ECE 385

Fall 2020

Experiment 8

SOC with USB and VGA Interface in SystemVerilog

Zhicong Fan/Xin Jin
AB2/Online
Yucheng Liang

Introduction:

In this lab, we used an external keyboard to implement a ball game. Most of the skeleton code is given to us, we just to need implement the bidirectional data-reads and data-writes between the interface of MAX3421E and NIOS II USB driver. To implement USB, we have to use SPI core which is designed for connect microprocessors to connect in embedded systems. After that, we need to implement all the sub modules in the high level lab8.sv so that the all the modules work as a whole.

Written Description:

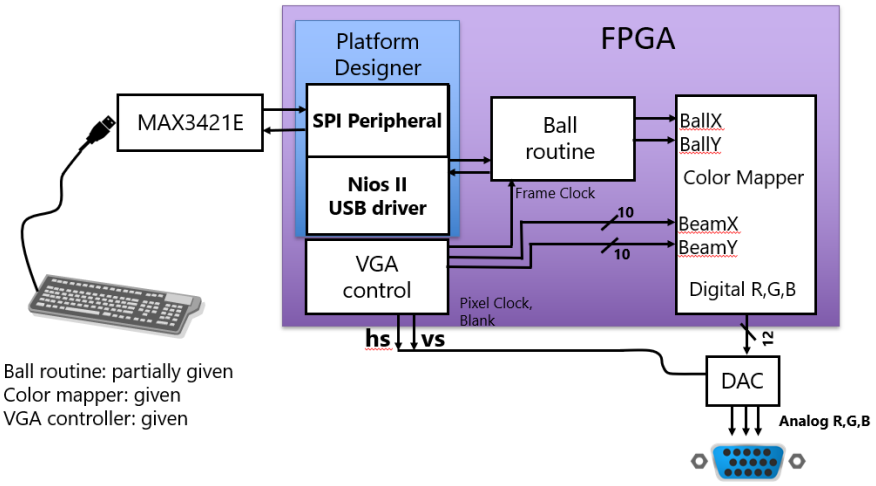
Description:

Basically, in lab 8, a connection between MAX3421E and Nios II USB driver is the core. To make sure the connection work, an SPI peripheral should be implemented first in the platform designer. And then for modules parts, the lab 8 system contains the following:

1. A lab8_soc that represents the NIOS II processor.
2. A VGA_controller that sketches the corresponding VGA signals on the screen.
3. A color-mapper that decides what the shape of the object is and its colors.
4. A ball that controls the ball routine with respect to different cases.
5. A Hex driver that displays what keys are pressed on the keyboard.

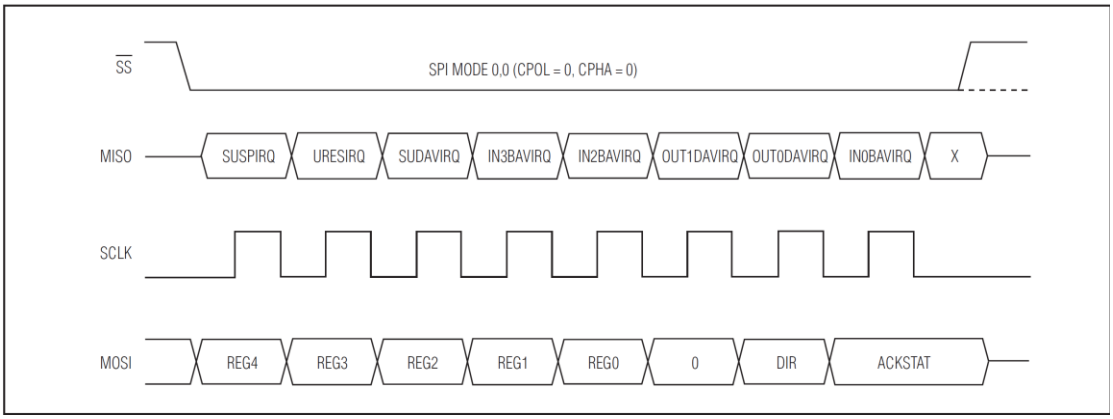
More about interactions among NIOS, MAX3421E and VGA:

First and foremost, as before, the NIOS in the platform designer generates the basic addresses of the key elements such as the processor, the keycode, and SPI protocol. With these basic addresses, in the C code parts, main.c could read correct inputs from the keyboard and match that with the symbol on keycode table. Also, in MAX3421E.c file, it could send data packet that communicates between the processor and the USB driver: whether the USB is reading or writing. For the VGA, it only interacts with color_mapper and has nothing to do with NIOS II processor. VGA controller and color_mapper directly control the output on the VGA screen



SPI Protocol:

The SPI Protocol is used to connect microprocessors to other different kinds of control devices, such as off-chip sensor, conversion, memory and control devices. However, in this lab, we only use this to connect the microprocessor NIOS II with the control device. In the SPI, it mainly use two data lines which are mosi(master out slave in) and miso(master in slave out). In mosi, the data flows from the master to slave and for the second one, vice versa. In our design, the clock of SPI is directly linked to the original clock. In the below graph, we are able to see how the MAX3421E implements the SPI interface.



Functions Implemented:

1. BYTE SPI_wr(BYTE data)
   a) This function is used to write single byte to MAX3421E via SPI, simultanously reads status register and returns it
   b) In this function we create a new variable status_register that stores the return value when we call the function alt_avalon_spi_command. However, notice that we don't directly make status_register = alt_avalon_spi_command, instead, the address of status_register was passed in to store register value. The return value of alt_avalon_spi_command is just used to determine whether the function successfully ended.
2. void MAXreg_wr(BYTE reg, BYTE val)
   a) this function is used to write register to MAX3421E via SPI
   b) Notice that, in this function, we use a data packet to store which register first and then store the value we want to write. We need to add 2 to the register due the protocol of SPI which means we need to set write flag to high so that we could write.
3. BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data)
   a) This function is used to multiple-byte write and would return a pointer to a memory position after last written
   b) This is very similar to MAXreg_wr function, we just need to add all the data after we specify which register instead of just 1 data.
4. BYTE MAXreg_rd(BYTE reg)
   a) This function is used to read register from MAX3421E via SPI
   b) As in the SPI_wr function, we don't directly set val = alt_avalon_spi_command, instead we pass in the address of val so that the

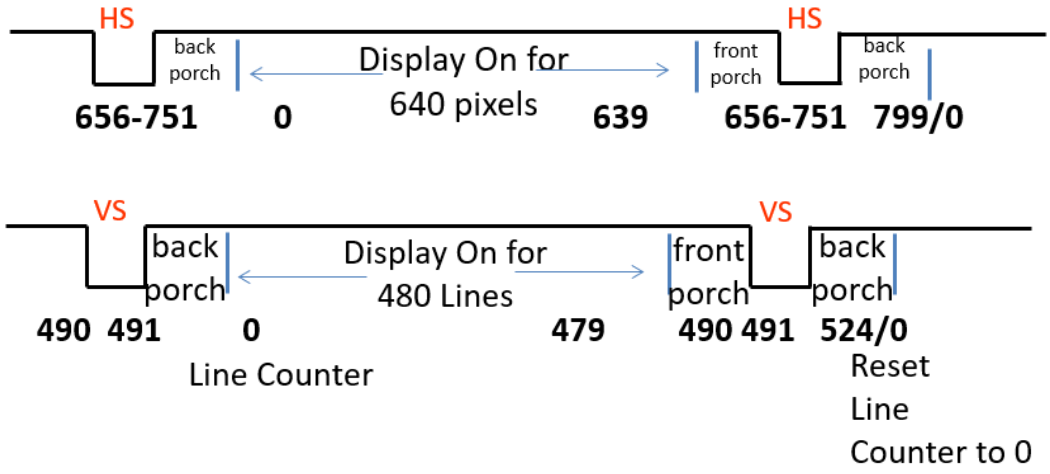
function could write the value in the specified register to the val address.

5. BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data)
   a) This function is used to multiple-byte read and would return a pointer to a memory position after last read.
   b) In this function, no local variables are created, we just need to pass in nbytes + 1 since we have to read the register also.

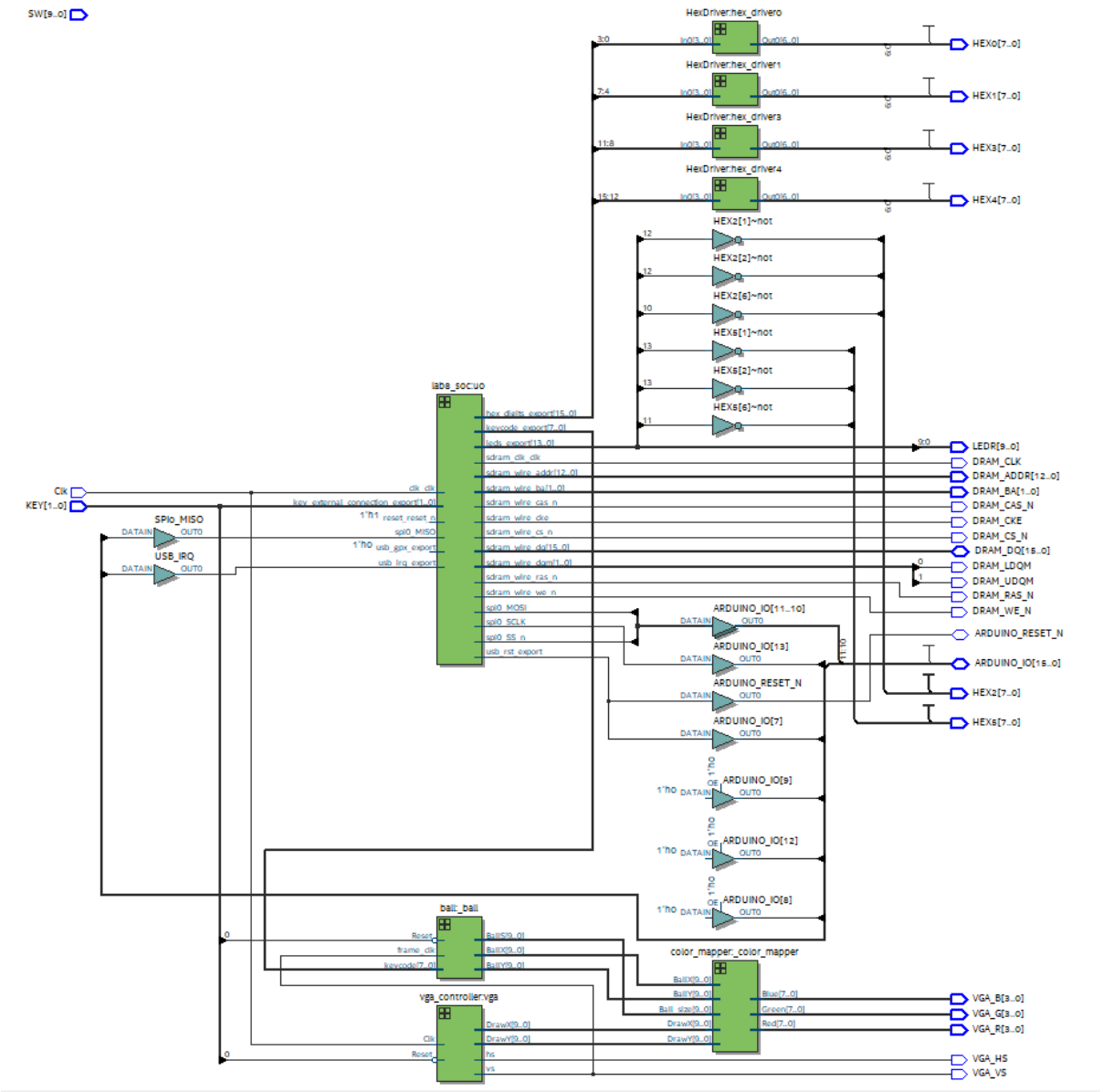
VGA, Ball and Color Mapper Interacions:

All these 3 sub modules are connected through the top level lab8.sv. First of all, the ball module reads the keycode that's pressed by the keyboard, and performs corresponding scenarios. Then the ball module outputs Ballx and Bally to color_mapper. Meanwhile, the VGA determines where to draw the corresponding scene, outputting DrawX and DrawY to color_mapper. Once receiving these, the color_mapper calculates which part should be the ball and which part should be something else. Inside the color_mapper, there is an if statement to determine if the object going to be drawn is a ball or other shapes. And inside the VGA, it has vertical and horizontal outputs that decide where the electron bean should paint as counters to help decide the locations.

Based on the different pulse, different operations are done for vertical and horizontal.



Top Level Block Diagram:



.sv Modules:

Module: lab8.sv
Inputs: Clk, [1:0] KEY, [9:0] SW
Outputs: [7:0] HEX0,HEX1,HEX2,HEX3,HEX4,HEX5, DRAM_CLK, DRAM_CKE, DRAM_ADDR, DRAM_BA, DRAM_LDQM, DRAM_UDQM, DRAM_CS_N,DRAM_WE_N,DRAM_CAS_N,DRAM_RAS_N,VGA_HS,VGA_VS,VGA_R,VGA_G,VGA_B,
Inout: [15: 0] DRAM_DQ, ARDUINO_RESET_N, [15: 0] ARDUINO_IO
Description: gets the switches input and buttons from FPGA and then outputs the control signals to the VGA, color_mapper and Ball.
Purpose: The top-level module that connects the NIOS II to software and board.

Module: lab8_soc.v
Inputs: clk_clk, [1:0] key_external_connection_export, reset_reset_n, accumulate_export, reset_1_export, [7:0] sw_export, spi0_MISO, usb_gpx_export, usb_irq_export
Outputs: [15:0] hex_digits_export, [7:0] keycode_export, [13:0] leds_export, sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [1:0] sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export
Inout: [15:0] sdram_wire_dq
Description: This module links the all the declared components, which are LED, clock, switches and processor, SPI, sdram and HEXs
Purpose: This file is generated by the Platform Designer base on the settings during initialization.

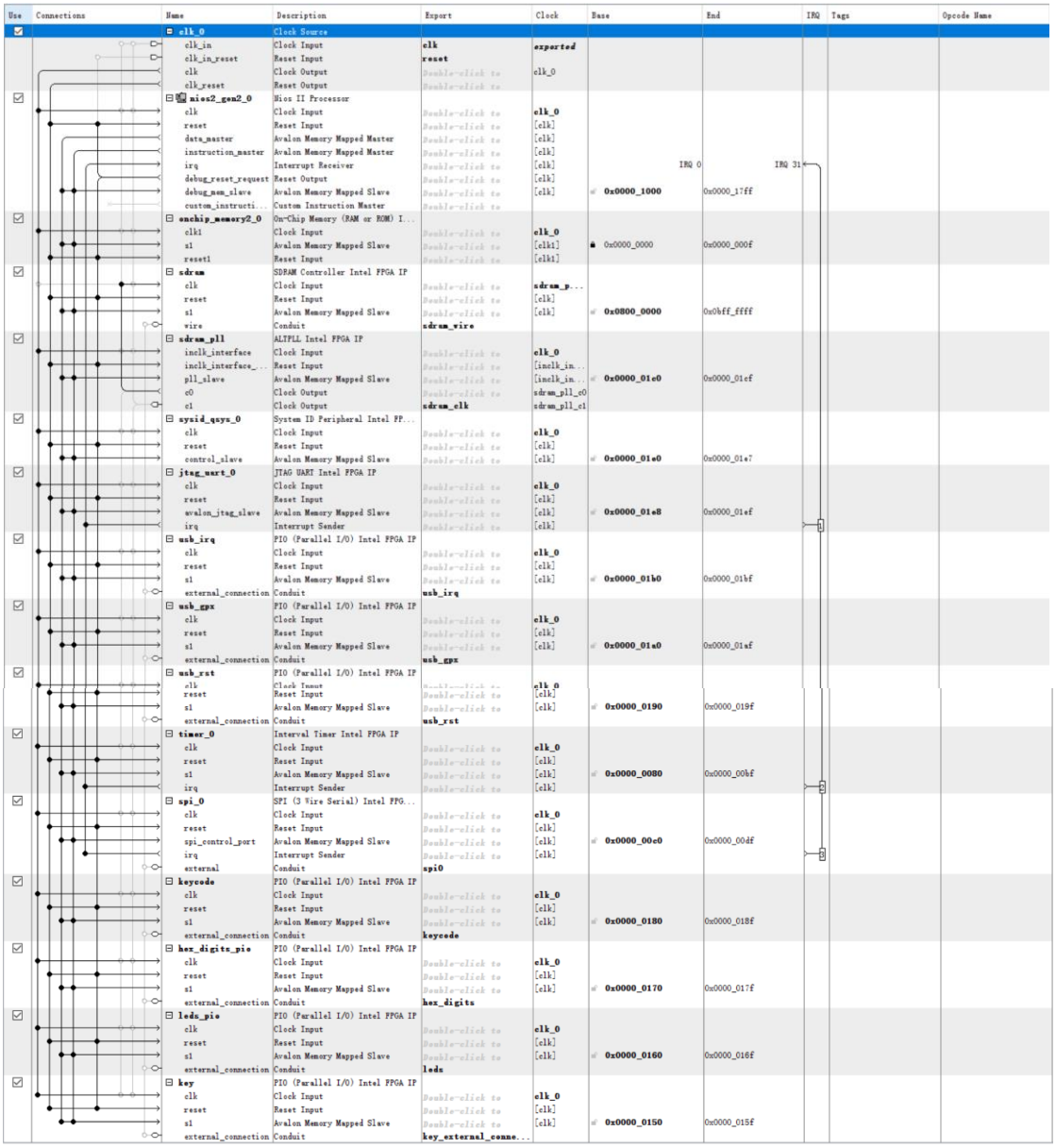
Module: VGA_controller.sv
Inputs: Clk, Reset
Outputs: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, DrawY
Desciption: This module generates how the pixels should be drawn. And within the module, it's been done by generating the horizontal and vertical puls and blanking interval.
Purpose: The module is used as the VGA protocol

Module: ball.sv
Inputs: Reset, frame_clk, [7:0] keycode,
Outputs: [9:0] BallX, BallY, BallS
Desciption: This module diverts the direction when an corresponding input is pressed or if the ball hits the wall.
Purpose: calculates the position of the ball under different situations

Module: Color_Mapper.sv
Inputs: [9:0] BallX, BallY, DrawX, DrawY, Ball_size
Outputs: [7:0] Red, Green, Blue
Desciption: This module generates what color should be and what the shape of the object is.
Purpose: Use to determine shape and color

Module: HexDriver.sv
Inputs: [3:0] In0
Outputs: [6:0] Out0
Description: A 16-to-1 mux control the value based on In0
Purpose: Use to change the HEX value on FPGA

System Level Block Diagram:



Functionalities of each blocks:

clk_0:

The clock for the other modules, although some modules will have their own clocks.

nios2_gen2_0:

The NIOS II processor, which will be responsible for the computation of the program.

onchip_memory2_0:

The on-chip memory, which will be used by the processor above to do the read and write during computation.

sdram:

This is the DRAM module. It is connected to the processor so the processor can get the memory contents from it.

sdram_pll:
This module generates the clock for the sdram which will be 1ns slower than the main clock. The slower clock for sdram is used to prevent the sdram from read/write corruption.

leds_pio:
The LED outputs to the board. The address of the LED will be used to control the on/off of each LED.

Key:
The key outputs to the board.

Hex_digits_pio:
Hex outputs to the FPGA.

Jtag_uart_0:
you can use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II (using print and scan statements in c).

Keycode:
The keycode read from the keyboard, the address is used in main.c to determine the starting point to read.

Spi_0:
Use to implement to SPI protocol to communicate between USB and processor.

Timer_0:
This is needed in the USB driver code in order to keep track of the various time-outs that USB requires.

Usb_rst:
Use to reset usb

Usb_gpx:
Not used actually

Usb_irp:
Use to send usb interrupts.

sysid_qsys_0:
This module is used to check whether the hardware and the software are connected

correctly.

LUT	3419
DSP	0
Memory (BRAM)	55296 bits
Flip-Flop	2512
Frequency	149.16 MHz
Static Power	96.18 mW
Dynamic Power	0.68 mW
Total Power	109.17 mW

Conclusion:

a. Our design works perfectly except for the glitch that Professor Cheng also had when he demoed his game to us.

b. The SPI core does not give enough information on how to implement the function.