ECE 385

Spring 2023

Lab 6

Experiment #6

\

Aleena Majeed & Ali Chaudry
aleenam2 & achau9
17:50 - 17:55

1. **Introduction**
    a. **Summarize the basic functionality of the NIOS-II processor running on the MAX10 FPGA**
    b. **Briefly summarize the operation of the USB/VGA interface**
2. **Written Description and Diagrams of NIOS-II System**
    i. **Describe in words the hardware component of the lab, in this lab, only the Platform Designer module is here**
    ii. **Describe in Lab 6.1 how the I/O works**
    iii. **Describe in words how the NIOS interacts with both the MAX3421E USB chip and the VGA components**
    iv. **Written description of the SPI protocol**
    v. **Describe the purpose of each function you filled in the C code (you do not need to describe the functions you did not modify)**
    vi. **Describe in detail the VGA operation, and how your Ball, Color Mapper, and the VGA controller modules interact**
3. **Top Level Block Diagram**
    a. **This diagram should represent the placement of all your modules in the top level. Please only include the top-level diagram and not the RTL view of every module**
4. **Written Description of all .sv Modules**
5. **System Level Block Diagram**
    a. **The Platform Designer view of the SoC module should be found here, describe the functionality of each block (including those which are part of the SoC, such as the memories)**
6. **Describe in words the software component of the lab**
7. **Answers to all INQ & Post lab questions**
    a. **What are the differences between the Nios II/e and Nios II/f CPUs?**
    b. **What advantage might on-chip memory have for program execution?**
    c. **Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?**
    d. **Why does SDRAM require constant refreshing?**
    e. **What is the maximum theoretical transfer rate to the SDRAM according to the timings given?**
    f. **Note that there is one 32M*16 chips, so the total amount of memory should be 512MBits (64 Mbytes), make sure this is consistent with your above numbers; you will need to justify how you came up with 512 Mbit to your TA.**
    g. **Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. Why might this be the case?**
    h. **The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?**
    i. **You must now make a second clock, which goes out to the SDRAM chip itself, as recommended by Figure 11. Make another output by clicking clk c1,**

and verify it has the same settings, except that the phase shift should be -1ns. This puts the clock going out to the SDRAM chip (clk c1) 1ns behind the controller clock (clk c0). Why do we need to do this? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.

j. **What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?**
k. **Look at the various segment (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment, e.g. the code:**
l. **const int my_constant[4] = {1, 2, 3, 4} will place 1, 2, 3, 4 into the .rodata segment.**
m. **Document any problems you encountered and your solutions to them**

8. **Design Resources and Statistics in table provided in the lab**
9. **Conclusion**
    a. **Discuss the functionality of your design. If parts of your design did not work, discuss what could be done to fix it?**
    b. **Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester?**

1. **Introduction**
    a. **Summarize the basic functionality of the NIOS-II processor running on the MAX10 FPGA**

    The NIOS II processor is a soft processor that can be customized and integrated into an FPGA device such as the MAX10. It is a 32-bit CPU that supports a wide range of peripherals and interfaces, making it suitable for various embedded applications. The NIOS II processor is designed to offer a balance between high performance and low power consumption.

    The NIOS II processor has flexibility in terms of programming. It can be programmed using a high-level language like C, which simplifies the development process.

    When running the MAX10 FPGA, the NIOS II processor can interfere with other peripherals and IP blocks via the FPGA fabric. For example, in the experiment hand out, the lab mentioned how the NIOS II processor is integrated with an SDRAM controller and a PIO block. The SDRAM controller allows the NIOS II processor to access external memory, which can be useful for storing large amounts of data. The PIO block provides a parallel interface for the nIOS II processor to communicate with other devices and peripherals, such as LEDs and switches.

    The NIOS II processor running on the MAX10 FPGA offers a flexible and customizable platform for embedded system development.

    b. **Briefly summarize the operation of the USB/VGA interface**

    The operation of the USB/VGA interface correlates computer input from the USB interface to an external display using the VGA interface. The USB connected devices to a computer. In this lab, we connected a keyboard to our computer using our FPGA, and used SPI peripherals to connect the FPGA to our USB controller. We also used a VGA port to connect our FPGA to some external devices such as a monitor or a TV, so that we could display our output. Our VGA interface in this lab was the DE10-Lite, to convert our digital signals to analog so that the data could be displayed on an external screen. So, combining the USB/VGA interface allows the user to control what is being displayed on the external application.

2. **Written Description and Diagrams of NIOS-II System**
    Describe in words the hardware component of the lab, in this lab, only the Platform Designer module is here
    Describe in Lab 6.1 how the I/O works
    Describe in words how the NIOS interacts with both the MAX3421E USB chip and the VGA components
    Written description of the SPI protocol
    Describe the purpose of each function you filled in the C code (you do not need to describe the functions you did not modify)

Describe in detail the VGA operation, and how your Ball, Color Mapper, and the VGA controller modules interact

The NIOS II processor is a configurable 32-bit processor architecture developed by Intel. It is designed to be embedded into various devices, including FPGAs, ASICs, and SoCs.

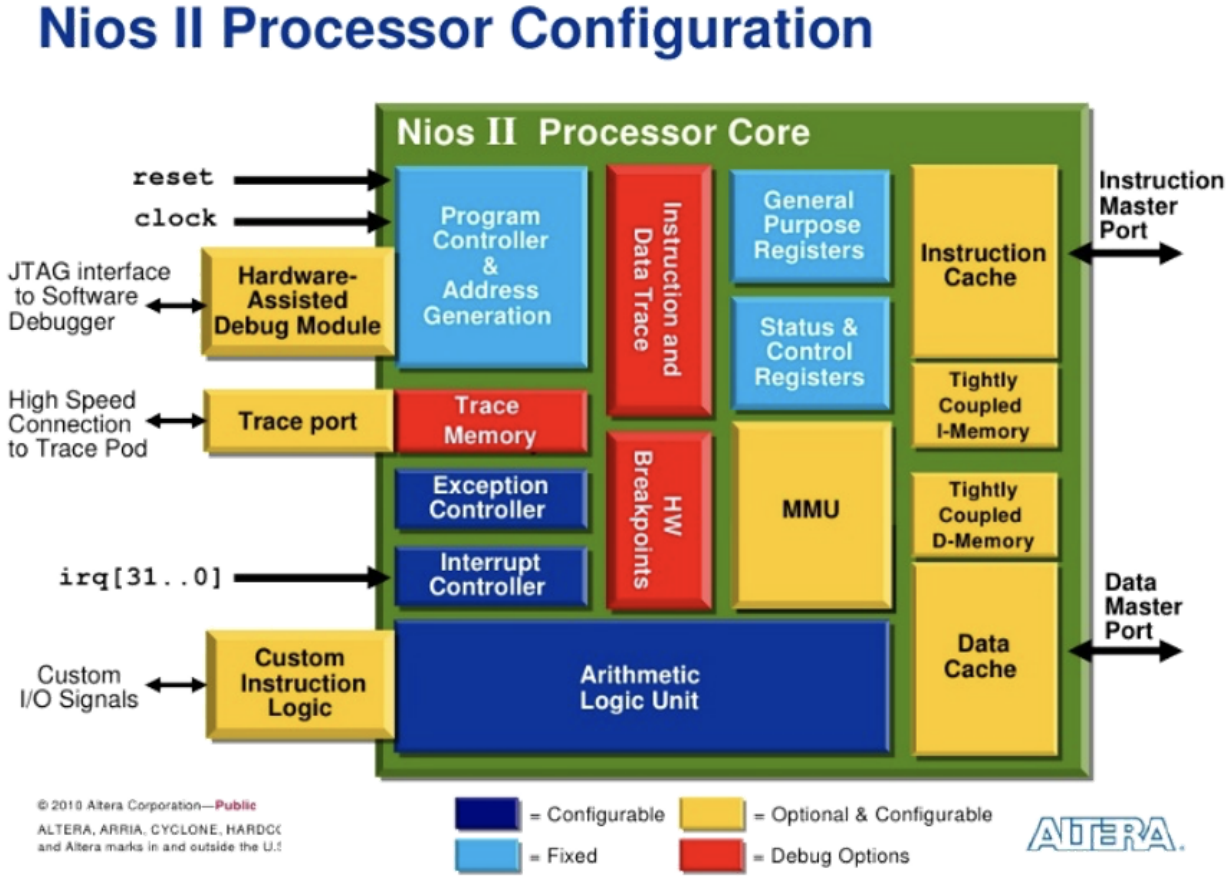
The NIOS II processor has a Harvard architecture, meaning that it has separate instruction and data memory spaces. It also supports both big and littleendian byte ordering. The processor is highly configurable, allowing designers to customize the processor to meet their specific needs. The NIOS II processor supports a wide range of peripherals, including UART, Ethernet, SPI, and I2C interfaces, as well as timers, interrupt controllers, and DMA (Direct Memory Access) controllers.

The processor has a five-stage pipeline, including the fetch, decode, execute, memory, and writeback stages. This pipeline design allows for efficient processing of instructions, and the processor can execute up to six instructions per clock cycle.

The NIOS II processor supports three different instruction set architectures (ISAs): standard, compact, and fast. The standard ISA provides a rich set of instructions for general-purpose computing, while the compact and fast ISAs are optimized for specific applications, such as digital signal processing (DSP) and floating-point operations.

The NIOS II processor is programmed using C, C++, and assembly languages. It also supports debugging and profiling tools, such as the NIOS II Integrated Development Environment (IDE), which provides a graphical user interface for developing and debugging NIOS II applications.

Overall, the NIOS II processor is a highly flexible and customizable processor architecture that is well-suited for embedded systems. It has a wide range of peripherals, configurable design, and support for multiple ISAs.



Top level diagram of the NIOS Processor architecture

- 32-bit modified Harvard RISC architecture
- User configurable tradeoffs (performance vs. LEs)
- We'll typically use "IIe" configuration (very small < 700 LEs)
- Used with Avalon bus
- Full C compiler included

Describe in words the hardware component of the lab, in this lab, only the Platform:

The hardware components on this lab are created by the platform designer. The platform designer uses IP blocks which are used to make a FPGA or application-specific integrated circuit for a product. IP cores are reusable units of logic. The IP blocks included are NIOS II, PLL, SDRAM controller, and other PIO blocks. Some other hardware components include the On-Chip Memory, ATPLL Intel FPGA IP, the System ID Peripheral, JTAG UART, PIOs, and SPI.

Clk_0: This clock runs at 50 MHz and clocks all hardware components

SDRAM Controller: This is a hardware component that manages the flow of data between the CPU and the SDRAM memory modules.

It acts as an interface between the CPU and the SDRAM memory chip and is responsible for controlling the data transfers to and from the memory. It ensures that data is transferred to and from the SDRAM memory at the appropriate clock rate and in the correct sequence.

This initializes the memory devices and translates read-and-write instructions from the local interface into SDRAM command signals.

ATPLL Intel FPGA IP: The sdram_pll makes two 50 MHz clock signals, one of these clocks has a 1ns delay. This is because the SDRAM, requires exact time and need time to stabilize, which the two clock sensures.

System ID Peripheral: The function of sysid_qsys_0 is to assign unique identifiers to both the hardware and software components and then verify their compatibility when the software is executed. This feature helps to prevent the installation of software onto an FPGA that doesn't have a compatible NIOS II configuration, ensuring that the software and hardware are compatible with each other.

NIOS II Processor: One of the key advantages of the NIOS II processor is its ease of use. It is designed to be programmed in C/C++, and its software development tools are integrated with widely used development environments like Eclipse. This makes it easy for software developers to write and debug code for the processor, reducing the time and effort required for development.

One of the key advantages of the NIOS II processor is its ease of use. It is designed to be programmed in C/C++, and its software development tools are integrated with widely used development environments like Eclipse. This makes it easy for software developers to write and debug code for the processor, reducing the time and effort required for development.

JTAG UART: A JTAG UART is a hardware module that enables serial communication with a device via the JTAG (Joint Test Action Group) port. This interface is widely used for programming and debugging embedded devices such as FPGAs, microcontrollers, and system-on-chip (SoC) devices.

Through the JTAG UART, users can effortlessly and consistently communicate with the device using a standard serial communication protocol. They can transmit and receive data, control, and configure the device's settings and behavior.

One of the primary benefits of the JTAG UART is its ability to provide a debugging interface for embedded devices, allowing developers to test and debug the device during the development process. Developers can accomplish various debugging tasks such as memory reading and writing, setting breakpoints, and stepping through code by connecting to the JTAG port.

Overall, the JTAG UART is a crucial tool for developers involved in embedded systems development. Its dependable and flexible interface provides a reliable method of communicating with and debugging devices through the JTAG port.

PIO: These are connected to the data bus of the processor. PIOs like usb_irq, usb_gpx, usb_rst are needed for the connection to the MAX3421E chip. Hex_digits_pio, leds_pio, and keycode are all utilized to display information from the data bus. Key is used with the peripheral keyboard input.

SPI: The spi_0, also known as the serial peripheral interface, enables communication between the FPGA and multiple peripherals. In this lab, the MAX3421 USB peripheral/host controller was the only peripheral utilized, and it was connected to the spi_0. The transmission of SS, MISO, MOSI, and SCLK signals via the SPI facilitates the transfer of data to and from the connected peripherals.

Describe in Lab 6.1 how the I/O works:

Lab 6.1 featured an accumulator that could be managed through a pair of buttons, namely, one for resetting and another for accumulating, along with some switches. Upon pressing the run button, a C program would read the values from the switches and add them to the current value in the accumulator. Pressing the reset button would reset the accumulator to 0. Additionally, a PIO block was employed to govern the LEDs that displayed the values stored in the accumulator.

Describe in words how the NIOS interacts with both the MAX3421E USB chip and the VGA components

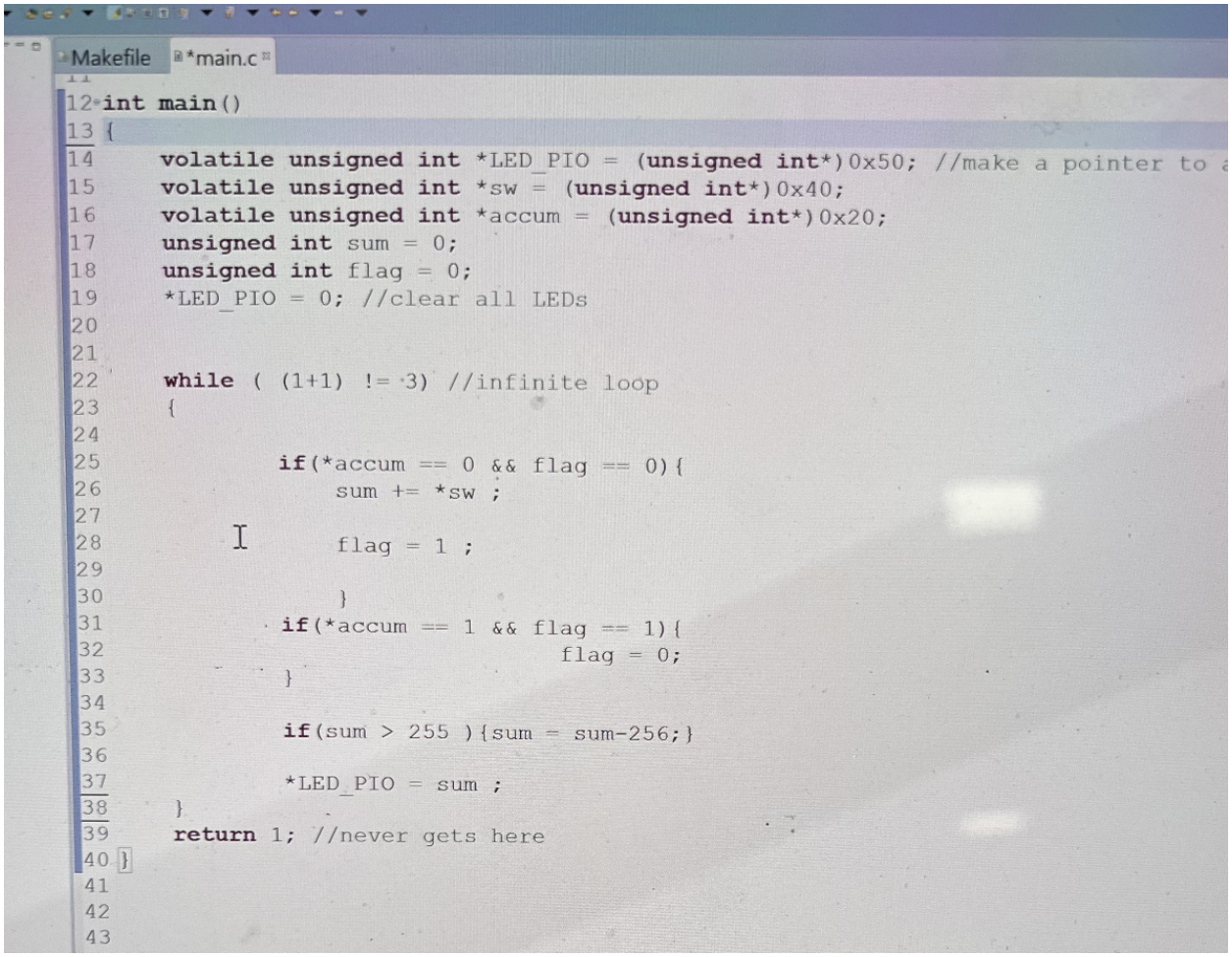
The NIOS utilizes four functions, namely MAXreg_wr, MAXbytes_wr, MAXreg_rd, and MAXbytes_rd, all communicated via SPI, to interact with the MAX3421E USB chip and VGA components. The information obtained by the NIOS is then transmitted to the FPGA's top-level module and ball module,

which utilize it to determine the position and movement of the ball. The resulting data is then conveyed to the color_mapper and VGA_controller modules, which operate the VGA components through the horizontal sync, vertical sync, red, green, and blue signals.

Written description of the SPI protocol

The Serial port interface (SPI) utilizes four signals, namely the slave select (SS), master in slave out (MISO), master out slave in (MOSI), and SCLK, to control several peripherals. In cases where multiple peripherals are attached, the SS signal is responsible for selecting the peripheral that the FPGA is communicating with. On the other hand, the MISO and MOSI signals determine the data that is being read from or written to the peripheral by the FPGA. Specifically, the MISO signal is read by the FPGA from the peripheral, while the MOSI signal is read by the peripheral from the FPGA. Furthermore, the SCLK is responsible for clocking the signals to ensure that data is read within the designated stable window.

Describe the purpose of each function you filled in the C code (you do not need to describe the functions you did not modify)



Picture of the C code that we wrote for lab 6.1 - this code makes an accumulator

6.1 Code:

In lab 6.1, we implemented an accumulator on the FPGA LEDs. The while loop ensures that the accumulator logic runs continuously so that it is checked whether the accumulator button is being pressed.

The first conditional statement checks whether the flag variable and accumulate button are both set to 0. If this condition is satisfied, it indicates that the accumulate button was pressed since it is active low. Consequently, the current sum is set to the previous sum added with the new values set onto the switches.

In the next conditional statement, if the accumulate button is not pressed and the flag is set to 1, the flag is reset to 0. The accumulate button will then wait to be pressed again to go high so that the first conditional statement can be activated again.

The last conditional statement is triggered when the sum overflows. It ensures that if a number that makes the sum over 255 is accumulated, then the accumulator will reset. For example, if 255 is accumulated with 1, then the accumulator will reset to 0. If it accumulates with 2, then the sum will be set to 1, and so on.

```c
//writes register to MAX3421E via SPI
void MAXreg_wr(BYTE reg, BYTE val) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write val via SPI
    BYTE return_code;
    alt_u8 arr[2] = {reg +2, val};  //passing in pointers so use an array
    return_code = alt_avalon_spi_command(SPI_0_BASE, 0,
                                2,arr,
                                0, NULL,
                                0);
    //if return code < 0 print an error
    if(return_code < 0 ){
        printf("error");}
}
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)}
```

This is the C code written for writing data into a register using the function: int alt_avalon_spi_command

6.2 MAXreg_wr

This function is responsible for writing data into a register. An array named arr is created with 2 elements. The two elements are reg+2 and val. reg + 2 indicates that we want to write to a register and val is the value to be written into a register. The function specifies a write length of two because the register and the value both need to be written. We then check if the return code (checking if the write command is valid) is valid. His reading length is zero and we are not reading anything.

```c
//multiple-byte write
//returns a pointer to a memory position after last written
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write data[n] via SPI, where n goes from 0 to nbytes-1
    //read return code from SPI peripheral (see Intel documentation)
    BYTE return_code;
    alt_u8 arr[nbytes + 1]; //making an array of size nbytes+1 bc. array size is nbytes
    arr[0] = reg + 2;       // first element is instructing write instruction
    for(int i = 0; i<nbytes; i++){  //for loop to iterate through each byte of data
        arr[i + 1] = data[i];    //setting data[i] to the first index of the array since the zero index is reg+2
    }
    return_code = alt_avalon_spi_command(SPI_0_BASE, 0,  //BASE is the SPI address
            nbytes+1,arr, //nbytes+1 is size of array, arr holds all the data[n] which it was assigned in the for loop
            0, NULL,
            0);
    if(return_code < 0 ){
            printf("error");}
    return data+nbytes;

    //if return code < 0  print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return (data + nbytes);
}
```

This is the C code written for writing several bytes into a register using the function: int alt_avalon_spi_command

6.2 MAXbytes_wr

This function is responsible for writing several bytes. An array with a size of nbytes + 1 is created to account for the number of bytes and the register to write to. A for loop is then created to assign all the data to this array. The alt_avalon_spi_command function then writes the data in the array and assigns a write length of the number bytes +1 to account for the register to write to. The return value is the data + nbytes.

```c

//reads register from MAX3421E via SPI
BYTE MAXreg_rd(BYTE reg) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return val
    BYTE return_code;
    BYTE val;
                    //passing in pointers so use an array
        return_code = alt_avalon_spi_command(SPI_0_BASE, 0,
                                    1,&reg,
                                    1,&val,
                                    0);
        //if return code < 0 print an error
        if(return_code < 0 ){
            printf("error");}
        return val;
}

```

This is the C code written for reading data from a register using the function: int alt_avalon_spi_command

6.2 MAXreg_rd

This function is responsible for reading data from a register. A variable, val, is created to hold the value read from the register. The function specifies a write length of 1 and read length of 1 because the register to write to needs to be written and then we need to read a value from this register. We then check if the return code (checking if the read command is valid) is valid. If it is valid then we return the value.

```
1
2  //multiple-byte read
3  //returns a pointer to a memory position after last written
4  BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
5      //psuedocode:
6      //select MAX3421E (may not be necessary if you are using SPI peripheral)
7      //write reg via SPI
8      //read data[n] from SPI, where n goes from 0 to nbytes-1
9      //read return code from SPI peripheral (see Intel documentation)
10     //if return code < 0 print an error
11     //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
12     //return (data + nbytes);
13     BYTE return_code;
14     //alt_u8 arr[nbytes]; //making an array of size nbytes+1 bc. array size is nbytes
15         //arr[0] = reg;       // first element is instructing read instruction
16 //      for(int i = 0; i<nbytes-1; i++){  //for loop to iterate through each byte of data
17 //          arr[i] = data[i];    //setting data[i] to the first index of the array since the zero index is reg+2
18 //      }
19         return_code = alt_avalon_spi_command(SPI_0_BASE, 0,  //BASE is the SPI address
20                 1, &reg,            //nbytes+1 is size of array, arr holds all the data[n] which it was assigned in the for loop
21                 nbytes, data,
22                 0);
23         if(return_code < 0 ){
24                 printf("error");}
25         return data+nbytes;
26 }
27
28
29
30
```

This is the C code written for reading multiple bytes from a register using the function: int alt_avalon_spi_command

6.2 MAXbytes_rd

This function is responsible for reading multiple bytes of data. The function specifies a write length of 1 and read length of nbytes because the register to write to needs to be written and then we need to read the bytes of data.. We then check if the return code (checking if the read command is valid) is valid. If it is valid then we return the data+nbytes.

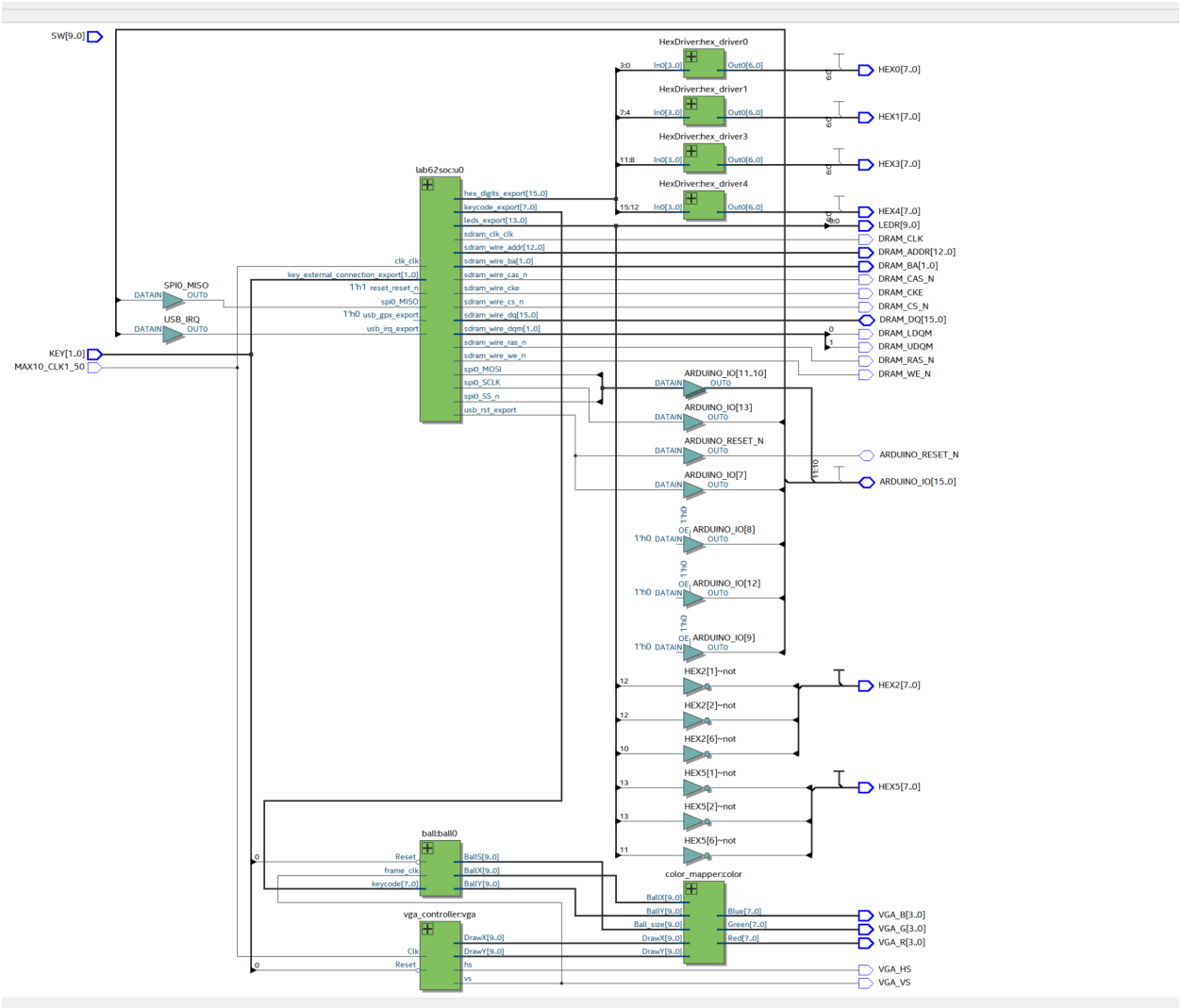
Describe in detail the VGA operation, and how your Ball, Color Mapper, and the VGA controller modules interact:

The Ball module plays a significant role in keeping the ball in motion by continuously updating its position and trajectory. Its behavior is influenced by key presses or boundary detection.

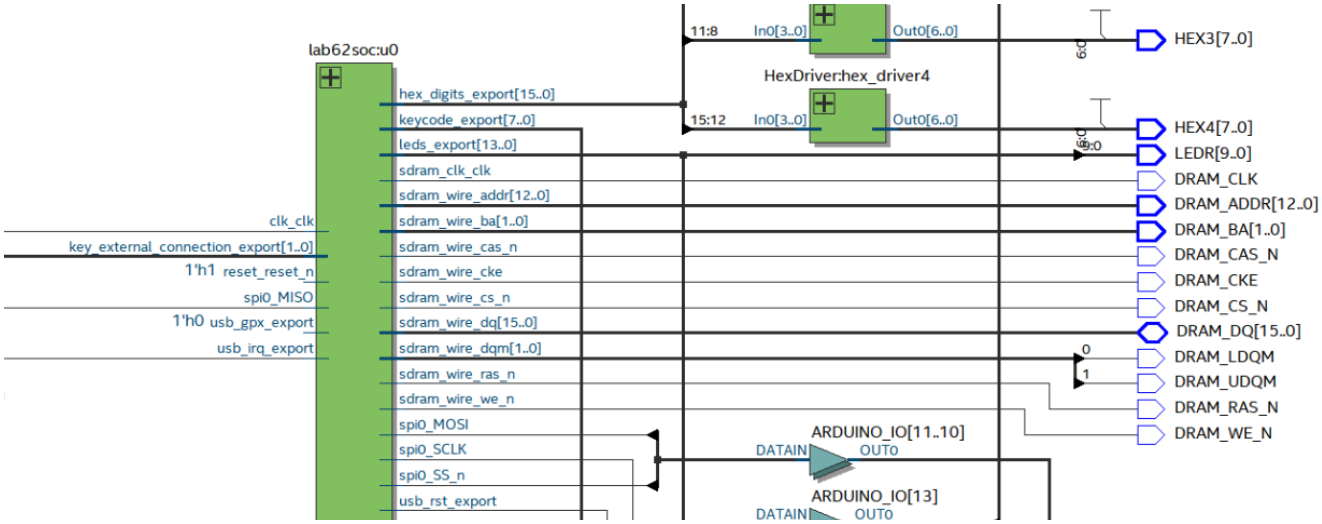
The VGA controller module is tasked with synchronizing the vertical and horizontal sync signals. This synchronization is essential in determining the precise position of the electron gun on the screen and when it should be turned on or off.

The color mapper module is responsible for identifying the shape and position of the ball by processing the inputs received from the Ball module. Its role is to map the right colors to each pixel on the screen. To achieve this, it uses the signals DrawX and DrawY from the VGA controller module to determine the pixel's location and the appropriate RGB signals to draw it accurately.

3. Top Level Block Diagram



RLT block diagram generated in Quartus



RTL diagram of the Top level

4. Written Description of all .sv Modules

- **lab61.sv :**
  - **module lab61()**
    - **input MAX10_CLK1_50,**
    - **input [1:0] KEY,**
    - **output [7:0] LEDR,**
    - **output [12:0] DRAM_ADDR,**
    - **output [1:0] DRAM_BA,**
    - **output DRAM_CAS_N,**
    - **output DRAM_CKE,**
    - **output DRAM_CS_N,**
    - **input [15:0] DRAM_DQ,**
    - **output DRAM_LDQM,**
    - **output DRAM_UDQM,**
    - **output DRAM_RAS_N,**
    - **output DRAM_WE_N,**
    - **output DRAM_CLK,**
    - **input [7:0] SW //added**
    -

FUNCTION: This module was responsible for instantiating the lab61soc module that was generated from our platform designer that contains all the PIO blocks. The instantiations and top level module connects the hardware components with the software and allows the keyboard peripherals to interact with the monitor peripherals.

- **lab62_toplevel.sv :**
  - **Module lab62()**
    - **input MAX10_CLK1_50,**
    - **input [1: 0] KEY,**
    - **input [9: 0] SW,**
    - **output [9: 0] LEDR,**
    - **output [7: 0] HEX0,**
    - **output [7: 0] HEX1,**
    - **output [7: 0] HEX2,**
    - **output [7: 0] HEX3,**
    - **output [7: 0] HEX4,**
    - **output [7: 0] HEX5,**
    - **output DRAM_CLK,**
    - **output DRAM_CKE,**
    - **output [12: 0] DRAM_ADDR,**
    - **output [1: 0] DRAM_BA,**
    - **inout [15: 0] DRAM_DQ,**
    - **output DRAM_LDQM,**
    - **output DRAM_UDQM,**
    - **output DRAM_CS_N,**
    - **output DRAM_WE_N,**
    - **output DRAM_CAS_N,**
    - **output DRAM_RAS_N,**
    - **output VGA_HS,**
    - **output VGA_VS,**
    - **output [3: 0] VGA_R,**
    - **output [3: 0] VGA_G,**
    - **output [3: 0] VGA_B,**
    - **inout [15: 0] ARDUINO_IO,**
    - **inout ARDUINO_RESET_N**

FUNCTION: This module was responsible for instantiating the vga_controller module, ball module, and color_mapper module. The instantiations and top level module connects the hardware components with the software and allows the keyboard peripherals to interact with the monitor peripherals.

- **color_mapper.sv :**
  - **Module color_mapper()**
    - **input [9:0] BallX, BallY, DrawX, DrawY, Ball_size,**
    - **output logic [7:0] Red, Green, Blue**

FUNCTION: The colors mapper determines if the ball is a circle, and based on that is draws the ball a certain color and otherwise draws the background blue. It helps figure out what to color a pixel on the monitor screen based on if it is the background or the ball.

- **ball.sv :**
    - **Module ball()**
        - **input Reset, frame_clk,**
        - **input [7:0] keycode,**
        - **output [9:0] BallX, BallY, BallS**

FUNCTION: The ball module helps determine the location of the ball and sets constraints on the ball's path so it can bounce only on screen. It helps determine the ball's path on where to bounce based on its current position and trajectory and updates its path once a key is pressed. Based on its trajectory, BallX and BallY are used to tell the location of the center of the ball which is received by the color mapper module to draw the ball again.

- **vga_controller.sv :**
    - **Module vga_controller()**
        - **input Clk,**
        - **Reset,**
        - **output logic hs, vs,**
        - **pixel_clk,**
        - **blank,**
        - **sync,**
        - **output [9:0] DrawX, DrawY**

FUNCTION: Every screen consists of an electron gun that traverses the display to fill in individual pixels with color. The vga_controller module is responsible for managing the vertical and horizontal synchronization, as well as the electron gun's movements, to enable the accurate drawing of these pixels.

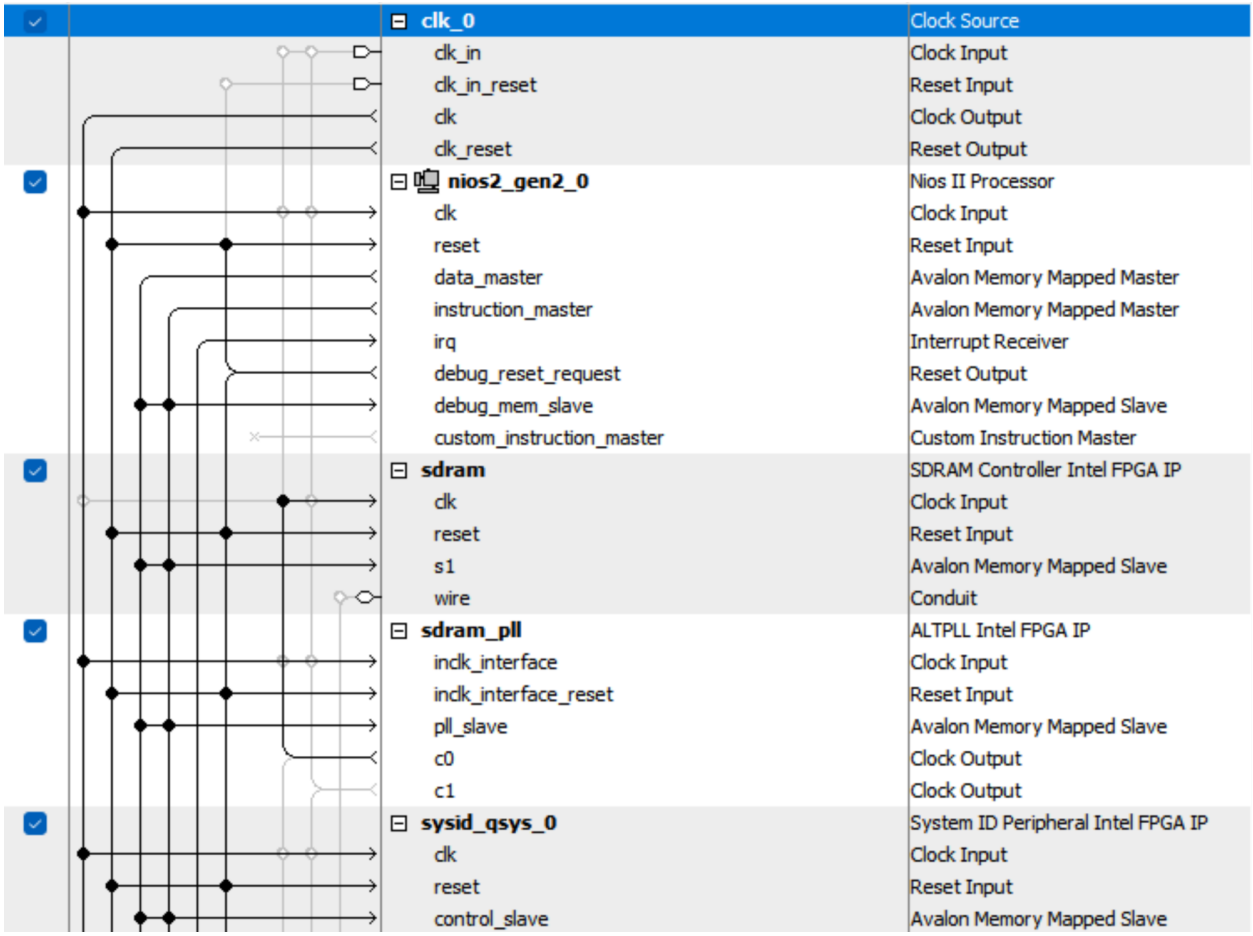
- **lab6.sdc :**

FUNCTION: Sets delays in the outputs for correct time signals and sets false paths from the switches, LEDs, and keys.

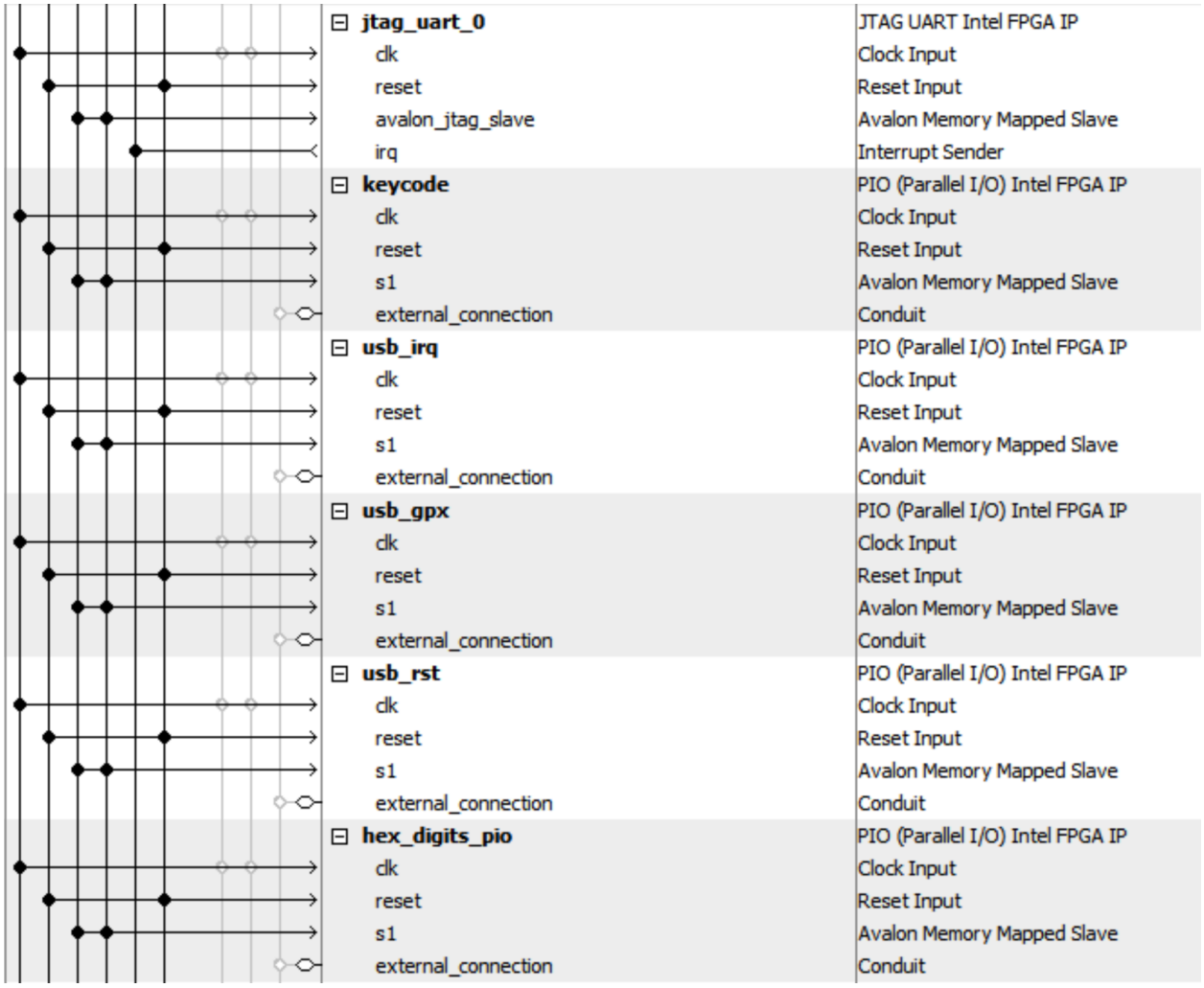
- **lab62soc.v:**

FUNCTION: The platform design generates these hardware components, and allows for connection between the hardware, keyboard, and peripherals.

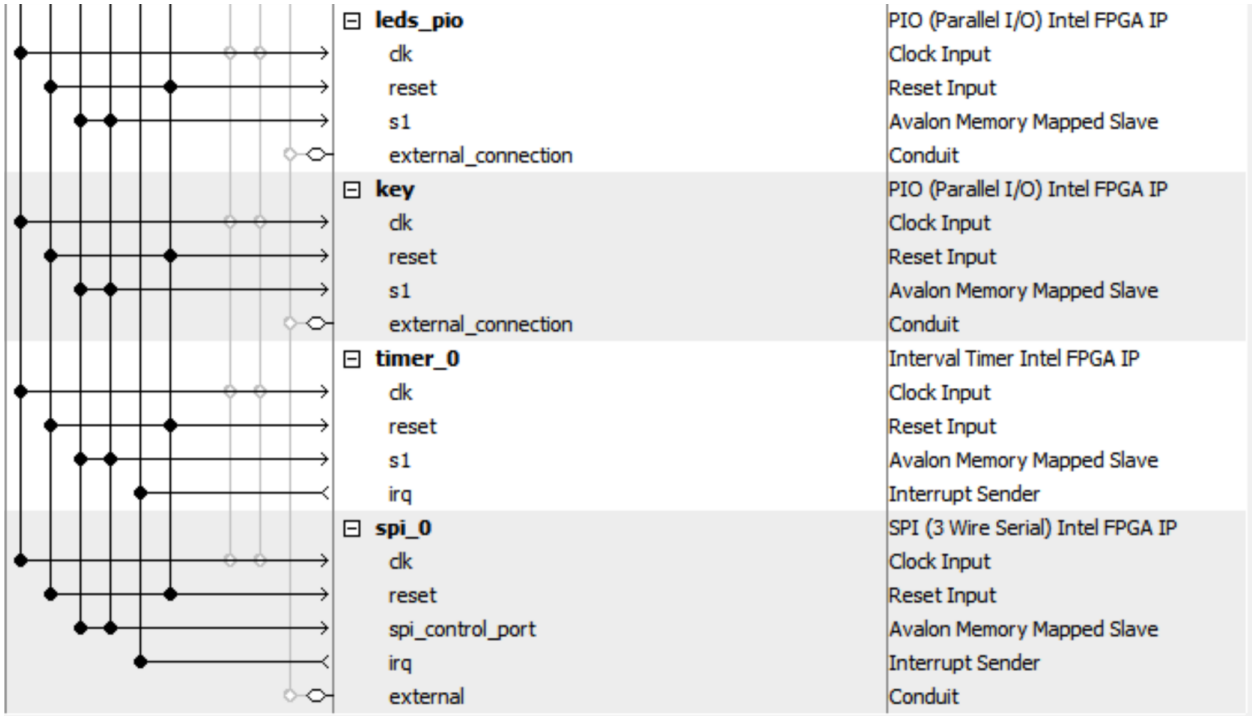
5. System Level Block Diagram



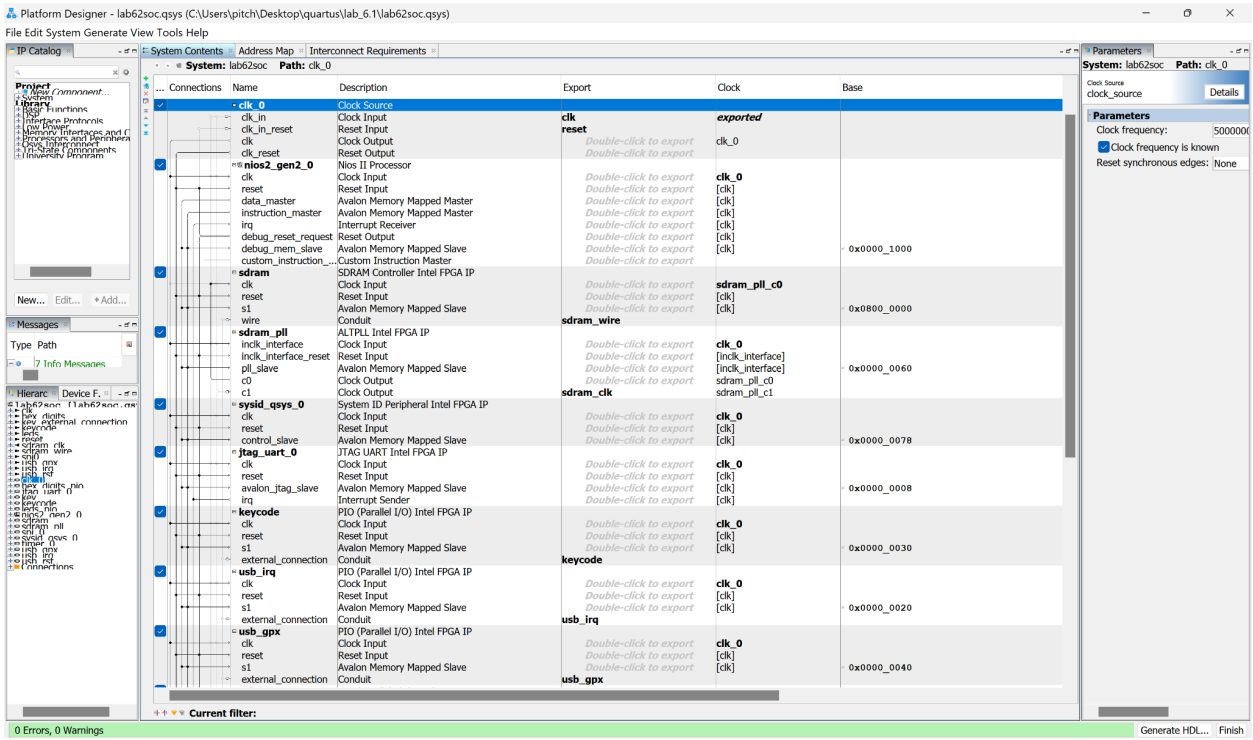
☑	**clk_0**	Clock Source
	clk_in	Clock Input
	clk_in_reset	Reset Input
	clk	Clock Output
	clk_reset	Reset Output
☑	**nios2_gen2_0**	Nios II Processor
	clk	Clock Input
	reset	Reset Input
	data_master	Avalon Memory Mapped Master
	instruction_master	Avalon Memory Mapped Master
	irq	Interrupt Receiver
	debug_reset_request	Reset Output
	debug_mem_slave	Avalon Memory Mapped Slave
	custom_instruction_master	Custom Instruction Master
☑	**sdram**	SDRAM Controller Intel FPGA IP
	clk	Clock Input
	reset	Reset Input
	s1	Avalon Memory Mapped Slave
	wire	Conduit
☑	**sdram_pll**	ALTPLL Intel FPGA IP
	inclk_interface	Clock Input
	inclk_interface_reset	Reset Input
	pll_slave	Avalon Memory Mapped Slave
	c0	Clock Output
	c1	Clock Output
☑	**sysid_qsys_0**	System ID Peripheral Intel FPGA IP
	clk	Clock Input
	reset	Reset Input
	control_slave	Avalon Memory Mapped Slave

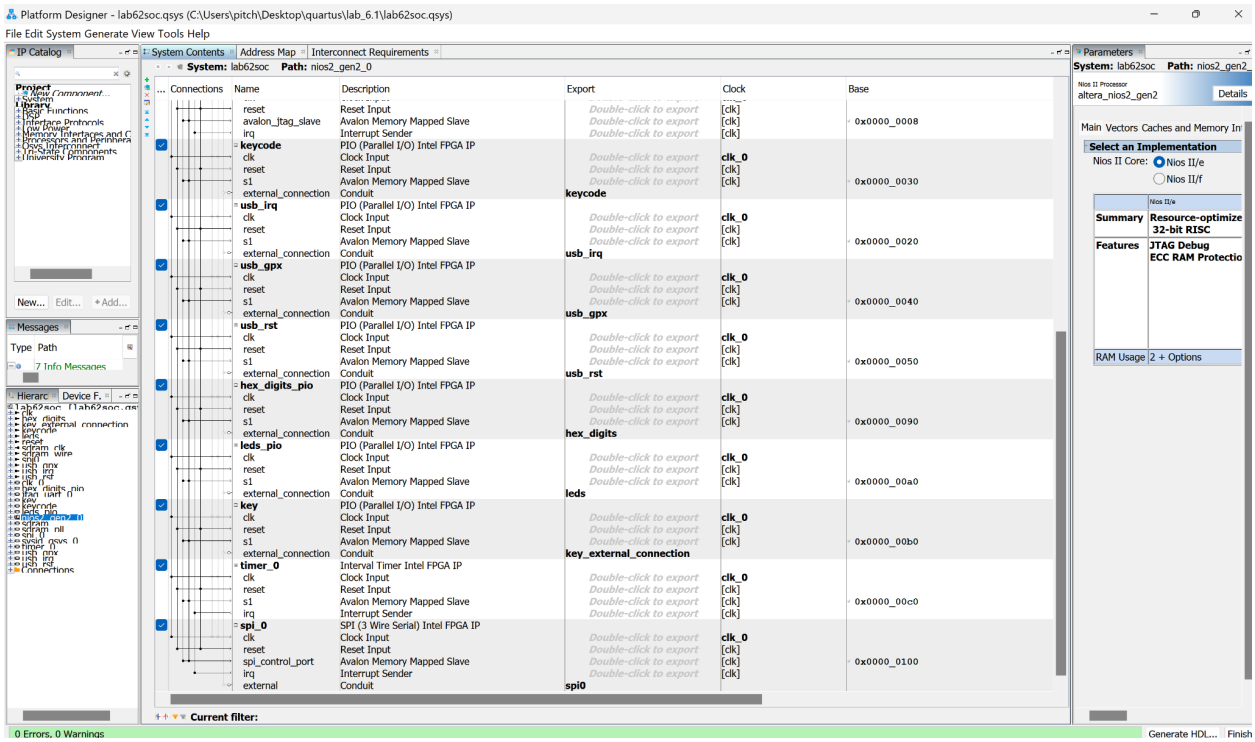


jtag_uart_0	JTAG UART Intel FPGA IP
clk	Clock Input
reset	Reset Input
avalon_jtag_slave	Avalon Memory Mapped Slave
irq	Interrupt Sender
keycode	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit
usb_irq	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit
usb_gpx	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit
usb_rst	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit
hex_digits_pio	PIO (Parallel I/O) Intel FPGA IP
clk	Clock Input
reset	Reset Input
s1	Avalon Memory Mapped Slave
external_connection	Conduit



PIO blocks created in Platform Designer for Lab 6.1





PIO blocks created in Platform Designer for 6.2

- **clk_0 -** This is a 50 MHz clock from the FPGA and includes a clock and a reset which are used to clock and reset other modules.

- **nios2_gen2_0 -** One of the key advantages of the NIOS II processor is its ease of use. It is designed to be programmed in C/C++, and its software development tools are integrated with widely used development environments like Eclipse. This makes it easy for software developers to write and debug code for the processor, reducing the time and effort required for development.

- **sdram -** In Nios II, the SDRAM is typically connected to the Nios II processor through a memory controller, which manages the transfer of data between the processor and the SDRAM. The memory controller provides an interface between the processor and the memory module, and handles tasks such as initializing the memory, issuing read and write commands, and managing the refresh cycles. When the Nios II processor needs to access data from the SDRAM, it sends a read request to the memory controller. The memory controller retrieves the requested data from the SDRAM and sends it back to the processor. Similarly, when the processor needs to write data to the SDRAM, it sends a write request to the memory controller, which writes the data to the appropriate memory location. The SDRAM's speed and capacity can affect the overall performance of a Nios II system. To optimize performance, the memory controller must be configured to work efficiently with the specific SDRAM device used in the system. This can involve setting timing parameters, configuring the refresh rate, and configuring other memory controller

settings. Overall, the use of SDRAM in a Nios II system allows for efficient data storage and retrieval, enabling the system to run faster and more efficiently.

- **sdram_pll -** The SDRAM clock is linked to a clock signal that is intentionally delayed by 1 ns. This delay is implemented to compensate for any phase discrepancies that may arise between the master and slave clocks.

- **sysid_qsys_0 -** This is used to check for errors in the connections from the hardware to software components that are connected. The hardware and software must be updated to the latest changes so they are compatible. This is why steps such as generating BSP are necessary.

- **jtag_uart__0 -** The JTAG UART is a hardware module that provides a communication interface between the FPGA and the computer or another device. The JTAG UART module allows for bi-directional communication between the device and an external device or computer. It provides a standard serial interface, which allows for the transfer of data in a standardized format. This interface is typically used for debugging and diagnostic purposes, as well as for firmware updates.

- **keycode -** The Keycode refers to an essential 8-bit PIO input that enables the processor to retrieve the specific code corresponding to a pressed key on a USB keyboard.

- **usb_irq -** Refers to the interrupt signal used by the USB device to notify the host (computer) about an event that requires attention, such as the insertion of a new device or the completion of a data transfer. This is needed to utilize the USB keyboard with the FPGA.
- **usb_gpx -** Stands for USB General Purpose Input/Output. It provides a set of pins that can be configured by the device designer for various purposes, such as controlling LEDs, reading switches, or interfacing with sensors. This is needed to utilize the USB keyboard with the FPGA.

- **usb_rst** - A signal used to reset the USB device or put it in a low-power mode. It is typically activated by the host, either through a software command or a hardware pin. This is needed to utilize the USB keyboard with the FPGA.

- **hex_digits_pio -** The term hex_digits_pio refers to a parallel input/output interface that can communicate hexadecimal data between a microcontroller or processor and an external device. In the context of the DE-10 board, this interface can be used to output the keycode of the currently pressed key on the hex display. By utilizing a 16-bit parallel

interface, the transfer of multiple digits can be accomplished simultaneously, resulting in faster and more efficient communication.

- **key -** This was used in lab 6.1 as a PIO input that was needed for the accumulate button's signal in the function we wrote.
- **leds_pio -** This was used in lab 6.1 as a PIO output that was needed for the LED's to light up to show the sums of our accumulation.
- **timer_0 -** Used to send a signal so that NIOS II can check time passed.
- **spi_0** - SPI_0 is a module that provides a means for devices to communicate with each other using the Serial Peripheral Interface (SPI) protocol. The SPI interface facilitates the transfer of data between a master device and one or more slave devices using two communication channels, MOSI (Master Output Slave Input) and MISO (Master Input Slave Output). Depending on the specific mode of operation, the transfer of data can occur simultaneously in both directions. The SPI interface also includes a slave select line that enables the master device to select the target slave device for communication.

6. Describe in words the software component of the lab

The main function in lab 6.1, implemented an accumulator on the FPGA LEDs. The while loop ensures that the accumulator logic runs continuously so that it is checked whether the accumulator button is being pressed. The first conditional statement checks whether the flag variable and accumulate button are both set to 0. If this condition is satisfied, it indicates that the accumulate button was pressed since it is active low. Consequently, the current sum is set to the previous sum added with the new values set onto the switches. In the next conditional statement, if the accumulate button is not pressed and the flag is set to 1, the flag is reset to 0. The accumulate button will then wait to be pressed again to go high so that the first conditional statement can be activated again. The last conditional statement is triggered when the sum overflows. It ensures that if a number that makes the sum over 255 is accumulated, then the accumulator will reset. For example, if 255 is accumulated with 1, then the accumulator will reset to 0. If it accumulates with 2, then the sum will be set to 1, and so on.

The MAXreg_wr function is responsible for writing data into a register. An array named arr is created with 2 elements. The two elements are reg+2 and val. reg + 2 indicates that we want to write to a register and val is the value to be written into a register. The function specifies a write length of two because the register and the value both need to be written. We then check if the

return code (checking if the write command is valid) is valid. His reading length is zero and we are not reading anything.

The MAXbytes_wr function is responsible for writing several bytes. An array with a size of nbytes + 1 is created to account for the number of bytes and the register to write to. A for loop is then created to assign all the data to this array. The alt_avalon_spi_command function then writes the data in the array and assigns a write length of the number bytes +1 to account for the register to write to. The return value is the data + nbytes.

The MAXreg_rd function is responsible for reading data from a register. A variable, val, is created to hold the value read from the register. The function specifies a write length of 1 and read length of 1 because the register to write to needs to be written and then we need to read a value from this register. We then check if the return code (checking if the read command is valid) is valid. If it is valid then we return the value.

The MAXbytes_rd function is responsible for reading multiple bytes of data. The function specifies a write length of 1 and read length of nbytes because the register to write to needs to be written and then we need to read the bytes of data.. We then check if the return code (checking if the read command is valid) is valid. If it is valid then we return the data+nbytes.

7. Answers to all INQ & Post lab questions

a. **What are the differences between the Nios II/e and Nios II/f CPUs?**
   These are two different types of a NIOS processor. the NIOS II/e is the "economy" version, which is the less efficient type as it's slower and requires more clock cycles in order to execute. The NIOS II/f is the "fast" version of the processor. This type is faster and utilizes efficiency and prioritizes having high performance.
b. **What advantage might on-chip memory have for program execution?**
   On chip memory is advantageous for program execution because it allows for faster access to the memory. It essentially has fewer data paths that need to be traveled across and thus, we can access memory faster and more frequently due to this efficiency.
c. **Note the bus connections coming from the NIOS II; is it a Von Neumann, "pure Harvard", or "modified Harvard" machine and why?**

The bus connection coming from the NIOS II is a modified Harvard machine. This is because both the instruction and data master are able to access the same memory, but the datapath (bus) for each of these components is separate. On the other hand the Von Nuemann model operates where the instruction and data master share a bus, and the pure Harvard implementation operates where both masters have their own separate buses and memory.

d. Why does SDRAM require constant refreshing?

The SDRAM requires constant refreshing because it is made up of capacitors and transistors. The refresh specifically is necessary for the capacitors, as there is leakage of charge/data over time, so refreshing the SDRAM allows us to keep our data accurate as we continue to access it.

e. What is the maximum theoretical transfer rate to the SDRAM according to the timings given?

We were provided with the following timing information:

- *Data width is 16 bits*
- *Access time is 5.4 ns*
- *740.74 MB/s*

In order to calculate the theoretical transfer rate, we can use the following equation:

$$(1/access\ rate)\ *\ bytes\ =\ 370.37\ MB/s$$

f. Note that there is one 32M*16 chips, so the total amount of memory should be 512MBits (64 Mbytes), make sure this is consistent with your above numbers; you will need to justify how you came up with 512 Mbit to your TA.

To accomplish this, we can use the following equation:

$$Rows\ *\ columns\ *\ chip\ selects\ *\ data\ width\ =\ total\ memory$$

For this equation, the banks are not needed in the calculation as they are just used for storing data. Now, to use this equation and have an answer of 512 MBits, we can set the parameters to be:

- *Data width=16*
- *rows= 13*
- *columns=10*
- *banks=4*
- *Chip selects=2*

$$2^{13} * 2^{10}\ * 2^{2} *\ 16\ \approx\ 512\ MBits$$

This information comes from the datasheet of the IP blocks. Once this calculation is complete with these entries, we get 512 MBits.

g. Note that while the on-chip memory needs access to both the data and program bus, the led peripheral only needs access to the data bus. Why might this be the case?

The LEDs only need to access the data bus as they are output ports to display data. Thus, the LEDs don't need to access the program bus. However, the on-chip memory needs access to both the data and program bus because it needs to be able to store both the program instructions as well as input and output data.

h. The SDRAM also cannot be run too slowly (below 50 MHz). Why might this be the case?

Part of the reason that the SDRAM cannot run too slowly is because it works synchronously with the clock, so the clock determines how long the SDRAM has to access memory, and how long it has to read and write data from memory. So, if the clock is too slow then the SDRAM will not be able to keep up with the transferring of data.

However, the most specific problem is that the SDRAM needs time to refresh so it can maintain the data that it is storing. This refresh time depends on the clock frequency. So, if the clock is running too slow, has a low frequency, this will result in the refresh cycle being too long, and this will cause the SDRAM to be refreshing data that was not intended to be refreshed, and therefore there will be data loss.

i. You must now make a second clock, which goes out to the SDRAM chip itself, as recommended by Figure 11. Make another output by clicking clk c1, and verify it has the same settings, except that the phase shift should be -1ns. This puts the clock going out to the SDRAM chip (clk c1) 1ns behind of the controller clock (clk c0). Why do we need to do this? Hint, check Altera Embedded Peripheral IP datasheet under SDRAM controller.

This delay is necessary because it accounts for a propagation delay between the controller and the SDRAM chip. This also gives the controller time to determine the control signals because the delay will make sure that the clock cycles will match up for the controller and chip when the controller receives the control signals. Thus, it ensures we get the correct values.

j. What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?

The NIOS II starts execution at the address x0800. We perform this step of assigning memory addresses after writing the code to ensure that the processor has a designated location to start execution in the event of a reset or exception. Additionally, this step helps prevent memory overlaps, ensuring that the processor can safely return from an operation without conflicts with other data structures in the system.

k. Look at the various segment (.bss, .heap, .rodata, .rwdata, .stack, .text), what does each section mean? Give an example of C code which places data into each segment, e.g. the code: const int my_constant[4] = {1, 2, 3, 4} will place 1, 2, 3, 4 into the .rodata segment.

- .bss
    - region with uninitialized data containing global and static variables that typically holds zeros.
        - static int sum;
- .heap
    - This segment is for dynamic memory allocation where the program can add additional memory to the OS in order to store more data. Memory can be allocated and deallocated.
        - int* **ptr** = (**int***)malloc(sizeof(int));
- .rodata
    - .rodata contains data that can only be read: “read only data”. This consists of constant values and string data.
        - const int x = 0;
- .rwdata
    - .rwdata contains data that can be either written to or read from. This can include global variables, static variables, arrays, etc. This segment can be modified during the program execution.
        - int x = 1;
- .stack
    - region of stack where the activation record and function calls are stored
        - int func(int x, int y){

- .text
    - region of text/strings
        - char x = "text";

l. You will need to determine the following parameters to instantiate the SDRAM controller. Refer to the DE10-Lite schematic and the IS42S16320D SDRAM (datasheet (PDF). Make sure you are looking at the correct part of the datasheet

SDRAM parameter	Short name	Parameter value (fill in from datasheet)
Data Width	[width]	16 bits
# of Rows	[norws]	13 rows
# of Columns	[ncols]	10 columns
# of Chip Selects	[ncs]	1 chip select
# of Banks	[nbanks]	4 banks

Parameters of SDRAM controller - from datasheet

m. Document any problems you encountered and your solutions to them

I think one of the biggest problems we encountered was just conceptually understanding 6.2, as we had to constantly refer to the IP User's Guide in order to understand the function, " int alt_avalon_spi_command" that was to be used to read and write into the registers. We had a bit of a design dilemma, where we were confused if we should call the function two times to do the read/write operations, but were able to optimize space by calling the function once, and using arrays and the concept of passing by reference in order to make the lab work successfully.

We also ran into an error with 6.1, as we did not initially have a flag value and this was causing the accumulation to be inaccurate. We added the flag as a way to tell the program that the switch value needs to be updated when the flag is high, and this addition ultimately lead to our lab working properly.

8. Design Resources and Statistics in table provided in the lab

LUT	**3397**

DSP	10
Memory (BRAM)	55,201/1,677,312
Flip - Flop	2532
Frequency	90.03 MHz
Static Power	96.44 mW
Dynamic Power	60.07 mW
Total Power	177.07 mW

Design resources and statistics table - data from the compilation summary in Quartus

9. Extra Credit

a. For extra credit, the provided ball.sv has a bug where it is possible to make the ball glitch off the top of the screen. Identify, document, and fix the bug in the code (and document your fix) in time for your lab report.

For the extra credit, we were instructed to fix a glitch in the code of the ball.sv file to allow the ball to bounce off when it hits the side of the screen, rather than just continuing forward. This bug was easily fixed by adding a “begin” and “end” statement to the “keycode” case in this file. Adding these “begin” and “end” allowed for the program to check the proceeding if statements, which handled the case of the ball trying to be moved beyond the scope of the screen. These statements check what keycode is currently being pressed, and if it reaches the end of the screen then the ball will “bounce” by going in the opposite direction of what the keycode instructs.

10. Hidden Questions

a. Note that Ball_Y_Motion in the above statement may have been changed at the same clock edge that is causing the assignment of Ball_Y_pos. Will the new value of Ball_Y_Motion be used, or the old? How will this impact behavior of the ball during a bounce, and how might that interact with a response to a keypress? Can you fix it? Give an answer in your Post-Lab.

Since it is possible that Ball_Y_Motion has been changed at the same time we are assigning Ball_Y_pos, then the new value of Ball_Y_Motion will not be used when making the assignment to the variable Ball_Y_pos. Since this is all happening at the same edge of the clock, the Ball_Y_Motion is being read at the same time we are assigning Ball_Y_pos, so it is not getting the updated value. This would cause the ball to continue going in the same direction as it was previously, no matter the keycode instruction that was just pressed, even if it was to change the direction. Thus, this causes a delay in the accuracy of the value of

Ball_Y_pos in response to a key press. In order to fix this, we could create temp variables for the Ball_Y_pos and Ball_Y_Motion to hold the updated and calculated values, and then set the temp variables equal to the actual variables that are used in the program. This ensures that Ball_Y_Motion and Ball_Y_pos contain the correct values.

11. Conclusion

a. Discuss the functionality of your design. If parts of your design did not work, discuss what could be done to fix it?

Our design works as intended. Both lab 6.1 and 6.2 are working correctly as we successfully created an accumulator and successfully made the ball move with key controls. There were definitely times during the lab where it wasn't working, but oftentimes these errors came down to minor mistakes like instantiation errors or having infinite loops with our while loop in lab 6.1.

b. Was there anything ambiguous, incorrect, or unnecessarily difficult in the lab manual or given materials which can be improved for next semester?

N/A