DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
UNIVERSITY OF BRITISH COLUMBIA
EECE 353 – Digital Systems Design
May – June 2015

Lab 2: Finite State Machines and Synchronous Systems

In this lab, you will create a controller for the LCD on the DE2 board. The controller will be a simple synchronous finite state machine (FSM) that you will describe using VHDL. Recall that in a synchronous system, a global clock signal is used to synchronize the movement of data. For an FSM, the clock coordinates when state-transitions occur. You will first build the FSM and clock it using a push-button switch. You will then replace the push-button clock with an on-board oscillator that is scaled down using a frequency divider that you will describe with VHDL.

LCD CONTROLLER OVERVIEW

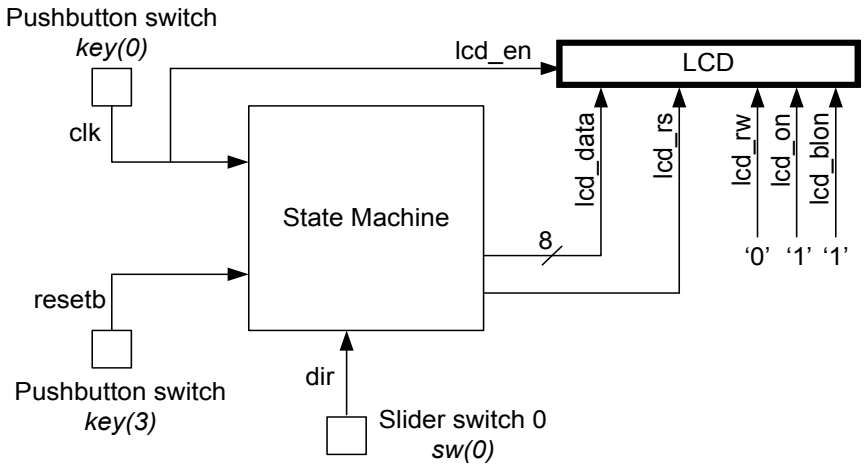
The system you will build will cycle through and display the first five characters of your name. Each clock cycle, the LCD will display one character. In the first version of your implementation (Task 2) the clock comes from a push-button switch **key(0)**. Every time you depress the switch, another character appears. So if your name is "Abcde" (such a great name), the LCD would display an "A" in the first cycle, add a "b" in the second cycle (so the display is "Ab"), add a "c" in the third cycle (so the display is "Abc"), etc. On the six cycle, it would cycle back to "A". Each character is displayed to the right of the previous characters, so after 12 cycles (for example), the LCD would display "AbcdeAbcdeAb".

To make things interesting, the user can, using the slider switch **sw(0)**, change the direction in which the letters cycle. If this switch is "down" (0), the system operates as described above. If this switch is "up" (1), the system cycles through the letters in reverse order (but still starts with the first character). So after 12 clock cycles, if your name is "Abcde" and **sw(0)** is "up" from the start, the LCD would display "AedcbAedcbAe". In this example, it started with A, but then cycled backwards ("e", "d", "c", etc).

To make things *even more* interesting, the user can change the slider switch during any cycle. So, for example, you might count "forwards" for 4 cycles, "backwards" for 2 cycles, and "forwards" for 4 cycles, giving an LCD display of "AbcdcbcdeA".

There is also a reset input; this will be controlled by the pushbutton switch **key(3)**. When this pushbutton switch is depressed, the system resets immediately. After a reset (and at the start), the state machine takes 6 cycles before starting with the first cycle of your name (this is due to the need to reset the LCD display; this will be explained later).

The following diagram shows the overall system you will build:



TASK 1: LEARN HOW TO USE THE LCD

Read this section to learn how the LCD operates. Then download the **test_lcd.vhd** design from Connect to further familiarize yourself with the LCD. **There are no deliverables for this section.**

The LCD has five single-bit control inputs, and one 8-bit wide data bus. The control inputs are as follows:

lcd_on	'1' turns the LCD on. In this lab, this should always be '1'
lcd_blon	'1' turns the backlight on. In this lab, this should always be '1'
lcd_rw	Specifies whether you want to read or write to the LCD. In this lab, we will only be writing, meaning this should always be '0'.
lcd_en	This is an enable signal. It is also used to latch in data and instructions (see below)
lcd_rs	This allows you to indicate whether the **lcd_data** data input is being used to send characters or instructions (see below).

The 8-bit data bus is called **lcd_data** and is used to send instructions or characters to the LCD display.

You can communicate with the LCD using either instructions (to set the LCD in a certain mode or tell it to do something like clear the display) or using characters (in which case the character is displayed on the screen). Each cycle, you can send either one instruction or one character on the 8-bit bus. If you are sending an instruction, the **lcd_rs** signal should be set to 0, and if you are sending a character, the **lcd_rs** signal should be set to 1.

The data bus is sampled on the *falling* edge of the **lcd_en** signal. In this lab, we will drive **lcd_en** with the system clock (which comes from one of the pushbuttons). It is important to remember that the LCD instruction or character is accepted on the falling edge of this clock (this is different than the state machine, which changes states on the rising edge of the clock).

So, to be clear, to send an instruction or character, you would do the following. First, **lcd_on**, **lcd_blon**, **lcd_rw** should be set as described above. **lcd_en** would initially be 1. You would then drive **lcd_rs** with a 0 (if you want to send an instruction) or 1 (if you want to send a character). At the same time, you would drive either the instruction code or character code (8 bits) on **lcd_data**. Then, **lcd_en** would drop to 0, and the LCD would either accept and execute the instruction, or accept and display the character.

There are several instructions that the LCD accepts. This handout will not describe all of them in detail. Instead, this handout will indicate a sequence of instructions used to set up the LCD properly. To set up the LCD, you should send the following instructions, in this order, once per reset:

00111000 (hex "38")
00001100 (hex "0C")
00000001 (hex "01")
00000110 (hex "06")
10000000 (hex "80")

In fact, the first instruction (00111000) should be sent twice, since depending on how you implement the reset, the LCD might miss the first one. Therefore, resetting the LCD will require 6 cycles. If you want to understand what these instructions mean, you can consult the LCD datasheet, which is on the Connect site.

Once you have set up the LCD as described above, you can send characters, one character per cycle. The following diagram shows the character encoding.

Character	Code	
	Binary	Hex
Space	00100000	20
!	00100001	21
"	00100010	22
#	00100011	23
$	00100100	24
%	00100101	25
&	00100110	26
'	00100111	27
(	00101000	28
)	00101001	29
*	00101010	2A
+	00101011	2B
,	00101100	2C
-	00101101	2D
.	00101110	2E
/	00101111	2F
0	00110000	30
1	00110001	31
2	00110010	32
3	00110011	33
4	00110100	34
5	00110101	35
6	00110110	36
7	00110111	37
8	00111000	38
9	00111001	39
:	00111010	3A
;	00111011	3B
<	00111100	3C
=	00111101	3D
>	00111110	3E
?	00111111	3F

Character	Code	
	Binary	Hex
@	01000000	40
A	01000001	41
B	01000010	42
C	01000011	43
D	01000100	44
E	01000101	45
F	01000110	46
G	01000111	47
H	01001000	48
I	01001001	49
J	01001010	4A
K	01001011	4B
L	01001100	4C
M	01001101	4D
N	01001110	4E
O	01001111	4F
P	01010000	50
Q	01010001	51
R	01010010	52
S	01010011	53
T	01010100	54
U	01010101	55
V	01010110	56
W	01010111	57
X	01011000	58
Y	01011001	59
Z	01011010	5A
[	01011011	5B
¥	01011100	5C
]	01011101	5D
^	01011110	5E
_	01011111	5F

Character	Code	
	Binary	Hex
`	01100000	60
a	01100001	61
b	01100010	62
c	01100011	63
d	01100100	64
e	01100101	65
f	01100110	66
g	01100111	67
h	01101000	68
i	01101001	69
j	01101010	6A
k	01101011	6B
l	01101100	6C
m	01101101	6D
n	01101110	6E
o	01101111	6F
p	01110000	70
q	01110001	71
r	01110010	72
s	01110011	73
t	01110100	74
u	01110101	75
v	01110110	76
w	01110111	77
x	01111000	78
y	01111001	79
z	01111010	7A
(	01111011	7B
|	01111100	7C
)	01111101	7D
→	01111110	7E
←	01111111	7F

So, for example, if you wanted to display an "a", you would send 01100001 on the **lcd_data** bus. Note that the table above includes both binary and hexadecimal (base-16) for each code; computer engineers like to talk in hexadecimal, since it is more convenient than binary. Other characters are available, and you can even design your own characters. See the datasheet on the web site if you want more information.

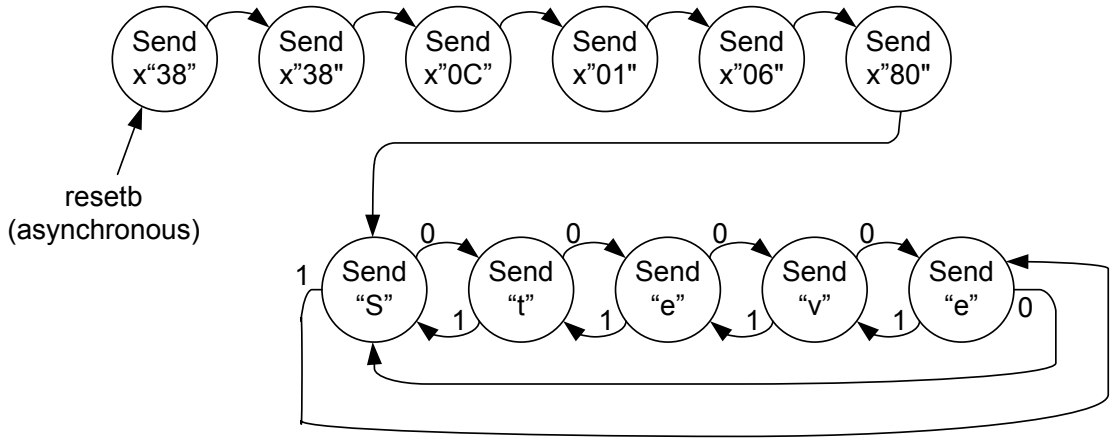
There are stringent timing requirements that must be met using the LCD (meaning that there is a limit to how fast you can send data to the LCD). However, in this lab, we are using the pushbutton switch as a clock, and it is not possible for you to push the button so fast that you are in danger of violating any of these minimum times. All that matters for this lab is that you need to make sure that the control lines are steady when the clock (**lcd_en**) switches from high to low.

If you want more information on the LCD, see the datasheet on the Connect site.

Download the **test_lcd.vhd** file from Connect and compile it to your DE2 board. This is a simple design that connects the **lcd_data** and **lcd_en** signals to switchs on the DE2 board. You can use this to play with the LCD and make sure that you understand how to control the LCD.

TASK 2: MANUALLY-CLOCKED FINITE STATE MACHINE

Design a state machine to implement the circuit as described in the "LCD Controller Overview" section of this handout. The state diagram might be something like this (if your name is "Steve"):



Upon reset, the FSM cycles through the first six states regardless of the input. This is to send the instructions to set up the LCD as described in Task 1. The reset is asynchronous; review the course notes to make sure you remember what this means. The reset is also active low, meaning that a "0" means reset, and a "1" means normal operation (this makes it easier to use the pushbutton switch). The FSM is positive-edge triggered; this means that the transition from one state to the next occurs on the rising edge of the clock. The FSM has a single input **sw(0)** used to control the direction of letter cycling. The outputs of the FSM are the signals **lcd_rs** and **lcd_data**; given the discussion on the previous pages, you should be able to figure out what should be driven on these signals each cycle. Note that this is a Moore state machine, meaning the output depends only on the current state.

Start with the template (**lab2.vhd**) on the Connect site. You should make all your changes to this file.

Simulate your design using Modelsim, and make sure that it works as expected. A very simple testbench is provided. This testbench resets the system, and toggles the clock. The clock cycle in this testbench is set to 6ns, so I would suggest selecting a run-time length of 80 ns to ensure you see all the state transitions. Don't forget to Zoom Full to see the whole thing, however, you'll probably have to Zoom in to see enough detail to convince you it is working. Manually observe the waveform and make sure it matches what you expect.

Once you are satisfied with your simulation, download your design to the board. Remember to use the pin assignments file from Lab 1. Cycle through the states and show that it operates as expected. Test the reset button to make sure that works too. You will probably find it easier to see what is going on by wiring the state bits (probably called something like "present_state" or "current_state" in your VHDL code) to the green LEDs so you can easily see what state you are in. Interestingly, this highlights a fundamental limitation of debugging directly on hardware – unlike software, you cannot automatically see any of the internal signals unless you have manually connected them to an output port before compilation.

Hint: Earlier I mentioned that the LCD accepts data on the falling edge of the clock. Don't be confused. In the state machine you design here, the state changes (and hence output changes) all happen on the rising clock edge. This is a normal state machine, just like we discuss in class.

➔ **BE PREPARED TO DEMO YOUR SIMULATION FOR 3 MARKS.**

➔ **BE PREPARED TO DEMO ON THE DE2 BOARD FOR 3 MARKS.**

TASK 3: ON-BOARD CLOCK AND CLOCK FREQUENCY DIVIDER

In Task 2, you used **KEY(0)** as your clock. In a real system, you would have a clock that is automatically generated. To generate a clock signal (with, say, a 40ns period), you might be tempted to use VHDL code that looks something like this:

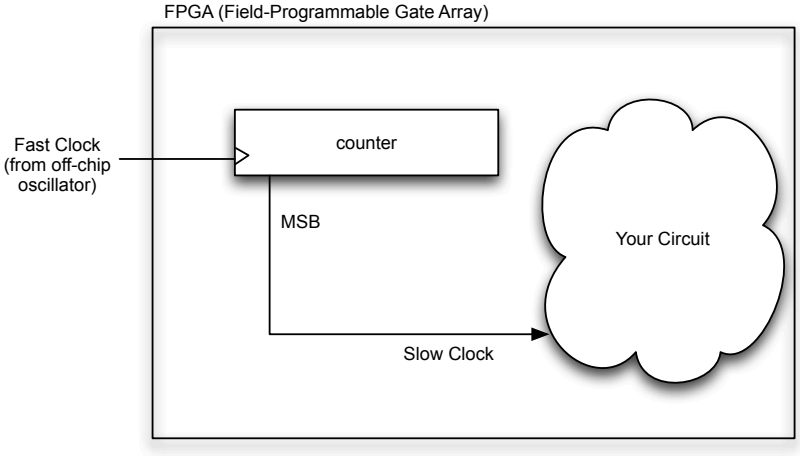
```
clk <= '0';
WAIT FOR 20 ns;
clk <= '1' ;
WAIT FOR 20 ns;
clk <= '0';
WAIT FOR 20 ns;
… etc
```

WRONG WAY
to do it!
(not synthesizable)

While this is legal VHDL, it is not synthesizable (often, this sort of code appears in testbenches, and as we talked about in class, testbench code does not need to be synthesizable). In most designs, the clock signal is not generated on-chip, but instead comes from an off-chip source (somewhere outside of your FPGA). The DE2 board has two oscillators (clock sources) that are permanently wired to specific pins of our FPGA device. Based on our pin assignments, these two clock sources are called **CLOCK_50** (a 50MHz clock) and **CLOCK_27** (a 27MHz clock).

If you wanted to replace **KEY(0)** with the clock from the oscillator, you could change **KEY(0)** to **CLOCK_50** everywhere it appears in your Task 2 code (in the process sensitivity list, the if statement inside the process and the port in the entity part of the description). You can try this and recompile; your circuit will now be clocked by the 50MHz clock rather than the **KEY(0)**.

If you do this, you will run into another problem. The 50MHz clock is too fast for you to see anything happen. Even if you could see things happen that fast, it is too fast for the LCD to respond. Therefore, we need to slow down the clock. One way to do this is shown below.



In the above diagram, the "fast clock" (in our case, 50MHz) is an input to our chip (it is a top-level port in your VHDL design). It feeds the clock input of a counter (you will write VHDL for this). Suppose, for the purpose of this discussion, your counter is 8 bits wide (so it can count from 0 to 255, and then rolls back to 0). On each rising edge of the fast clock, the 8-bit counter increments by 1. If we think about the bits inside the counter, we can see that bit 0 (the least significant bit) changes every 1/50Mhz = 20ns. Bit 1 changes every 40ns. Bit 2 changes every 80ns. Bit 3 changes every 160ns…. (if you don't understand why this is so, write down the binary representations of 1, 2, 3, 4, 5, 6, 7, etc and observe how often each bit changes). Notice that any one of these bits could be used as a slow clock signal. As described above, bit 0 changes every 20ns. This means that if bit0 was used as a clock signal, it would go through the complete clock cycle (low to high and high to low) every 40ns – this translates to a clock of 25MHz. Bit 1 changes at half the frequency of bit 0, meaning if we used bit 1 as a slow clock, it would change at a rate of

12.5MHz. If we go all the way to the Most Significant Bit (MSB) which is bit 7 of the counter (because this is an 8 bit counter), we see that the MSB could be used as a 50MHz/256 = 195KHz clock. In the above diagram, this bit is used to clock the user circuit, meaning the user circuit is clocked at 195KHz.

In our design, even this is too fast, since we want to be able to see the effects of transitions. But, as you can deduce from the above, by simply adding more bits to the counter, we can slow down the clock as much as we want.

Your first step in this Task is to determine how many bits we would need in the counter to slow down a 50 MHz clock to 1 Hz (if the clock is running at 1 Hz, that should be slow enough for us to observe the effect of state transitions). *Hint: the number of bits required is more than 16 and less than 64.*

The second step in this task is to integrate this clock divider into your LCD controller from Task 2. Your design will *no longer have* **KEY(0)** *as a top-level input port*. Instead, it will have **CLOCK_50** as a top-level input port. When you download and run your design, it will step through the states approximately one state per second. Since the first six states are sending control characters, you should not expect to see anything on the LCD until 6 seconds after you run it.

> *Hint #1: Slide Set 3 has an example of a counter circuit which you will probably find useful.*
>
> *Hint #2: The counter itself can be descibed as a process; you can either add this process to your existing architecture (remember that an architecture can have as many processes in it as you like) or you can create an architecture/entity for the counter, and architecture/entity for the state machine, and combine them structurally (you saw examples of structural descriptions in Slide Set 2)*
>
> *Hint #3: Consider connecting the slow clock signal to an LED on the board, so you can observe whether the clock signal is working. It will make debugging easier.*

➔ **BE PREPARED TO DEMO ON THE DE2 BOARD FOR 3 MARKS.**

CHALLENGE TASK (OPTIONAL)

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 technical score you can get on this lab is 9/10 (which is still an A+).

In this challenge task, you are to modify your circuit from Task 3 such that it keeps track of the number of characters printed and clear the screen when the screen is full. The reason this is not trivial is because you don't know exactly what characters you will have displayed (because the user can change the sequence using **SW(0)** switch).

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. 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).

➔ **BE PREPARED TO DEMO ON THE DE2 BOARD FOR 1 MARK.**

SUMMARY ON WHAT TO DEMO:

Demo your simulation of the design (on ModelSim) as requested from Task2 for 3 Marks.

For the remaining 7 marks (board demos), each section of this lab builds on the previous section. Due to limited demo time with your TA, you don't need to demo each on-board task individually. If you successfully demo the challenge task, we can infer that all the other sections also work. If you successfully completed Task 3, you can demo that, and we can infer that Task 2 worked. If you only have part of Task 2 working, you can demo what you have, and explain what you think is wrong for part marks.