EE3 FPGA Lab 2 Theoretical Background

1. Asynchronous reset

In this lab, all your modules must use asynchronous, active-low, reset signal. This type of reset is a common approach in practice, due to 2 main reasons:

1. High-to-low transition: Most logic families can sink more current (to GND) than they can source (to VDD). The reset signal is required simultaneously in many places in chip, which requires a high fan-out. Therefore, the ability to transit from high to low is preferable.
2. Asynchronous signal forces an immediate simultaneous reset through the chip. This reset is regardless of the clock, which might be in different modes (toggling or idle) in different clock/power domains in the chip. Whereas in the synchronous reset case, the first condition to begin a reset is that the clock will reach its active edge.

One must keep in mind that asynchronous reset is prone to metastability issues, therefore an asynchronous reset must be provided for a long enough time.

To avoid confusion, the active low asynchronous reset port is named **rstn**.

2. PS2 interface

The PS2 protocol is a simple two-wire scheme that uses serial transmission to transmit the data to the board. On our board, this 2-wire PS2 communication from the USB port is provided to FPGA pins C17 and B17 by the USB-HID controller, as depicted in Figure 1. More information on the PS2 protocol can be found here and in the Basys3 reference manual.

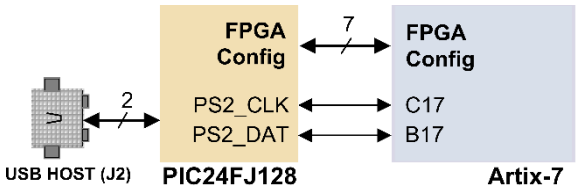

Figure 1 - BASYS3 board connections from FPGA to the USB port

From the moment user presses (and not yet releases) a key, the keyboard transmits a sequence of bits called "make-code". This sequence is carrying a special 1-Byte scan-code of the keyboard key, and each key on the keyboard is given a unique "scan-code", see Figure 2. If the user is holding the key, the "make-code" packet is retransmitted every 100ms.

When a key is released, the keyboard transmits a 2-Byte "break code" sequence consisting of the first byte being 0xF0, followed by the 2nd byte being the scan-code of the key that was released.

↑ key up code

For example, consider the following scenario, where user presses the letter 'a':

1. User presses the 'a' key
2. Keyboard sends "make code" (which is 0x1C for the 'a' key) serially. The keyboard keeps sending the "make code" every 100ms until the user releases the key.
3. User releases the 'a' key
4. Keyboard sends the "key up" code 0xf0 serially
5. Keyboard sends the "break code" (which is 0x1C for the 'a' key) serially

Byte - 8 bits.

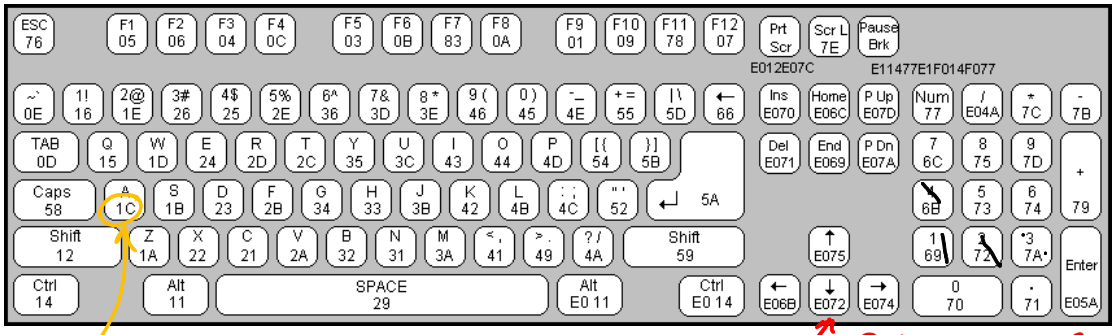

Figure 2 - Keyboard keys and their scan-codes in hexadecimal representation.

To transmit the sequence of bytes, the keyboard first forces the *DATA* line low to create the start bit. Bits are transmitted using the **negative edge of *CLK*** for synchronization. This is illustrated in Figure 3. The *DATA* signal changes state when the *CLK* signal is high, and *DATA* is valid for reading on the falling edge of *CLK*. This *CLK* signal is slow, 20-30KHz, and this clock is kept constantly high when idle. **Namely, this clock doesn't always toggle!**

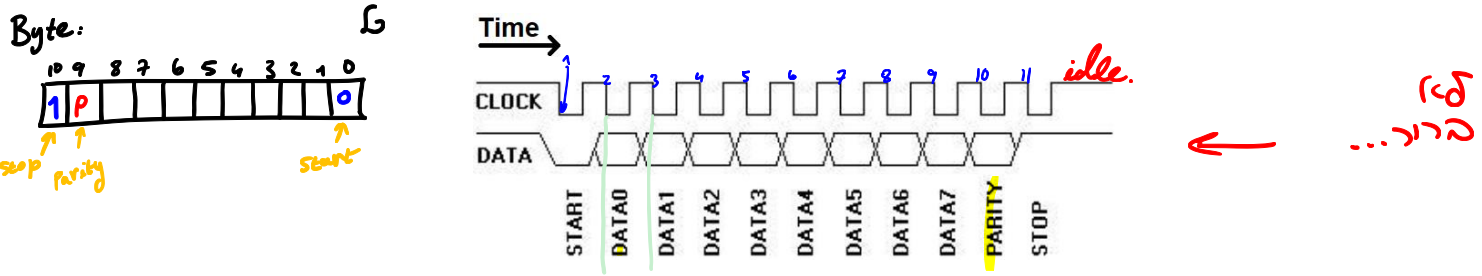

Figure 3 - Timing diagram of PS2 data and clock signals

The basic operation of a PS2 interface is as follows:

1. On the falling edge of *CLK*, use a shift register to capture each bit of data.
2. When enough bits have been sent (i.e. when start bit and scan-code bits are captured and the parity bit is on the PS2Data line, no need to wait for the deterministic stop bit), you can look at the scan-code and decide what to do:
   2.1. If the received Byte is neither an extended key qualifier (0xE0) nor a "key up" (0xF0), you know that this is a key-pressed event indication.
   2.2. If the received byte is "keyp up" (0xF0) then the following packet will tell the scan-code of the key that was released

For example, when "CAPS-lock" key is released, and your design receives the incoming bits into a 22-bit long shift register (input to the shift-register is better be through its **MSB**[1]), then after 22 clock cycles the following content will be in the shift register:

From Ps2Data →

R[21]	R[20]	R[19]	R[18]	R[17]	R[16]	R[15]	R[14]	R[13]	R[12]	R[11]	R[10]	R[9]	R[8]	R[7]	R[6]	R[5]	R[4]	R[3]	R[2]	R[1]	R[0]
1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0
Stop	Par	Data[7:0]=0x58								Start	Stop	Par	Data[7:0]=0xF0								Start

Important Information

1. The PS2 uses a simple error detection scheme called **Odd Parity**. The sender asserts the parity bit value such that the total number of high bits in data and parity together is odd. When the receiver receives the data frame, he checks odd-parity condition to detect 1/3/5/7/9-bit flip errors.

[1] Maintaining the shift-register in this manner (pushing new bit into the MSB) helps you in receiving the packets in a correct order for a convenient further processing. If you'd push new bits into the LSB of the shift register, you will probably need to reverse the content of this register.

2. Some keys on the keyboard (like the arrow keys) are special in the sense that they send an additional code “E0” ahead of the scan code. Such keys are called extended keys. When an extended key is released and “E0 F0” code is sent followed by the scan code. So, regardless of the key, the last two chunks of data when a key is released will be “F0” and the scan code.
3. When your keyboard will be connected to the BASYS3 board, **no lights should be lit on the keyboard.** The keyboard LEDs, such as caps-lock, are lit after the host PC sends messages to keyboard, however in our lab you are not supposed to send anything to the keyboard. But you should be able seeing an orange led blinking next to the USB port on the BASYS board.
4. Although the keyboard clock and data wires are bidirectional, we will only be using it as inputs for this lab. [Typically writing to the keyboard is used to reset, turn on the various indicator lights, etc.] The bidirectionality of the line is provided by the **open-collector** driver at each terminal connected to the wire.
5. Constraints file: you will need to enable the following pins (by uncommenting them and verifying that they are mapped to correct IO ports in your Verilog top module):

FPGA pin names	IO port name (from top module in Verilog)	Description
U18	btnC (for reset)	for the active-high reset
U2, U4, V4, W4	an[0:3]	Active-low digit selection (7-segment)
V7	dp	Active-low dot (7-segment)
W7,W6,U8,V8,U5,V5,U7	seg[0:6]	Active-low 7 segments (7-segment)
U16, E19, U19	led[0:2]	Active high user signals to the LEDs
W5	clk	System clock 100MHz
C17, B17	PS2Clk, PS2Data	PS2 protocol communication lines (fed to FPGA from USB port via HID controller)

6. The 2 signals PS2Clk and PS2Data require the 3rd (pull-up-enable) constraint line to be uncommented for them. The purpose is to drive the high-Z output of keyboard into a logical 1.
```
set_property PULLUP true [get_ports PS2Data]
set_property PULLUP true [get_ports PS2Clk]
```

7. In case you are getting an error related to the keyboard clock during the Place & Route step in Xilinx ISE (which says something like "Clock IOB/ clock component is not..."), please add the following line to your XDC file and then re-run the Place & Route step again:
```
set_false_path -from [get_ports PS2Clk] -to [get_clocks sys_clk_pin]
```
where *PS2Clk* is the name of keyboard clock signal in your design.

3. VGA interface

A VGA monitor scans the screen row by row, starting at the upper left corner and ending at the lower-right corner. This scan order also known as Raster-scan. The scan is synchronized using two signals, called *Hsync* (horizontal synchronization), and *Vsync* (vertical synchronization). The *Hsync* signal is normally high and it goes low to signal the end of the line scan. The *Vsync* signal is normally high and it goes low to signal the end of the whole screen scan, meaning that the scan should re-start back in the upper left corner of the screen. In addition to these synchronization signals, the VGA screen is fed with the color of the current pixel, which consists of red, green and blue color channels.

Screen Synchronization

We will focus on a 800×600 pixel screen display. A pixel clock will be 50 MHz. To get 800 pixels horizontally, a horizontal synchronization frequency of approximately 48.07 KHz is required. This corresponds to approximately 1040 clock periods of the pixel clock. During the first 800 clock periods of the pixel clock for a row, visible pixels can be displayed; however, the last 240 clock periods for the row are called the "retrace period" or "blanking region" where nothing is displayed while the beam retraces back to the left of the screen. To get 600 pixels vertically, a vertical synchronization frequency of 72 Hz is required. This corresponds to approximately (1040x)666 clock periods of the pixel clock. This can be thought of as generating 600 visible rows followed by 66 blank rows during which the beam retraces back to the top of the screen. This is illustrated in Figure 4. A summary of numbers of the pixel clocks for each region can be found here.

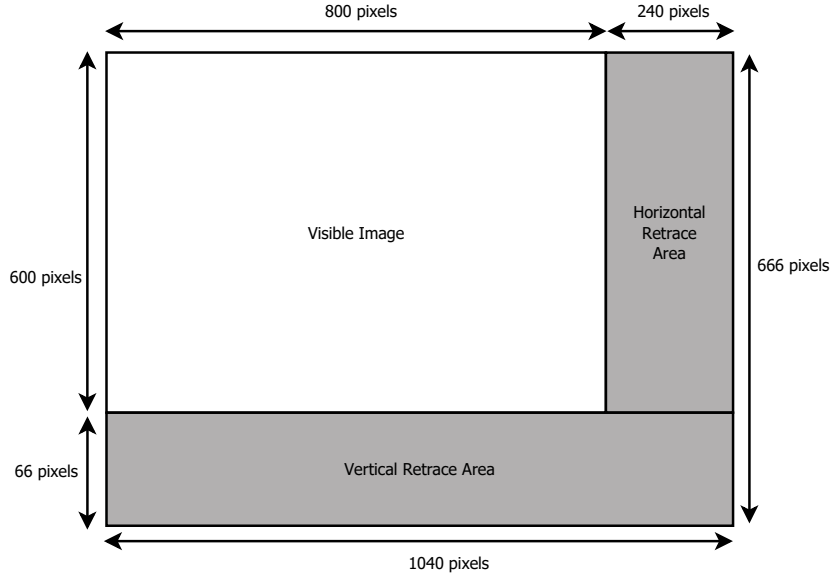

Figure 4 - VGA standard display regions for 800x600 resolution

Figure 5 shows the timing of the *Hsync* signal. It is made low starting on the 856th pixel clock period for the row and made high again on the 976th pixel clock period. During the first 800 pixel-clock periods, visible pixels are generated. During the last 240 pixel-clock periods, nothing is generated on the screen, and a black color is expected from the pixel source.

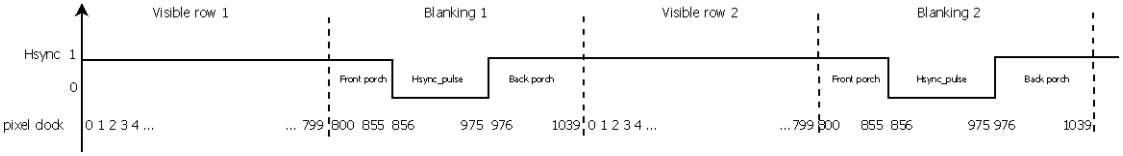

Figure 5 - Horizontal sync timing example

Figure 6 shows the relationship between the *Vsync* signal and the *Hsync* signal. The digits represent the line count. As stated earlier, the first 600 lines are displayed while the last 66 lines correspond to the retrace period for the beam to get back to the top left corner of the screen.

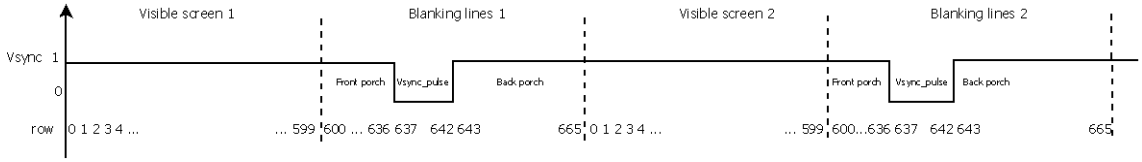

Figure 6 - Vertical sync timing example

Colors: The VGA connector is analog, meaning that the red, green and blue colors are must be expressed by continuous voltages on 3 separate wires. However the Artix-7 FPGA chip has digital output pins dedicated for the R,G,B colors (namely, each color is expressed by 4 bits). This gives a total of 16 values levels per color-channel, and a total of 4096 different colors possible to output. To convert the digital color channels to analog ones, a simple DAC circuit is implemented on the Basys3 board, which takes the digital bits and performs a weighted sum of voltages. See Figure 7 for illustration.

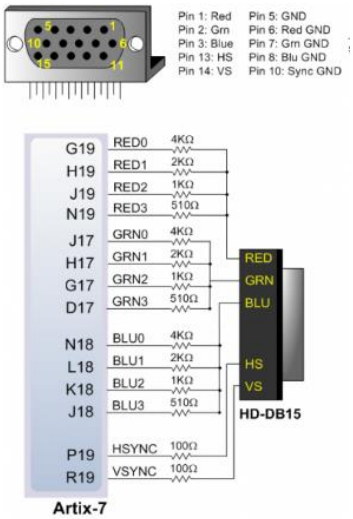

Figure 7 - The conversion of the digital FPGA signals into analog signals for the VGA connector

In this lab, you may find this table useful to determine the digital RGB color codes. Pay attention that you don't have 256 (8b) but only 16 (4b) levels per channel in your implementation. Think about ways to properly down-sample the 8bits to 4bits color code.

Constraints file: to connect the design to the VGA pins, go to you xdc file and uncomment the lines that define the pins that appear in Table 1. Note that additional pins should be enabled to receive a clock, reset etc.

FPGA Pin names	Verilog output port names
G19,H19,J19,N19	vgaRed[0:3]
N18,L18,K18,J18	vgaBlue[0:3]
J17,H17,G17,D17	vgaGreen[0:3]
P19	Hsync
R19	Vsync

Table 1 - Artix-7 VGA related pin names