

TSIU03: Lab 2 - Keyboard

Petter Källström, Mario Garrido

September 1, 2015

Abstract

This lab aims to create a system that decodes PS/2 keyboard signals. The goal is to detect the keys 0 to 9, and print the last typed key on a 7-segments display.

1 Introduction

You will read the scan codes that a keyboard sends on the PS2 interface. If the scan code belongs to a numerical key, the corresponding number should be displayed on a 7-seg display. Otherwise it should show “E” on the display. In addition, the code will be shown on the LED bar on the DE2 board.

The system contains two parts: decoding the PS/2 interface (extract the scan codes), and recode the scan code into 7-seg codes.

As a second task, you will create a test bench, that tests the the system for errors.

The web page [1] (section “Communication: Device-to-Hose”) gives a good description on how the PS/2 protocol works, and linked from that page, how the keyboard uses the PS/2 communication. The protocol is also summarized in App A. Read this!

In this lab, we will only care for the last byte in each scan code.

The 7-segment display interface is described in Appendix B. Read this!

2 Common Errors

Some common errors for this lab:

- Trying to use `falling_edge` with “PS2_CLK” while synchronizing to the FPGA clock.
- Most of the errors mentioned in the section “Common errors” in the FAQ [2].

3 Your Task 1: The Keyboard Decoder

Create the module “Lab2_KB”. The module shall:

- be implemented in VHDL.
- scan the PS/2 port for key actions from a keyboard.
- decode the keys 0 to 9 (above the letters — not those on the numerical keyboard).
- print the last byte of the scan code on the red LED bar.
- print the last typed number on a 7-segment display, or “E” if the last keystroke was not a number.
- print your group number on two 7-segment displays.

It is acceptable that the LEDs and 7-segment display flashes while the data is transmitted.

The module should have the inputs/outputs listed in Table 1. **Important:** Those names are exact, stick to them, as well as the top module name “Lab2_KB”. If you change a name, you will get problem during the simulation (since a given help file assumes those names).

3.1 Implementation Hints

Since you are going to read data that is sent in serial, you need a shift register. Each byte is followed by two more bits (parity bit and stop bit), and hence you need ten bits to read the last byte (otherwise the most

Name	I/O	type	Comment
rstn	in	std_logic	Reset, active low.
clk	in	std_logic	System clock, 50 MHz.
PS2_CLK	in	std_logic	PS/2 clock line.
PS2_DAT	in	std_logic	PS/2 data line.
HEX0	out	std_logic_vector(6 downto 0)	The 7-seg for the number.
LEDR	out	std_logic_vector(7 downto 0)	The LED bar for the scan code.
HEX7,HEX6	out	std_logic_vector(6 downto 0)	Two 7-seg for your group number.

Table 1: Port contents of the design

significant bits of the byte will be shifted away when the entire transmission is sent. The recommendation is to use a 10 bit signal and use it as a shift register:

```
signal shiftreg : std_logic_vector(9 downto 0);
```

At every falling flank of PS2_CLK, shift in PS2_DAT2 into the shift register, from left (so you will have the least significant bit (LSB) to the right when done). Since this is a register, it must naturally be assigned in a process. When all shifts are done for a byte (within a millisecond), the transmitted byte is located in `shiftreg(7 downto 0)`, as illustrated in Table 2.

Flank number	On the PS2_DAT	shiftreg(9 downto 0) after the flank									
		sr ₉	sr ₈	sr ₇	sr ₆	sr ₅	sr ₄	sr ₃	sr ₂	sr ₁	sr ₀
1	Start bit	0									
2	D ₀	D ₀	0								
3	D ₁	D ₁	D ₀	0							
4	D ₂	D ₂	D ₁	D ₀	0						
5	D ₃	D ₃	D ₂	D ₁	D ₀	0					
6	D ₄	D ₄	D ₃	D ₂	D ₁	D ₀	0				
7	D ₅	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	0			
8	D ₆	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	0		
9	D ₇	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	0	
10	Parity	P	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	0
11	Stop bit	1	P	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀

Table 2: The shift register content directly *after* the clock flanks.

Since only the last byte of each scan code is of interest in this lab, and there is no need to detect when a complete new byte has arrived, you can decode `shiftreg(7 downto 0)` continuously, which suggests that you do that in a combinational statement outside any process.

Table 3 shows a summary of the codes you need to recode the scan code into 7-segments display codes. **It is important that you understand the table, and what each '1' and '0' in the 7-seg code means.**

Figure 1 depicts the structure of the hardware, and Code 1 depicts the suggested file structure. Use Table 3 to get the 7-seg code for your lab group number.

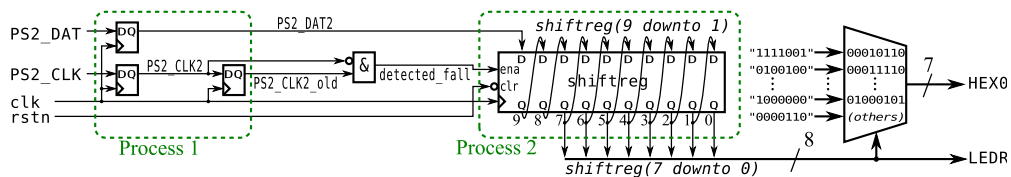


Figure 1: Suggested hardware structure.

Key/number	Scan Code	7-seg
1	00010110	1111001
2	00011110	0100100
3	00100110	0110000
4	00100101	0011001
5	00101110	0010010
6	00110110	0000010
7	00111101	1111000
8	00111110	0000000
9	01000110	0010000
0	01000101	1000000
Else	Else	0000110

Table 3: The used keys, their scan codes, and corresponding 7-segments codes.

```

library ieee;
use ieee.std_logic_1164.all;

entity Lab2_KB is
  port(...); -- Fix this
end entity;

architecture rtl of Lab2_KB is
  -- Declare signals here
begin
  -- Process 1: Synchronize the input
  p1 : process(clk) begin
    if rising_edge(clk) then
      -- Assign input DFFs here
      end if; -- rising_edge(clk)
    end process;
    detected_fall <= ...; -- Fix this

    -- Process 2: Handle shiftreg:
    p2 : process(clk,rstn) begin
      if rstn = '0' then
        -- Insert reset values here
      elsif rising_edge(clk) then
        -- Assign shift register here
      end if;
    end process;
    -- Output the last byte of the scan code:
    LEDR <= shiftreg(7 downto 0);
    -- Recode the scan code (shiftreg(7..0)) here:
    HEX0 <= ... -- Fix this
    -- Also write your group number on HEX{7,6}.
    HEX7 <= "00000000"; -- "8"
    HEX6 <= "0000010"; -- "6" } Example for group 86
  end architecture;

```

DFF: D-type Flip Flop (Swedish: "D-vippa")

Code 1: Suggested file structure for the hardware in Fig. 1.


In the **declaration part**, you need to declare the signals PS2_CLK2, PS2_CLK2_old, PS2_DAT2 and detected_fall as std_logic, and the signal shiftreg as std_logic_vector.

In **process 1**, synchronize the inputs to the system clock. This means that all signals now changes directly after the system clock's rising flanks.

Assign the detected_fall signal as depicted in Fig. 1.

In **process 2**, shift in PS2_DAT2 into shiftreg when detected_fall indicates that PS2_CLK just changed from '1' to '0'.

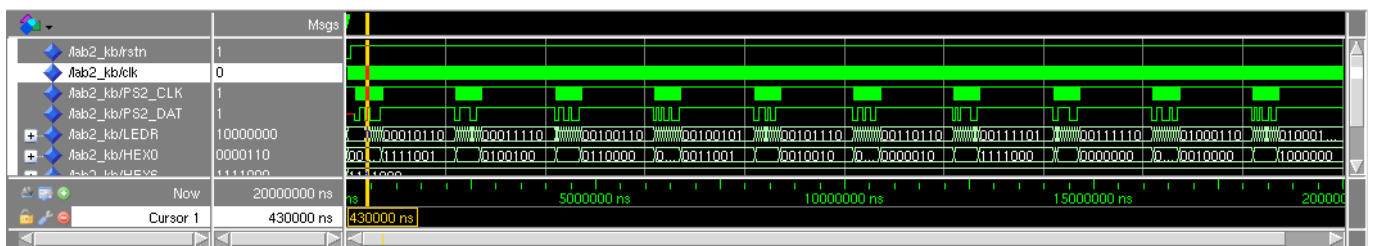
To assign HEX0, use the "when-else" or "with-select" statement.

Hint: In Quartus, you can find VHDL construction templates in Edit→Insert Template... . Look in VHDL / Constructs / Concurrent Statement / Conditional Signal Assignment.

3.2 Simulate

Simulate your design in Modelsim:

The .do file contains the stimuli in this lab. The result should look as depicted in Fig. 2. The stimuli gives a system clock of 50 MHz, a PS2 clock of 20 kHz, and the keys 1, ..., 9, 0 are fed with 2 ms interval. The simulation is 20 ms long. Verify that you get the correct “digits” at HEX0, and that it is “E” during the shifts.



If something is wrong, locate and fix the error in the VHDL file, recompile the unit, simulate again, and check. Loop until it seems to work.

Keep the waveform open, for the demonstration.

In Task 2, you will create a VHDL test bench.

3.2.1 How to handle compilation errors?

If you get compilation errors, solve them using the following (priority) list:

- Read the error code, and discuss within the lab group if it can give any hints (E.g. a missing “;”).
- Try to find some hints in the FAQ [2].
- Consult the VHDL book, or google the error codes and/or VHDL syntax.
- If the lab assistant is free, ask him/her.
- Ask some class mates. If they had the same problem, they should be able to help you. Otherwise they may later get the same problem, and then you should be able to help them.
- If the lab assistant is busy, raise a hand and wait for help. Continue with the googling/books/discussions/etc in the meanwhile.

3.3 Test on the DE2 Board

Now it's time to test the design on the DE2 board. In order to do so, you need to enter a lot of pin names. Those are given in Table 4. The `rstn` signal (pin G26) is connected to KEY0 on the DE2 board.

If you have not synthesized yet, it's time to do so, to get a list of the pins in the pin assignment.

Don't forget to Synthesize again after pin assignment, so the changed pins take effect.

Program the DE2 board using Petters hand-on method, and verify the functionality.

If the connection to the DE2 cannot be established (does not detect the server), please contact the teacher. It may be that the computer to which the DE2 is connected must be (re)started.

Signal	Name	Signal	Name	Signal	Name	Signal	Name
HEX0[0]	AF10	HEX6[0]	R2	HEX7[0]	L3	LEDR[0]	AE23
HEX0[1]	AB12	HEX6[1]	P4	HEX7[1]	L2	LEDR[1]	AF23
HEX0[2]	AC12	HEX6[2]	P3	HEX7[2]	L9	LEDR[2]	AB21
HEX0[3]	AD11	HEX6[3]	M2	HEX7[3]	L6	LEDR[3]	AC22
HEX0[4]	AE11	HEX6[4]	M3	HEX7[4]	L7	LEDR[4]	AD22
HEX0[5]	V14	HEX6[5]	M5	HEX7[5]	P9	LEDR[5]	AD23
HEX0[6]	V13	HEX6[6]	M4	HEX7[6]	N9	LEDR[6]	AD21
clk	N2	PS2_CLK	D26			LEDR[7]	AC21
rstn	G26	PS2_DAT	C24				

Table 4: The pinout on the EP2C35F672 FPGA used in this lab.

3.4 Demonstration

When the hardware works, demonstrate the implementation on the DE2 board, and show the waveform (the one generated using the TCL script). You can start on Task 2 if there is a queue to the lab assistant.

Keep the VHDL file, the waveform and the Programmer open, so you can quickly configure the FPGA board and show the waveform/VHDL file when it's your time to demonstrate.

4 Your Task 2: A Test Bench

In order to verify a system, a test bench is really useful. In the simulation in task 1, you had a predefined TCL script file, that tests everything. However, such an extensive script is not efficient to write in general. It's better to write a test bench in VHDL that generates the stimuli for the result.

Therefore, you will now write your own test bench, as a VHDL file. We recommend to use another text editor than Quartus for the test bench, e.g. the Modelsim built in editor. In this way you get a more intuitive separation between synthesizable code and the non-synthesizable test benches.

4.1 About Test Benches and Simulations

A system that should be checked is often called “design under test” (DUT) or “unit under test” (UUT). In this part, the `Lab1_KB` is the DUT.

A test bench typically contains two parts: A stimuli part (that generates inputs to the system), and a sanity check part (that checks the output from the system).

When synthesizing against the FPGA, the VHDL statements are translated to logic functions. In a simulation, the VHDL statements are executed. Each concurrent statement (outside a process) is executed as soon as any signal it depends on is changed. A process is executed as soon as any signal in the sensitivity list (`process(<sensitivity list>)...`) is changed. Within a process, the statements are executed sequentially, which opens up for advanced statements like `for` loops, file system I/O, pauses of the execution etc. We can use this in test benches. Many of those Tricks that can of course not be synthesized into an FPGA.

4.2 Your Test Bench

The test bench you are going to implement will only test one keyboard key (“4”).

The file structure of the test bench is given in Code 2. Some explanations follow.

The **library work**; gives you access to the other modules you have compiled (**work** is the default library where your stuffs goes).

The **entity** is empty, since you don't need any inputs/outputs. In Modelsim, you will look at the signals inside the module(s).

In **TOD01**, you should declare all signals needed for the test bench. This includes the I/O signals used from Table 1. In this test bench, no other signals are needed.

```

library ieee;
use ieee.std_logic_1164.all;
library work;

entity Lab2_KB_TB is
end entity;

architecture sim of Lab2_KB_TB is
  -- TODO1: Declare signals here.
  -- TODO2: Declare DUT component here.
begin
  -- Generate system clock/reset:
  clk <= not clk after 20 ns;
  rstn <= '0', '1' after 100 ns;

  process begin
    -- Generate stimuli (the scancode for key 4):
    PS2_CLK <= '1'; PS2_DAT <= '1';
    wait for 200 ns;
    -- First bit:
    PS2_DAT <= '0'; -- Start bit.
    PS2_CLK <= '0' after 10 us, '1' after 35 us;
    wait for 100 us;
    -- TODO3: More stimuli.

    -- Sanity check:
    assert HEX0 = "0011001" report "HEX0 failed" severity error;
    wait; -- do not restart the process
  end process;

  -- Instantiate DUT (TODO4: Complete this):
  DUT : Lab2_KB ...
end architecture;

```

Code 2: File structure of the test bench.

In **TODO2**, declare the Lab2_KB component. This looks exactly like the entity for the Lab2_KB, except the keyword “**entity**” is replaced with “**component**”. Copy-paste the entity from Lab2_KB, and change that. You can find an explanation for this in [R10.2.2].

The **clk** works like this: The row is executed every time **clk** changes, and forces the signal to flip after 20 ns. This gives a period of 40 ns, e.g. 50 MHz.

For this to work, **clk** must be initiated to '0' or '1'. This initialization should be done in the declaration part (**signal clk : std_logic := '0';**).

The **rstn** does not depend on another signal, so the assignment is made once only. First **rstn** is set to zero, and after 100 ns it's set to one.

The **process** has no sensitivity list, so it is executed once, but “loops”, e.g., when the execution comes to **end process**, it starts over again.

The **wait** statement pauses the execution for some (simulation) time. This time has nothing to do with the time it takes to simulate...

In **TODO3**, you can copy the code for the first bit, ten more times, and provide the bit pattern for the key “4”. Change the PS2_DAT bit pattern accordingly, see Table 3. The parity bit can be set to zero. This is a clumsy way to provide input stimuli, but it will do for this lab (however, see Section 4.4).

In the **sanity check**, the output of the DUT is checked. If the test **HEX0 = "0011001"** fails, this is reported, indicated as an error. This will be prompted in the transaction window in Modelsim.

The single **wait;** statement will wait forever, so the process will halt here, and never start over.

In the **TODO4**, you should complete the instantiation of Lab2_KB. You can read how to instantiate a component in [R10.2.1].

4.3 Simulation

Add the test bench to your Modelsim project and compile. Fix possible compilation errors and recompile until the compilation works.

If you have not demonstrated Part 1 yet, it's time to do so. Wait here until it's done.

Simulate with the test bench as the top module. Do not call the TCL file (U:/.../Stimuli.do) this time. Add signals, and run for 2 ms. Verify that no error message has appeared in the transcription, and the waveform behaves like expected.

4.4 Alternatives

Here are some alternatives to the proposed implementation.

4.4.1 For Loop Stimuli

When simulating sequential code, for loops can be useful, as illustrated in Code 3.

```
process
  constant key4 : std_logic_vector(1 to 11) := "...";
begin
  ...
  for i in 1 to 11 loop
    PS2_DAT <= key4(i);
  end loop;
```

Code 3: For ... loop solution for the Stimuli.

A for loop is simple for Modelsim to execute, but is typically not suitable to implement in hardware. If you try to synthesize a for loop using Quartus, it will probably not treat the loop in the same way as you intended. It may give a cryptic warning, or it may not.

4.4.2 Stopping the Simulation

If we want to run the simulation for a limited amount of simulation time, we can specify it, for instance `> run 100ns`. We can also run the simulation for an unlimited time, using the command `> run -a` (called “Run all”). If we want to stop the simulation we can press the “Stop” button in Modelsim. Modelsim will also stop the simulation if there are no more changes to perform to the signals. This can be done in the test bench by defining a “done” signal, that tells the VHDL system clock to stop.

```
signal done : boolean := false; In the "TODO1: Declare signals"
...
clk <= not clk after 20ns when not done; New
...
done <= true;
wait; } After the assert at the end of the process
```

Code 4: Automatic halt the simulation when done.

In the clock generation, “... when <test>” means that the signal is not assigned (keeps its value) when the <test> evaluates to false.

4.4.3 Good Looking Pass/Fail Outputs

The `assert` command can be changed to Code 5, printing either “OK” or “NOK” (not OK) in the transcript window.

```
if HEX0 = "0011001" then
  report "Result: OK" severity note;
else
  report "Result: NOK" severity error;
end if;
```

Code 5: Result printing.

5 Requirements to Pass

General requirements are given in the “Lab Demonstration” in the FAQ [3].

- The system must fulfill the list in top of section 3 (“Your Task 1”).
- The system must be synchronized to the 50 MHz clock (clk).

- The system must be simulated in Modelsim.
- You must understand the implementation.
- You must understand Table 3. *Each* student will most likely get a question on that part of the implementation (you do not have to know the scan codes by heart, but you need to understand how the 7-seg code works).
- You must have a working test bench. You may be asked to inject an error in the DUT, and show that your test bench finds the error.

If there is a queue to the lab assistant, take the moment to discuss the design. Read App. B again, and compare with the HEX0 assignment in your code.

References

- [1] <http://www.computer-engineering.org/ps2protocol/>
- [2] TSIU03: FAQ, section *Common Errors*.
- [3] TSIU03: FAQ, section *Lab Demonstration*.
- [4] <http://www.computer-engineering.org/ps2keyboard/scancodes2.html>

Appendix A PS/2 and Keyboard

In short, the communication can be summarized as:

- When a key is pressed/released, the keyboard sends a scan code.
- A scan code consists of one or several bytes.
- Each byte is sent serially over the PS/2 bus, according to the PS/2 interface.

A.1 The PS/2 Interface

The PS/2 bus has two wires, `PS2_CLK` and `PS2_DAT`, both are high ('1' in VHDL) when the bus is idle.

For each byte sent from the keyboard, there will be eleven “falling flanks” (transitions from '1' to '0') on `PS2_CLK`. On each falling flank, the current bit should be ready on `PS2_DAT`. The bits are:

- **0:** Start bit, always 0.
- **1-8:** Data bits, least significant bit (LSB) first.
- **9:** A parity bit (odd parity).
- **10:** Stop bit, always 1.

The waveform for a byte is illustrated in Fig. 3.

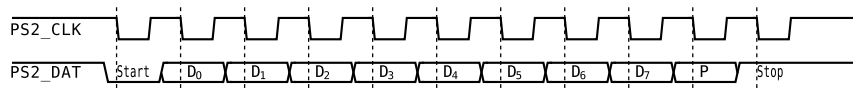


Figure 3: The PS/2 bus waveform, when the keyboard transmits a byte. A clock period is 50–100 μ s.

A.2 The Scan Codes

The scan codes can be divided into *make codes* and *break codes*, consisting of one or more bytes each.

When you press the key on the keyboard, the make code for that key is sent over the PS/2 bus. If you hold the key down for a while, the make code will start to repeat fast until you release the key. When the key is released, the break code is sent.

The make codes are a byte, sometimes preceded by the byte `E016`. The break code is the same, preceded by the byte `F0` (after the `E0` byte, if that is included). A few examples are shown in Table 5.

Key	Make code	Break code
1	16	F0, 16
P	4D	F0, 4D
(num pad) 4	6B	F0, 6B
(left arrow)	E0, 6B	E0, F0, 6B
(num pad) 3	7A	F0, 7A
(page down)	E0, 7A	E0, F0, 7A
(left ctrl)	14	F0, 14
(right ctrl)	E0, 14	E0, F0, 14

Table 5: Some examples of a few scan codes.
Values are given in hexadecimal form.

There are different sets of scan codes (make and break codes). The keyboard default to **set 2** upon start or reset. The complete scan codes set 2 are listed on for instance the web page [4].

A.3 Byte and Scan Code Synchronization

THIS IS NOT REQUIRED IN THIS LAB. But it can be a useful hint for the project.

In this lab, you just shift in the bits, and decodes what is on the shift register (`shiftreg`) in every moment. If an action is to be done when a key is pressed (which is the case for normal keyboard usage), then you must keep track of when an entire byte has arrived, and when an entire scan code has arrived.

One way to keep track of when an entire byte has arrived, is to reset the shift register into only ones, and then shift in. When the start bit (which is always zero) are shifted out, then you know that the entire byte has arrived, and you can “report” that byte, and also reset the shift register to only ones again.

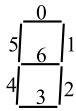
One way to keep track of the entire scan code, is to implement two flags, `e0` and `f0`. If the byte is “11100000” ($E0_{16}$), you set the `e0` flag. Handle the `f0` flag in a similar way. If the byte is something else, then the scan code is done. Report the content of the `e0` and the `f0` flag, together with the latest byte, and then reset the `e0` and the `f0` flags. The procedure is illustrated as:

- if `byte == 111000002` \implies `e0 <= '1'`;
- if `byte == 111100002` \implies `f0 <= '1'`;
- if something else \implies `scancode <= e0 & f0 & byte`; `e0 <= '0'`; `f0 <= '0'`;

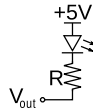
where `scancode` then is a 10 bit vector.

Appendix B 7-Segment Display

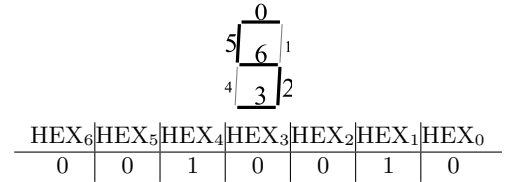
The 7-segments display (also called “hex display”, since it is suitable to display hexadecimal values on it), contains seven LEDs and seven resistors. The LEDs are indexed according to Fig. 4a. Since the LEDs are connected between +5V and the output of the FPGA, as depicted in Fig. 4b, the FPGA must output a '0' (0 V) to light a LED.



(a) The index of the LEDs in the 7-seg display.



(b) The electrical circuit. Set $V_{out}=0$ (0V) to light the LED.



(c) Example for the digit “5”. Bold lines are lit. Segment 1 and 4 are turned off.

Figure 4: The 7-segment display.

For example, if you want to show the digit 5, illustrated in Fig. 4c, you should set elements number 0, 2, 3, 5 and 6 to '0', and the other ones to '1'. Since the `HEX` signal are indexed (6 **downto** 0), and not (0 **to** 6), the bit vector should be "0010010", where the left most "0" is `HEX(6)` (the middle LED), and the right most "0" is `HEX(0)`.