**ECE385**

Fall 2022

Experiment 6

# System-on-Chip with NIOS II in System Verilog

Lai Shu Xian
Section NL
TA: Nick Lu

## Introduction

The NIOS II processor running on the MAX-10 FPGA is a simple, IP based 32-bit CPU. The NIOS II processor is used to handle simple tasks: to read in the user input from external device (keyboard) and use that keycode to control the movement of a ball on the displayed monitor. The NIOS II processor also reads the keycode (based on user's presses on the keyboard) and displays those keycodes on the hex segment displays of the FPGA board.

The operation of the USB interface for this lab is handled in the software on NIOS II, what we worked on was setting up the interrupt based I/O interface for NIOS II to read/write MAX3421E's registers via a low-level serial peripheral interface (SPI) (which is also something we have to produce). The USB host driver provided was mostly complete and usable.

We also use a VGA display for the purposes of displaying a ball with simple background. For this lab, the entities and implementation required to draw onto the screen is provided. The entities involved included a simple colour mapper and a VGA controller. On a high-level, the VGA controller outputs the x and y-coordinates of which pixel we are currently drawing on our screen. With those coordinates, the colour mapper checks for within-bounds condition and produce foreground/background colour depending on the value of the pixel.

## Written Descriptions and Diagrams of NIOS-II System

Lab6.2 Hardware Component – Platform Designer module

I/O operation for lab 6.1

In the 6.1 section of the lab, we implemented an accumulator that takes in the input value of the switches and accumulates that with the current value of the "lit-up" LEDs. The switches act as inputs for the accumulator, whereas the LEDs act as inouts.

Upon the pressing of the "Accumulate" button, the value of the switches is summed with current value of LEDs and that sum updates the status of the LEDs.

This program is done in NIOS II software development environment whereas the hardware connections is set up in Platform designer using PIO (programmable I/O) that uses parallel communication. The PIO blocks are used to read data from the switches and buttons, then perform the addition logic and eventually pass that LED status information to the LEDs.

NIOS II interaction with MAX3421E USB chip and VGA components

The interaction we implemented between NIOS II and MAX3421E is the reading/writing to the MAX3421E registers. The NIOS II can perform single and multiple bytes read and write. We used a readily available function alt_avalon_spi_command to perform the reads and writes based on the input parameters. The register of which we want to perform the write/read will be the MOSI signal, and the slave (MAX3421E) will respond accordingly.

The interaction between NIOS II and VGA controller is mainly only through the clock and reset signals.

## SPI protocol description

SPI communication between devices is based upon a master-slave relationship where the master transmits instructions to one or multiple slaves. In a SPI communication, there are four transmission lines in which signals are exchanged:

1. Master Output/Slave Input (MOSI)
2. Master Input/Slave Output (MISO)
3. Clock (CLK)
4. Chip Select (CS)

CLK signal is configured by the master and fed to the slave to determine the rate of information transfer. Each bit is transferred every clock cycle. With the presence of a clock, SPI is a synchronous communication protocol.

CS signal is an active-low signal from master to slave as an indicator that the master is communicating to the slave.

MOSI and MISO are like their names, signals being exchanged between the master and slave devices.

## C code that we written

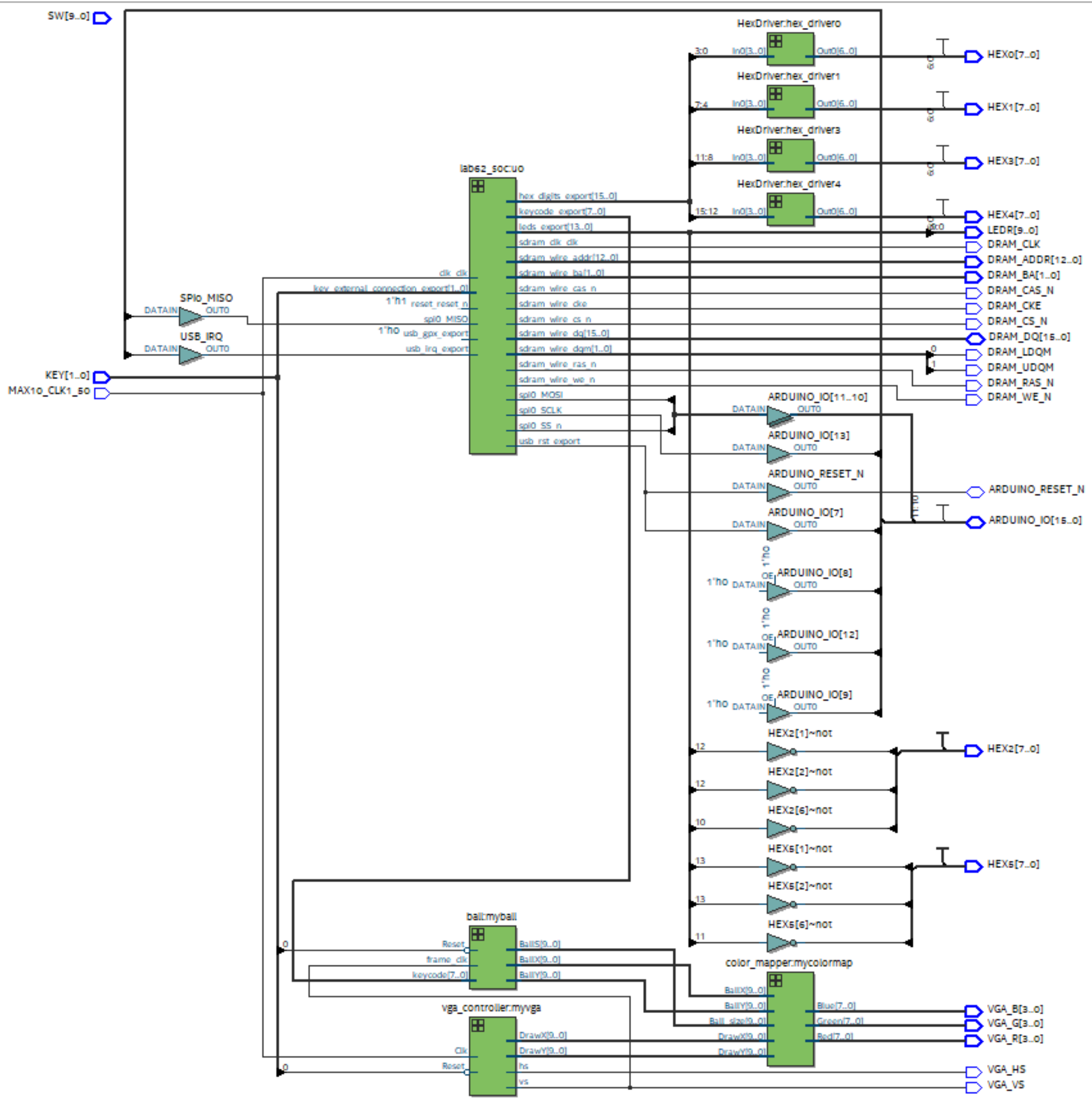
## VGA operation (Ball, Color Mapper, VGA controller)

Based on the horizontal and vertical sync pulse for the VGA controller.sv as inputs, the coordinates being drawn on the screen is determined (conventionally named DrawX and DrawY). The DrawX and DrawY signals are fed to the colour_mapper.sv to determine which colour to output.

Based on the keycode (WASD movement direction), the ball.sv module calculates the ball coordinates on the screen and feeds that information for the colour mapper as inputs for coordinates.

The colour_mapper.sv takes in the coordinates of the centre of the ball, as well as the coordinates of the pixel that we are intending to draw. By checking whether the draw coordinates fall within the ball, corresponding RGB values are displayed. In other words, if the draw coordinates are within the ball, then the RGB values outputted represent the colour of the ball. If coordinates are outside the ball, then background colour is displayed instead.

## Top level Block Diagram



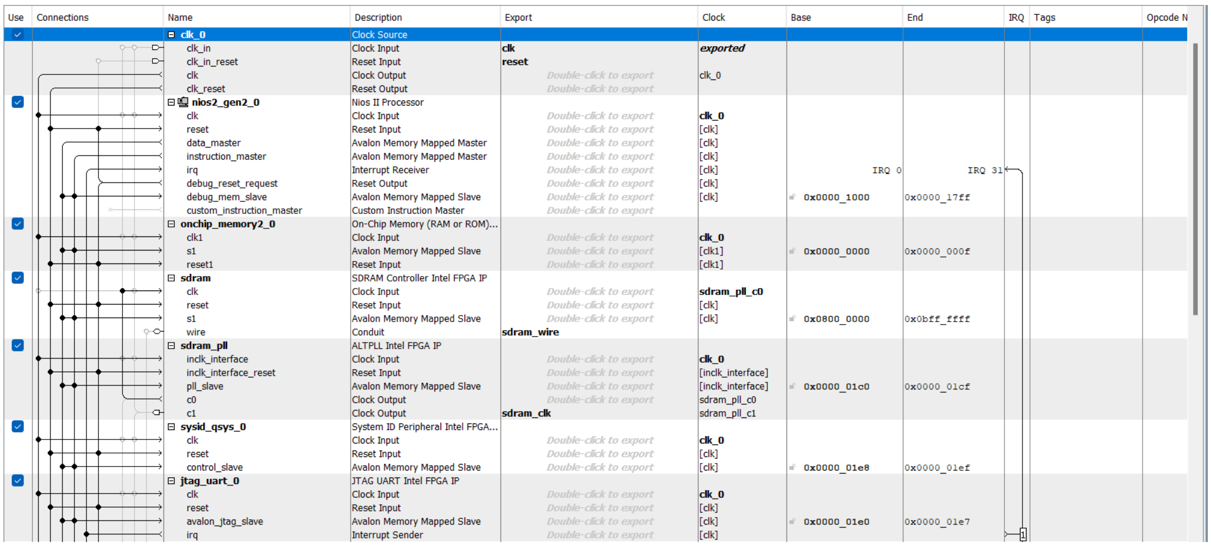
## Written description of all .sv Modules

| | |
|---|---|
| Module: VGA_controller.sv<br><br>Inputs: Clk, Reset<br><br>Outputs: hs, vs, pixel, blank, sync, [9:0] DrawX, [9:0] DrawY | Description:<br>Takes in the Clk and Reset signals, reset signals is used to set the screen position to top left corner.<br><br>Within clock cycles, update the horizontal counter and vertical counter and check bounds condition. If a new vertical/horizontal shift is required, then the vertical/horizontal sync is set.<br><br>DrawX and DrawY is set accordingly based on the calculation of horizontal counter and vertical counter. |

| | |
|---|---|
| | Purpose: Provides important information on the coordinates of the pixel being drawn. |
| Module: lab62.sv<br><br>Inputs: MAX10_CLK1_50, [1: 0]KEY, [ 9: 0] SW<br><br>Outputs: [ 9: 0] LEDR,<br>///////// HEX /////////<br>[ 7: 0] HEX0,<br>[ 7: 0] HEX1,<br>[ 7: 0] HEX2,<br>[ 7: 0] HEX3,<br>[ 7: 0] HEX4,<br>[ 7: 0] HEX5,<br><br>///////// SDRAM /////////<br>DRAM_CLK,<br>DRAM_CKE,<br>[12: 0] DRAM_ADDR,<br>[ 1: 0] DRAM_BA,<br>DRAM_LDQM,<br>DRAM_UDQM,<br>DRAM_CS_N,<br>DRAM_WE_N,<br>DRAM_CAS_N,<br>DRAM_RAS_N,<br><br>///////// VGA /////////<br>VGA_HS,<br>VGA_VS,<br>VGA_R,<br>VGA_G,<br>VGA_B<br><br>In-outs:<br>[15: 0] DRAM_DQ,<br>[15: 0] ARDUINO_IO,<br>ARDUINO_RESET_N | Description:<br>Connects everything to where they need to be.<br><br>Purpose: Top-level entity that wraps all the required modules together. |
| Module: Color_Mapper.sv<br><br>Inputs: [9:0] BallX, BallY, DrawX, DrawY, Ball_size<br><br>Outputs: [7:0] Red, Green, Blue | Description:<br>Perform calculations based on ball coordinates and the current pixel coordinates to determine whether we are presenting foreground or background.<br><br>Purpose:<br>Logic for foreground and background to output their corresponding RGB values. |

| | |
|---|---|
| Module: Ball.sv<br><br>Inputs: Reset, frame_clk, [7:0] keycode<br><br>Outputs: [9:0] BallX, BallY, BallS | Description:<br>Depending on the keycode we pressed WASD, calculate the new ball motion and therefore new coordinates.<br><br>Purpose:<br>Provide new ball position to be used as inputs in ColorMapper.sv. |
| Module: HexDriver.sv<br><br>Inputs: [3:0] In<br><br>Outputs: [6:0] Out | Description: Takes in a 4-bit binary number and outputs its corresponding hexadecimal value on a 7-segment LED display setup using switch case statements.<br><br>Purpose: To light up the correct LED segments to show a 4-bit binary number as a single hex number on the LED display segment. |
| Module:<br>Lab62_soc.v<br><br>Inputs:<br>Clk_clk,<br>[1:0]key_external_connection_export,<br>reset_reset_n,<br>Spi0_MISO, usb_gpx_export,<br>usb_irq_export<br><br>Outputs:<br>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,  sdram_wire_dqm, sdram_wire_ras_n, sdram_wire_we_n, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export<br><br>inout: [15:0] sdram_wire_dq, | Description:<br>Exported signals are used for other modules such as keycode and hex digits signals. SDRAM wires are for SDRAM connections and related modules, SPI0 wires are for SPI connections and related modules, USB wires are for USB connections and related modules.<br><br>Purpose:<br>File generated by Platform Designer, used as SOC, and be instantiated within the top-level design entity. |

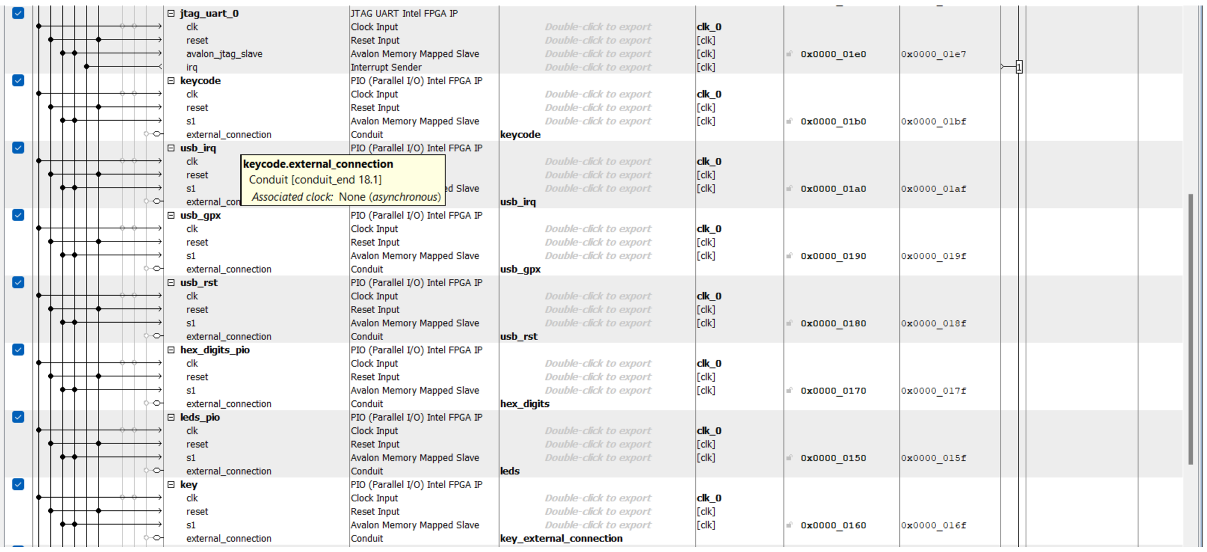
## System Level Block Diagram



| Use | Connections | Name | Description | Export | Clock | Base | End | IRQ | Tags | Opcode N |
|---|---|---|---|---|---|---|---|---|---|---|
| ✓ | | ⊟ **clk_0** | Clock Source | | | | | | | |
| | | clk_in | Clock Input | **clk** | ***exported*** | | | | | |
| | | clk_in_reset | Reset Input | **reset** | | | | | | |
| | | clk | Clock Output | *Double-click to export* | clk_0 | | | | | |
| | | clk_reset | Reset Output | *Double-click to export* | | | | | | |
| ✓ | | ⊟ **nios2_gen2_0** | Nios II Processor | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | data_master | Avalon Memory Mapped Master | *Double-click to export* | [clk] | | | | | |
| | | instruction_master | Avalon Memory Mapped Master | *Double-click to export* | [clk] | | | | | |
| | | irq | Interrupt Receiver | *Double-click to export* | [clk] | IRQ 0 | IRQ 31 | | | |
| | | debug_reset_request | Reset Output | *Double-click to export* | [clk] | | | | | |
| | | debug_mem_slave | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x0000_1000 | 0x0000_17ff | | | |
| | | custom_instruction_master | Custom Instruction Master | *Double-click to export* | | | | | | |
| ✓ | | ⊟ **onchip_memory2_0** | On-Chip Memory (RAM or ROM)... | | | | | | | |
| | | clk1 | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | s1 | Avalon Memory Mapped Slave | *Double-click to export* | [clk1] | 0x0000_0000 | 0x0000_000f | | | |
| | | reset1 | Reset Input | *Double-click to export* | [clk1] | | | | | |
| ✓ | | ⊟ **sdram** | SDRAM Controller Intel FPGA IP | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **sdram_pll_c0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | s1 | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x0800_0000 | 0x0bff_ffff | | | |
| | | wire | Conduit | **sdram_wire** | | | | | | |
| ✓ | | ⊟ **sdram_pll** | ALTPLL Intel FPGA IP | | | | | | | |
| | | inclk_interface | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | inclk_interface_reset | Reset Input | *Double-click to export* | [inclk_interface] | | | | | |
| | | pll_slave | Avalon Memory Mapped Slave | *Double-click to export* | [inclk_interface] | 0x0000_01c0 | 0x0000_01cf | | | |
| | | c0 | Clock Output | *Double-click to export* | sdram_pll_c0 | | | | | |
| | | c1 | Clock Output | **sdram_clk** | sdram_pll_c1 | | | | | |
| ✓ | | ⊟ **sysid_qsys_0** | System ID Peripheral Intel FPGA... | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | control_slave | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x0000_01e8 | 0x0000_01ef | | | |
| ✓ | | ⊟ **jtag_uart_0** | JTAG UART Intel FPGA IP | | | | | | | |
| | | clk | Clock Input | *Double-click to export* | **clk_0** | | | | | |
| | | reset | Reset Input | *Double-click to export* | [clk] | | | | | |
| | | avalon_jtag_slave | Avalon Memory Mapped Slave | *Double-click to export* | [clk] | 0x0000_01e0 | 0x0000_01e7 | | | |
| | | irq | Interrupt Sender | *Double-click to export* | [clk] | | | 1 | | |

| | |
|---|---|
| Block Name: clk_0<br><br>Type: Clock Source | Description: Acts as the global clock for the NIOS II processor and all other entities except for the SDRAM chip. The clock frequency we use for the lab is 50 MHz. Also contains a clock reset signal, which serves as the reset signal to all the other entities. |
| Block Name: nios2_gen2_0<br><br>Type: NIOS II processor | Description: The hardware block that contains the CPU and the supporting hardware. Contains general signals that determines the setup of a basic processor such as data line, instruction line, as well as interrupt request line. |
| Block Name: onchip_memory2_0<br><br>Type: On-chip Memory (RAM or ROM) | Description: Acts as on-chip memory block to provide lower latency and higher bandwidth in memory accesses for the processor. |
| Block Name: sdram<br><br>Type: SDRAM Controller Intel FPGA IP | Description: Off chip memory to store the software program(s) that we want to run (e.g. the program that displays blinking LED/accumulator) since on_chip_memory is very limited. |
| Block Name: sdram_pll<br><br>Type: ALTPLL Intel FPGA IP | Description: Intel provided IP that acts as an intermediary that outputs two clocks so to overcome the clock skew due to the board layout. One clock is used direct for the |

| | |
|---|---|
| | SDRAM chip (c1), the other clock is used for the SDRAM controller core (c0). The clock that goes to the SDRAM chip has a phase shift 1ns behind the controller clock. |
| Block Name: sysid_qsys_0<br><br>Type: System ID Peripheral Intel FPGA IP | Description: System ID checker to ensure the compatibility between hardware and software so that we do not load software onto FPGA that has an incompatible NIOS II configuration. |
| Block Name: jtag_uart_0<br><br>Type: JTAG UART Intel FPGA IP | Description: An interface that gives us the capability to use console commands from NIOS II. In general serves as a debugging component for software development. |

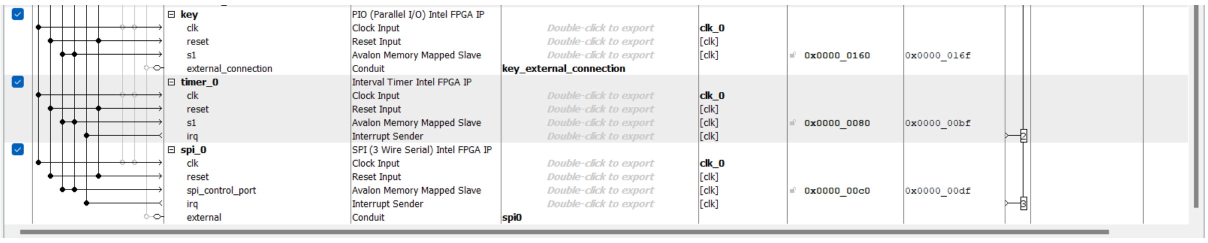


| | | | | | | |
|---|---|---|---|---|---|---|
| ⊟ jtag_uart_0 | JTAG UART Intel FPGA IP | | | | | |
| clk | Clock Input | Double-click to export | clk_0 | | | |
| reset | Reset Input | Double-click to export | [clk] | | | |
| avalon_jtag_slave | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_01e0 | 0x0000_01e7 | |
| irq | Interrupt Sender | Double-click to export | [clk] | | | 1 |
| ⊟ keycode | PIO (Parallel I/O) Intel FPGA IP | | | | | |
| clk | Clock Input | Double-click to export | clk_0 | | | |
| reset | Reset Input | Double-click to export | [clk] | | | |
| s1 | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_01b0 | 0x0000_01bf | |
| external_connection | Conduit | keycode | | | | |
| ⊟ usb_irq | PIO (Parallel I/O) Intel FPGA IP | | | | | |
| clk | | Double-click to export | clk_0 | | | |
| reset | | Double-click to export | [clk] | | | |
| s1 | d Slave | Double-click to export | [clk] | 0x0000_01a0 | 0x0000_01af | |
| external_con | | usb_irq | | | | |
| ⊟ usb_gpx | PIO (Parallel I/O) Intel FPGA IP | | | | | |
| clk | Clock Input | Double-click to export | clk_0 | | | |
| reset | Reset Input | Double-click to export | [clk] | | | |
| s1 | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_0190 | 0x0000_019f | |
| external_connection | Conduit | usb_gpx | | | | |
| ⊟ usb_rst | PIO (Parallel I/O) Intel FPGA IP | | | | | |
| clk | Clock Input | Double-click to export | clk_0 | | | |
| reset | Reset Input | Double-click to export | [clk] | | | |
| s1 | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_0180 | 0x0000_018f | |
| external_connection | Conduit | usb_rst | | | | |
| ⊟ hex_digits_pio | PIO (Parallel I/O) Intel FPGA IP | | | | | |
| clk | Clock Input | Double-click to export | clk_0 | | | |
| reset | Reset Input | Double-click to export | [clk] | | | |
| s1 | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_0170 | 0x0000_017f | |
| external_connection | Conduit | hex_digits | | | | |
| ⊟ leds_pio | PIO (Parallel I/O) Intel FPGA IP | | | | | |
| clk | Clock Input | Double-click to export | clk_0 | | | |
| reset | Reset Input | Double-click to export | [clk] | | | |
| s1 | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_0150 | 0x0000_015f | |
| external_connection | Conduit | leds | | | | |
| ⊟ key | PIO (Parallel I/O) Intel FPGA IP | | | | | |
| clk | Clock Input | Double-click to export | clk_0 | | | |
| reset | Reset Input | Double-click to export | [clk] | | | |
| s1 | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_0160 | 0x0000_016f | |
| external_connection | Conduit | key_external_connection | | | | |

**keycode.external_connection**
Conduit [conduit_end 18.1]
*Associated clock:* None (*asynchronous*)

| | |
|---|---|
| Block Name: keycode<br><br>Type: Programmed Input/Output (Parallel I/O) | Description:<br>Output for the keycode will be exported and used as input for the ball.sv module (indicates direction of which the ball moves). |
| Block Name: usb_irq<br><br>Type: Programmed Input/Output (Parallel I/O) | Description:<br>Necessary connection ports to the USB chipset (MAX3241E), this indicates the interrupt request line. This is assigned to the 10th interrupt request line of the Arduino. |

| | |
|---|---|
| Block Name: usb_gpx<br><br>Type: Programmed Input/Output (Parallel I/O) | Description:<br>Necessary connection ports to the MAX3241E but not used for standard USB host so set to 1’b0. |
| Block Name: usb_rst<br><br>Type: Programmed Input/Output (Parallel I/O) | Description:<br>Necessary connection ports to the MAX3241E. Connects to the Arduino Reset. |
| Block Name: hex_digits_pio<br><br>Type: Programmed Input/Output (Parallel I/O) | Description:<br>Acts as output signals to display the digits on the hex displays. The digits represent the keycode of the keyboard presses. These output signals are used as inputs for the hex driver modules. |
| Block Name: leds_pio<br><br>Type: Programmed Input/Output (Parallel I/O) | Description:<br>Similar to hex_digits_pio but represent the LEDs on the FPGA board. |
| Block Name: key<br><br>Type: Programmed Input/Output (Parallel I/O) | Description:<br>Represents the input signals coming from the two buttons on the FPGA board, the first (KEY[0]) signal signifies the reset, the second is not used in this lab. |



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| ✓ | ⊟ **key** | PIO (Parallel I/O) Intel FPGA IP | | | | | |
| | clk | Clock Input | Double-click to export | **clk_0** | | | |
| | reset | Reset Input | Double-click to export | [clk] | | | |
| | s1 | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_0160 | 0x0000_016f | |
| | external_connection | Conduit | **key_external_connection** | | | | |
| ✓ | ⊟ **timer_0** | Interval Timer Intel FPGA IP | | | | | |
| | clk | Clock Input | Double-click to export | **clk_0** | | | |
| | reset | Reset Input | Double-click to export | [clk] | | | |
| | s1 | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_0080 | 0x0000_00bf | |
| | irq | Interrupt Sender | Double-click to export | [clk] | | | 2 |
| ✓ | ⊟ **spi_0** | SPI (3 Wire Serial) Intel FPGA IP | | | | | |
| | clk | Clock Input | Double-click to export | **clk_0** | | | |
| | reset | Reset Input | Double-click to export | [clk] | | | |
| | spi_control_port | Avalon Memory Mapped Slave | Double-click to export | [clk] | 0x0000_00c0 | 0x0000_00df | |
| | irq | Interrupt Sender | Double-click to export | [clk] | | | 3 |
| | external | Conduit | **spi0** | | | | |

| | |
|---|---|
| Block Name: timer_0<br><br>Type: Interval Timer Intel FPGA IP | Description:<br>Timer needed in the USB driver code to keep track of different time-outs that USB requires. |
| Block Name: spi_0<br><br>Type: SPI (3 Wire Serial) Intel FPGA IP | Description:<br>SPI port for the communication interface between the USB chip and the NIOS II processor. |

**Software Component of the lab (snapshots of code provided after explanation)**

If we consider the functionality of the software C code that we wrote for NIOS II processor, we technically only wrote "two" functions, one for read and one for write. We in reality have four functions because we perform two ways of reading and writing each:

- Single byte read/write from/to a register
- Multiple bytes read/write from/to a register

The four functions that we have are:

1. void MAXreg_wr(BYTE reg, BYTE val);
2. BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data);
3. BYTE MAXreg_rd(BYTE reg)
4. BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data);

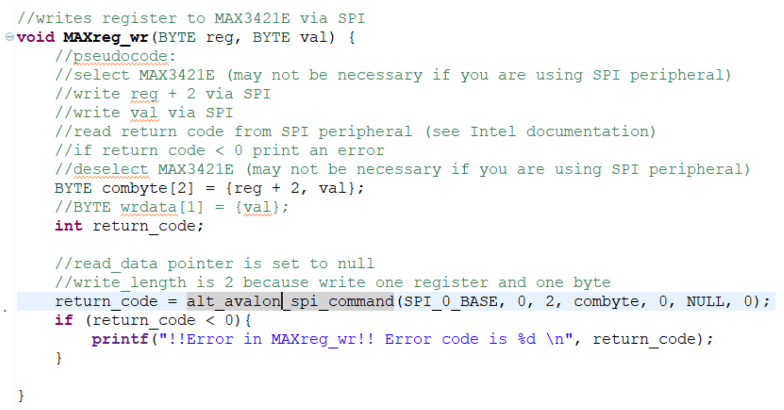
The functionalities and actual implementation of all four functions are included in the next page.

Across all the four functions that we coded for read/write, the alt_avalon_spi_command plays an integral role as a means of communication per the SPI protocol. Quoting from the Intel's documentation:

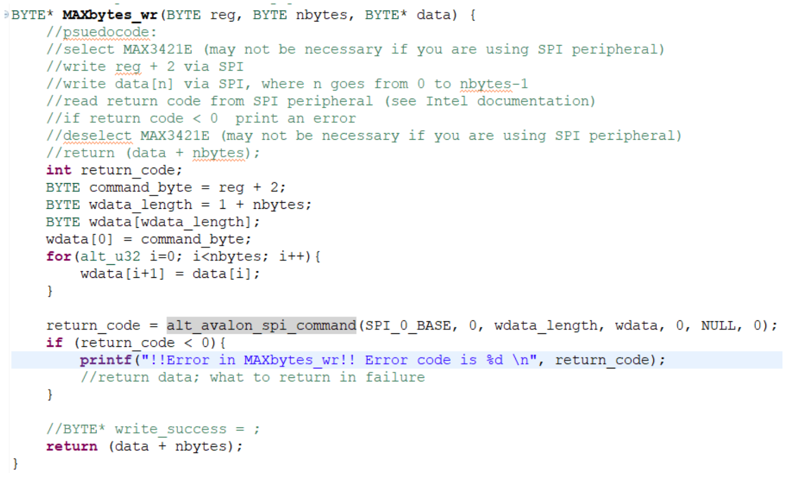
> This function (alt_avalon_spi_command()) performs a control sequence on the SPI bus. It supports only SPI hosts with data width less than or equal to 8 bits. A single call to this function writes a data buffer of arbitrary length to the mosi port, and then reads back an arbitrary amount of data from the miso port.

On writes, we pass in the register which we intend to write to and the byte(s) that we want to write into the register. These parameters are wrapped in an array and used as argument for the alt_avalon_spi_command. If multiple bytes are written to the register, the endpoint of which the bytes were written to is returned. Else, nothing is returned.

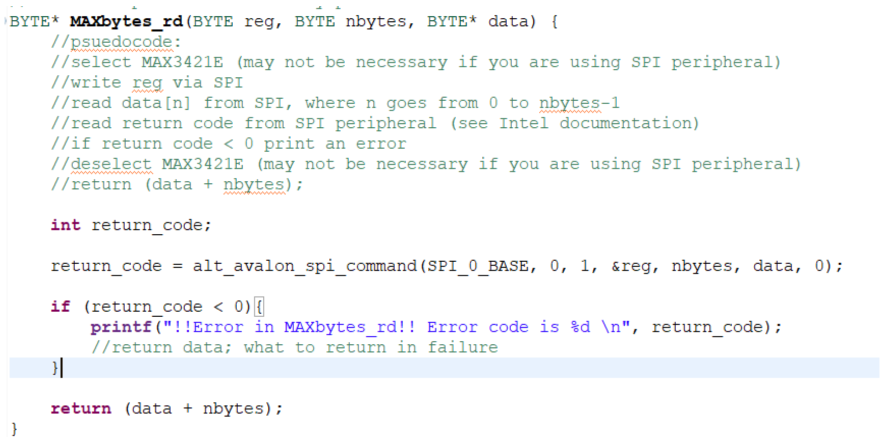
On reads, we pass in the register which we intend to read from and a placeholder to hold the read value. On reading however, the alt_avalon_spi_command first writes the register information first via MOSI channel to the slave, then only the read value is transmitted back via the MISO channel.



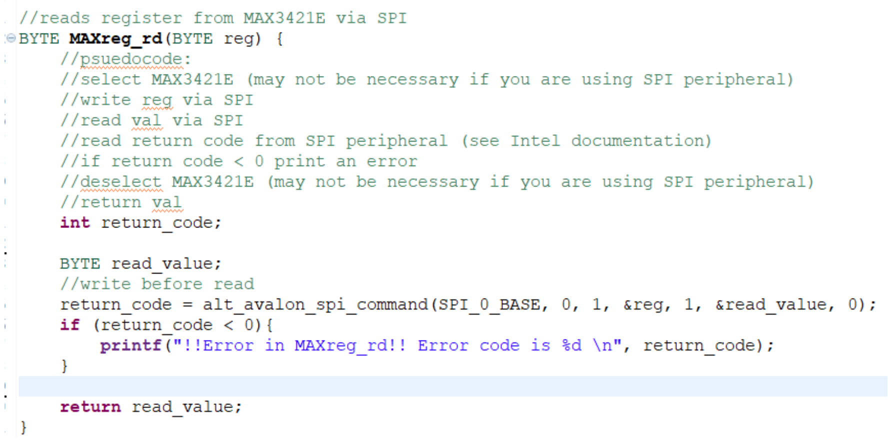
```
//writes register to MAX3421E via SPI
void MAXreg_wr(BYTE reg, BYTE val) {
    //pseudocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg + 2 via SPI
    //write val via SPI
    //read return code from SPI peripheral (see Intel documentation)
    //if return code < 0 print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    BYTE combyte[2] = {reg + 2, val};
    //BYTE wrdata[1] = {val};
    int return_code;

    //read_data pointer is set to null
    //write_length is 2 because write one register and one byte
    return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 2, combyte, 0, NULL, 0);
    if (return_code < 0){
        printf("!!Error in MAXreg_wr!! Error code is %d \n", return_code);
    }

}
```



```
BYTE* MAXbytes_wr(BYTE reg, BYTE nbytes, BYTE* data) {
    //pseudocode:
    //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)
    //if return code < 0  print an error
    //deselect MAX3421E (may not be necessary if you are using SPI peripheral)
    //return (data + nbytes);
    int return_code;
    BYTE command_byte = reg + 2;
    BYTE wdata_length = 1 + nbytes;
    BYTE wdata[wdata_length];
    wdata[0] = command_byte;
    for(alt_u32 i=0; i<nbytes; i++){
        wdata[i+1] = data[i];
    }

    return_code = alt_avalon_spi_command(SPI_0_BASE, 0, wdata_length, wdata, 0, NULL, 0);
    if (return_code < 0){
        printf("!!Error in MAXbytes_wr!! Error code is %d \n", return_code);
        //return data; what to return in failure
    }

    //BYTE* write_success = ;
    return (data + nbytes);
}
```



```
BYTE* MAXbytes_rd(BYTE reg, BYTE nbytes, BYTE* data) {
    //psuedocode:
    //select MAX3421E (may not be necessary if you are using SPI peripheral)
    //write reg via SPI
    //read data[n] from SPI, where n goes from 0 to nbytes-1
    //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 (data + nbytes);

    int return_code;

    return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, nbytes, data, 0);

    if (return_code < 0){
        printf("!!Error in MAXbytes_rd!! Error code is %d \n", return_code);
        //return data; what to return in failure
    }

    return (data + nbytes);
}
```



```
//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
    int return_code;

    BYTE read_value;
    //write before read
    return_code = alt_avalon_spi_command(SPI_0_BASE, 0, 1, &reg, 1, &read_value, 0);
    if (return_code < 0){
        printf("!!Error in MAXreg_rd!! Error code is %d \n", return_code);
    }

    return read_value;
}
```

**Post-lab/INQ Questions**

| | | |
|---|---|---|
| Data Width | Width | 16 |
| # of rows | Nrows | 13 |
| # of columns | Ncols | 10 |
| # of chip selects | Ncs | 1 |
| # of banks | nbanks | 4 |

Calculation to get 512Mbit:

(2^13) * (2^10) * 1 * 4 * 16 = 536870912 = 512Mbit

1. What are the differences between NIOS II/e and NIOS II/f?

Nios II/e is resource optimized whereas NIOS II/f is performance optimized. NIOS II/f has more powerful features in terms of operations it can perform as well as memory capacity. For example, NIOS II/f has hardware for multiply and division, as well as the shadow register sets, and cache for instructions and data.

2. What advantage might on-chip memory have for program execution?

On chip memory provides lower latency (faster memory access) compared to SDRAM as well as higher memory bandwidth (can access more memory at a time) because it does not have to wait compared to off-chip memory.

3. Is NIOS II a Von Neumann, pure Harvard, or modified Harvard?

NIOS II is a modified Harvard machine because the contents of the instruction memory is accessible as data.

4. Why does SDRAM require constant refreshing?

SDRAM requires constant refreshing because of the hardware used in storing DRAM memory are capacitors. Capacitors can experience leakage of charge over time. Therefore, it is important to constant "recharge".

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

We have a data width of 16 bits and given access time is 5.4 ns, maximum theoretical transfer rate to SDRAM = (approximately) 24000 bits per second

6. SDRAM cannot be run too slowly below 50 MHz, why might this be the case?

If the SDRAM is run too slowly below the clock frequency, we might lose data due to the slow refreshing to retain our memory.

7. Why do we need to make a second clock for the SDRAM chip that is 1ns behind the controller clock?

The second clock is necessary to alleviate the clock skey between the SDRAM controller core and the SDRAM chip, especially when we are using a high clock rate. This is to ensure the SDRAM clock edge arrives after synchronous signals have been stabilized.

8. What address does the NIOS II start execution from? Why do we do this step (assign the reset and exception vectors) after assigning the addresses?

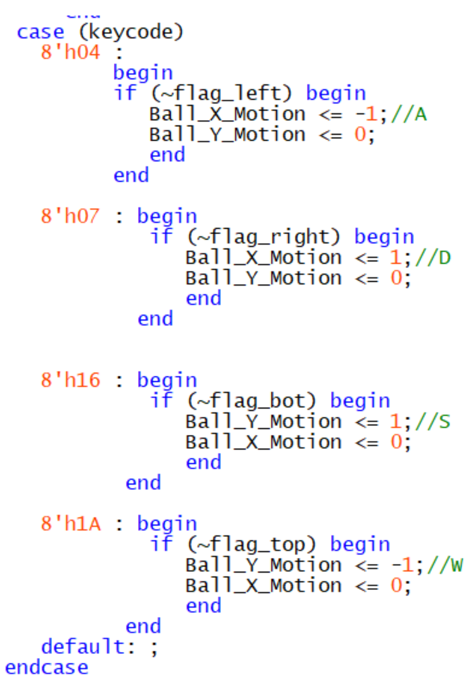
NIOS II start execution from the address set at Reset vector offset, in this case it will be 0x0000_0000. This is because the calculation to determine where the reset vector resides cannot be done until all the memory components of the system is set in place.

**Bug log**

| | |
|---|---|
| Bug description:<br>When setting up the lab for 6.1 for the LED to be blinking, the LED did not blink. Did not know that the assign base addresses can change as I add different blocks and I simply used the provided C code function. | Resolve:<br>Fixed the pointer of the LED_PIO to match with the addresses in Platform designer. |
| Bug description:<br>In lab 6.2, if the user holds the movement keys without lifting, the ball on screen will continue move in the same direction and exceeds the bounds when we expect it to bounce back.<br><br>This is due to the ball motion being updated with the non-blocking assignment in the always_ff block without safety checking. | Resolve:<br>Added an additional always_comb block that sets the flags to check for bouncing condition. These flags are then used in the always_ff block to use as an additional conditional check before updating the ball's motions.<br><br>Note: code is included below the bug log table. |

Code modifications for bug 2:

```
always_comb begin
  if ( (Ball_Y_Pos + Ball_Size) >= Ball_Y_Max ) flag_bot = 1'b1;
  else flag_bot = 1'b0;

  if ( (Ball_Y_Pos - Ball_Size) <= Ball_Y_Min ) flag_top = 1'b1;
  else flag_top = 1'b0;

  if ( (Ball_X_Pos + Ball_Size) >= Ball_X_Max ) flag_right = 1'b1;
  else flag_right = 1'b0;

  if ( (Ball_X_Pos - Ball_Size) <= Ball_X_Min ) flag_left = 1'b1;
  else flag_left = 1'b0;
end
```



```
 case (keycode)
   8'h04 :
         begin
         if (~flag_left) begin
            Ball_X_Motion <= -1;//A
            Ball_Y_Motion <= 0;
            end
         end

   8'h07 : begin
            if (~flag_right) begin
               Ball_X_Motion <= 1;//D
               Ball_Y_Motion <= 0;
               end
         end


   8'h16 : begin
            if (~flag_bot) begin
               Ball_Y_Motion <= 1;//S
               Ball_X_Motion <= 0;
               end
          end

   8'h1A : begin
            if (~flag_top) begin
               Ball_Y_Motion <= -1;//W
               Ball_X_Motion <= 0;
               end
          end
   default: ;
endcase
```

## Design Resources and Statistics

| | |
|---|---|
| LUT | 3438 |
| DSP | 10 |
| Memory (BRAM) | 55296 bits |
| Flip-flop | 2528 |
| Frequency | 66.15MHz |
| Static Power | 96.51mW |
| Dynamic Power | 59.54mW |
| Total Power | 177.71mW |

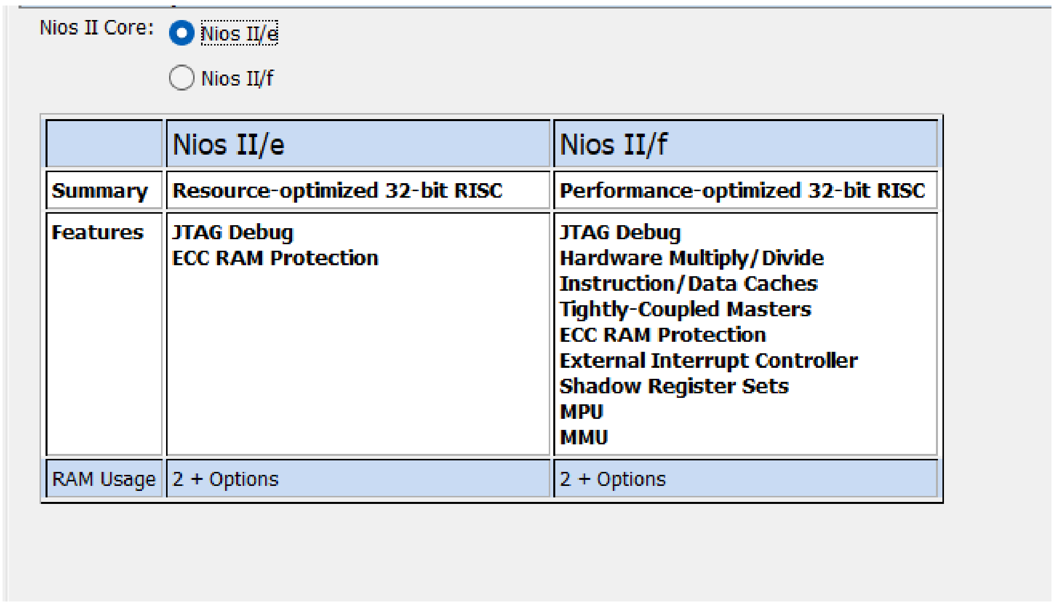
Question: PowerAnalyzer specs, what do they represent and why do we not care about I/O Thermal Power Dissipation? PowerAnalyzer Report confidence: Low: user provided insufficient toggle rate data?

## Conclusion

My circuit functions completely as expected, and I managed to fix the bug that ball.sv originally has. This is an extremely pointless lab within the timeframe where we learned more about the lab when we write the lab report but learned nothing when we were doing the lab. And even the lab report is unnecessarily long, and we don't rationally assume graders to grade them carefully, yet we are expected to write them with all our effort. Again, nobody cares, and nothing will change.

I have run into several times where some staff told me to separate the reads and write of the alt_avalon_spi_command, and some told me to combine them. And somehow along that process of seemingly doing the same things in different ways, and it worked. This lab only makes sense if this class itself is given at least 6 credit hours.

- What are the differences between NIOS II/e and NIOS II/f?

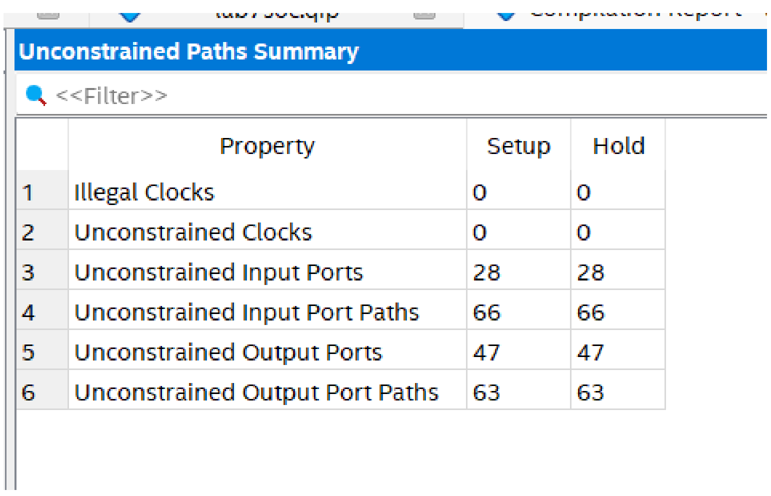


Nios II Core: [● Nios II/e] ○ Nios II/f

| | Nios II/e | Nios II/f |
|---|---|---|
| **Summary** | **Resource-optimized 32-bit RISC** | **Performance-optimized 32-bit RISC** |
| **Features** | **JTAG Debug**<br>**ECC RAM Protection** | **JTAG Debug**<br>**Hardware Multiply/Divide**<br>**Instruction/Data Caches**<br>**Tightly-Coupled Masters**<br>**ECC RAM Protection**<br>**External Interrupt Controller**<br>**Shadow Register Sets**<br>**MPU**<br>**MMU** |
| RAM Usage | 2 + Options | 2 + Options |

- What advantage might on-chip memory have for program execution?
- Note the bus connections coming from the NIOS II, is it a Von Neumann, pure Harvard, or modified Harvard? Why?
- Why does SDRAM require constant refreshing?

- What is the maximum theoretical transfer rate to the SDRAM according to the timings given?
- What address does the NIOS II start execution from? Why do we do this step after assigning the addresses?



**Unconstrained Paths Summary**

<<Filter>>

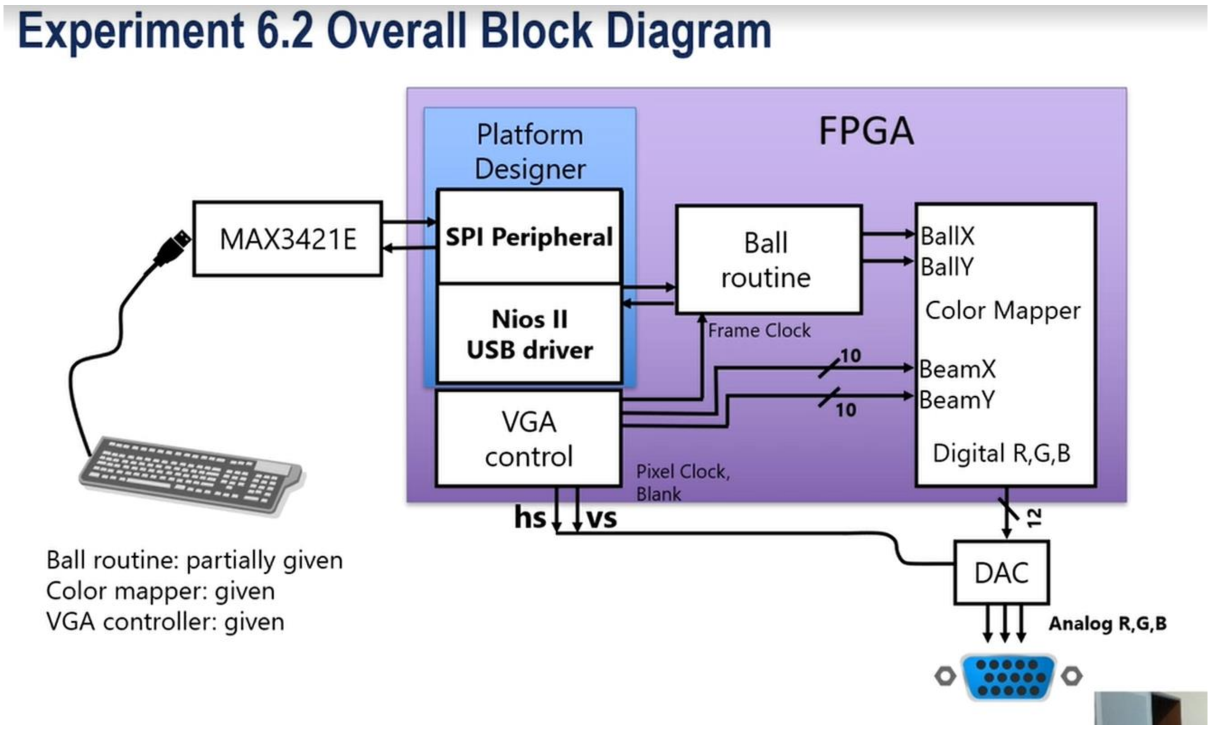
| | Property | Setup | Hold |
|---|---|---|---|
| 1 | Illegal Clocks | 0 | 0 |
| 2 | Unconstrained Clocks | 0 | 0 |
| 3 | Unconstrained Input Ports | 28 | 28 |
| 4 | Unconstrained Input Port Paths | 66 | 66 |
| 5 | Unconstrained Output Ports | 47 | 47 |
| 6 | Unconstrained Output Port Paths | 63 | 63 |

Part 6.2

JTAG-UART: we can use the terminal of the host computer (the one running eclipse) to communicate with the NIOS II

Reason for interrupts set up in JTAG UART = transmit/receive text over the console is very slow and we do not want to block the CPU

Interval Timer IP = needed in the USB driver code to keep track of the different time-outs that that USB requires.



## Experiment 6.2 Overall Block Diagram

Ball routine: partially given
Color mapper: given
VGA controller: given