Interfacing Keyboard with Basys3 Board

Supervisor : Dr. Farhan Khan
Muhammad Anas & Hussain Mustansir

January 22, 2024

Department of Computer Science
Habib University

1 Introduction

Introduction: Interfacing an external USB keyboard input in Basys3 FPGA is quite simple. Basys3 FPGA board has a built-in USB port for connecting the keyboard at pin B-17. This pin provides 11 bit input to the FPGA board out of which 8 bits give the information about the particular key pressed. Each key having its particular ASCII value(mentioned below in 1.1) is used to capture key pressed on a qwerty keyboard.

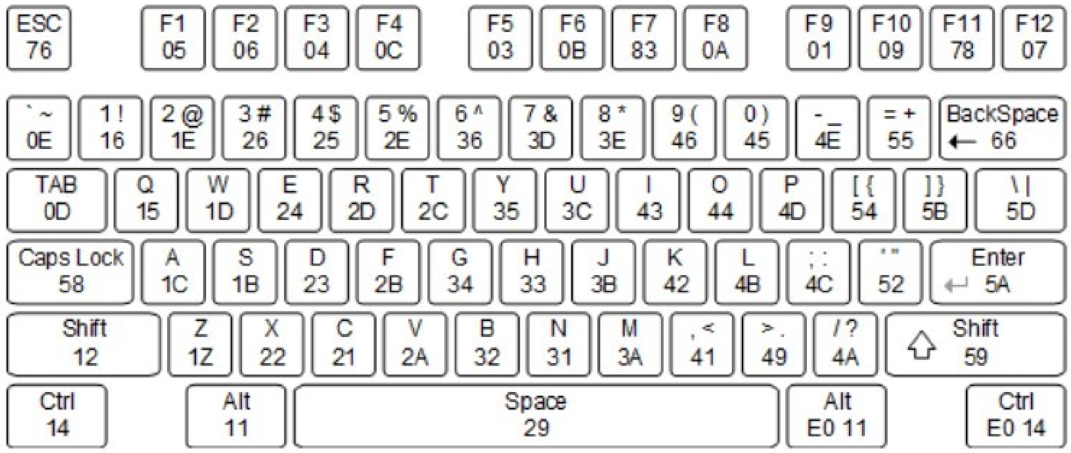


fig:1.1

2 Explanation:

To start off we define the keyboard port in constraints files i.e. I/O ports (this has usually extension of .xdc in vivado) The keyboard pin is named in I/O ports as B17 alongwith clock of keyboard at pin C17 as follows :

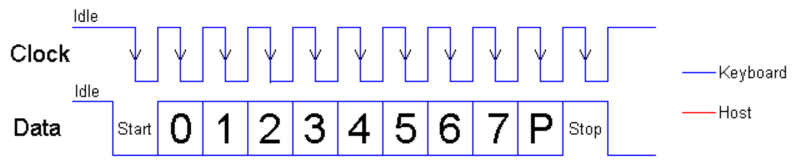
```
set_property PACKAGE_PIN C17 [get_ports ps2c]
set_property IOSTANDARD LVCMOS33 [get_ports ps2c]
set_property IOSTANDARD LVCMOS33 [get_ports ps2d]
set_property PACKAGE_PIN B17 [get_ports ps2d]
```

The ps2d pin receives the input of any keypressed from the keyboard, and we can use it in the project. In our top module we define the keyboard clock and keyboard data which we will use in our keyboard module as follows:

```
module main_control(
input logic clk,
input logic start,
input logic reset,
input logic ps2d,
input logic ps2c, )
```

note: since this example uses .sv extension of for the files so we have used logic in input logic ps2d.

Next step comes the keyboard Module. In the main module just call your keyboard module with the necessary variables. As a keyboard has a lot of keys, and each key has a different ASCII value. In this example we are only interested in a few keys, which are Q, A, W, S, P, L, O, K, space bar, and enter key. The ASCII values of these keys elaborated in fig 1.1. Pressing a key on the keyboard sends a 11 bit signal to the FPGA board, out of which 8 bits are the ASCII value of the key pressed. The other 3 bits are used for other purposes, which we are not interested herein.



3 Sample Code:

```
‘timescale 1ns / 1ps
// Keyboard module__
module keyboard_1(
    input wire clk, reset,
    input wire ps2d, ps2c,
    output wire start_ball, KEY_UP_4, KEY_DOWN_4, KEY_UP_3, KEY_DOWN_3, KEY_UP_2, KEY_DOWN_2, KEY_UP
    output reg key_release,
    output  reg state
    );

// Declare variables
wire [7:0] dout;
wire rx_done_tick;
supply1 rx_en; // always HIGH
reg [7:0] key;


// Instantiate models
// Nexys4 converts USB to PS2, we grab that data with this module.
// ps2_rx module(defined below) extracts the useful 8 bits from 11 bit ps2 data and returns in dout
ps2_rx ps2(clk , ps2d, ps2c, rx_en, rx_done_tick, dout);
```

```
// Sequential logic
always @(posedge clk)
begin
    if (rx_done_tick)
    begin
        key <= dout; // key is one rx cycle behind dout
        if (key == 8'hf0) // check if the key has been released
            key_release <= 1'b1;
        else
            key_release <= 1'b0;
    end
end

// Output control keys of interest
//assign reset = ((key == 8'h2D) & !key_release) ? ~reset : reset; // 1C is the code for 'R'
assign start_ball = (key == 8'h29) & !key_release; // 1C is the code for 'Space Bar'
assign KEY_UP_4 = (key == 8'h4D) & !key_release; // 1C is the code for 'P'
assign KEY_DOWN_4 = (key == 8'h4B) & !key_release; // 1F is the code for 'L'
assign KEY_UP_3 = (key == 8'h44) & !key_release; // 20 is the code for 'O'
assign KEY_DOWN_3 = (key == 8'h42) & !key_release; // 42 is the code for 'K'
assign KEY_UP_2 = (key == 8'h1D) & !key_release; // 1C is the code for 'W'
assign KEY_DOWN_2 = (key == 8'h1B) & !key_release; // ___1F is the code for 'S'
assign KEY_UP_1 = (key == 8'h15) & !key_release; // 20 is the code for 'Q'
assign KEY_DOWN_1 = (key == 8'h1C) & !key_release; // 32 is the code for 'A'

always @(posedge clk ) begin
     begin
        // Check if the key is pressed and assign the state accordingly
        case(key)
            8'h5A : state <= 1'b1; // Set state to 1 when Enter key is pressed
            8'h66 : state <= 1'b0; // Set state to 0 when Backspace key is pressed
            default: state <= state; // Maintain state otherwise
        endcase
    end
end


endmodule

//ps2_rx module

module ps2_rx(
    input wire clk, reset,
    input wire ps2d, ps2c, rx_en,
    output reg rx_done_tick,
    output wire [7:0] dout
);

// Symbolic state declaration
localparam [1:0]
    idle = 2'b00,
    dps  = 2'b01, // data, parity, stop
    load = 2'b10;

// Signal declaration
reg [1:0] state_reg, state_next;
reg [7:0] filter_reg;
wire [7:0] filter_next;
reg f_ps2c_reg;
wire f_ps2c_next;
reg [3:0] n_reg, n_next;
reg [10:0] b_reg, b_next;
wire fall_edge;


// Filter and falling-edge tick generation for ps2c
always @(posedge clk, posedge reset)
if (reset)
begin
    filter_reg <= 0;
    f_ps2c_reg <= 0;
end
else
begin
    filter_reg <= filter_next;
    f_ps2c_reg <= f_ps2c_next;
end

assign filter_next = {ps2c, filter_reg[7:1]};
assign f_ps2c_next = (filter_reg==8'b11111111) ? 1'b1 :
                     (filter_reg==8'b00000000) ? 1'b0 :
                     f_ps2c_reg;
assign fall_edge = f_ps2c_reg & ~f_ps2c_next;

// FSMD state & data registers
always @(posedge clk, posedge reset)
if (reset)
begin
    state_reg <= idle;
    n_reg <= 0;
    b_reg <= 0;
```

```
end
else
begin
    state_reg <= state_next;
    n_reg <= n_next;
    b_reg <= b_next;
end

// FSMD next-state logic
always @*
begin
    state_next = state_reg;
    rx_done_tick = 1'b0;
    n_next = n_reg;
    b_next = b_reg;
    case (state_reg)
        idle:
            if (fall_edge & rx_en)
            begin
                // Shift in start bit
                b_next = {ps2d, b_reg[10:1]};
                n_next = 4'b1001;
                state_next = dps;
            end
        dps: // 8 data + 1 parity + 1 stop
            if (fall_edge)
            begin
                b_next = {ps2d, b_reg[10:1]};
                if (n_reg==0)
                    state_next = load;
                else
                    n_next = n_reg - 1;
            end
        load: // 1 extra clock to complete the last shift
            begin
                state_next = idle;
                rx_done_tick = 1'b1;
            end
    endcase
end

// Output
assign dout = b_reg[8:1]; // Data bits

endmodule
```

4 Explanation of the sample code

The above mentioned code uses clock signals (ps2c) and data (ps2d) declared in the I/O ports. Whereafter the keyboard module is called in the main module. The module keyboard contains declaration of : ps2c(clock signals),
ps2d(the data received from the user)
reset signals
dout: An 8-bit wire to hold the data from the PS/2 interface.
rx_done_tick: A wire indicating the completion of a PS/2 reception.
key: An 8-bit register to hold the processed key value.

Thereafter ps2_rx module extracts and stores the the 8-bit information of the key pressed in dout. Which is, whereafter, stored in "key" if previous key is released on positive edge of clock. Thus the "key" is used to match the ASCII code of the key pressed and store in corresponding variable. Thus high output on any variable indicates the key is pressed and, wherefore, can be used to perform various actions.

References (Hover below to be direct to relevant links)

1. Basys 3 Keyboard Demo - Digilent Reference
2. Keyboard data form
3. Foosball Game on Basys 3
4. Youtube Reference
5. Our Github Repository For Full Code