Simple PS/2 Interface

ADVANCED LOGIC SYNTHESIS FOR ELECTRONICS
A.L.S.E.

Introduction

This Interface was created after the examination of an existing VHDL module (source unknown). Since this module was poorly coded and had no test bench, I wrote this small project in a couple of hours to let it be used as a real interface, and for teaching purpose.

I kept the original architecture (after some fixings in the VHDL style) under the name "**Plain_Wrong**" and wrote a new one under the name "**ALSE_RTL**". Obviously, only the latter should be used ! "*Plain_wrong*" is left to compare both solutions and understand what errors were made (I inserted a few hints).

I have then developed a more sophisticated controller (with bi-directional communication), which should probably be adopted in a serious design. If you are interested, please contact ALSE at mailto:info@alse-fr.com.

As usual (especially for such simple functions), there is more complexity and more value in the **simulation code** (self-testing VHDL Test Bench) than in the RTL code. The test bench does include a behavioural model for a PS/2 Keyboard (transmit only).

However, the structure used for the RTL description is suitable and efficient for much more complex tasks. It should be also very simple to add extra features to the code provided. As this is often the case, the "clean" solution is as compact (or more) than the "dirty" solution, it has more features (Parity + Overflow + Stop Bit errors detection), and it is easier to enhance or modify.

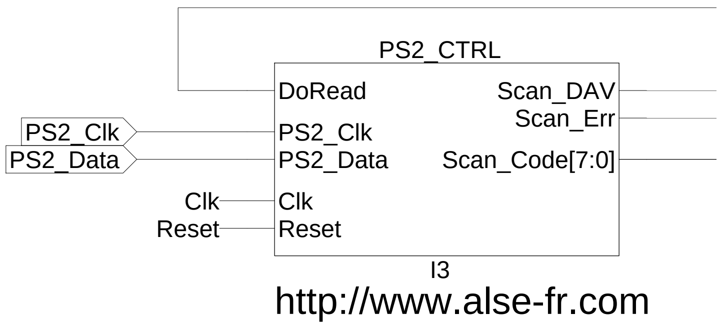
Features

- Fully synchronous solution: one clock, one global reset.
- Suitable for PS/2 peripherals (keyboard, mouse…).
- PS/2_Clock is filtered digitally.
- Parity is verified
- Overflow detection (when the host does not read the value received in time).
- ALSE_RTL2 architecture only: timeout protection (against incomplete or wrong frames)
- Simplified interface: the host does not send data to the peripheral, it can only receive data. This is viable for a Keyboard (though the LED indicators NumLock, ScrollLock, CapsLock can only be changed by the host). This poses more problem with the recent mice which require a complex dialog with the host before operating in the expected way.
  For details about the PS/2 serial bus format, please refer to some of the abundant literature widely available. There is a good article from Adam Chapweske available at :
  http://govschl.ndsu.nodak.edu/~achapwes/PICmicro/PS2/ps2.htm
  in which information can be found on the connectors and cabling, the hardware interface, the protocol, details on mouse and keyboard data, etc…

Note: we have left the digital filter on PS2_Clk though appropriate hardware (Schmidt trigger) should be used to ensure that the signal has steep enough edges, in which case a simple re-synchronizing + synchronous edge detection (2 Flip-Flops) would suffice. The larger of the shift register used as a digital filter could/should be reduced when the system clock is very slow (like 1 MHz or less).

This module can also be used as a transparent "spy", to listen to the traffic on a PS/2 port.

Module Description



The graphic symbol of the Controller is represented on the right, with inputs on the left side, and outputs on the right side.

Reset	is used inside the module as an asynchronous and global reset.
Clk	is the sole clocking source for all the Flip-Flops (fully synchronous design).
PS2_Clk, and **PS2_Data**	are the two signal lines of the PS/2 Interface. They are inputs (only) to our module.
DoRead	is an input pulsed by the user when the Data out code is read. This clears the Scan_DAV bit.

The outputs are:

Scan_DAV	Goes to 1 when a word has been received. Remains there until DoRead is asserted then goes low at the next clock cycle.
Scan_Err	Is set when the data received is incorrect (parity, stop…) or when there is an overflow (user has not read the previous code in time). This flag is automatically cleared when the reception of a new character begins.
Scan_Code	(8 bits) is the data word received. The value remains stable until the next word is received.

Cosmetic note: the naming “Scan_xxx” is very appropriate when a keyboard is attached, but makes less sense when another device (mouse) is attached.

Implementation

Please refer to the widely available PS/2 documentation to fully understand the specific serial protocol. You will notice that, when the host is only reading (and does not send data), the basic function to accomplish is to sample the Data line just after each Clock line’s falling edge.
This determines a word made of 11 bits, including in this order a Start bit (=0), the 8-bits code (appearing LSB first), an Odd parity bit, and a Stop bit (=1). See the waveform next page.

The most compact solution one could imagine would be asynchronous, and we will indeed avoid this bad practice! Our solution is fully synchronous. We have implemented a small filter on the PS2_Clk line: we sample it at the system clock rate and wait for 8 consecutive identical samples before accepting the value. The number “8” is a generic parameter and can be easily changed. For values above 8, a different and more efficient implementation of this filter (based on a counter) should be used. When ad-hoc hardware is used (Schmidt-trigger logic on the PS2 lines) this could be simplified into a standard synchronous edge detector. Note however hat a bounce (or glitch) on the PS2_Clk line would be catastrophic (it would desynchronize the frame).

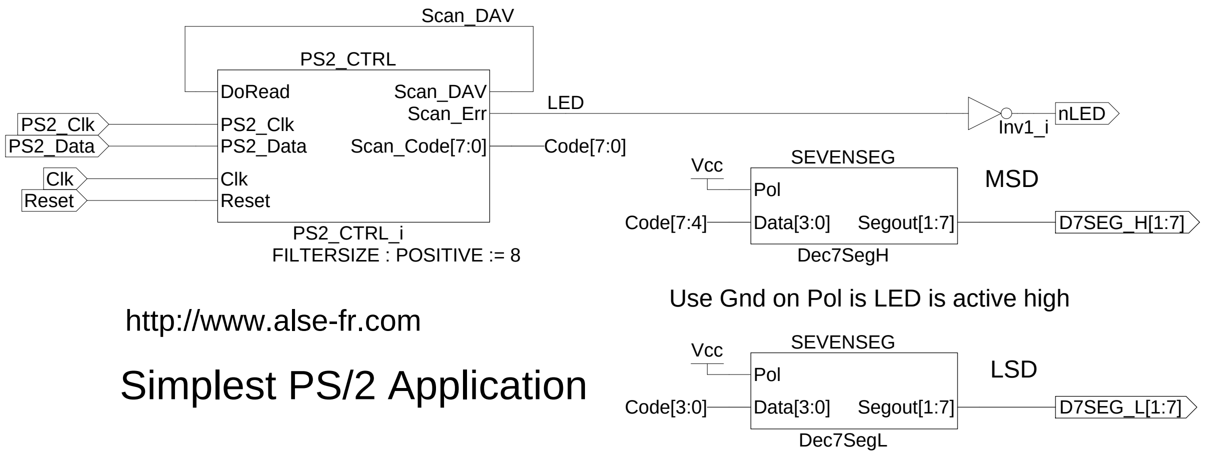
The reception of the frame is quite simple since we use a 2-states FSM.
If you wish to make this module specific to the AT keyboard, you can add a few lines of code to : check if the code received is x”E0” or x”F0”, then arm a flag and do not generate DAV, else, test if the flag is armed, then ignore the code and disarm the flag. This way, your application will only get the key pressed information (a.k.a. “make codes”).

When you connect a device, make sure (1) that there is a Pull-Up resistor somewhere on each PS/2 line (both Clock and Data) and (2) that the resistors’ value is adequate. A too high value can create lazy rising edges, which may in turn create problems on the FPGA side. You may find that your keyboard already has pull-up resistors wired.

Simplest Application

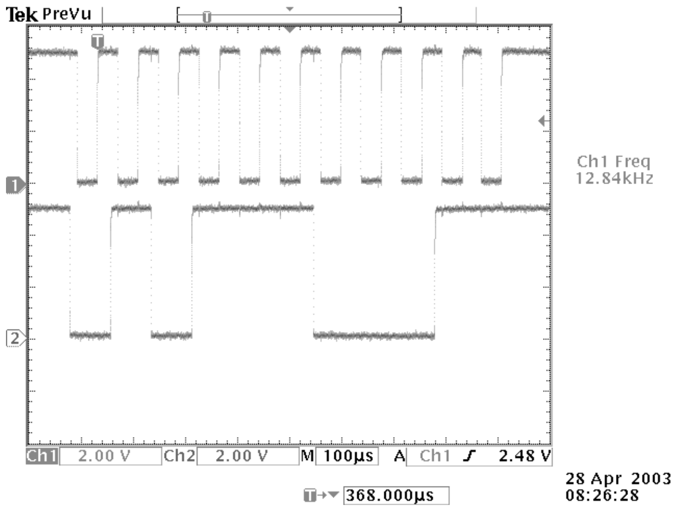
For very simple applications, or when the user modules are capturing the arriving Data without delay, it is possible to connect Scan_DAV to DoRead.

Since the output code remains stable, it is then possible to build a very simple application:



The VHDL code for the schematics above is included, and can be tried on any FPGA/CPLD board. It was tried on an old Lattice CPLD board and on an XESS board (XS40-010XL) and worked directly on both.

If you connect a keyboard to PS2_Clk and PS2_Data (and power the keyboard with +5Vdc), pressing or releasing a key should display the corresponding scan code. Verifying first these signals with an oscilloscope is definitely a good idea, if only to verify the rising edges' slope.



Note that lighting the keyboard's LED indicators (CapsLock etc…) would require the PS/2 Controller to actually **send** data to the keyboard. This feature (*"host to device" communication*) is *not available* in this simple controller. This would require bi-directional I/Os for PS2_Data and PS2_Clk (and appropriate control logic indeed).

Note also that when you depress a key, the keyboard issues two codes in sequence: x"F0" followed immediately by the key's scan-code. Since we display the codes on the LEDs when they arrive (without extra delay), you will probably miss the F0 and only observe the scan code.

A more sophisticated application could ignore scan codes preceded by x"F0", or handle a matrix of 128 bits where each bit represents the status of one key (pressed, not pressed), etc…

If you connect a mouse, then you need to handle the incoming codes in a much different manner, and this is outside the scope of this document. Just be aware that you will also need to *send information* to the mouse, which this version of the controller cannot do.

This trivial application can help verify that the hardware is correctly connected and in working order.

PS/2 Controller Source code *(correct architecture only)*

```
-- PS2_Ctrl.vhd
-- ------------------------------------------------
--   Simplified PS/2 Controller  (kbd, mouse...)
-- ------------------------------------------------
-- Only the Receive function is implemented !
-- (c) ALSE. http://www.alse-fr.com

library IEEE;
  use IEEE.STD_LOGIC_1164.all;
  use IEEE.Numeric_std.all;

-- --------------------------------------
    Entity PS2_Ctrl is
-- --------------------------------------
  generic (FilterSize : positive := 8);
  port( Clk       : in  std_logic;  -- System Clock
        Reset     : in  std_logic;  -- System Reset
        PS2_Clk   : in  std_logic;  -- Keyboard Clock Line
        PS2_Data  : in  std_logic;  -- Keyboard Data Line
        DoRead    : in  std_logic;  -- From outside when reading the scan code
        Scan_Err  : out std_logic;  -- To outside : Parity or Overflow error
        Scan_DAV  : out std_logic;  -- To outside when a scan code has arrived
        Scan_Code : out std_logic_vector(7 downto 0) -- Eight bits Data Out
        );
end PS2_Ctrl;

-- --------------------------------------
    Architecture ALSE_RTL of PS2_Ctrl is
-- --------------------------------------
-- (c) ALSE. http://www.alse-fr.com
-- Author : Bert Cuzeau.
-- Fully synchronous solution, same Filter on PS2_Clk.
-- Still as compact as "Plain_wrong"...
-- Possible improvement : add TIMEOUT on PS2_Clk while shifting
-- Note: PS2_Data is resynchronized though this should not be
-- necessary (qualified by Fall_Clk and does not change at that time).
-- Note the tricks to correctly interpret 'H' as '1' in RTL simulation.

  signal PS2_Datr  : std_logic;

  subtype Filter_t is std_logic_vector(FilterSize-1 downto 0);
  signal Filter    : Filter_t;
  signal Fall_Clk  : std_logic;
  signal Bit_Cnt   : unsigned (3 downto 0);
  signal Parity    : std_logic;
  signal Scan_DAVi : std_logic;

  signal S_Reg     : std_logic_vector(8 downto 0);

  signal PS2_Clk_f : std_logic;

  Type   State_t is (Idle, Shifting);
  signal State : State_t;

begin

Scan_DAV <= Scan_DAVi;

-- This filters digitally the raw clock signal coming from the keyboard :
--   * Eight consecutive PS2_Clk=1 makes the filtered_clock go high
--   * Eight consecutive PS2_Clk=0 makes the filtered_clock go low
-- Implies a (FilterSize+1) x Tsys_clock delay on Fall_Clk wrt Data
-- Also in charge of the re-synchronization of PS2_Data

process (Clk,Reset)
begin
  if Reset='1' then
    PS2_Datr  <= '0';
    PS2_Clk_f <= '0';
    Filter    <= (others=>'0');
    Fall_Clk  <= '0';
  elsif rising_edge (Clk) then
    PS2_Datr <= PS2_Data and PS2_Data; -- also turns 'H' into '1'
    Fall_Clk <= '0';
    Filter   <= (PS2_Clk and PS2_CLK) & Filter(Filter'high downto 1);
    if Filter = Filter_t'(others=>'1') then
```

```
      PS2_Clk_f <= '1';
    elsif Filter = Filter_t'(others=>'0') then
      PS2_Clk_f <= '0';
      if PS2_Clk_f = '1' then
        Fall_Clk <= '1';
      end if;
    end if;
  end if;
end process;


-- This simple State Machine reads in the Serial Data
-- coming from the PS/2 peripheral.

process(Clk,Reset)
begin

  if Reset='1' then
    State     <= Idle;
    Bit_Cnt   <= (others => '0');
    S_Reg     <= (others => '0');
    Scan_Code <= (others => '0');
    Parity    <= '0';
    Scan_Davi <= '0';
    Scan_Err  <= '0';

  elsif rising_edge (Clk) then

    if DoRead='1' then
      Scan_Davi <= '0'; -- note: this assgnmnt can be overriden
    end if;

    case State is

      when Idle =>
        Parity  <= '0';
        Bit_Cnt <= (others => '0');
        -- note that we dont need to clear the Shift Register
        if Fall_Clk='1' and PS2_Datr='0' then -- Start bit
          Scan_Err <= '0';
          State <= Shifting;
        end if;

      when Shifting =>
          if Bit_Cnt >= 9 then
            if Fall_Clk='1' then -- Stop Bit
              -- Error is (wrong Parity) or (Stop='0') or Overflow
              Scan_Err  <= (not Parity) or (not PS2_Datr) or Scan_DAVi;
              Scan_Davi <= '1';
              Scan_Code <= S_Reg(7 downto 0);
              State <= Idle;
            end if;
          elsif Fall_Clk='1' then
            Bit_Cnt  <= Bit_Cnt + 1;
            S_Reg <= PS2_Datr & S_Reg (S_Reg'high downto 1); -- Shift right
            Parity <= Parity xor PS2_Datr;
          end if;

      when others => -- never reached
        State <= Idle;

    end case;

  end if;

end process;

end ALSE_RTL;
```

Unitary Test Bench Source code

```
-- KBD_TST.vhd
--------------------------------------------------
-- Self-Testing PS/2 Keyboard Test bench (c) ALSE
--------------------------------------------------
-- Author : Bert Cuzeau - ALSE - http://www.alse-fr.com
-- No part of this code can be used without the prior
-- written consent of ALSE.
-- This is a simplified model.
-- Compile with '93 option (hex constants...)

  use STD.Textio.all;
library IEEE;
  use IEEE.Std_Logic_1164.all;
  use IEEE.Numeric_std.all;
  use IEEE.Std_Logic_Textio.all;

-------------------
entity Kbd_tst is
  constant Period    : time := 40 ns; -- 25 MHz System Clock
  constant BitPeriod : time := 60 us; -- Kbd Clock is 16.7 kHz max
end;
-------------------

-------------------
architecture Test of Kbd_tst is
-------------------

Component PS2_Ctrl
port( Clk       : in  std_logic; -- System Clock
      Reset     : in  std_logic; -- System Reset
      Kbd_Clk   : in  std_logic; -- Keyboard Clock Line
      Kbd_Data  : in  std_logic; -- Keyboard Data Line
      DoRead    : in  std_logic; -- From outside when reading the scan code
      Scan_Err  : out std_logic; -- To outside if wrong parity or Overflow
      Scan_DAV  : out std_logic; -- To outside when a scan code has arrived
      Scan_Code : out std_logic_vector(7 downto 0) ); -- scan code
end component;

signal Clk       : std_logic := '0';
signal Reset     : std_logic;
signal Kbd_Clk   : std_logic := 'H';
signal Kbd_Data  : std_logic := 'H';
signal DoRead    : std_logic := '0';
signal Scan_Err  : std_logic;
signal Scan_DAV  : std_logic;
signal Scan_Code : std_logic_vector(7 downto 0);

signal Succeeded : boolean := true;

type Code_r is
record
  Cod : std_logic_vector (7 downto 0);
  Err : Std_logic;  -- note: '1' <=> parity error
end record;

type Codes_Table_t is array (natural range <>) of Code_r;
constant Codes_Table : Codes_Table_t -- if you need more codes: just add them!
                     := ( (x"A5",'0'), (x"5A",'0'), (x"00",'0'), (x"FF",'0'),
                          (x"12",'0'), (x"34",'0'), (x"56",'1'), (x"56",'0'),
                          (x"78",'0'), (x"BC",'0') );

-- in Verilog, the function below is just : ^V    ;-)
function Even (V : std_logic_vector) return std_logic is
  variable p : std_logic := '0';
begin
  for i in V'range loop p := p xor V(i); end loop; return p;
end function;

-------------------
begin
-------------------
```

```
-- Instanciate the UUT (PS/2 Controller) :
UUT: PS2_Ctrl
  port map ( Clk       => Clk,
             Reset     => Reset,
             Kbd_Clk   => Kbd_Clk,
             Kbd_Data  => Kbd_Data,
             DoRead    => DoRead,
             Scan_Err  => Scan_Err,
             Scan_DAV  => Scan_DAV,
             Scan_Code => Scan_Code );

-- System Clock & Reset

Clk <= not Clk after (Period / 2);

Reset <= '1', '0' after Period;

-- Keyboard sending Data to the Controller

Emit: process

  procedure SendCode ( D   : std_logic_vector(7 downto 0);
                       Err : std_logic := '0') is
  begin
    Kbd_Clk  <= 'H';
    Kbd_Data <= 'H';
    -- (1) verify that Clk was Idle (high) at least for 50 us.
    -- this is not coded here.

    wait for (BitPeriod / 2);
    -- Start bit
    Kbd_Data <= '0';
    wait for (BitPeriod / 2);
    Kbd_Clk  <= '0'; wait for (BitPeriod / 2);
    Kbd_Clk  <= '1';
    -- Data Bits
    for i in 0 to 7 loop
      Kbd_Data <= D(i);
      wait for (BitPeriod / 2);
      Kbd_Clk  <= '0'; wait for (BitPeriod / 2);
      Kbd_Clk  <= '1';
    end loop;
    -- Odd Parity bit
    Kbd_Data <= Err xor not Even (D);
    wait for (BitPeriod / 2);
    Kbd_Clk  <= '0'; wait for (BitPeriod / 2);
    Kbd_Clk  <= '1';
    -- Stop bit
    Kbd_Data <= '1';
    wait for (BitPeriod / 2);
    Kbd_Clk  <= '0'; wait for (BitPeriod / 2);
    Kbd_Clk  <= '1';
    Kbd_Data <= 'H';
    wait for (BitPeriod * 3);
  end procedure SendCode;

begin -- process Emit
-----
  Wait for BitPeriod;

  -- Send the Test Frames
  for i in Codes_Table'range loop
    SendCode (Codes_Table(i).Cod,Codes_Table(i).Err);
  end loop;
  if not Succeeded then
    report "End of simulation : " & Lf &
           "  There have been errors in the Data / Err read !"
      severity failure;
  else
    report Lf & "  SUCCESSFULL End of simulation : " & Lf &
           "  There has been no (zero) error !" & Lf & Ht
      severity note;
    report "End of Simulation" severity failure;
  end if;

end process Emit;
```

```
-- Host reading (& verifying) Data :

Host: process
  variable L : line;
  variable Index : natural := 0;
begin
  wait until Scan_DAV='1';
  wait for 300 * Period;
  DoRead <= '1';
  write (L,now,right,12);
  write (L,Ht&"Scan code read (hex) = ");
  hwrite (L,Scan_Code);
  if Scan_Err='1' then
    write (L,ht&" >>> Scan_Err <<<");
  end if;
  -- Compare with the original Data-Error :
  if   (Scan_Code /= Codes_Table(Index).Cod)
    or (Scan_Err  /= Codes_Table(Index).Err) then
    Succeeded <= False;
    write (L, Ht&"!!! Mismatch !!!");
  end if;
  Index := Index + 1;
  writeline (output,L);
  wait for Period;
  DoRead <= '0';
end process Host;

end Test;
```

When the simulation runs, the following messages appear in the simulation transcript:

```
--#    732380 ns  Scan code read (hex) = A5
--#   1602380 ns  Scan code read (hex) = 5A
--#   2472380 ns  Scan code read (hex) = 00
--#   3342380 ns  Scan code read (hex) = FF
--#   4212380 ns  Scan code read (hex) = 12
--#   5082380 ns  Scan code read (hex) = 34
--#   5952380 ns  Scan code read (hex) = 56  >>> Scan_Err <<<
--#   6822380 ns  Scan code read (hex) = 56
--#   7692380 ns  Scan code read (hex) = 78
--#   8562380 ns  Scan code read (hex) = BC
--# ** Note:
--#   SUCCESSFULL End of simulation :
--#   There has been no (zero) error !
--#
--#    Time: 8760 us  Iteration: 0  Instance: /kbd_tst
--# ** Failure: End of Simulation
--#    Time: 8760 us  Iteration: 0  Instance: /kbd_tst
```

Simulation script *(ModelSim)*

```
# kbd.do
# PS/2 Controller Simulation Script
# -----------------------------------

vlib work

vcom -93 PS2_Ctrl.vhd
vcom -93 Kbd_Tst.vhd

vsim kbd_tst
view structure
view signals
do wkbd.do

puts "(c) ALSE - http://www.alse-fr.com"
run -a
```

Launch ModelSim, “**Change Directory**” to the project’s directory, and type “`do kbd.do`”.

VHDL Code for the Simplest Application top-level

```
-- PS2SIMPL.vhd
-- ------------------------------------------------
-- Simplest PS/2 application
-- ------------------------------------------------
-- (c) ALSE - http://www.alse-fr.com
--
LIBRARY ieee;
  USE ieee.std_logic_1164.ALL;

-- ------------------------------------------------
    Entity PS2SIMPL is
-- ------------------------------------------------
      Port ( Clk      : In  std_logic;
             Reset    : In  std_logic;
             D7SEG_L  : Out std_logic_vector (1 to 7);
             D7SEG_H  : Out std_logic_vector (1 to 7);
             PS2_Data : In  std_logic;
             PS2_Clk  : In  std_logic;
             nLED     : Out std_logic );
end PS2SIMPL;

-- ------------------------------------------------
    Architecture SCHEMATIC of PS2SIMPL is
-- ------------------------------------------------
  component SEVENSEG
     Port (    Data : in  std_logic_vector (3 downto 0);
                Pol : in  std_logic;
             Segout : out std_logic_vector (1 to 7) );
  end component;
  component PS2_CTRL
  Generic ( FILTERSIZE : POSITIVE := 8 );
     Port (     Clk : in  std_logic;
             DoRead : in  std_logic;
            PS2_Clk : in  std_logic;
           PS2_Data : in  std_logic;
              Reset : in  std_logic;
          Scan_Code : out std_logic_vector (7 downto 0);
           Scan_DAV : out std_logic;
           Scan_Err : out std_logic );
  end component;

  signal Gnd,Vcc : std_logic;
  signal LED     : std_logic;
  signal DoRead  : std_logic;
  signal Code    : std_logic_vector (7 downto 0);

begin
  Gnd <= '0';  Vcc <= '1';

  PS2_CTRL_i : PS2_CTRL
    Generic Map ( FILTERSIZE => 8 )
    Port Map ( Clk=>Clk, Reset=>Reset, DoRead=>DoRead,
               PS2_Clk=>PS2_Clk, PS2_Data=>PS2_Data,
                Scan_Code=>Code, Scan_DAV=>DoRead, Scan_Err=>LED );

  -- Note: use Pol=>Gnd if display is active high type.
  Dec7SegL : SEVENSEG
     Port Map ( Data => Code(3 downto 0), Pol=>Vcc, Segout => D7SEG_L );

  Dec7SegH : SEVENSEG
     Port Map ( Data => Code(7 downto 4), Pol=>Vcc, Segout => D7SEG_H );

  nLED <= not LED; -- Note: remove the "not" if nLED is active high

end SCHEMATIC;
```

Note on VHDL style: in VHDL ’93, it is possible to get rid of the component declaration, and avoid Gnd and Vcc in the port map. On the other hand, the style above is very conservative and will be accepted by all the synthesis tools.

Conclusion

This small example is very simple and easy to understand, but it is based on principles that are used for the most complex projects. The simulation Test Bench includes a number of nice tricks (like for writing efficiently in the simulation transcript, for self-testing, for the automatic dimensioning of tables, etc…).

If you decide to build a more sophisticated controller, you should find that the code provided is a good basis. The self-testing test bench will also help you verify the non-regression.

The benefits of a good methodology (especially the efforts spent on the simulation) are visible, even on a simple design as this one (which is basically a shift register).
Even the simplest code is prone to errors, and taking them out as early as possible in the design flow is a great advantage.

If you need a more sophisticated controller, do not hesitate to contact us!

For other examples or other IPs, visit our Web site: http://www.alse-fr.com/English.

Caveat Students!

If you have an assignment that is precisely a PS/2 interface, it would do you no good to simply copy and paste from this project, even in the unlikely case your teacher isn't aware of it.

If you persist in using this material, I strongly suggest that you very carefully ignore the RTL code included (not even a glance, if possible) and the section which describes its implementation.

You might try to understand and use the testbench, design your own solution, verify that it simulates correctly, then only have a look at the provided RTL code. You may afterwards find reasons to modify your own code; this is fine.

Since you have received a little help, you should add at least a timeout to return to Idle state when the synchronization is lost (this is very easy) and handle the x"F0" code.

Then, if you do not already have to, you could also add the communication in the opposite direction (host to keyboard). This is slightly trickier.

Appendix A
Scripts for specific FPGA/CPLD Boards

For the Lattice ispLSI / ispGDS Demo board, the free software “ispLEVER” is sufficient to completely implement this simple design. The project is provided under the Lattice directory.
The synthesis is done by the OEM version of Synplify (but you can select Leonardo instead).
Note that the device selected is rather old (ispLSI1032) and requires some manual changes in the ispTools installation to be accepted. We have a document indicating how to make these changes, which is freely available upon request.

For the Xess XC40-010XL board, the device is not supported directly by recent versions of ISE. This is why we included two **Synthesis scripts** that we’ve designed to ensure the proper VHDL synthesis using either Leonardo Spectrum or Synplify. The “Project File’ for Synplify is in fact a Tcl script… There is not much know-how in this script, see next page.

For Leonardo Spectrum, the script is a bit trickier. It has the advantage over a Project File (which is not a Tcl script) to be path-independent. We use a few Tcl instructions to pick the current path and use it to grab the VHDL source files:

```
puts ""
puts " Synthesis script for Leonardo Spectrum. (c) ALSE. http://www.alse-fr.com "
puts ""

# Very simple script for Leonardo Spectrum, (c) ALSE.
# Author : Bert Cuzeau
# Here, we target # a Xilinx XC4010XLpc84
# From Leonardo, simply use "File > Run Script" command.

# Global Clock timing constraints (12.5 MHz)
set register2register 80.0
set input2register 80.0
set register2output 80.0
set input2output 80.0

set synthdir [pwd]
set srcdir [file join ${synthdir} ..]
set fitdir $synthdir

_gc_read_init
_gc_run_init

set input_file_list "${srcdir}/sevenseg.vhd
                     ${srcdir}/ps2_ctrl.vhd
                     ${fitdir}/ps2simpl.vhd "
set module ps2simpl
set wire_table 4010xl-3_avg
set chip TRUE
set macro FALSE
set optimize_for area
set -hierarchy auto
set maxdelay 0
set hierarchy_auto TRUE
set hierarchy_preserve FALSE
set hierarchy_flatten FALSE
set report brief
set edif_write_arrays TRUE

set output_file "${fitdir}/PS2simpl.edf"

set novendor_constraint_file FALSE
set part 4010xlPC84
set process 3
set pack_clbs FALSE
set target xi4xl

_gc_read
_gc_run
```

`Leo_Script.tcl` : **Synthesis script for Leonardo Spectrum**

```
#-- Synplicity Project file

add_file -vhdl -lib work "../sevenseg.vhd"
add_file -vhdl -lib work "../ps2_ctrl.vhd"
add_file -vhdl -lib work "ps2simpl.vhd"

#implementation: "Xess"
impl -add Xess

#device options
set_option -technology XC4000XL
set_option -part XC4010XL
set_option -package PC84
set_option -speed_grade -3

#compilation/mapping options
set_option -default_enum_encoding onehot
set_option -symbolic_fsm_compiler 1
set_option -resource_sharing 1

#map options
set_option -frequency 12.000
set_option -disable_io_insertion 0
set_option -force_gsr auto

#simulation options
set_option -write_verilog 0
set_option -write_vhdl 0

#automatic place and route (vendor) options
set_option -write_apr_constraint 0

#set result format/file last
project -result_file "./ps2simpl.edf"
impl -active "Xess"
```

`Synplify.prj` : **Synplify Project File**

The Xess sub-directory contains a specific top-level and the constraints file (ps2simpl.ucf) for the XC40-010XL board. The top level does add a few pins and set them to a fixed logic level of ‘1’ to ensure that the peripherals locate on the board (Ram and µP) will not create conflicts. Moreover, since this board has only one LED digit (except when plug into an extension board), we display only the low nibble of the scan code received. Download the bit file, plug a keyboard, and type !
The constraints are :

```
#NET CLK  LOC=P13; # CLOCK FROM EXTERNAL OSCILLATOR

# LED DRIVER OUTPUTS
NET D7SEG_L(1)  LOC=P19; # a
NET D7SEG_L(2)  LOC=P23; # b
NET D7SEG_L(3)  LOC=P26; # c
NET D7SEG_L(4)  LOC=P25; # d
NET D7SEG_L(5)  LOC=P24; # e
NET D7SEG_L(6)  LOC=P18; # f
NET D7SEG_L(7)  LOC=P20; # g

NET NLED        LOC=P70; # Parallel Port status S3

# PS/2 Keyboard Connector
NET PS2_CLK     LOC=P68;
NET PS2_DATA    LOC=P69;

# DATA BITS FROM THE PC PARALLEL PORT
NET RESET       LOC=P46; # We use D2 as Reset

# RAM CONTROL PINS
NET nOE         LOC=P61; # ACTIVE-LOW OUTPUT ENABLE
NET nOE         PULLUP;
NET nCE         LOC=P65; # ACTIVE-LOW CHIP ENABLE
NET nCE         PULLUP;
# MICROCONTROLLER PINS
NET RST         LOC=P36; # uC ACTIVE-HIGH RESET
NET RST         PULLUP;
```

PS2simpl.ucf for Xess XC40-010XL