description.



You will be given the specifications for a Moore-type finite-state machine in the form of a detailed state diagram with input conditions for state transitions and output settings for each state.

Based on the number of states, the number of flip-flips required for the state machine will be two.

You will implement a VHDL process to describe the behavior of the state-register flip-flops.

You will derive optimized logic expressions for the next-state logic and output logic, and then you will implement gate-level signal assignment statements for those expressions.

Finally, you will use high-level VHDL syntax to describe behavior for an equivalent state machine, using an enumerated state type, a process for state transitions, and output signal assignments.

Use the suffix `_gl` for the **g**ate-**l**evel primary outputs, and `_hl` for the **h**igh-**l**evel primary outputs.

The state bit output ports from the gate-level version can be called `q1_out` and `q0_out`.

Because *both* versions of the state machine implementation have the *same* primary inputs, the primary outputs across multiple cycles should be identical, if the systems are specified correctly.

Part 1: Gate-Level and High-Level Specification of Finite-State Machine

- On your USB drive, create a folder *<USB drive letter>*`:\ELEC271\LAB4`.
- Place your template file `lab4.vhd` in the above folder.
- Start Altera Quartus II, and create a new project called `lab4` in the folder you created above.
- In the New Project Wizard dialog box, when you reach the Add Files step, use the "…" button to select your `lab4.vhd` file and be certain to click on the Add button to make it appear in the list.
- In the device selection step, choose *Cyclone III*, the *FBGA* package, *484* for the pin count, *6* for the speed grade, and then click on *EP3C16F484C6* to highlight that choice in blue. Click *Next*.
- Click *Next* to skip the EDA Tool portion, then click *Finish* to close the New Project Wizard.
- From the main menubar, select *Assignments → Device…* and then *Device and Pin Options…*
- For *Unused Pins,* change the option for 'Reserve all unused pins' to *As input tri-stated.*
- For *Voltage*, change the 'Default I/O standard' to *3.3-V LVTTL.*
- Select *OK* to close the Device/Pin Options dialog box and *OK* again to close the Device box.
- *On your worksheet*, draw the state diagram given to you at the start of the laboratory session.
- *On your worksheet*, fill in the combined state table, making appropriate choices for the assignment of state bits. *Double-check your work*.
- *On your worksheet*, derive the optimized expressions for the next-state logic. *Use appropriate care as you pursue your derivation in order to reduce the probability of making an error.*
- *On your worksheet*, derive the optimized expressions for the output logic *using appropriate care*.
- From the main menubar, select *File → Open...* and select your `lab4.vhd` file for editing.
- Ensure that you have a proper process to describe the behavior of the state bit flip-flops.
- Use your derived expressions to complete the ***g**ate-**l**evel* signal assignment statements for the next-state logic and the output logic. Use the suffix `_gl` for the related entity outputs. In addition, use `q1_out` and `q0_out` as entity outputs to allow the current state bit values to be visible.
- Using the given state diagram, complete the ***h**igh-**l**evel* VHDL description for the state machine in a process that uses `case` … `end case` syntax and `if` statements, as well as signal assignments for the outputs using `when` ... `else` syntax. Use the suffix `_hl` for the related entity outputs.
- From the main menubar, select *File → Save* to save the modified file.
- Select *Processing → Start Compilation* from the main menubar in order to synthesize the circuit.
- After synthesis, select *Assignments → Pin Planner*. Using the Terasic DE0 User Manual, make the pin assignments as indicated below.

  `reset_n`: BUTTON0 `clk`: BUTTON1 `q1_out`: LEDG9 `q0_out`: LEDG8

  …`_gl` outputs: LEDG5 and LEDG4 …`_hl` outputs: LEDG1 and LEDG0

  common primary inputs to both state machines: SW1 and SW0

- Close Pin Planner. Select *Processing → Start Compilation* **again** to resynthesize with new pins.
- Plug the USB cable into the DE0 board. Turn on the power. Select *Tools → Programmer.* Ensure "USB-Blaster" appears beside *Hardware Setup*. Press the *Start* button to program the FPGA chip.
- Use the clock/reset pushbuttons and the switches for the primary inputs to verify that both versions of the state machine implementation produce identical outputs. Also verify that state bit outputs from the gate-level implementation match the expected state bit values in your combined table. If the behavior is incorrect, review your derivations and/or your code to make corrections.
- *For credit, demonstrate your working systems on the DE0 board when you complete this part.*

Part 2: Code-Converter Logic for Hex-Display Output of FSM Current State

- Part 2 involves the extension of the overall system in the `lab4.vhd` file to use the right-most hex display to display a symbolic representation of the current state, in addition to the raw bit pattern for the state bit outputs that appear on LEDG9 and LEDG8.
- The letters representing the states in the FSM specification given to you have been chosen on the basis that they can be properly represented in the available seven segments on the hex display.
- In some cases, a lowercase representation of the state letter is more appropriate to display for clarity. Suggestions for such cases are provided in the specifications given to you.
- Having demonstrated the correct operation of the two FSM implementations, the activity in Part 2 will be applied to the same project as in Part 1.
- Open the the `lab4.vhd` file in the Quartus editor.
- In the entity definition, add a new 7-bit output port at the end of the port list called `hex_out`.
- In the architecture *before* the `begin`, define a new internal 7-bit signal called `hex`.
- In the architecture body, add a new signal assignment statement that will choose different result vectors (see below) for `hex` using *when…else…* syntax to check the values of `q1` and `q0`.
- In the architecture body, add a new signal assignment statement `hex_out <= not hex;` (This statement is necessary because each LED segment turns if the actual pin output is '0'.)
- To prepare the specific output bit patterns, consult the details provided in Section 4.? of the Terasic DE0 User Manual. Figure 4.9 provides the pin assignments for the seven segments in the right-most hex display (the same information is in the first part of Table 4.4). Figure 4.10 provides a simplified diagram that applies more generally to each of the hex displays. When using a logic bit vector of the form `(6 downto 0)` to connect to a 7-segment (hex) display, the top LED segment labelled '0' in Figure 4-10 would correspond to bit 0 of the vector, segment '1' would correspond to bit 1 of the vector, and so on.
- *On your worksheet*, prepare the four result bit patterns corresponding to each state identifier.
- Insert those bit patterns in the signal assignment statement for the `hex` signal. Remember that to ensure purely combinational logic, the final *else* should not have a *when*.
- From the main menubar, select *File* → *Save* to save the modified `lab4.vhd` file.
- Select *Processing* → *Start Compilation* from the main menubar in order to synthesize the circuit for syntax checking and population of the pin planner with elements of the new port. If necessary, make corrections to the VHDL file to address any syntax errors and recompile.
- After synthesis, select *Assignments* → *Pin Planner*. Using the information in Figure 4.9 (or the first part of Table 4.4) in the Terasic DE0 User Manual, make the pin assignments for the seven bits of the hex output. Double-check the pin assignments. Close the Pin Planner.
- Select *Processing* → *Start Compilation* **again** to resynthesize with new pins.
- Make certain the DE0 board is connected and powered. Select *Tools* → *Programmer.* Ensure "USB-Blaster" appears beside *Hardware Setup*. Press the *Start* button to program the FPGA chip.
- With an automatic reset when the chip is programmed, you should see the same initial outputs on the green LEDs, but you should also see the corresponding symbolic representation of the initial state of the FSM on the right-most hex display.
- Use the clock pushbutton and the switches for the primary inputs to verify that both versions of the state machine implementation still produce identical outputs. Verify that state bit outputs from the gate-level implementation on the green LEDs match the expected state bit values in your combined table, and also that the hex-display output in symbolic form also matches the expected indication of the current state. For incorrect behavior, make appropriate corrections and try again.
- *For credit, demonstrate your hex display symbols properly matching the binary state bit outputs.*

Part 3: Use of ModelSim to Simulate FSM Operation and Probe Internal Signals

- The previous parts have verified correct operation in hardware. In typical digital design, hardware testing is normally preceded by simulation (of individual sub-circuits, then high-level circuits that integrate those subcircuits). For this course, however, there is somewhat higher emphasis on using the hardware because it is available in the laboratory sitting.
- Nonetheless, a final activity to pursue for this exercise on finite-state machines is to use the more sophisticated ModelSim environment with a testbench to simulate behavior, to revisit clock/reset simulation as initially considered in Part 5 of Lab 2, and to highlight the ability of a simulation tool such as ModelSim to observe internal signals in support of debugging.
- First, to enable proper ModelSim usage, perform the three following setup actions.
    - Select *Tools* → *Options*, then *EDA Tool Options*. For the box labelled ModelSim-Altera, the path should be set properly. See the description under the Lab 1 on the course Webpage.
    - Select *Assignments* → *Settings …*, then *EDA Tool Settings*. For *Simulation*, the tool name should be ModelSim-Altera, and the *Format* (i.e., language) should be VHDL.
    - *Recompile* the project in Quartus to properly generate information needed for ModelSim
- Now, select *File* → *New …*, then select *VHDL File*. Before typing in the text editor, select *File* → *Save As …*, and type `tb_lab4.vhd` as the filename. **UNCHECK** the box for *Add file to current project*. The testbench file is only for ModelSim. It is not part of the Quartus project. Quartus is being used for editing convenience (e.g., to copy/paste text from `lab4.vhd` to `tb_lab4.vhd`).
- Fill in the contents of `tb_lab4.vhd` guided by previous experience in Lab1 and Lab 2.
- Because the system to be simulated involves sequential circuitry, a reset process and a clock process are required, as pursued in Part 5 of Lab 2.
- For the actual testing of the system, the third `test_driver` process can be (initially) as simple as setting all FSM inputs to logic-0, then waiting forever.
- Once th `tb_lab4.vhd` file is complete, save it.
- Select *Tools* → *Run Simulation Tool* → *RTL Simulation*. Once ModelSim opens, refer to earlier laboratory exercise descriptions for the procedures to (a) compile the testbench file with `vcom`, (b) prepare for simulation with `vsim`, (c) create a waveform window with `add wave *`.
- *Do not run the simulation yet.*
- In previous ModelSim usage, only the entity-level input/output ports have been of interest for simulation. Even if internal signals are defined within the architecture body, the focus has been on the ports for external connection. ModelSim, however, is a powerful tool with the ability to allow designers to look within an entity. In the case of this particular exercise, a feature of particular interest within the architecture body is the state signal that uses an abstract enumerated type.
- Having used `add wave *` for the entity ports, now type `add wave the_lab4/state` (This command assumes that you named the instance of lab4 in the testbench file '`the_lab4`' and that you named the signal for reflecting the state in the architecture body '`state`'; if you used somewhat different names, adjust the parts of the command above as appropriate.).
- The command above is asking ModelSim to look inside the entity that is being tested and probe an signal internal to the architecture by adding it for waveform viewing. In this manner, a designer can debug by viewing not only the top-level port signals, but also internal signals of a sub-circuit or even a sub-sub-circuit, i.e., at any depth of a hierarchically-organized design.
- The waveform window should show that another signal has been added to the list. The 'path' to the new signal is longer, so the signal name may be not be fully visible. To make the name fully visible on the left-hand side of the waveform window, click *and hold* on the vertical dividing line to the right of the names, and drag the line further to the right so that all names are fully visible.

- Now, type `run 200ns` to initiate simulation for a number of clock cycles. Recall that it is necessary to zoom out and scroll in the waveform window so that all of the cycles are visible.
- For the `state` signal, states with the abstract single-character names should be visible.
- In each cycle, the single-character representation appearing on the `state` signal should match the gate-level binary state bits `q1_out` and `q0_out`.
- With the initially simple `test_driver` process, the state machine is certainly not being fully tested in the same manner as the manual testing of the state machine on the DE0 board using the switches and pushbuttons. Preparing a test_driver process that exhaustively tests all states and all transitions between those states would merely involve replicating the manual testing sequence on the DE0 board. For the purposes of this final part of the exercise, it is sufficient to extend the `test_driver` process just enough for the sequence to return to the initial state. Depending on the nature of the state diagram, a minimum of three transitions would likely be necessary, and possibly four transitions.
- Examine the waveform window. If the constant-logic-0 input signals are already enabling the return of the state machine to the initial state, then that is sufficient. If not, then determine what input-signal setting(s) is(are) necessary, and at what time(s) in the simulation that(those) setting(s) is(are) necessary, to cause one (or two) additional transition(s) to reach the initial state again. It is suggested that the input signal transitions be timed to coincide with the *falling* clock edge so that they appear ready well in advance of the rising clock edge where state changes occur.
- In the Quartus text editor, extend the `test_driver` process of the testbench file based on the above determination(s).
- Save the testbench file.
- Switch back to the ModelSim window.
- *Recompile* the testbench file.
- Use the `restart` command to reinitialize the simulation (recall that previously-added waveforms will be re-added to an empty waveform window)
- Use `run 200ns` to simulate again.
- Confirm that the extension of the `test_driver` process has cause the state sequence to return to the initial state (and possibly repeat through the same sequence until the end of simulated time). If the behavior is not correct, verify the input settings and the timing of those input settings, make appropriate revisions of the `test_driver` process, recompile, restart, and resimulate until correct behavior is obtained.
- *For credit, show the waveform output and the testbench code.*

Part 1: Draw state diagram below.

Combined state table

	Next State				
Curr. State sym. *q1q0*	___ = ___ sym. *d1d0*	___ = ___ sym. *d1d0*	___ = ___ sym. *d1d0*	___ = ___ sym. *d1d0*	Outputs __ __

Derivation of next-state logic (two Karnaugh maps)

Derivation of output logic (two Karnaugh maps)

Part 2: Use the technical information for the DE0 board to generate the necessary bit patterns.

state: '__' '__' '__' '__' (fill in letters)

`hex`: _____ _____ _____ _____ (fill in binary values)