TSIU03: Lab 2 - Keyboard

Petter Källström, Mario Garrido

August 24, 2021

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.

Contents

1 Lab Introduction

This lab contains three tasks.

Task 1 Here you will decode individual bytes from the PS/2 port. The byte will be shown on the LED bar. The keys 0 to 9 will recognized and presented on the 7-segment display. You will also test you design in a simple simulation.

Task 2 Here you will write your first VHDL test bench for your design.

Task 3 (optional) Here you will practice the concept of enable signals, by synchronizing the packages sent over the PS/2 protocol.

But first, of course, you must read up on how the PS/2 keyboard protocol works.

Hint: Before starting to code: Read the appendix "Common Error". When you have implemented everything, and it does not work, read the appendix again.

2 About Keyboard Interface

In short, the communication can be summarized as:
- When a key is pressed/released, the keyboard sends a scan code as a package.
- 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.

2.1 The PS/2 Interface: Sending One Byte at a Time

The PS/2 bus has the task to transmit bytes, typically from a device (keyboard) to a host (computer). It 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. 1.

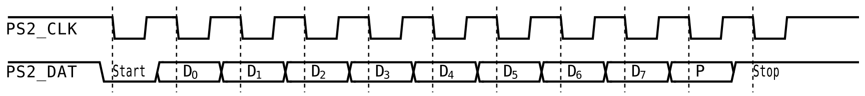


Figure 1: The PS/2 bus waveform, when the keyboard transmits a byte. A clock period is 50–100 µs.

Note: The PS/2 protocol also admit sending bytes to the keyboard, although this is not covered here. This feature is used e.g. to control the LEDs on the keyboard.

2.2 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 a 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 typically one byte, but sometimes preceded by the byte `E0`$_{16}$. 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 1.

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

There are different sets of scancodes. This lab manual describes the scancode **set 2**, which is default for the keyboards. You can google the complete scan code set 2.

3 About 7-segment Display and LED bar

We need to output the result, and to do so, we learn about the LED bar and 7-seg display.

3.1 LED Bars

You have already used the LEDs in the introduction lab. The hardware connection is depicted in Fig. 2, and works like FPGA pin → resistor → LED → Ground. Hence, when the FPGA outputs a `'1'` (2.5 V), an electric current is flowing through the diod and it is lit.

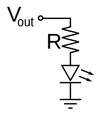


Figure 2: The LEDR and LEDG connections.

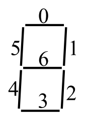
An unused pin on the FPGA is by default set to “weak pull-up”, using an internal resistor between 2.5 V and the FPGA pin (in addition of the external). That is, it can output a weak current to pull it up to 2.5 V. The intention is to provide a logical `'1'` for surrounding component’s input ports, while not being strong enough to cause trouble when connected to their output pins. As a consequence, the unused LEDs are dimly lit, while the used one(s) are clearly ON or OFF.

3.2 7-Segment Display

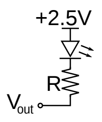
The 7-segments display (also called “hex display”, since it is suitable to display hexadecimal values on it), contains seven LEDs and seven resistors. To show a number, we must control each LED separately, to produce a pattern that looks like the number.

Each LED is connected just like those in the LED bar, but “upside down”, as depicted in Fig. 3b. That is - produce a current (light the LED) by forcing the FPGA pin to logical `'0'` (0 V). Remember this - The 7seg LED’s are *active low*, since they are lit by a `'0'`.

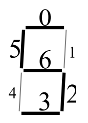
The LEDs are indexed according to Fig. 3a.



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



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



HEX_6	HEX_5	HEX_4	HEX_3	HEX_2	HEX_1	HEX_0
0	0	1	0	0	1	0

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

Figure 3: The 7-segment display.

For example, if you want to show the digit `5`, illustrated in Fig. 3c, you should set elements number 0, 2, 3, 5 and 6 to `'0'`, and the other ones to `'1'`. Since the `HEX` signals are indexed `(6 downto 0)` in this lab, 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)`.

4 Task 1: Keyboard Decoder

You will soon, with lots of help, create the module “Lab2_KB”. It shall:
- be implemented in VHDL.
- read key actions from a keyboard, via the PS/2 port.
- decode the keys `0` to `9` (above the letters — not those on the numerical keyboard).
- print the latest byte of the scan code on the red LED bar.
- print the latest 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.

► Start with creating the Quartus project (the FPGA is still EP4CE115F29C7). Create the empty VHDL file `Lab2_KB.vhd`.

4.1 Implementation

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). Hence, the shift register must be (at least) ten bit signal in order to read the latest byte (what happens otherwise?).

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

At every falling flank of `PS2_CLK`, shift in `PS2_DAT` 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	On the	`shiftreg(9 downto 0)` *after* the flank									
number	`PS2_DAT`	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. Do that in a combinational statement outside any process.

Table 3 shows a summary of the codes you need in order 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**, but you don't need to know the scan codes by heart. During the demonstration, *each* student will most likely get a question related to those bits.

Key/number	**Scan Code**	**7-seg**
1	`16 = 00010110`	`1111001`
2	`1e = 00011110`	`0100100`
3	`26 = 00100110`	`0110000`
4	`25 = 00100101`	`0011001`
5	`2e = 00101110`	`0010010`
6	`36 = 00110110`	`0000010`
7	`3d = 00111101`	`1111000`
8	`3e = 00111110`	`0000000`
9	`46 = 01000110`	`0010000`
0	`45 = 01000101`	`1000000`
Else	Else	`0000110`

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

Figure 4 depicts the structure of the hardware, and Code 1 depicts the file structure.

▶ Copy the code from Code 1, by manual typing. Solve the missing pieces (according to comments).

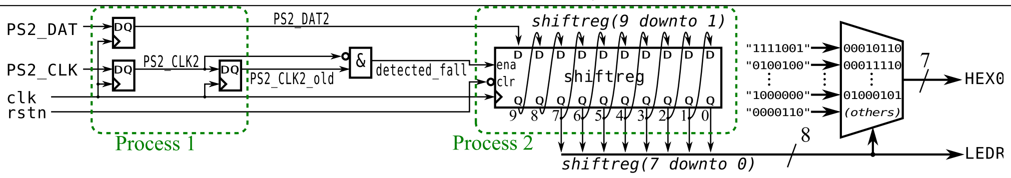


Figure 4: Hardware structure.

```
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 <= "0000000"; -- "8"
  HEX6 <= "0000010"; -- "6"
end architecture;
```

DFF: D-type flip flop (Swedish: "D-vippa")

Example for group 86

Code 1: File structure for the hardware in Fig. 4.

The module must have the inputs/outputs listed in Table 4. **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).

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 4: Port contents of the design

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 a `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. 4.

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. Use Table 3 to assign your lab group number to HEX7 and HEX6.

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

4.2 Simulate

▶ Simulate your design in Modelsim:

- Create a Modelsim project.
  - Project Name=Sim_KB .
  - Project Location=X:/TSIU03/Lab2/MSim .
- Compile the design (`> vcom ../Lab2_KB.vhd`). `If errors then goto 4.2.1; end if;`
- Load the design ("simulate it" - `> vsim work.lab2_kb`).
- Hint: Start the synthesis in Quartus while Modelsim loads the design. Then the synthesis is already done later.
- Add all your signals to the wave (`> add wave sim:/lab2_kb/*`).
- Generate stimuli, run, and check result via a predefined TCL script:
  `> do K:/TSIU03/Labs/Lab2_KB/Stimuli.do`
- Zoom in around the falling flanks of `PS2_CLK`, and check how the flank detection works.
- If there are any NOK's in the transcript, you have to figure out why, and fix the problem. Go back to recompilation.

The `.do` file contains the stimuli in this part. The result should look as depicted in Fig. 5. The stimuli gives a system clock of 5 MHz[1], 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.

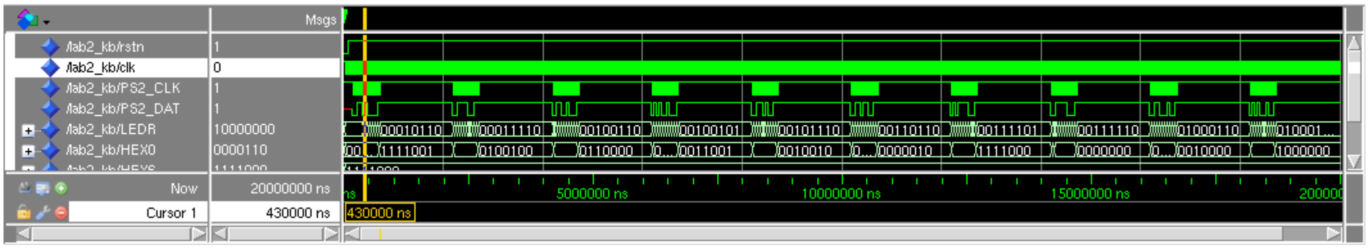


Figure 5: Simulation result.

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.

4.2.1 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 Appendix A.
- 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.

[1]The clock on the DE2-115 board is 50 MHz. To speed up the simulation, a much slower clock is used.

4.3 Test on the DE2-115 Board

Now it's time to test the design on the DE2-115 board. In order to do so, you need to enter a lot of pin names. Those are given in Table 5. The `rstn` signal (pin M23) is connected to KEY0 on the DE2-115 board.

Corona edition: Copying all those pins is a tedious work. Instead, $\boxed{\text{Assignments}\rightarrow\text{Import Assignments}}$. Use file `K:\TSIU03\Labs\Lab2_KB\some_pinns.csv`. Then, do pin assignments on the remaining pins.

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

Signal	Name	Signal	Name	Signal	Name	Signal	Name
HEX0[0]	G18	HEX6[0]	AA17	HEX7[0]	AD17	LEDR[0]	G19
HEX0[1]	F22	HEX6[1]	AB16	HEX7[1]	AE17	LEDR[1]	F19
HEX0[2]	E17	HEX6[2]	AA16	HEX7[2]	AG17	LEDR[2]	E19
HEX0[3]	L26	HEX6[3]	AB17	HEX7[3]	AH17	LEDR[3]	F21
HEX0[4]	L25	HEX6[4]	AB15	HEX7[4]	AF17	LEDR[4]	F18
HEX0[5]	J22	HEX6[5]	AA15	HEX7[5]	AG18	LEDR[5]	E18
HEX0[6]	H22	HEX6[6]	AC17	HEX7[6]	AA14	LEDR[6]	J19
clk	Y2	PS2_CLK	G6			LEDR[7]	H19
rstn	M23	PS2_DAT	H5				

Table 5: The pinout on the EP4CE115F29 FPGA used in this lab.

Hint: In the pin planner, you can assign the HEX0[6] to HEX0[0] by placing the cursor in the "Location" column, and typing "H22" (enter) "J22" (enter) "L25" etc.

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

▶ Program the DE2-115 board using the hand-on method, and verify the functionality.

If the connection to the DE2-115 cannot be established (does not detect the server), please contact the teacher. Perhaps the computer to which the DE2-115 is connected must be (re)started.

4.4 Demonstration

When the hardware works, demonstrate the implementation on the DE2-115 board, and show the waveform (the one generated using the TCL script Stimuli.do). During the demonstration, you also have to show your code. You must convince the lab assistant that it is beautiful enough (as easy as possible to read and understand). You must have a good understanding of what you have done.

You can start on Task 2 if there is a queue to the lab assistant, or you can take the moment to discuss the design. You can e.g. read the section about the 7-seg display again, and compare with the HEX0 assignment in your code.

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.

5 Task 2: VHDL Testbench

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.

5.1 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 `Lab2_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 constructions like `for` loops, file system I/O, pauses of the execution, text printouts etc. We can use this in test benches. Many of those tricks can of course not be synthesized into an FPGA.

5.2 Your Test Bench

The test bench you are going to implement will only test the keyboard key "4".

▶ Create a VHDL file, `Lab2_KB_TB.vhd` in the MSim folder. The file structure of the test bench is given in Code 2. Some explanations follow.

```vhdl
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 100 ns;
  rstn <= '0', '1' after 300 ns;

  process begin
    -- Generate stimuli (the scancode for key 4):
    PS2_CLK <= '1'; PS2_DAT <= '1';
    wait for 100 us;
    -- First bit: Start bit.
    PS2_DAT <= '0';
    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; -- forever. 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.

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

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

In **TODO1**, you should declare all signals needed for the test bench. This includes the I/O signals used from Table 4. In this test bench, no other signals are needed.

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 100 ns. This gives a period of 200 ns, i.e. 5 MHz. For this to work, `clk` must be initiated to `'0'` or `'1'`. This initialization is done in the declaration part as `signal clk : std_logic := '1';`.

The **rstn** does not depend on another signal, so the assignment is made once only. `rstn` is set to "`'0'` now, and then `'1'` after 100 ns".

The **process** has no sensitivity list, so it is executed once, but starts over again every time the executions comes to `end process`.

The **wait for** 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 5.5).

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 transcript 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].

5.3 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 Task 1 yet, it's time to do so. Wait here until it's done.

► Simulate with the test bench as the top module (load the test bench). If you had the simulation from task 1 open, you will be prompted to close it. Do not call the TCL file (K:/.../Stimuli.do) this time. Add signals, and run for 2 ms. Verify that no error message has appeared in the transcription, and that the waveform behaves as expected.

5.4 Demonstration

When you think your test bench works, it is time to demonstrate. You may optionally demonstrate with one or more of the alternatives below.

To pass this task, you must be able to show that your test bench actually works. You might be asked to inject an error in the Lab2_KB.vhd file, to show that your test bench reports it.

Test benches are in general hard to write, and often a real nightmare to read and understand. Therefore, it is important that you practice on writing them well, in order to be a good engineer.

5.5 Alternatives

Here are some alternatives to the implementation, that you can try.

5.5.1 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.

5.5.2 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 100ns when not done;
                                   New
...
done <= true; }  After the assert at the end of the process
wait;         }
```

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.

5.5.3 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.

6 Task 3: Synchronizing the Decoder (optional)

THIS IS OPTIONAL IN THIS LAB. But it can be useful for the project.

In Task 1, you just shift in the bits, and decodes what is on the shift register (`shiftreg`) in every moment. In normal keyboard usage, an action is performed once per make code. For this, you must keep track of when an entire byte has arrived, and which bytes belong to the same scan code.

For this, we also practice the concept of *enable signals*.

6.1 About Enable Signals

When two modules communicate, there use to be some kind of "hand shaking", where the modules agree on the transmission of some data (a package). Here we introduce the simplest version: Just an enable signal (often the signal name ends with "`_en`").

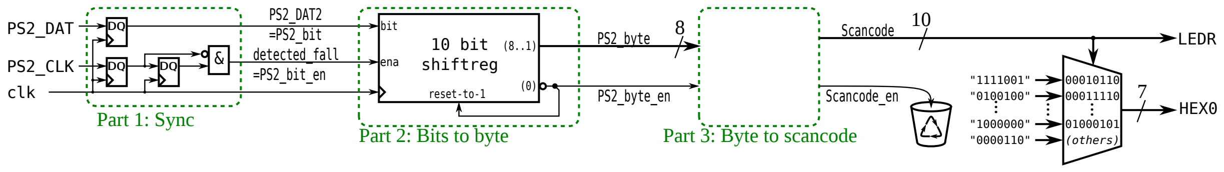


Figure 6: Hardware structure for Task 3.

Fig. 6 shows the structure of Task 3. We can see a hint that the `detecting_fall` is actually an enable signal telling there is *one* new bit on the `PS2_DAT2` wire. As we know, this is `'1'` during exactly one clock cycle for each bit. Two new names are suggested for those: `PS2_bit` and `PS2_bit_en` in the figure.

In the same way, when the PS/2 bus has sent a complete byte, we would like to know that there is *one* new byte available. This is done by a `PS2_byte_en` that is '1' during exactly one clock cycle for each byte. In this way, we avoid counting each byte several times.

Typically, we want a `scancode_en` signal to actually do something with the scancode, but in this lab we ignore this.

6.2 Task 3.1: Byte Synchronization (Part 2)

To start with, we solve the byte synchronization.

Normally, you implement a counter that counts to the 11 bits (0 to 10), and then starts over. Here we will instead use a clever solution, that some students came up with in 2009.

Initiate the shift register to all-'1'. Note that the start bit is always '0'. We can use this, and detect when the LSB of the shift register is '0'. Then we know that the completed byte is in bits 8 downto 1. The parity is in bit 9, which we ignore. As soon as the LSB is '0', we take this as a `PS2_byte_en`, and reset the shift register to all-'1' again. The stop bit has yet not come, but this is always '1', so when it comes, the entire shift register will still have a '1'.

Do you remember the timing table above? Table 6 is a modified version, where $\texttt{sr}_0$ is used to immediately reset the vector.

Flank number	On the PS2_DAT	`shiftreg(9 downto 0)` *after* the flank									
		$\texttt{sr}_9$	$\texttt{sr}_8$	$\texttt{sr}_7$	$\texttt{sr}_6$	$\texttt{sr}_5$	$\texttt{sr}_4$	$\texttt{sr}_3$	$\texttt{sr}_2$	$\texttt{sr}_1$	$\texttt{sr}_0$
1	Start bit	0	1	1	1	1	1	1	1	1	1
2	D_0	D_0	0	1	1	1	1	1	1	1	1
...											
9	D_7	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	0	1
10	Parity	P	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	0
+1 cc	Parity	1	1	1	1	1	1	1	1	1	1
11	Stop bit	1	1	1	1	1	1	1	1	1	1

Table 6: The shift register content directly *after* the clock flanks, and one clock cycle after resetting.

Declare three new signals:

- `PS2_byte` – 8 bits, directly assigned from `shiftreg(8..1)`.
- `PS2_byte_en` – directly assigned from `not shiftreg(0)`.
- `scancode` – 10 bits, assigned in “Part 3”.

When declaring `shiftreg`, you have to initiate it using `:= (others=>'1')`. You also need to modify Process 2, such that `rstn` and `PS2_byte_en` sets the shiftreg to `(others=>'1')` (asynchronously and synchronously respectively).

With this modification, the `PS2_byte` will flash for one clock cycle, which is too short for a human eye to see. We solve this with a temporary version of “Part 3”, where a register catches the data when enabled: A process that `scancode(7 downto 0) <= PS2_byte;` when `PS2_byte_en`. Don't forget `rising_edge(clk)` etc. Then, you use those eight scancode bits to assign `LEDR`, and to decode into `HEX0`.

You should be able to simulate this new code using either the Stimuli.do from Task1, or your test bench from Task2. If testing on the DE2-115 board, you will get brief flashes only for scancodes containing several bytes.

6.3 Task 3.2: Scancode Synchronization (Part 3)

A scancode consists of a byte code, possibly preceded by 0xE0 and/or 0xF0. This suggests that the “reported” scancode, is actually ten bits: The 8 normal bits, plus two that indicate if there was an E0 respectively F0 byte as well.

For this, you need two new signals:

- `E0` – flag
- `F0` – flag

In Part 3, whenever there is a `PS2_byte_en`, you should check the byte value.

- If `ps2_byte` = 11100000_2, then set the E0 flag to '1'.
- If `ps2_byte` = 11110000_2, then set the F0 flag to '1'.
- Else: Set `scancode` to `F0 & E0 & ps2_byte`, and reset the E0 and F0 flags to '0'. If `scancode_en` was used, this should be set for one clock cycle now.

The HEX decoder can still operate on the 8 LSBs of scancode, but the LEDR should get all 10. Hence, you need to modify the entity, and add two extra bits to LEDR. LEDR[8] connects to pin J17, and LEDR[9] connects to pin G17 (synthesize first, to simplify the pin assignment).

You can still simulate using Stimuli.do, but if you want to simulate using your VHDL testbench, you have to do some updates to match the longer LEDR.

Now try on the DE2-115 board. The keys 0 to 9 should still work, but without flashes during the transmit this time. Unlike before, however, you should see difference between left and right Ctrl, and between key press and release.

Discuss with supervisor on how to implement the enable signal here. Why is it synchronous in Part3 but not in Part2?

Challenge: Can you make the HEX0 show blank when no key is pressed?

Appendix A Common Errors

Some typical compile time errors:
- Typos (misspellings, forgotten “;” and so on).
- Mixing word length (e.g. `scancode = "00010110"`).
- Wrong syntax (e.g. “if-then” outside a process, or “when-else” inside a process).

Behaviour errors (might show up in simulation and/or hardware):
- Trying to use `falling_edge` with “`PS2_CLK`” while synchronizing to the FPGA clock.
- Inverting the `detected_fall` signal.

Some “Load” errors in simulation:
- No library ”work” – `> vlib work`
- “Can’t load the design” – Did you compile? Are sub modules correctly instantiated?
- Some essential signals in your design is missing – You have probably done something wrong, so they does not affect the result. You can resimulate “without optimization”.
- Still complaining on the repaired VHDL? – (Re)compile → restart/load.
- Anything – Consult compilation log.

Logic errors (hopefully found in simulations):
- Here you have to figure out how to read the signals, in order to locate the error.
- Still complaining on the repaired VHDL? – (Re)compile → restart/load.

A.1 Errors/warnings in Quartus

There are always a number of warnings in the synthesis. Some can be ignored, some should be considered.

Warnings to Ignore:
- ”Synopsys Design Constraints file ...” – This file is for advanced stuffs.
- ”Found ¡xx¿ output pins without output pin load capacitance ...” – For advanced timing.
- ”The Reserve All ... ” – All unused pins are weakly pulled high, since you didn’t specify anything else.
- ”Output pins are stuck at VCC or GND” – The HEX6 and HEX7 pins should be stuck. Look through the list.

Warnings to Consider:
- ”No exact pin location...” – Do pin placement of all pins.
- ”No clocks defined” – Well...
- ”Output pins are stuck at VCC or GND” – HEX0 and LEDR should *not* be stuck. If they are, you must find out why.

Common problems when programming the DE2-115 board:
- Make sure you send the program to the right board (in the current room).
- Can’t find the Ethernet Blaster server:
  - Make sure the DE2-115 board is on.
  - A (re)start of the computer connected to the board is needed.
- The [Start] button is grey in the programmer:
  - No hardware set – [Hardware Setup] etc.
  - The .sof file is missing – [Add File], check in the “output files”.

Common errors when testing on the DE2-115 board:
- Nothing works, not even the group number:
  - Did you send it to *this* board? Check the LEDs during the programming).
  - Did you do the pin placement, and then recompiled?
- The group number works, but nothing else (but it worked in the simulation)
  - Do you send the latest .sof file? If you have copied or moved the project folder/files, it might compile old files. Check the file list in Quartus.
  - After the last change, did you resynthesize in Quartus?
- Something else? Look into the synthesis warning list.