DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
CPEN 311 – Digital Systems Design
2015/2016 Term 1

Lab 1: State Machines

Work week: September 14-18, 2015
Marking week: September 21-25, 2015

Work in Pairs: Choose your partners during the lab period Sept 14-18th. If you do not have a partner, your TA can help pair people up during your lab period Sept 14-18th. Do not wait until Sept 21st to partner up.

In this lab, you will become familiar with the software and hardware we will be using in the labs, and use the hardware and software to implement a state machine. You will also be re-acquainted with VHDL, which most of you will have been introduced to in EECE 259 or EECE 355.

We will be using two pieces of software for most of this course: Quartus II, which is produced by Altera, and ModelSim, which is produced by Mentor Graphics. There are several versions of ModelSim available; we will be using one that has been modified by Altera for use with Quartus II. You can download these programs as described below (they are free), or use them on the departmental computers in the lab.

Like all labs in this course, this lab will span two weeks. The first week is intended to be a work-week, where you spend the three hours in the lab working on the tasks. A TA will be available to help you if you run into problems. The second week is primarily for marking (you should not expect help from the TA during the second week). You may have to do some work on your own (at home or in the lab outside of lab hours) as well. You most certainly can not finish the lab if you don't start until the marking week.

PHASE ONE: GETTING READY

TASK 1.1: The first step is to install Quartus II and the Altera version of ModelSim on your home PC or laptop, or find a place you can run the software (it will be available on most of the computers available throughout the department). Installation instructions can be found in the "Digital System Design using Altera Quartus II and ModelSim" document (called **tutorial.pdf**) located in the Lab 1 folder on Connect.

You should use Version 13.0 SP1 (*do not use Version 13.1 or later*, *since it does not support the device you will be targeting*). If you took EECE 259 or EECE 355 in 2014/2015, this is the version you have already installed. If you took the course earlier, check which version you have, and upgrade if necessary.

TASK 1.2: Work through the rest of the "Digital System Design using Altera Quartus II and ModelSim" tutorial document located in the Lab 1 folder of Connect. There are some download files for this tutorial, which you can find in Lab->Download Files->Task1.2 on the Connect site. This will re-introduce you to Quartus II and ModelSim, which are the tools we will use throughout the course. Even if you have used Quartus II and Modelsim, you might have forgotten some of the details, so it will save you a lot of time if you work through the tutorial carefully.

PHASE TWO: REVIEW OF STATE MACHINES

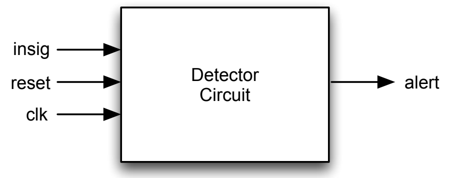
In EECE 259 or EECE 355, you learned about state machines, but may have forgotten some of the details. In this phase, you will step through an example state machine to remind yourself how state machines work.

TASK 2.1: You should start by reviewing Chapter 14 of the Dally book (available for free on-line to CPEN 311 students in 2015/2016, see textbook handout for link and password) or reviewing your EECE 259 or EECE 355 notes.

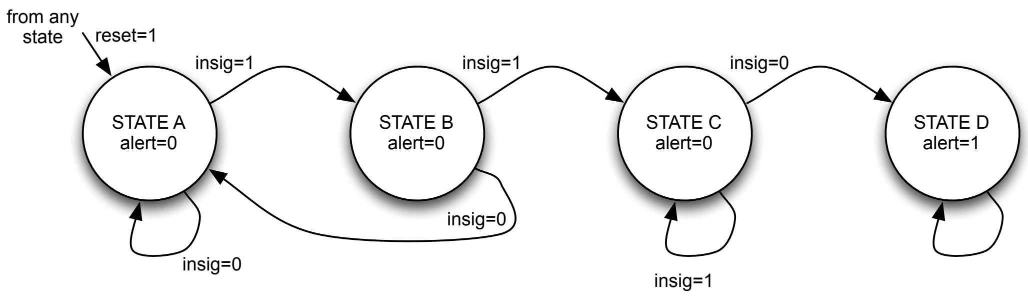
TASK 2.2: In this task, you will be led through an example state machine, just to remind you of what you learned last year, and get you ready for this lab. The circuit we are using in this section is a purposely simple; you won't use it in the rest of the lab.

From last year, you might recall that a state machine is a special type of sequential circuit, in which the output(s) of the circuit depends on the *state* that the machine is in; the state depends on the history of previous states and inputs. State machines are often used as controllers for larger systems – if you took EECE 259 in 2014/2015, you would have implemented a state machine that controls a simple RISC processor in Lab 6.

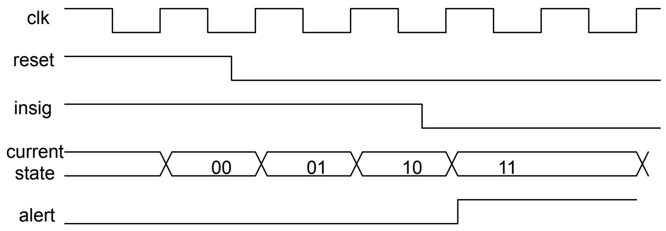
Example State Machine: As an example, in this section we will study a simple state machine that might be used within a network switch implementing a deep packet inspection algorithm. Deep packet inspection is used to "look inside" an incoming stream of data to identify packet headers, virus, or other known patterns. We will simplify the situation significantly, and only look for the pattern "110" in an incoming data stream. A block diagram of the circuit is below. The circuit has three one-bit inputs (***clk, reset,*** and ***insig***) and an output (***alert***). Each cycle, one bit of the input stream arrives at the ***insig*** input. If the machine detects a 1, 1, and 0 in three consecutive cycles, it raises ***alert***; at all other times, ***alert*** is low. Once ***alert*** goes high, it remains high until the machine is reset.



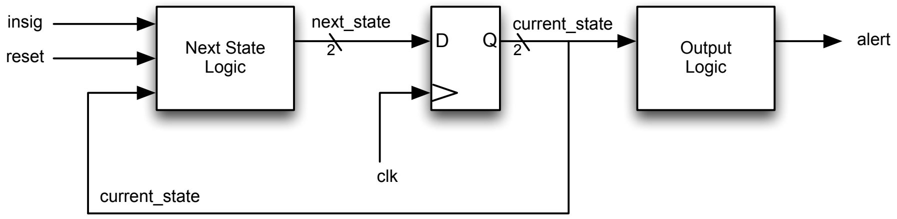
A state diagram that illustrates the state transitions and detailed operation of this machine is shown below. Each circle represents a state that the machine might be in (A, B, C, or D). As the diagram shows, when the machine is in States A, B, and C, the alert output is 0; when the machine is in state D, the alert output is 1. Each transition is labeled with the input value (on the input ***insig***) that causes that transition. You may have seen state diagrams in which the output values are associated with transitions rather than states (so each transition is labeled with input/output); we will see examples of this type of state machine later in the course.



The operation of this state machine is as follows. The machine starts in State A. If you recall from EECE 259, transitions between states occur only on the rising clock edge (when the ***clk*** signal transitions from 0 to 1). At each rising clock edge, the machine will transition to a new state (or possibly stay in the same state) depending on the value of the current state and the input signal(s). As you can see in the diagram, while in State A, if a '1' is received on input ***insig***, the machine goes to state B; otherwise, it waits in state A. If the machine is in state B and another '1' is received, the machine goes to state C; otherwise, the machine goes back to state A. If in state C, the machine receives a '0', the correct three-bit code has been received, so the machine goes to state D; otherwise, the machine remains in state C (can you see why it does not go to state A? If not, ask your T.A.). In state D, the ***alert*** output is raised, and the machine remains in state D until ***reset*** goes high (at which time it goes back to state A). Whenever ***reset*** goes high (regardless of the current state), the system goes back to the start and the process repeats. A timing diagram is shown below. Examine these diagrams until you are sure you understand the behaviour of the state machine.



Circuit-Level Implementation: In EECE 259 or EECE 355, you learned the general structure of a digital circuit that implements a state machine. As a reminder, consider the following diagram.



The circuit in the above diagram can be understood as follows. The heart of the circuit is the two-bit wide register in the middle of the diagram. Since this state machine has four states, we need two state bits to encode the state. Thus, the register that holds the states is two bits wide. On each rising clock edge, the input ***next_state*** is stored in the register, and appears at the internal signal ***current_state***. The ***current_state*** does not change until the next rising clock edge. Thus, you can think of this register as a memory that is updated on each rising clock edge. The value in the register indicates the current state of the machine.

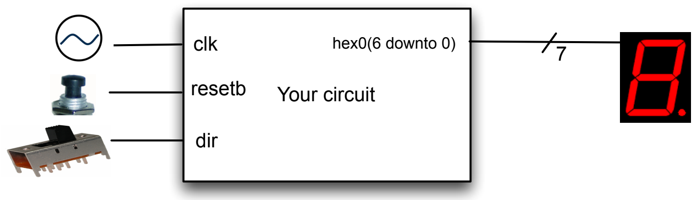
Now consider the block labeled **Output Logic**. This block contains combinational logic (gates) that determines the value of the output ***alert***. As described earlier, the value of ***alert*** depends on the current state (if the machine is in state D, alert is a 1, otherwise it is a 0). The block labeled **Next State Logic** is also combinational. The output of this block is a two bit-wide signal that encodes the next state given the 2-bit current state, the value of the input data ***insig***, and the ***reset*** signal.

Taken together, this circuit operates as follows. Suppose the circuit starts in State A; in this case, the state register encodes State A using two bits, and this value appears on the two-bit signal ***current_state***. The combinational block labeled **Output Logic** computes that the output signal alert should be 0 (since we are in state A). Meanwhile, the combinational block labeled **Next State Logic** computes the next state; this will depend on the inputs ***reset*** and ***insig***. Assuming ***reset*** is 0 and ***insig*** is 1, this block will determine that the next state should be State B, and this value will appear encoded on the two bit signal ***next_state***. At the next rising clock edge, this value is latched into the two-bit state register, and the machine is deemed to have entered state B. At this point, the **Output Logic** and **Next State logic** blocks will compute the new value of the output and the new value of the next state respectively. It is important to note that the state only gets updated at the rising edge of the clock signal.

VHDL Specification of a State Machine: One can create a VHDL specification for this circuit by considering each block individually. Download the file **detector.vhd** from the Connect site (Lab1->Download Files->Task2.2) and examine it closely. You should be able to recognize three processes: one that implements the Next State Logic, one that implements the Output Logic, and one that implements the two-bit State Register. Examine this carefully to make sure you understand how this specification works. You may need to review your notes from EECE 259 or EECE 353, or review Chapter 14 of the Dally book.

PHASE THREE: STATE MACHINE DESIGN

In this phase, you will design a simple state machine, simulate it using Modelsim, and then implement and test it on the DE2 board. The following diagram shows the inputs and outputs of your circuit.

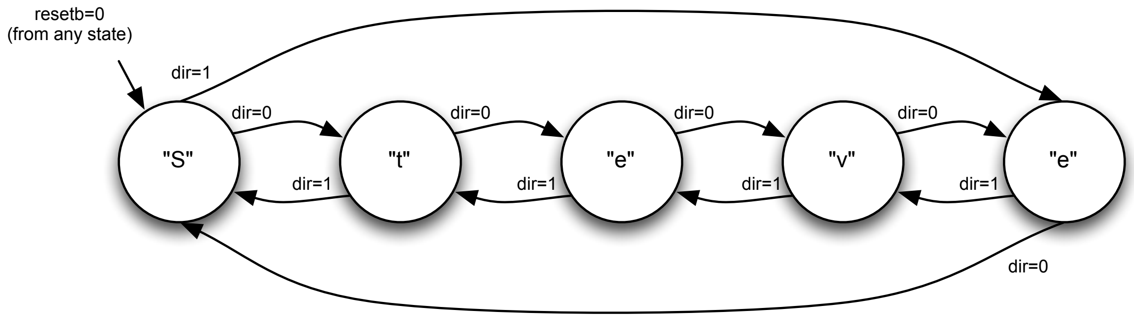


As shown in the above figure, your state machine has three inputs: a clock, an *active-low* reset signal, and a direction switch. The circuit will drive one seven-segment digit on the DE2 board. When the direction switch is in the "up" position (indicated when input ***dir*** is 0), the circuit drives the seven-segment digit with each letter in your name, one letter per clock cycle. In other words, if your name is "Steve", the circuit will drive the letter "S" for one clock cycle, then the letter "t" for the second clock cycle, the letter "e" for the third clock cycle, the letter "v" for the fourth clock cycle, and the letter "e" for the fifth clock cycle. On the sixth clock cycle, the process repeats starting with "S". If the switch is in the "down" position (indicated when the input ***dir*** is 1), the circuit steps through the characters in your name *backwards*, again one letter per clock cycle. The switch can be raised or lowered between clock cycles. So, if the machine operates with the switch in the up position for three cycles, the down position for four cycles, and then the up position for three more cycles, it would display the characters (one per cycle) S, t, e, t, S, e, v, e, S, t.

You can see what this should look like by downloading **Lab1->Movies->phase3.mov** from Connect.

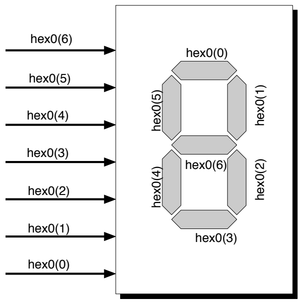
The circuit has an active-low reset signal. When this signal is true, the state machine goes to the first state, regardless of what state it is currently in. Recall that an "active low" signal is one where the "true" value of the signal is a 0 and the "false" value is a 1. In this case, the machine should be reset when **resetb** is 0 (this is different than the example from Phase 2 where the machine was reset when reset is 1; that type of signal is called an "active high" signal). We use active low signals here since the signals connected to switches on our board are 0 when pressed, and 1 otherwise.

A state diagram that describes this behaviour might look something like the following (if your name is "Steve"):



Note that if the reset signal goes to 0, the circuit goes back to the first letter in your name at the next rising clock edge.

The output of your circuit is a seven-bit wide bus called ***hex0(6 downto 0).*** This bus will be connected to one of the seven-segment digits on your DE2 board. Each wire in this bus corresponds to a specific segment in the seven-segment digit, as shown in the following diagram. Each segment is turned on if the corresponding wire is a 0, and turned off if the corresponding wire is a 1. So, for example, to display the letter S, your circuit would drive ***hex0(0), hex0(2), hex0(3), hex0(5),*** and ***hex0(6)*** with a 0 (to turn them on) and ***hex0(1)*** and ***hex0(4)*** with a 1 (to turn them off).

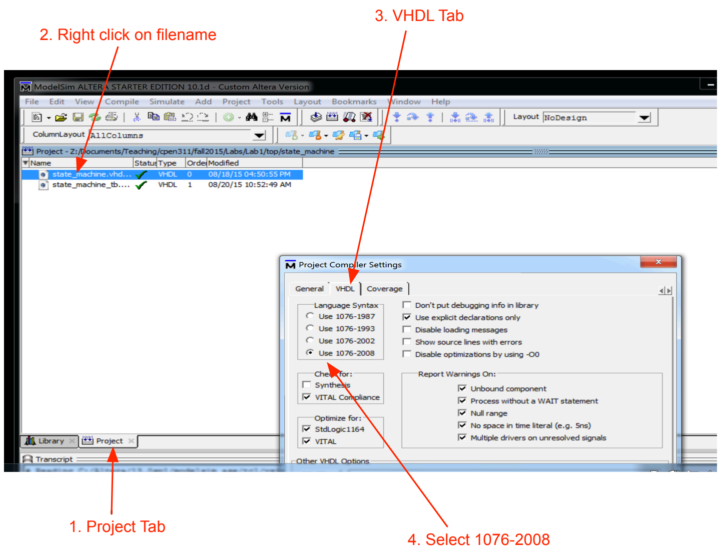


Task 3.1: Write a VHDL specification of this state machine. You should use the first name of one of your two group members (do not use "Steve" unless that is actually your first name). If the name has more than 8 letters, just use the first 8 letters. If the name has fewer than 4 letters, repeat the name twice. Note that some letters are difficult to display on a seven-segment display (not clear how to display an "m" for example). In that case, do the best you can, or use an "all-on" pattern to represent letters that you can't display.

You should start with the skeleton file "**state_machine.vhd**" which is on the Connect site (Lab1->Download Files->Task 3.1). You may find the code you examined in Phase 2 to be a good reference.

Task 3.2: Simulate the design using Modelsim. The tutorial you went through in Phase 1 will help you do this. As in Phase 1, you will compile your design (which you created in Task 3.1) along with a *testbench* file. As a reminder, a testbench is non-synthesizable code that will *not* be synthesized into hardware. A testbench is used during simulation only to drive waveform patterns on input signals and (optionally) monitor output signals to detect incorrect behaviour. To save you time, you can download a testbench "**state_machine_tb.vhd**" from the Connect site (Lab1->Download Files->Task3.2). Examine **state_machine_tb.vhd** carefully so you understand how it works. In this case, we are using an extremely simple testbench that supplies a clock waveform to the clock input, a short pulse on reset at the start of the simulation, and a pattern on the **dir** input (so you can observe the state machine going both forward and backwards). As you test your design, you may want to enhance your testbench to better test your design. As in the tutorial in Phase 1, your Modelsim project will contain two files: your design itself and the testbench file.

In this course, we will primarily be using VHDL 2008 (which has a few new features compared to older versions of VHDL). Modelsim defaults to an older version of VHDL. If you want to use any VHDL 2008 features (such as the ***all*** keyword inside the sensitivity list of a combinational block) you need to turn on VHDL 2008 in Modelsim. To do this, before compiling your files, in the project tab, right click on the filename, choose the VHDL tab, and select "Use 1076-2008" as shown in this diagram.



Project Compiler Settings

General | VHDL | Coverage

Language Syntax
- ○ Use 1076-1987
- ○ Use 1076-1993
- ○ Use 1076-2002
- ● Use 1076-2008

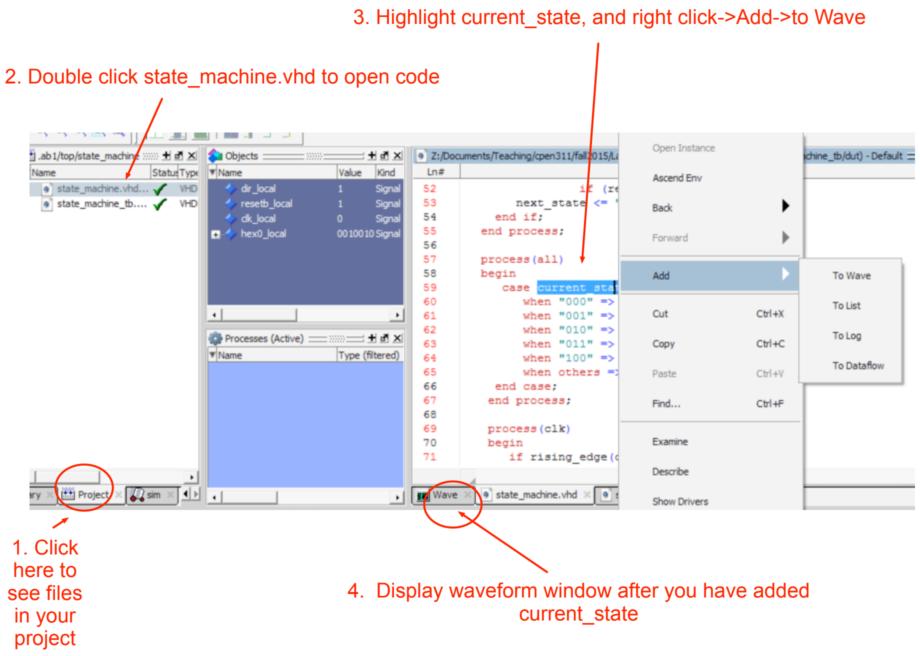
- ☐ Don't put debugging info in library
- ☑ Use explicit declarations only
- ☐ Disable loading messages
- ☐ Show source lines with errors
- ☐ Disable optimizations by using -O0

Check for:
- ☐ Synthesis
- ☑ VITAL Compliance

Optimize for:
- ☑ StdLogic1164
- ☑ VITAL

Report Warnings On:
- ☑ Unbound component
- ☑ Process without a WAIT statement
- ☑ Null range
- ☑ No space in time literal (e.g. 5ns)
- ☑ Multiple drivers on unresolved signals

In the tutorial from Task 1, you learned how to add input signals and output signals to the waveform. Follow those instructions to add the three inputs and one output to the waveform window. In addition, simulation lets you do something else: you can easily add internal signals (that are not connected to pins of your chip) to the waveform window. In this case, debugging might be made easier if you could see the value of **current_state** in the waveform window. To do this, after you have added your inputs and outputs to the waveform, click on the Project tab in the left-most panel. Double click on **state_machine.vhd**. The code for this file will open in a window to the right. Find **current_state** in your code, highlight it, and right-click -> Add -> ToWave. Finally, go back to your waveform window, and run the simulation as normal. This procedure allows you to add any internal signal to your waveform, which may be incredibly useful during debugging. Learn to do it now, because being good at debugging will save you a lot of time in later labs.

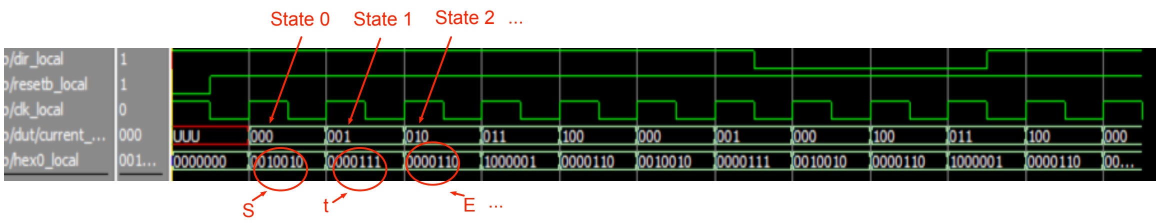

3. Highlight current_state, and right click->Add->to Wave
2. Double click state_machine.vhd to open code
1. Click here to see files in your project
4. Display waveform window after you have added current_state
Open Instance
Ascend Env
Back
Forward
Add
To Wave
To List
To Log
To Dataflow
Cut
Copy
Paste
Find...
Examine
Describe
Show Drivers

```
                        if (re
            next_state <= '
        end if;
    end process;

    process(all)
    begin
        case current_stat
            when "000" =>
            when "001" =>
            when "010" =>
            when "011" =>
            when "100" =>
            when others =>
        end case;
    end process;

    process(clk)
    begin
        if rising_edge(c
```

Your waveforms should look something like the following:



In this waveform you can examine the current state and the output signal. If your output values are correct, there is no debugging to do (move on to the next Task). If not, you can look at the **current_state** waveform. If the state transitions appear correct, but the output is wrong, it suggests the root cause of your error is in the output logic process. If the state transitions do not appear to be correct, it suggests the error is in the next state logic or the state register processes. This is the key to debugging: looking at internal signals to help narrow down the location of the bug. In this case, you could also look at the **next_state** signal using the same process as above.

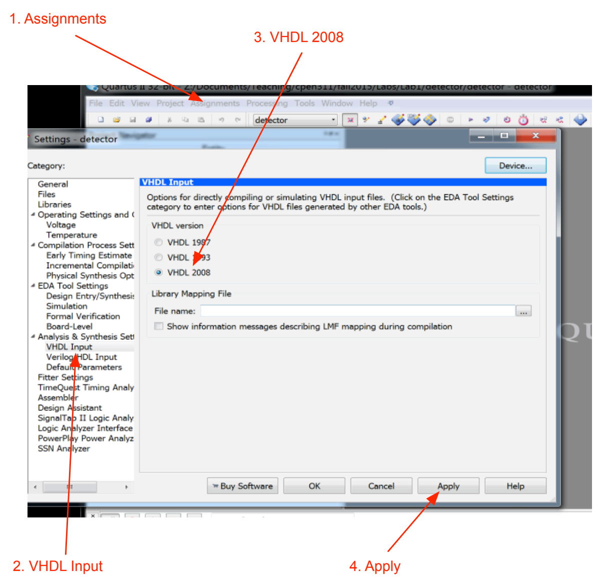
Do not move on until (a) you are sure your design is correct, and (b) you are confident that you are able to add signals to a waveform to help with debugging. This skill will save A LOT of time later in the course.

Don't be disappointed if your design doesn't work the first time. I don't know of any real designers (even the best designers at the best companies) who would expect their designs to work the first time. Debugging is an integral part of design, and is what most engineers spend the most time doing. This is true for both hardware and software.

Task 3.3: Download your Design

Now you will download your design to the DE2 board. Create a new Quartus II project called **phase3**. Add the **state_machine.vhd** file you created in Task 3.1 (and possibly debugged in Task 3.2). Do *not* add the testbench file from Task 3.2; testbenches are only used during simulation of the design, they do not represent actual hardware.

As in Modelsim, Quartus II defaults to an older version of VHDL. If you want to use VHDL 2008 features (which you probably do in this course), you can turn on the VHDL 2008 flag as follows. After creating the project, choose Assignments->Settings. In the left-hand Category window, open "Analysis and Synthesis Settings", and select VHDL input. In the dialog box, turn on VHDL 2008, and click Apply as shown below. Remember to do this for all your projects that use VHDL 2008 features:



VHDL Input

Options for directly compiling or simulating VHDL input files. (Click on the EDA Tool Settings category to enter options for VHDL files generated by other EDA tools.)

VHDL version: VHDL 1987 [✓ VHDL 2008] VHDL 1993

Library Mapping File

File name:

Show information messages describing LMF mapping during compilation

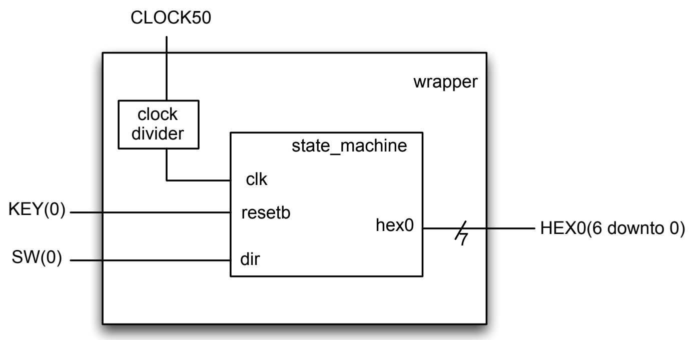
On the DE2 board, the hex digits, the lights, the switches, sliders, and other I/O devices are all hardwired to specific pins on the FPGA using metal traces. To understand this mapping, download the **DE2_pin_assignments_UBC.csv** file from Connect, and open it. You will see a list of pin names, each one associated with a pin number. For example, from the file, you can see that switch 0 is connected to Pin N25 of the FPGA. Scrolling down, you can see that the seven lines that drive the Hex0 digit are connected using Pins AF10, AB12, AC12, AD11, etc. (the nomenclature for these pins doesn't actually matter, so don't worry about the exact meaning of AF10, for example … just understand that this refers to one specific pin on the FPGA chip). When you compile your design, it is necessary to ensure that the input and output pins of your circuit are wired to the correct pins of the FPGA. For example, if you have an input that you want to connect to Switch 0, you need to make sure that input is wired to pin N25. To ensure this happens, you need to do the following:

1. Remember to download **DE2_pin_assignments_UBC.csv** and import the assignments by selecting Assignments->Import Assignments *before* you compile your design.
2. In your design, name your input and output pins using the names in the csv file (so for example, if you want to connect a signal to switch 0, call the signal SW(0) … note that round brackets are used to index into an array rather than square brackets as in the .csv file.

These are the same steps you would have used last year and are described in the tutorial from Phase 1.

Step 2 above requires you to rename your I/O signals. For example, in Task 3.1, you had an input called **dir**, but the above description suggests you should rename this input to SW(0). Although it is possible to go into your **state_machine.vhd** file and change the signal names, a more elegant way is to create a hierarchical design, in which your **state_machine.vhd** file is instantiated in a wrapper (which will be called **phase3.vhd**). The names of the wrapper I/O pins will be the names from the .csv file. This means that the pin names of **state_machine.vhd** do not need to change; you can use the file from Task 3.1 directly.

To better understand the function of the wrapper file, download **phase3.vhd** from the Connect site (Lab 1->Download Files->Task3.3). As you can see, the entity has input pins KEY, SW, and CLOCK50, and output pins HEX0. You can also see that one instance of state_machine is instantiated within the wrapper design. The following diagram shows the connections with **phase3.vhd** graphically.



Note that the wrapper also contains a clock divider. The DE2 board contains two clock sources: one that produces a 50MHz clock, and one that produces a 27MHz clock. In this Phase, we will use the 50MHz clock (this means there are 50,000,000 cycles per second… exercise: work out the period of each cycle. If you can't figure it out, be sure to ask your TA). With 50,000,000 cycles per second, your state machine will transition states far too fast for you to see them on the hex digit. The clock divider steps down the 50MHz frequency to a clock with a frequency of less than one Hz (a clock period of a few seconds). This allows you to observe the operation of the state machine. The clock divider is written as a process and is included in the **phase3.vhd** file you downloaded. Identify the process that makes up the clock divider, and try to understand it (we will talk more about these sorts of processes later in the course, but it should be fairly self-explanatory). Again, a TA can help you if you don't understand it.

Combine the wrapper file (**phase3.vhd**) and your **state_machine.vhd** into the single project (called **phase3**) and compile your design (remember to import your pin assignments from **DE2_pin_assignments_UBC.csv** first**)**. Once you successfully compile your design, download it to the board and test it. You should see the hex digit 0 changing roughly every second, and you should be able to adjust the direction of the transitions using SW0. You can also reset the state machine (go back to the first state) using KEY0.

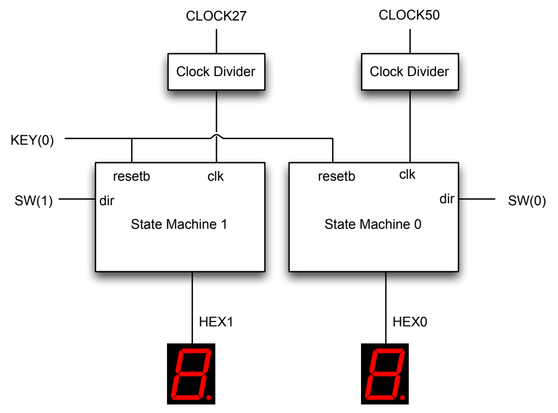
If the circuit does not work as expected, you have some debugging to do. When you were simulating in Task 3.2, you could simply add the state bits to the output waveform to help narrow down the root cause of observed incorrect behaviour. This is more difficult to do now; the state bits are internal wires within the FPGA, and are not connected to external pins for you to observe. Think about how you could solve this problem… what could you do so that you can observe the values of the state bits? If your design doesn't work, you may need to do this to help narrow down the cause of your problem.

PHASE 4: MULTI-CLOCK DOMAIN DESIGN

Real systems often have multiple clock domains; circuitry in different domains will be clocked at a different rate. This phase will introduce you to multi-clocked circuits, and will help emphasize the fact that circuitry is *spatial*; that is, a circuit is a collection of entities that operate "in parallel" rather than the sequential control-flow structure that is common in software.

In this phase, you will enhance your design from Phase 3. In particular, you will create a design that contains two state machines (called State Machine 0 and State Machine 1). Each state machine is an instantiation of the state machine you created in Phase 3. Each state machine drives a different hex digit in the same pattern as in Phase 3. The only difference between the two is that State Machine 0 is clocked with a 50MHz clock (and divided using a clock divider as in Phase 3), and State Machine 1 is called with a 27MHz clock (and divided using the same clock divider). This means that both state machines would cycle through the letters of your name, however, they will do so at a different speed. A separate DIR switch is used for each state machine, so one may be stepping forward through your name while another may be stepping backwards through your name. You can download **Lab1->Movies->phase4.mov** from the Connect site for a demonstration of how this circuit should work.

Your task in this Phase is to design this circuit using VHDL and demonstrate it working on a DE2 board. We are not giving you any skeleton files, but you should be able to use what you learned in Phase 3 to create the proper input/output signals and design hierarchy.



CHALLENGE PHASE: ADJUSTABLE SPEED CONTROLS FOR CIRCUIT FROM PHASE 4

Challenge tasks are tasks that you should only perform if you have extra time, are keen, and want to show off a little bit. This challenge task is only worth 1 mark, but is far more work than the other tasks in this lab. If you don't demo the challenge task, the maximum score you can get on this lab is 9/10 (which is still an A+).

In this challenge phase, you are to modify the circuit from Phase 4 so that the user can adjust the speed at which your circuit cycles through your name. The user should be able to use SW17 to SW10 to select the speed of State Machine 1 and SW9 to SW2 to select the speed of State Machine 0. The user would use each of these switch banks to indicate an 8-bit binary number, and this binary number is used to somehow adjust the speed of the state transitions that occur within the appropriate state machine. How best to do this is up to you. Note: each bank of 8 switches can encode $2^8 = 256$ different numbers; your circuit needs support all 256 different speeds – a different speed for each of the 256 different settings. It is not enough to have only 2 or 3 different possible speeds (eg. fast, medium or slow). The fastest speed should be *roughly* four transitions per second. The slowest speed should be *roughly* 10 seconds per transition.

For the challenge mark, you must demo your working circuit on the FPGA board (simulation is not enough). Demo and explain your circuit to the TA (1 mark). Remember, this part is optional, and not worth many marks, so do not spend all your time on it (at the expense of your other courses).

MARKING OF LAB 1:

During the second week of the lab (Sept 21-25, 2015), you will be marked by the TA. There are two components to the marking:

1. In-person demonstration: you must demonstrate your working circuit(s) (see below). Your TA will ask you questions to assess your knowledge of the material from this lab. You must sign up for a demo time before the start of the marking week; signup sheets will be available during your lab during the working week (Sept 14-18). If you do not sign up for a time, you will need to demo whenever there is an empty slot in the marking schedule.
2. After your demo, you must submit your code to the Lab 1 submission page on Connect. Your code must be submitted *no later than 24 hours after your demo*. You must ensure you submit exactly the code that you demoed; submitting code other than the code you demo is considered academic misconduct. You should submit all .vhd files as described below.

Exactly what you should demo and submit depends on how far you got in the lab.

If you finished Phases 1 to 4 *and* the Challenge Phase, you can demo your solution to the challenge phase only (you don't have to separately demonstrate your solutions to Phase 3 or 4). On the Connect site, submit all .vhd files that make up your Challenge Phase solution. The maximum mark you will receive is 10/10 (it may be lower if your code is not well-written or you can not adequately answer TA questions).

If you finished Phases 1 to 4 (but not the Challenge Phase), you can demo your solution to Phase 4 only (you don't have to separately demonstrate your solutions to Phase 3). On the Connect site, submit all .vhd files that make up your Phase 4 solution. The maximum mark you will receive is 9/10 (it may be lower if your code is not well-written or you can not adequately answer TA questions).

If you finished Phases 1 to 3 (but not Phase 4), you can demo your solution to Phase 3 only on the DE2 board (you don't have to separately demonstrate the Modelsim simulation for Phase 3). On the Connect site, submit all .vhd files that you modified in creating your working Phase 3 solution. The maximum mark you will receive is 6/10 (it may be lower if your code is not well-written or you can not adequately answer TA questions).

If you finished Task 3.2 (simulation) but not Task 3.3 (working DE2 implementation), you can demo your working Modelsim simulation. On the Connect site, submit all .vhd files that you modified in creating your working Modelsim simulation. The maximum mark you will receive is 4/10 (it may be lower if your code is not well-written or you can not adequately answer TA questions).

If you did not finish Task 3.2 (you did not get a working simulation), discuss your progress with your TA, show what you have accomplished, and try to explain why you think your code does not work. The TA will advise you what to submit on the Connect site.

Do not submit temporary files, or .vhd files that you did not modify. Please do not zip up your entire working directory and submit it; if everyone does this, it takes too much space.

You are doing this lab in pairs: only one person of the pair needs to submit the code on Connect. Both members of the group must be present during the in-person demonstration (if one person is not present, that person will not receive marks for this lab).