Vignesh Krishnan, Amith Kopparapu Venkata Boja, Archit Joshi

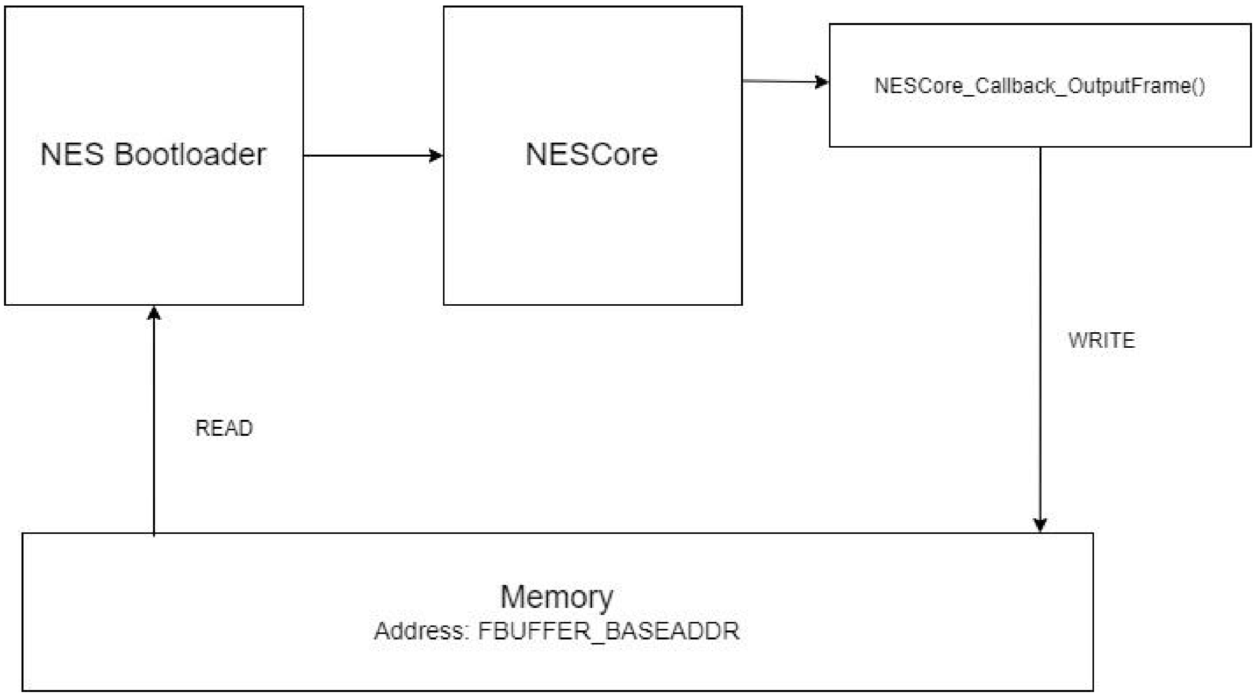
Cpre 488 MP0 Report

Section 2

February 3, 2020

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?

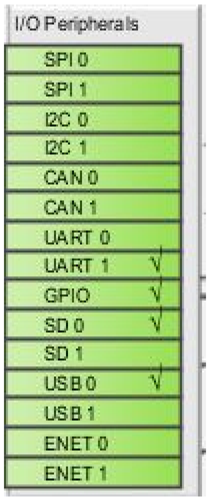
This code initializes the memory and cache on the board as well as the NES core that is used for the emulation. It loads the specific ROM of the game hardcoded in and begins cycling the NEScore to run the game. First, NESBootloader.c calls the NESCore_Cycle() function and this makes another call to NESCore_HSync() and that finally calls NESCore_Callback_OutputFrame() with the current workframe. This function gets called on every NESCore cycle.



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.

This is a high performance slave port that can be used for faster master/slave data interfacing with other components in the system from the processor. There are 4 different HP ports that can be enabled and each port can be configured for 32 or 64 bit data width.

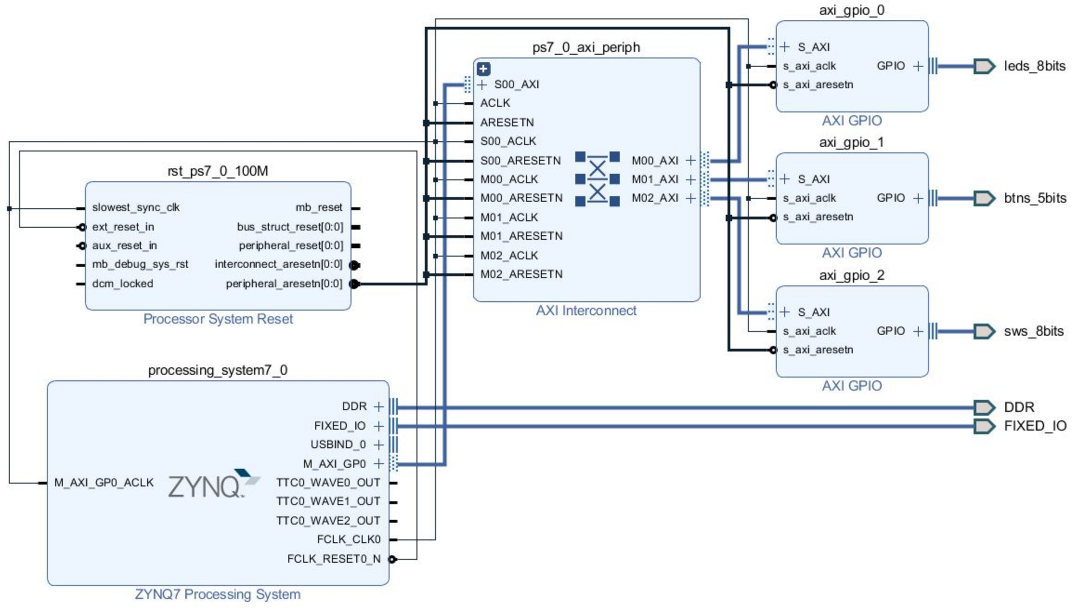
Clock generation can be used to modify the CPU, DDR, and PL Fabric clocks. It can also generate new clocks from the processor to be used for other IO as well as just more PL Fabric clocks. Configurations to the clock frequency can be made.



These are various IO peripherals that can be enabled such as UART, CAN, SPI, etc. These ports can be enabled, disabled, and configured individually.

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?

From the block diagram below, the cores that we created for the LEDs, buttons, and switches are in the PL subsystem because they are connected to the PS through the axi controllers.



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.

Using the XIL_In32 function, we can read the data in the axi_gpio_2 register to get the switch states. Using the macro for the axi_gpio_2 base register contained in xparameters.h, we can read the last 8 bits and find out the individual switch states. We can mask these bits to get the specific switches' states.

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?

The print function takes an 8 bit character pointer and writes each bit over UART. Printf uses a lot more memory due to the formatting aspect of it. Print uses much less.

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.

VDMA MM2S Circular Mode and Start Bits - We set this to 0x3 because we configured only one frame buffer in hardware and decided to leave circular mode on, meaning its continually cycling through single frame buffers. The start is also set to high to begin VDMA operation.

VDMA MM2S Reg_Index - Set this bit to 0 to select bank0 out of the two register banks since address is not greater than 32.

VDMA MM2S Frame Buffer Start Address - We set this to the memory location of our test_image because this is the address for which we want VDMA to start retrieving frame buffers for.

VDMA MM2S FRM_Delay and Stride - The stride indicates address bytes between first pixels of each video line and since each row has 640 pixels, 640*16 (bits in every pixel) / 8 = 1280 bytes. Frame delay is not needed because we're not using any genlock operations.

VDMA MM2S HSize - This field refers to the number of bytes per line. Since each row has 640 pixels, 640*16 (bits in every pixel) / 8 = 1280 bytes.

VDMA MM2S VSize - This field refers to number of vertical lines per frame which is simply 480 lines.

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

To convert 24 bit color value to 16 bit value, we trimmed the LSB from each component. Each 24 bit value contains 8 bits for each RGB color, so trimming the LSB results in a 12 bit color with 4 bits for each color and 4 bits of padding. This totals to 16 bits for the color. Based on our hardware implementation, we had to switch the location of R and B in the conversion to account for the way the pins are planned in the hardware configuration.

Ex. Cardinal Red - #C8103E
Trimming LSB - 0x0C13
Flipping R ad B - 0x31C

Teammate Contributions:

- Amith - 33.3%
- Archit - 33.3%
- Vignesh - 33.3%
- James and Joe - 0.1%