Iowa State University
Electrical And Computer Engineering

CprE 488

Machine Problem 0

February 4, 2023

Group : S1-4

Member 1: Nathan Thoms

Member 2: Taylor Johnson

Member 3: Jack Cassidy

Member 4 : John Lavigne

MP 0 Questions

1.) *In your report, describe how nes_bootloader.c currently works. Using a similar approach as what is presented in Chapter 1 of the Wolf textbook, draw a high-level structural diagram. How does NESCore_Callback_OutputFrame() get called?*

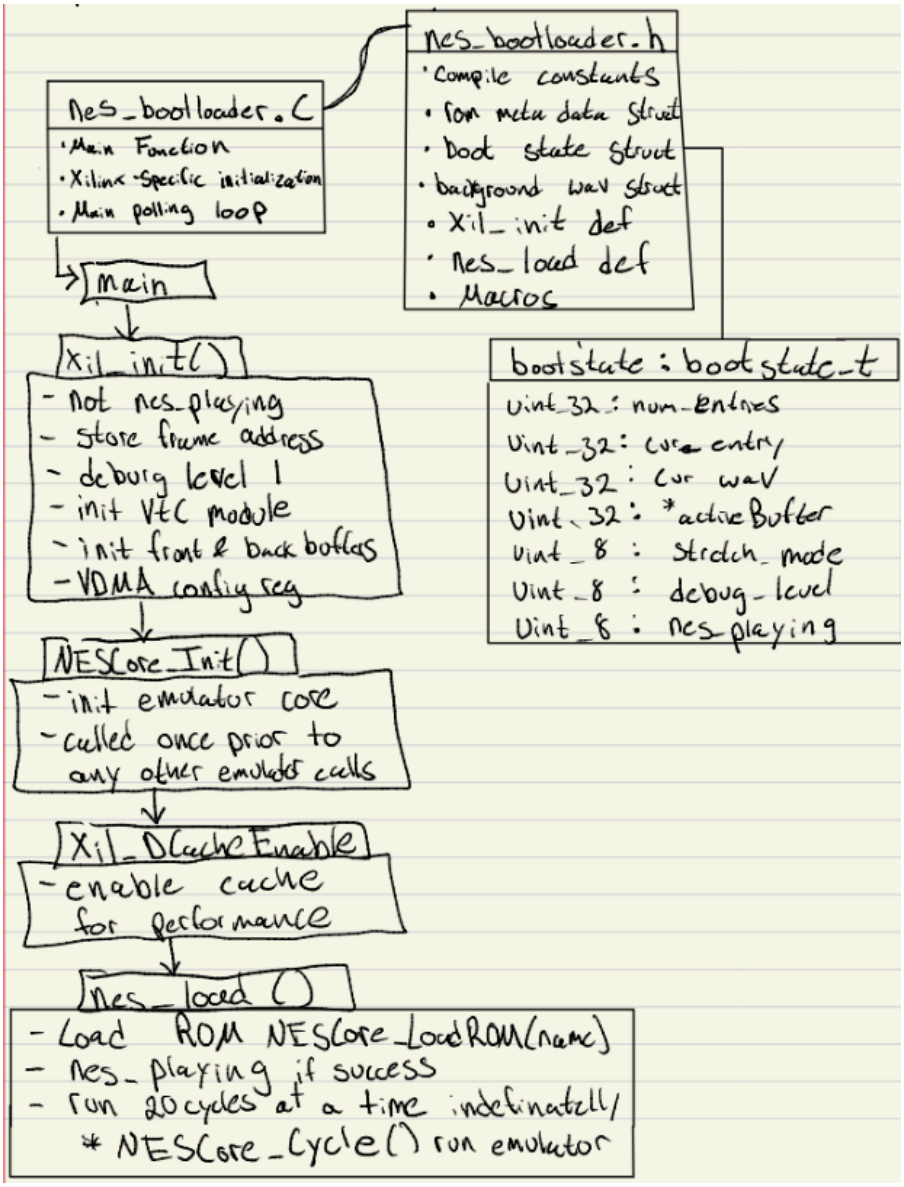

Figure 1 - *Structural explanation of bootloader.c functionality.*

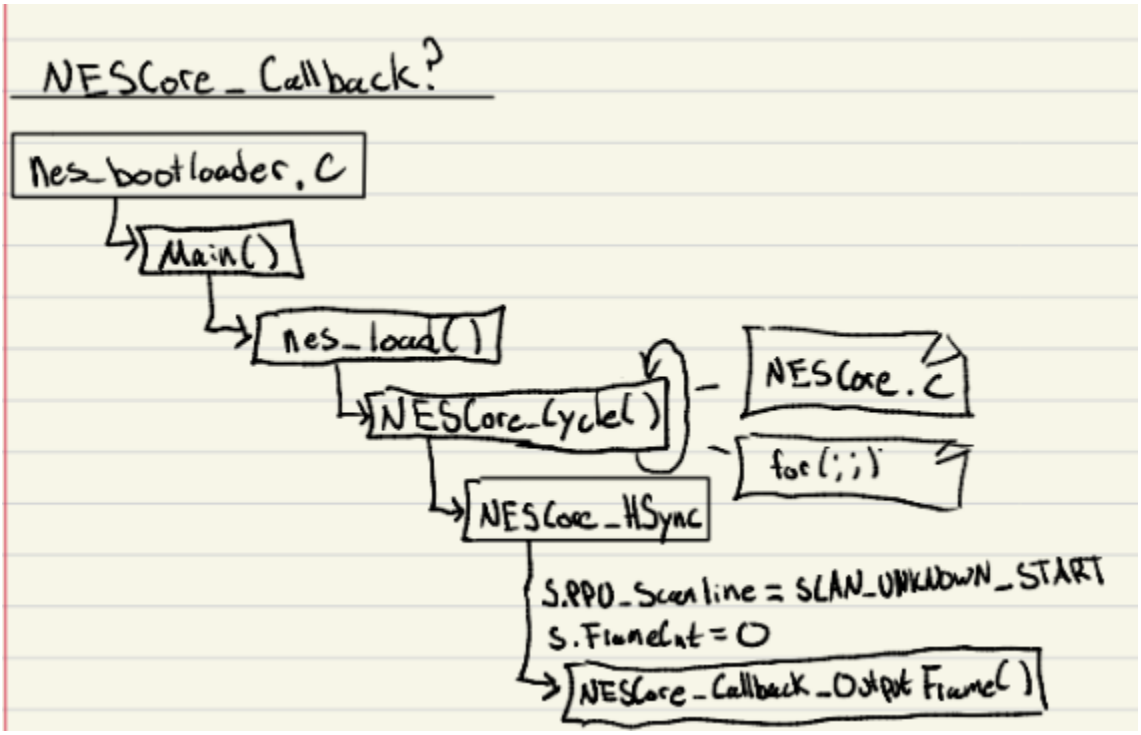

Figure 2 - *Calling NESCore_Callback_OutputFrame()*

2.) *Click three different green boxes and in your writeup, describe what configuration options are available and how they may be potentially useful in an embedded system.*

I/O Peripherals – Turn on and off the list of I/O devices depending on which one you would want to use. If the piece of hardware uses a certain I/O peripheral it would be useful to configure it and don't use things you don't need.

General Settings – Gives you a general description of what each setting does and gives you the option to change most of them based on what the user wants. If you need a certain setting for the embedded system, you can configure it and learn how it would works

DDR2/3, LPDDR2 – Controls a lot of aspects from the board's memory. Such as the DRAM Bus Width, the memory type and memory part. Useful in an embedded application where you need to specify the memory needed for the system and how it functions

3.) *Are these buttons, LEDs, and switches connected via the PS subsystem or the PL subsystem? Briefly defend your answer. Note also that all three peripherals appear to be the same exact IP type (axi_gpio) – how can this be possible?*

PL Subsystem because the peripherals are connected directly to the AXI controller and not a CPU device (Processing System). All three peripherals fall under the same IP axi_gpio because they all involve interaction with gpio pins, however the IP's are configured to a specific set of pins differentiating the IP's from one another.

4.) *Based on the datasheet and the address map shown in the "Address Editor" (mentioned in instruction 7 of Step 2: Use Designer Assistance), how would you (in software) read the current state of the switches? Be specific.*

Given the master base address in the address editor, in software you can read the data from the base address through pointer usage. The pointer is initialized to the base address given, and dereferencing the pointer will allow you to fetch the contents at that address. The value of that register holds the current states of the switches; with bit 0 corresponding to switch 0's state.

5.) *In your writeup, use this feature and describe what print() does, and how. Why do you believe this function is used by Xilinx for their Hello World application, as opposed to the more conventional printf() function?*

It works by going through a while loop that goes through the pointer and every bit sends it UART until every character gets printed out. Utilizes the hardware better than a traditional printf statement in C. printf doesn't have to worry about the base address while print does.

6.) *Take a screen capture of an LED wire turning on? Can you turn the LED on and off fast enough to get a screen capture of the Logic Analyzer displaying this pulse?*

If so, then provide this screen capture as well. For how long does the pulse stay high?

We were able to capture the LED pulse signal using the logic analyzer in Vivado. The cycle time of the pulse was 2 ms with a duty cycle of 50%. A screenshot of the captured pulse is provided below in **figure 3**.

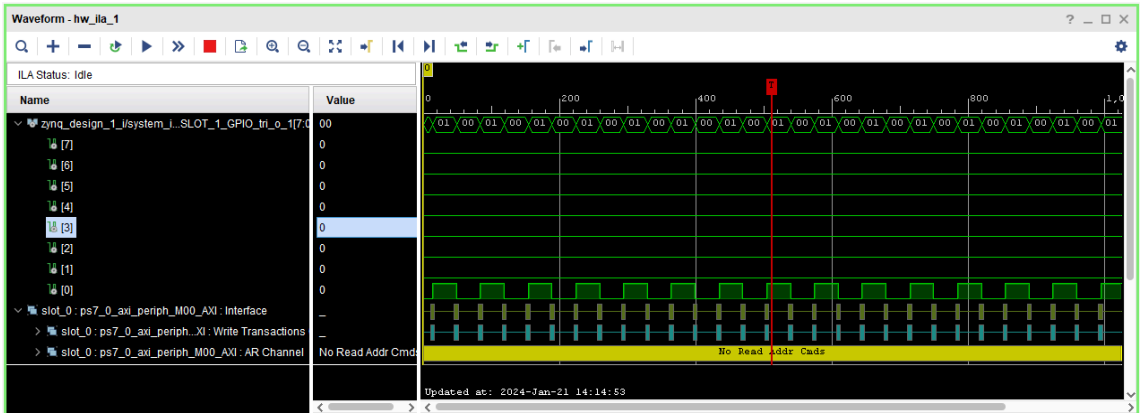

Figure 3 - *Using the ILA to capture pulsing LED*

7.) *Modify the hello_world application to also interface with the switches, buttons, and LEDs that are configured in the programmable logic. For example, have the application print out the state of the switches when a button is pressed, or light up certain LEDs given an input integer.*

```
for (int i = 0; i < 100; i++) {
    sleep(1);
    //sw_vals = *sw_base_addr;
    btn_vals = *btn_base_addr;
    *led_base_addr = btn_vals;
    sprintf(str, "%X\n\r", btn_vals);
    print(str);
}
```

Figure 4 - *Toggling LEDs based on button states*

The image provided in **figure 4** above provides a subsection of the modified hello_world application. The body of the for loop reads the state of the buttons and assigns the results to the LEDs state register. Button 1 ,when pressed, will turn on LED 1.

8.) *In VIVADO, add these peripherals to your project , connect and then configure them to generate a 640x480 output signal.*

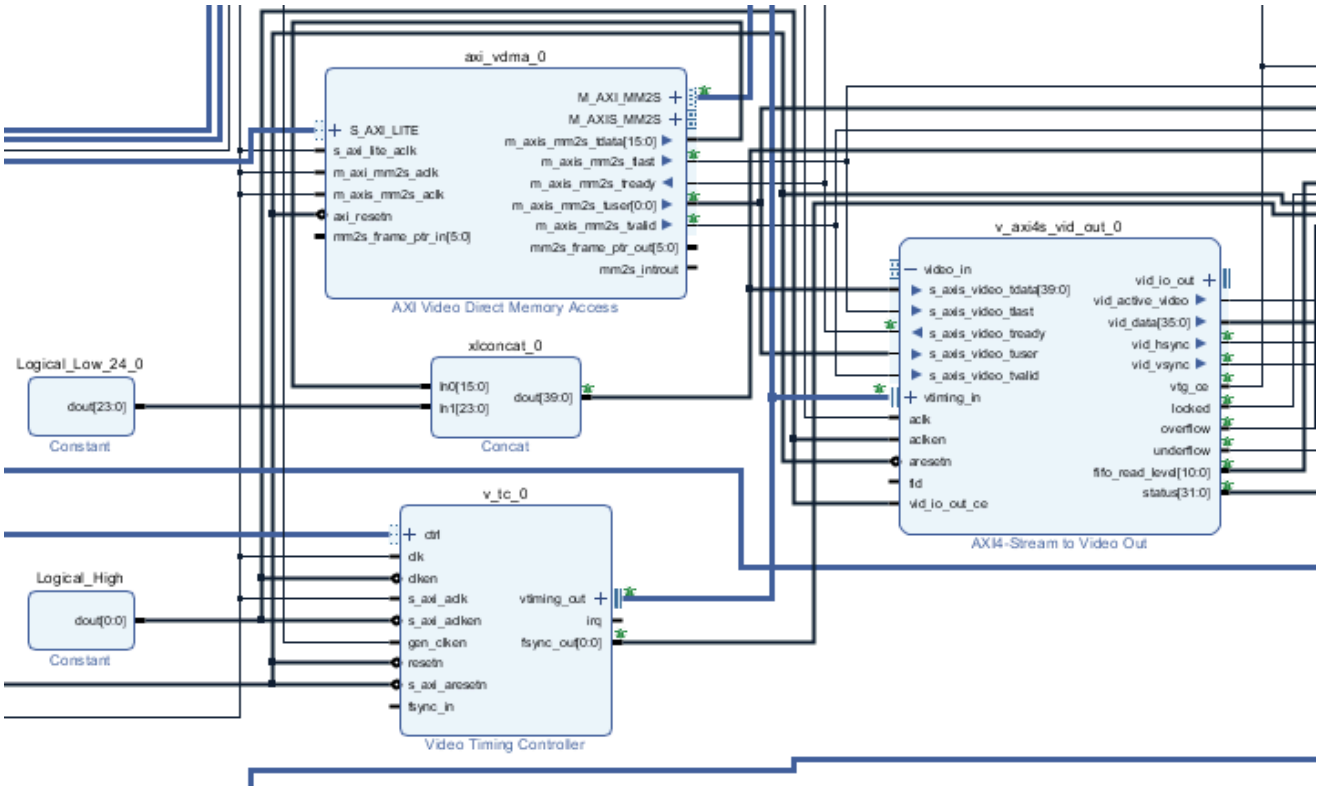

Figure 5 - *Interconnection of VTC, VDMA, Video Out IPs*

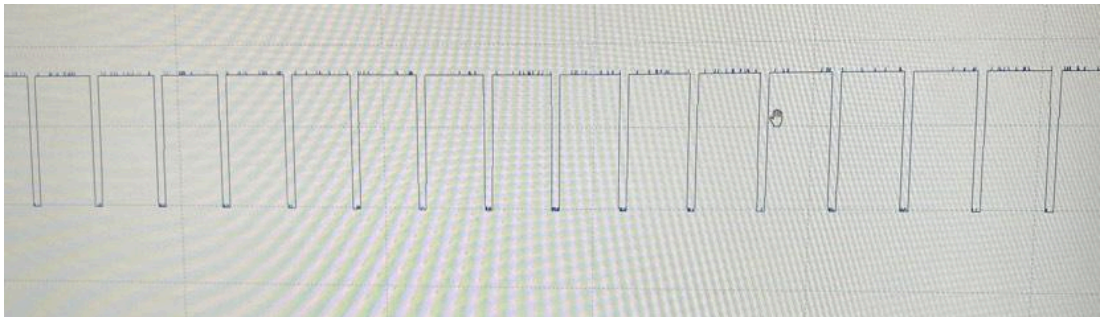

Figure 6 - *Measured horizontal sync pulse for 640x480 output*

Figures 5 & 6 above demonstrate the proper interconnection of the three IPs to produce the horizontal timing waveform shown.

9.) *Modify the configuration registers for correct VDMA operation, and in your writeup, provide a justification based on the VDMA documentation for how you set these values.*

```
// Simple function abstraction by Vendor for writing VDMA registers
XAxiVdma_WriteReg(XPAR_AXI_VDMA_0_BASEADDR, XAXIVDMA_CR_OFFSET,  (u32)0x00000003);  // Read Channel: VDMA MM2S Circular Mode and Start bits set,
XAxiVdma_WriteReg(XPAR_AXI_VDMA_0_BASEADDR, XAXIVDMA_HI_FRMBUF_OFFSET, (u32)0x00000000);  // Read Channel: VDMA MM2S Reg_Index
XAxiVdma_WriteReg(XPAR_AXI_VDMA_0_BASEADDR, XAXIVDMA_MM2S_ADDR_OFFSET + XAXIVDMA_START_ADDR_OFFSET, start_addr);  // Read Channel: VDMA MM2S Frar
XAxiVdma_WriteReg(XPAR_AXI_VDMA_0_BASEADDR, XAXIVDMA_MM2S_ADDR_OFFSET + XAXIVDMA_STRD_FRMDLY_OFFSET, (u32)(HORZ_SIZE_PIXELS * BYTES_PER_PIXEL));
XAxiVdma_WriteReg(XPAR_AXI_VDMA_0_BASEADDR, XAXIVDMA_MM2S_ADDR_OFFSET + XAXIVDMA_HSIZE_OFFSET,  (u32)(HORZ_SIZE_PIXELS * BYTES_PER_PIXEL));  // F
XAxiVdma_WriteReg(XPAR_AXI_VDMA_0_BASEADDR, XAXIVDMA_MM2S_ADDR_OFFSET + XAXIVDMA_VSIZE_OFFSET, (u32)VERT_SIZE_LINES);  // Read Channel: VDMA MM2S
```

Figure 7 - *Using SW to configure the VDMA registers*

CR Register - Enable the start bit and circular mode operation. Genlock and other interrupt enablers were not necessary in our read only usage.

Register Index - Set to 0, read from the bottom set of frame addresses. This is important if you want to read from the first frame address written to the start address register assignment.

Start Address - Store the base address of test_image to the register corresponding to the first frame location. This allows the VDMA to read the pixel values assigned in vga_test.c.

Horizontal Size - Tells the VDMA the number of bytes making up a row. The horizontal stride divided by the MM2S output width gives you the number of pixels per row which can be used by the VDMA to set EOL flags in its communication with the video out IP.

Stride - This sounded identical to the horizontal size in our usage, for that reason the same value assigned to horizontal size was assigned to the stride.

Vertical Size - This register tells the VDMA how many lines make up a frame. In our case this is 480, and is used by the VDMA to set SOF outputs to the video out IP.

10.) *In your writeup, explain how you converted these color values to valid values for the 16-bit framebuffer.*

Color channels were converted from 8 bits to 4 bits by truncating the 4 least significant bits and rounding up the 4 most significant bits by the lower bits.

For example: *0x3E -> 0x4 Round Up & 0xC8 -> 0xC Round Down*

11.) Modify the nes_bootloader code such that the NES games are reasonably playable. In your report, describe your general approach to implementing both of the NESCore_Callback functions.

```
void NESCore_Callback_OutputFrame(word *WorkFrame) {

    extern uint16_t NesPalette3[];
    uint32_t i, j, k, l;
    uint16_t *ptr = (uint16_t *)FBUFFER_BASEADDR;
    uint16_t tpixel;
    int pixel_index = 0;

    // Active Frame
    for (i = 0; i < NES_DISP_HEIGHT; i++) {
        for (j = 0; j < NES_DISP_WIDTH; j++) {
            // Grab a temporary pixel using the color palette lookup table.
            tpixel = NesPalette3[WorkFrame[NES_DISP_WIDTH * i + j]];
            ptr[pixel_index+BORDER_WIDTH] = tpixel;
            ptr[pixel_index + 1+BORDER_WIDTH] = tpixel;
            ptr[pixel_index+VGA_WIDTH+BORDER_WIDTH] = tpixel;
            ptr[pixel_index+VGA_WIDTH+1+BORDER_WIDTH] = tpixel;
            pixel_index+=2;
        }
        pixel_index += BORDER_WIDTH * 2 + VGA_WIDTH;
    }

    pixel_index = 0;
    for (i = 0; i < VGA_HEIGHT; i++) {
        for (j = 0; j < BORDER_WIDTH; j++) {
            ptr[pixel_index] = 0x0000;
            ptr[pixel_index + BORDER_WIDTH+ACT_WIDTH] = 0x0000;
            pixel_index++;
        }
        pixel_index += VGA_WIDTH - BORDER_WIDTH;
    }


    // Flush the cache since VDMA does not play nicely with the cache
    Xil_DCacheFlush();

    return;
}
```

In this code, it was originally mapped to a 256x240 image and we were tasked to figure out how to map it to the 640x480 frame buffer so we could get the whole screen to display it. To solve this, we would place the pixel in a 2x2 grid because if we placed a pixel in a grid, the image would map to 512x480 and if 64 pixels get added on each size for a border then we would get the 640x480 frame buffer that we want. We would also want to account for the borderwidth so we had the pixel index go up by the border width to account for the new border.

```
void NESCore_Callback_InputPadState(dword *pdwPad1, dword *pdwPad2) {

    dword current_states = 0;
    int strt_btn, sel_btn, other_btns, up_btn, down_btn, left_btn, right_btn, b_btn, a_btn, sel_sw;
    int* btn_base_addr =  XPAR_AXI_GPIO_1_BASEADDR;
    int* sw_base_addr = XPAR_AXI_GPIO_2_BASEADDR;
    b_btn = XGpioPs_ReadPin(&GPIO, 50);
    a_btn = XGpioPs_ReadPin(&GPIO, 51);
    other_btns = *btn_base_addr;

    up_btn = other_btns & 0x10;
    down_btn = other_btns & 0x02;
    left_btn = other_btns & 0x04;
    right_btn = other_btns & 0x08;

    strt_btn = other_btns & 0x01;
    sel_sw = *sw_base_addr & 0x01;

    xil_printf("ALL BTNS : %d\n\r", other_btns);

    current_states |= (sel_sw != 0)? NCTL_SELECT:0;
    current_states |= (a_btn != 0)? NCTL_A: 0;
    current_states |= (b_btn != 0)? NCTL_B: 0;
    current_states |= (up_btn != 0)? NCTL_UP: 0;
    current_states |= (down_btn != 0)? NCTL_DOWN: 0;
    current_states |= (left_btn != 0)? NCTL_LEFT: 0;
    current_states |= (right_btn != 0)? NCTL_RIGHT: 0;
    current_states |= (strt_btn != 0)? NCTL_START: 0;

    xil_printf("CUR STATES : %d\n\r", current_states);
    // Assign input to controller object.
    *pdwPad1 = current_states;

    *pdwPad2 = 0;

    return;
}
```

This function was necessary to initialize the buttons and switches required to emulate the NES games. We did this by initializing the button base address and the switch base address. Then using these we initialized a dpad, start button, and select switch by setting each button/switch variable to its corresponding address. However because the A and B buttons utilize a PS subsystem we initialized them using XGpioPs_ReadPin. Finally we checked the current states of each button/switch initialized and outputted the results into pdwPad1 to allow for easy reading when emulating.

Team - Percent Contributions

Nathan Thoms - 40%

Taylor Johnson - 25%

Jack Cassidy - 20%

John Lavigne - 15%