

Finite State Machine Design for the PS/2 and LCD Interfaces

Objective

To understand the importance of finite state machines (FSMs) in digital systems design. Gain experience with coding styles for FSMs and implement digital systems using the PS/2 and LCD peripherals on the DE2-115 board. PS/2 is a serial port used to interface a keyboard or a mouse to a personal computer. LCD is a low-power/flat-panel display that uses liquid crystals and it is predominant in embedded devices.

Preparation

- Revise the first lab and FSMs
- Read this document and get familiarized with the source code and the in-lab experiments

In-lab experiments

- [Experiment 1](#)
- [Experiment 2](#)
- [Experiment 3](#)
- [Experiment 4](#)
- [Experiment 5](#)

Evaluation

Take-home exercise (due 16 hours before the next lab is scheduled):

- [Exercise](#) (has a weight of 3% of your final grade)

Online workflow

- [Simulation of PS/2 and LCD events](#)
- [PS/2 keyboard codes](#)
- [LCD codes](#)

Experiment 1

Figure 1(a) shows the FSM that corresponds to the source code from **experiment 1**, which describes an FSM that detects if push-button 0 has been pressed three times consecutively. Whenever push-buttons 1, 2 or 3 are pressed, the FSM returns to an idle state and it waits until push-button 0 is pressed again. Before returning to the state that indicates whether push-button 0 has been pressed three times consecutively, the FSM goes through two intermediate states that record if push-button 0 has been pressed once or twice consecutively. In any other state except the one where push-button 0 has been pressed for three times consecutively, the value displayed on the 7-segment-display should be "F".

| □ | |:-:| **Figure 1** - State diagrams for *experiment 1*|

You have to perform the following tasks in the lab for this experiment:

- verify if the FSM shown in Figure 1(a) is described correctly and if its circuit implementation works
- change the FSM description in such way that if push-button 1 is pressed three times consecutively the number displayed on the 7-segment-display is "1"; note, if push-buttons 2 or 3 are pressed then the system returns to the idle state; for clarity Figure 1(b) shows the FSM to be implemented

Experiment 2

In this experiment you will learn about the PS/2 port and you will modify the FSM of the PS/2 controller.

Figure 2 shows the timing of the PS/2 interface. This interface has two bidirectional signals: `PS2_clock` and `PS2_data`. The given implementation supports only the receiver part of the PS/2 interface (i.e., we only receive data from the keyboard). If a key is pressed or released one byte of data is sent approx every ms from the PS/2 keyboard. There is a reference clock (`PS2_clock`), which is logic high when no data is sent. When the PS/2 device initiates the communication a `PS2_clock` pulse must be accompanied by a start bit (driving `PS2_data` to logic low). This is detected in the `S_PS2_IDLE` state. In the next state (`S_PS2_ASSEMBLE_CODE`), on each edge of the `PS2_clock`, the value of the `PS2_data` is sampled into the most significant position of a right-shift register (`PS2_shift_reg`). After 8 clock cycles the parity bit is checked in the `S_PS2_PARITY` state (in our implementation no action is taken on this parity bit, nonetheless the state must exist in order to comply with the PS/2 protocol initiated by the keyboard). The final state `S_PS2_STOP` checks if the stop bit (active high) is passed with the last `PS2_clock` pulse. In the same state we set a flag `PS2_code_ready` that together with the `PS2_code` (loaded from `PS2_shift_reg`) will be passed for further processing outside of the controller. It is important to note that the FSM is clocked at 50MHz and it is enabled when a rising edge is detected on the `PS2_clock`.



 |Figure 2 - Timing diagram for the PS/2 interface|

Figure 3 illustrates the processing of the PS2 codes received from the PS/2 controller. If an edge is detected on the `PS2_code_ready` signal, the PS2 code is placed into the 8 least significant bits of a 24-bit left-shift register. This register controls the 6 rightmost 7-segment displays. Note, when a key is pressed a “make code” is generated and when a key is released a “break code” is generated by the PS/2 keyboard. For the keys that we are concerned with in this lab, the make code is 1 byte and the break code is 2 bytes (the first byte is 8'hF0 and the second one is the same as the make code). The keys supported in our simplified implementation are available in the PS2 keyboard codes [table](#). The `PS2_make_code` flag that comes out of the PS/2 controller module can be used to differentiate between the make codes and the break codes.

 |Figure 3 - Interfacing the PS/2 controller to the top level design|

You have to perform the following tasks in the lab for this experiment:

- check on the 7-segment displays that if a PS/2 keyboard key is pressed and released then both the make code and the break code are properly assembled by the PS/2 controller
- implement the functionality of the `PS2_make_code` flag as follows: in the state machine from the PS2 controller set the `PS2_make_code` to a “1” only if a make code is generated; in the top level file the `PS2_code` should be passed to the shift register only if the `PS2_make_code` is set; this will enable us to “filter out” the break codes and display only the make codes on the 7-segment displays

Experiment 3

The aim of this experiment is to familiarize you with the LCD controller and its interface.

| □ | | :-: | **Figure 4** - LCD controller interface to top FSM and the state transition diagram of the top FSM|

Although the LCD interface is bidirectional, in our simplified implementation the user defined logic is only sending instructions to the LCD. The behavior of the LCD controller can be explained using Figure 4(a). After power-up an initialization sequence is provided after which the LCD controller waits for instructions, such as, for example, display a character at the current cursor position or change the line. When the user defined logic sends a new instruction to the LCD controller it must generate a rising transition on the `LCD_start` signal and provide the appropriate instruction. The instruction is on 9 bits and if the most significant bit is a "1" then the remaining 8 bits will uniquely identify the character to be displayed. Note, the supported LCD codes for the characters can be found in the LCD codes [table](#). If the most significant bit is a "0" then the instruction is a system instruction (in our implementation we cover only the "initialization" and "change line" instructions). Note, no new instruction can be generated until the LCD controller asserts the done signal.

The transition diagram of the top FSM that decides what kind of characters are displayed on the LCD screen is shown in Figure 4(b). The first two states are for initializing the LCD device and should not be modified. The bottom three states supply 32 character display instructions and one change line instruction. These instructions are initiated by toggling switch 0, which is required to leave the idle state. The LCD instructions are provided as constants to the `LCD_data_sequence` signal and they are looked-up using a counter (`LCD_data_index`) updated within the `S_LCD_FINISH_INSTRUCTION` state. Note, after `LCD_start` is asserted in the `S_LCD_ISSUE_INSTRUCTION`, it will be de-asserted immediately in the following clock cycle in the `S_LCD_FINISH_INSTRUCTION` state. Only then the `LCD_done` signal will be monitored to see if the LCD controller has finished its writing to the external LCD display. If this is the case, the next instruction will be sent to the LCD controller. We will return to `S_IDLE` when all the characters have been displayed.

You have to perform the following tasks in the lab for this experiment:

- understand the behavior of the FSM from the top module and verify if the design works correctly
- change the displayed message to show your group id on the top line and lab day on the bottom line [LCD codes](#)

Experiment 4

This experiment puts together the PS/2 and LCD controllers, thus displaying typed characters.

Embedded memories are part of the field-programmable array (FPGA) fabric and they can be configured either as a read-only memory (ROM), single-port random access memory (RAM) or dual-port RAM. In this lab we will be using only a ROM, which has the following ports: input address, clock and output data. The ROM can be initialized at compile time and its content is programmed into the FPGA together with the rest of the configuration stream required for logic elements and interconnect switches. The ROM content is specified in the memory initialization file ("MIF"). The ROM content can also be initialized with a "HEX" file format.

The previous two experiments have introduced the PS/2 and LCD controllers. You have displayed the make codes on the 7-segment displays and you have printed characters on the LCD. As it is obvious that one would like to print the typed characters, the question is how can we put together the two controllers? This objective is achieved using a ROM that is employed as a code converter. Figure 5 illustrates the basic principle on how the PS/2 and LCD controllers are bridged using a ROM.

| □ | |:-:| **Figure 5** – Using a ROM to bridge the PS/2 and LCD controllers|

The PS/2 controller from the figure has been updated to provide the make codes of the typed keys when the `PS2_make_code` flag is set. The ROM from the figure, stores the LCD character code in the location given by the make code. For example, since the make code for "z" is "8'h1A", (see [PS/2 keyboard codes](#)), and the LCD character code for "z" is "8'h7A" (see [LCD codes](#)), we will store 8'h7A in location 9'h01A. The ROM address is on 9 bits (and not on 8 bits as given by the make code width as shown in the figure) because the ROM has two segments. The lower segment (i.e., most significant bit is a "0") stores the LCD code values for the lower-case characters and the upper-segment (currently unused) can be used to store the LCD code values for the upper-case characters.

The content of the ROM is provided in the "MIF" file. When a character has been typed, it will be displayed by hooking up the lower 8 bits of the `LCD_instruction` word to the output of the ROM. The `S_IDLE` state is left when a new make code has been detected (by monitoring a transition on the `PS_code_ready` and ensuring that `PS_make_code` is set). Because the LCD instruction is on 9 bits, the most significant bit (i.e., "1" for printing a character) is provided by the FSM from the top level design file. To change the line on the LCD display, the state machine from the previous experiment has been extended with two extra states. These two states `S_LCD_ISSUE_CHANGE_LINE` and `S_LCD_FINISH_CHANGE_LINE` ensure that after 16 characters have been typed, the LCD display advances to a new line (i.e., from the top line to the bottom line or the other way around). The instruction for changing lines is issued also by the FSM.

You have to perform the following tasks in the lab for this experiment:

- understand the behavior of the FSM from the top module and verify if the design works correctly
- update the ROM file to support keys 0 to 9 (use [PS/2 keyboard codes](#) and [LCD codes](#))

Experiment 5

The objective of this experiment is to put together a large FSM by showing how to buffer several keys typed on the PS/2 keyboard and subsequently displaying all of them at once on the character LCD display.

| □ | |:-:| **Figure 6** - Using a shift register structure to buffer 4 typed keys before displaying all of them on the LCD|

Figure 6 shows how 4 data registers are connected into a shift-register structure between the PS/2 controller and the code-converter ROM. These 4 data registers are used to buffer the typed keys by storing and then shifting the last 4 make codes that were generated by the PS/2 controller. After all the 4 registers have been filled with new values, the top FSM (shown in Figure 7) will switch from the *S_IDLE* to *S_LCD_WAIT_ROM_UPDATE*. This state together with *S_LCD_ISSUE_INSTRUCTION* and *S_LCD_FINISH_INSTRUCTION* will enable the necessary steps to display the 4 pressed keys. When a key is printed, the shift register structure will provide the stored make code to the ROM by shifting `data_reg[i]` to `data_reg[i+1]`, while filling `data_reg[0]` with all zeros.

The output of the most significant register from the shift-register structure (i.e., `data_reg[3]` in this example) is connected to address lines of the code converter ROM. There is one counter `data_counter` that keeps track of how many make codes have been printed and another counter `LCD_position` that keeps track of the position on the LCD screen. When putting together the state machine shown below with the above-mentioned counters and registers, the ROM can translate the 4 buffered make codes to LCD codes such that the corresponding characters are displayed on the LCD.

| □ | |:-:| **Figure 7** - The revised FSM for buffering 4 typed keys before printing them on the LCD|

This last experiment puts together the key concepts explored so far: modulo counters, shift registers, state machines, ROMs, combinational and sequential circuit blocks, design hierarchy and component instantiations, as well as the behavior of the PS/2 and LCD interfaces. Therefore, it is essential that you take your time to understand the source code, the interactions between design components, the state transitions and the effects of these state transitions on the data path registers, and the relation between the source code and the implemented hardware.

You have to perform the following tasks in the lab for this experiment:

- understand the behavior of the FSM from the top module and verify if the design works correctly
- change the shift register structure to 16 data registers, so an entire line can be displayed at once

Exercise

In the reference design from **experiment 5**, only the lower memory half (i.e., address range 000h-0FFh) is used for storing the LCD codes of lower-case characters. First, extend the "MIF" file with the LCD codes such that the upper memory half (i.e., address range 100h-1FFh) stores the LCD codes for upper-case characters. Each letter character displayed on the LCD should be shown in either the *upper-case mode* or the *lower-case mode*, which are defined as follows. If the position (or index) of the *least* significant switch that is in the *low* position from the group of switches 15 down to 0 is an odd number then *mode* is *upper-case*; otherwise it is *lower-case* (this includes the scenario when no switches from 15 down to 0 are in the *low* position). For the sake of simplicity, it is assumed that the *mode* affects only the letter keys, i.e., 'a' to 'z'; digit keys ('0' to '9') and the space key are not affected by this *mode*. It is fair to assume that for this exercise, no other keys than the types specified above (i.e., 'a' to 'z', '0' to '9' and space) will be typed. It is important to note that the *mode* can be changed by toggling a switch at a time between the times when any two keys that have been pressed.

As for the in-lab experiment, an entire line is displayed at once after the last (sixteenth) character to be displayed on a line has been typed.

For the top line, if the sequence of the first eight characters is identical to the sequence of the last eight characters (a **match** status), display letter "d" (for "detected") on the leftmost 7-segment display until a new key is type. The behavior to be implemented for the bottom line is the same as for the top line, with one exception: letter "d" will be displayed on the leftmost 7-segment display only if the sequence of the first eight characters is a **reverse match** of the sequence of the last eight characters. After a key is pressed again, the behavior resumes as specified above for the top line. Note, at any other times than specified above the leftmost 7-segment display will be lightened off (as if the board is not powered). Note also, what gets displayed on the remaining 7-segment displays is not specified (you can use them to display some debug info, if you deem this to be useful).

Consider the following illustrative examples for clarification purposes:

Line	Printed Characters	Leftmost 7-Segment Display	Status
Top	'Ab01C23dAB01c23D'	d	match
Top	'aB01c23Dab01c230'		N/A
Bottom	'aB01c23Dd32C10bA'	d	reverse match

As shown above, it is important to emphasize that the *lower-case/upper-case mode* matters only for the letter keys when displaying a character and it does not matter when the match/reverse match comparisons are done. As a final point, you should also note that while a full line of 16 characters gets sent to the LCD controller, it will take just below 15 microseconds (us) in simulation. Hence, make sure the next key press in `board_events.txt` will be done after these 15 us have elapsed (this concerns simulation only, if you choose to use it).

Submit your sources and in your report write approx half-a-page (but not more than a full-page) that describes your reasoning. Your sources should follow the directory structure from the in-lab experiments (already set-up for you in the `exercise` folder); note, your report (in `.pdf`, `.txt` or `.md` format) should be included in the `exercise/doc` sub-folder.

Your submission is due 16 hours before your next lab session. Late submissions will be penalized.

Emulating board events

In addition to the conventions used for switches and push-buttons that were introduced for the sequential circuits from **lab 1**, in this lab we also have board events for the PS/2 keys defined in the `board_events.txt` file. Below are a few key points about the custom format used to define the interactions with a PS/2 keyboard in our custom simulation environment:

- events for PS/2 keys have a `PS` prefix; a specific key is identified through a single letter; for example `PSA` means PS/2 key 'A' was pressed;
- To simplify the parser used by the testbench to control the simulation, in addition to the alphanumerical keys ('A' to 'Z' and '0' to '9'), *only* four additional keys are supported: `PS_` means the Space key was pressed, `PS(` and `PS)` are for left-shift and right-shift respectively and `PS!` is for Enter; note also, the alphabet keys are identified **only** through upper-case letters ('A', 'B', ..., 'Z');
- The additional arguments for the PS/2 key events in the `board_events.txt` file have the same meaning as for push-buttons, i.e., the time the key was pressed, in terms of microseconds (us) relative to the start of simulation, and the duration the key was kept pressed (also in terms of us).

Consider the following example for `board_events.txt` :

```
SW16 300
PB0 8 5
PSA 500 10
SW0 5
PS_ 200 21
...
```

Based on the sequence of events from the above file, the following will occur.

```
- switch 0 toggles at 5 us
- push-button 0 pressed at 8 us
- push-button 0 released at 13 us
- PS/2 key space pressed at 200 us
- PS/2 key space released at 221 us
- switch 16 toggles at 300 us
- PS/2 key 'A' pressed at 500 us
- PS/2 key 'A' released at 510 us
...
```

A few additional points are worth making:

- A pulse on the PS2 clock line that is driven by an external hardware keyboard is in the range of approx 100 us. In order to save simulation time, we set the duration of this pulse to 80 nanoseconds (ns) in our simulation environment;
- For real-life keyboards, if a key is kept pressed, the same make code is resent at a pre-defined rate; this feature is **not** supported in our custom simulation environment; rather, the make code is sent only once and, after the key is released, the two bytes of the break code are sent from the testbench to the design; since the time needed to send one byte (assuming a PS2 clock of 80 ns) is just below one microsecond, the total amount of time that should pass before the next key is pressed should be the duration while the key is pressed **plus** the two microseconds needed to send the break-code; it is also important to note that the testbench will probably break if overlapping PS/2 events are specified - in simpler words, for the simulation to make sense you should wait for the time needed for the break code of the previously pressed key to be sent before pressing another key!
- Although in `board_events.txt` we define *only* input events, it is worth making a note about the LCD controller; the external character LCD display can accept a new command every approx five milliseconds (ms); this would obviously place too much burden on simulation time and therefore in our simulation environment we reduce the delay between instructions to only a couple of hundred ns;

PS/2 keyboard codes

Key	Make code	Break code	Key	Make code	Break code
`	0E	F0 0E	\	5D	F0 5D
1	16	F0 16	Q	15	F0 15
2	1E	F0 1E	W	1D	F0 1D
3	26	F0 26	E	24	F0 24
4	25	F0 25	R	2D	F0 2D
5	2E	F0 2E	T	2C	F0 2C
6	36	F0 36	Y	35	F0 35
7	3D	F0 3D	U	3C	F0 3C
8	3E	F0 3E	I	43	F0 43
9	46	F0 46	O	44	F0 44
0	45	F0 45	P	4D	F0 4D
-	4E	F0 4E	[54	F0 54
+	55	F0 55]	5B	F0 5B
A	1C	F0 1C	C	21	F0 21
S	1B	F0 1B	V	2A	F0 2A
D	23	F0 23	B	32	F0 32
F	2B	F0 2B	N	31	F0 31
G	34	F0 34	M	3A	F0 3A
H	33	F0 33	,	41	F0 41
J	3B	F0 3B	.	49	F0 49
K	42	F0 42	/	4A	F0 4A
L	4B	F0 4B	space	29	F0 29
;	4C	F0 4C	enter	5A	F0 5A
'	52	F0 52	left shift	12	F0 12
Z	1A	F0 1A	right shift	59	F0 59

Key	Make code	Break code	Key	Make code	Break code
-----	-----------	------------	-----	-----------	------------

LCD character codes

Character	LCD code	Character	LCD code	Character	LCD code
Space	20	0	30	@	40
!	21	1	31	A	41
"	22	2	32	B	42
#	23	3	33	C	43
\$	24	4	34	D	44
%	25	5	35	E	45
&	26	6	36	F	46
'	27	7	37	G	47
(28	8	38	H	48
)	29	9	39	I	49
*	2A	:	3A	J	4A
+	2B	;	3B	K	4B
,	2C	<	3C	L	4C
-	2D	=	3D	M	4D
.	2E	>	3E	N	4E
/	2F	?	3F	O	4F
P	50	`	60	p	70
Q	51	a	61	q	71
R	52	b	62	r	72
S	53	c	63	s	73
T	54	d	64	t	74
U	55	e	65	u	75
V	56	f	66	v	76
W	57	g	67	w	77
X	58	h	68	x	78

Character	Hex code	Character	Hex code	Character	Hex code
Z	5A	j	6A	z	7A
[5B	k	6B	{	7B
]	5D	l	6C		7C
^	5E	m	6D	}	7D
_	5F	n	6E	█	FF
o	6F				