Jared Danner
Brian Bradford
Leo Freier

MP-0 Report

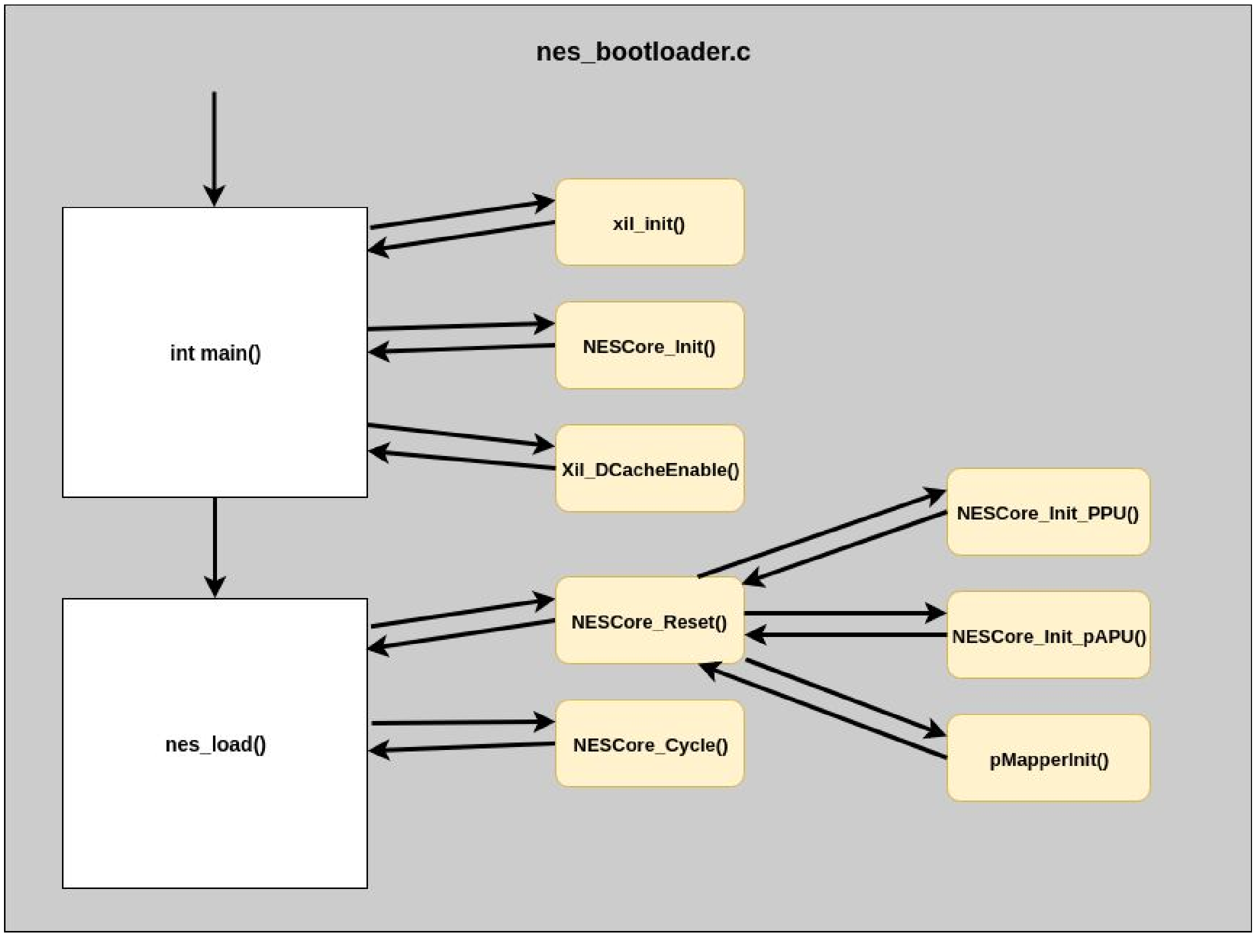
2) Getting Started.

- Describe the NES_bootloader.c

The bootloader begins by going through three different initialize methods `xil_init()`, `NESCore_Init()`, & `Xil_DCacheEnable()`. The first method `xil_init()` sets up and configures the bootloader along with the needed peripherals and finishes with the initialization of the different display buffers to be used in the emulator. The second method `NESCore_Init()` which is responsible for allocating and defaulting memory tables to be used by the emulator. The setup method `Xil_DCacheEnable()` that sets either the L1 or L2 cache to be active by configuring and defaulting certain registers.

The final section in the `main()` method is the polling infinite while loop that calls `nes_load()`. This `nes_load()` method begins to load the zelda.nes file out of memory while temporarily disabling the L memory cache. After the L memory cache has been re-enabled and the ROM image has been loaded, the bootloader `nes_load()` will reset itself by resetting RAM to a clear slate. Beyond this, the `NESCore_Reset()` method initializes the picture processing unit, Pseudo-Audio processing unit, memory mapper module, and finally the onboard CPU. To provide a brief description of each initialized unit, the PPU (picture processing unit) is the module responsible for managing the picture memory and its corresponding registers. The pAPU (pseudo-audio unit) is responsible for configuring and controlling the audio waveforms that will be played over speakers. Lastly, the memory mapper module is responsible for allocating and mapping the the memory to be used by the emulator.

- Bootloader Diagram



- How does `NESCore_Callback_OutputFrame()` get called?

`nes_load()` which is the main polling loop, calls `NESCore_Cycle()` in a loop for 20 cycles. `NESCore_Cycle()` instructs the emulator to perform its cycles until halted, it loops calling `NESCore_Step()` which executes CPU cycles and calls `NESCore_HSync()`. This function calls `NES_Callback_OutputFrame()` if the NES `PPU_Scanline` state is `SCAN_UNKNOWN_START` and the NES state `Framecnt` is 0.

3) Creating a Hardware System.

- Configuration options in the PS:
    - Clock Configuration

All of the different clocks can be configured with different sources and frequencies. The main use here is changing frequencies of clocks like the CPU. Lower frequencies will reduce power consumption, which is usually a large constraint of embedded systems. The frequency of the timer can also be changed which could potentially prove to be useful.

-
    - DDR Configuration

There are numerous DDR configuration options available. Memory type can be changed from DDR3 into other operating modes such as low voltage mode which can save power. Bus width, burst length, and memory clock frequency are other notable configurations that can decrease the size of the design. Memory can potentially be optimized through these kind of configurations to assist with speed, size, power consumption, and portability of the embedded system.

-
    - MIO Configuration

I/O peripherals can be configured based on the type of peripheral. Power to the peripherals can be changed to reduce power consumptions, but the MIO itself can be changed, allowing for multiple ports to be used at once. The ‘M’ in MIO stands for multiplexed, so changing the MIO for a peripheral would change which other peripherals are multiplexed together. This allows the embedded system to be further optimized to reduce redundancy and implementation size/overhead of the system. Speed of the I/O devices can be changed and different available peripherals can be enabled or disabled if they are needed or not.

- Connected via PS or PL subsystem?

These buttons, LEDs, and switches are implemented in the Programmable Logic, which are connected to physical pins on the FPGA. They are connected from PL to the PS subsystem via an AXI Interconnect IP Core. All of the modules can be of the same IP type because the AXI GPIO core’s map into the address space at different offset addresses. This allows the PS subsystem to read the state of all three AXI GPIOs.

- In software, how would you read the current state of switches?

Since each AXI GPIO core has its own address space, the PS subsystem can index the corresponding space for the switches. From this according to the AXI GPIO Register Space, the PS subsystem could read the state of the switches from Address Space Offset 0x0000, which is the GPIO_DATA register.

4) Creating a Software Development Environment for your Hardware System.

- Print() statement.
    - What it does.

The print() method writes the complete raw character array to the output device (hardware).

- How it works.

It loops over the given pointer and outputs character-by-character, incrementing the pointer each loop.

- Why does Xilinx use print() instead of printf().

Xilinx opts for their print() function over the printf() function because traditionally printf() from stdio.h brings with it high overhead and additional code that is not needed for this system. Since this program is running on a embedded platform, one would want the overhead and amount of resources used by a simple print function to be as low and small as possible.

5) Develop an interactive software application for your hardware system.

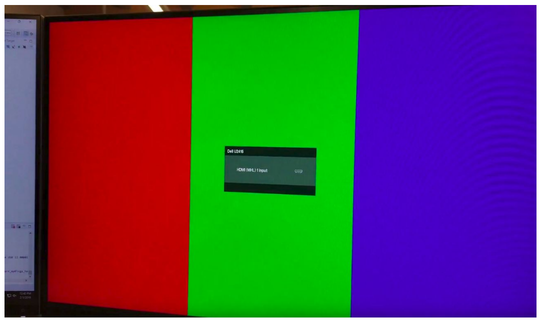
- Switches and Button Interaction

In order to get used to using the button and switches, we first hooked up the switches to the LEDs so each switch corresponds to the LED of the same name. To go beyond this basic functionality, we XORed the switches and the buttons so the LEDs will turn off if a button is pushed while a switch is turning the LED on. If a button is pushed corresponding to an LED without a switch triggering the LED, the LED will also turn on. We configured a little more with reading switch values, so if all the switches are on the XOR functionality will stop and all the LEDs will stay on.

6) Video Display

- VGA Display (RGB)

In order to get the test_image of red, green, & blue columns to display over the VGA cable, our group needed to figure out the needed hardware configurations, which of the bytes in the 16-bit color code display which colors, and overcome the issue of having the VGA port subtract out colors that were not black. Once these issues were fixed/completed, the below image shows the VGA output that our group generated.

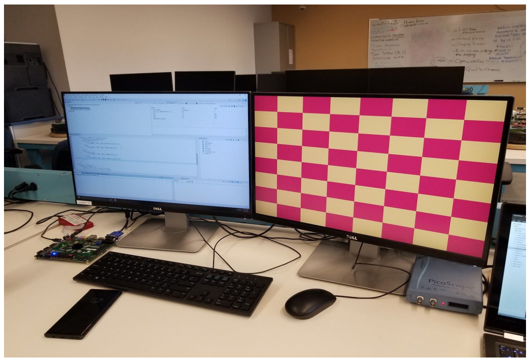


- VDMA Registers

Registers in the VDMA IP needed to be set based on our configuration. Most of this was figured through comparing the datasheet with the VMDA options we were using in hardware. First, the MM2S_VDMACR register is essentially the overall control settings of the VDMA block in MM2S mode, which we were using. This register was set to 0x0003, in order to set the VMDA to run, and to run in circular mode while all other settings did not matter. The next register to set is the MM2S_REG_INDEX register which was set to 0x0. This is because we were only using 1 frame buffer, and the register value is 1 only if more than 16 frame buffers are being used. Next, the MM2S Frame Buffer Start Address was set to be the value of the array being used to hold color values, since the start address of the array is where the frames need to start. The frame delay and stride register was set to be equal to the H_SIZE register, based on configuration that there should be no frame delay and no excess stride. The H_SIZE was set to 0x0500, which is 640 * 2. This value was chosen as there are 640 horizontal pixels at a size of 2 bytes, so the total bytes would be 1280, or 0x500 in hex. V_SIZE was the easiest, as it is based on pixels, and is just 480 in hex, or 0x01E0.

- Conversion of color values from 24-bit down to 16-bit.

To convert the Cyclone Cardinal & Cyclone Gold color codes from 24-bit down to 16-bit, our group split the 24-bit the hex color code into bytes, converted that value into decimal, divided that decimal value by 255 (the max color value that pair could hold), and then multiplied that ratio value by 15 to find the new hex value that would be placed into the 16-bit color code. This process was done for each pair of two until the full 16-bit color code was created. For example, we took the color code 0xF1BE48 and would section it into pairs “F1” “BE” & “48”. From this point F1’s equivalent in decimal is 241. This 241 value would be divided by 255 and then multiplied by 15 which would produce a result of roughly 14. When we converted 14 back to hex, that gave us the value “E”. This “E” would then be placed into the first spot of the 16-bit color code (0xExxx). This process was repeated for each pair until the color code was complete.



7) General approach to implementing both NESCore_Callback functions.

- NESCore_Callback_OutputFrame()

We approached this method by deciding that it would be more appropriate to scale the image from 240x256 resolution to a image twice the size with resolution of 480x512. This was done by taking the current pixel from the WorkFrame and applying that pixel's color to the pixels to the direct right, direct bottom, and downward right diagonal. To better explain this scaling, please see the picture below.

Original Pixel

Maps to

Original Pixel	Scaled Horizontal
Scaled Vertical	Scaled Diagonal

All pixels in the above pictures hold the same color value.

- NESCore_Callback_InputPadState()

Our group approached this method by first figuring out that we needed to implement GPIO AXI IPs in our design to be able to have hardware access to the buttons, and switches. Once these were integrated, our team used to bitmasks given to us in NESCore_callback.h to figure out what switches and buttons were being pressed. If any of the given IOs had an active singal, we ORd the corresponding bits with the player's input state.