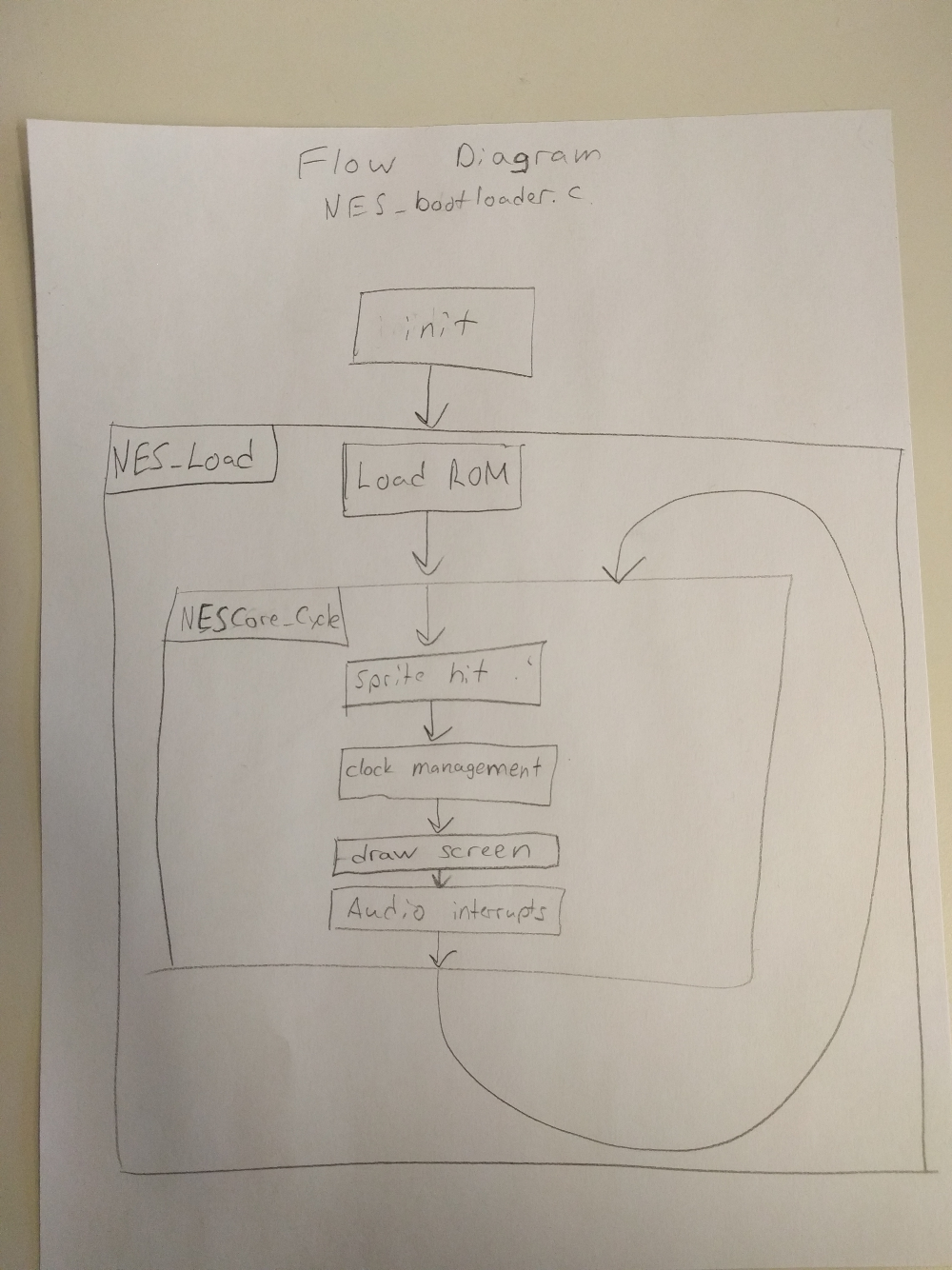
MP-0

# nes\_bootloader.c

The main function starts off with doing a few initialization steps. Making sure that everything is set up correctly. The interesting part comes in at nes\_load(). This function starts loading the correct ROM. In the default case, this game is “zelda.nes”. Next it sets the bootstate.nes\_playing = 1. Now it goes into a infinite loop of NESCore\_Cycles. This function first detects sprite hits in the game. Then it uses NESCore\_step to control frame rate and cpu timing so that the game runs and looks as it is expected to. It then aligns the H-sync mapping. Then it takes care of the audio using an IRQ.

NESCore\_Callback\_OutputFrame() gets called when we are drawing a row that isn’t the first row and isn’t a vertical sync. This is happening inside of the H-sync function.

# System Assembly View

The Green Boxes

1. 32b GP AXI Slave Ports

We can enable GP0 and GP1 interfaces. We can modify the address ranges of each. Controlling memory is always useful in an embedded system. It seems like we can set memory addresses ourselves or we can let the program decide them for us.

1. I/O Peripherals

We can enable or disable peripherals. This is useful because it lets us optimize our designs and be flexible to modifications.

1. General Settings

It has a lot of options that don’t fit anywhere else. We can change the baud rate of the UART as well as Enabling power on reset. These are very nice features to extend the ability of your design.

# PS and PL

We think the buttons, LEDs, and switches are connected via the PL subsystem. We wouldn’t write c code to change how the bus works so this doesn’t fall into the PS subsystem. Since it isn’t PS, it must be PL. Also, we know how the bus works so we make our design implement the bus protocol. We don’t implement the bus protocol in code.

The AXI\_GPIO IP core acts as an interface. The bus doesn’t care what is on the other side of the interface. There is a standard when talking to AXI\_GPIO IP cores. Since buttons, switches, and LEDs are all GPIO on a bus following the AXI protocol, they all get labeled as AXI\_GPIO.

# Read the Switches

In order to read the switches, we have to start off with the base address of the switches. In the Addresses tab we can see that “SWs\_8Bits” has a Base Name of C\_BASEADDR and a value of 0x4120FFFF with a size of 64k. Now taking that information over to the axi\_gpio datasheet, on page 8 it describes the registers of the IP. It describes 4 registers. 2 of which are data registers (one for each channel, if enabled) and 2 for tristate values. Reading or writing is done with the data registers. To read and write you use the following equation. C\_BASEADDR + 0x00. We know that the switches C\_BASEADDR is 0x4120FFFF. Therefore, 0x4120FFFF + 0x00 = 0x4120FFFF. Now simply read the data at that address.

# Print()

The print() function is a watered down version on printf(). Print() copies a string byte for byte while printf() does a lot more, such as format specifiers and data type conversions.

# LED’s Hello World

# CHANGE\_ME

In order to find the correct values for the VDMA register, we needed to look in the VDMA documentation. On page 31 there is the start to the Register Space section. This is where we will find the offsets and default values. More importantly, we will find which bits out of each register we need to set in order to get the desired output.

# Cyclone Checker

We found what fraction out of 255 each color channel was and then applied that percent to the 16 bits that we have to work with. Then to display a checker board we divided the screen into sections by using a range of indices. These ranges corresponded to different colors that make up the checker board design. Although we have written the code for the cyclone checker, it is untested.

# Conclusion

After spending the prescribed many hours for this lab just like every other group. We have been halted by what has been described to us as “Tool chain” issues. Our project has been looked over by every member of the teaching staff for this class. Unfortunately, the consensus is that our hardware implementation looks fine and that everything should work. With that being said we haven’t been able to get VGA output working. We have tried to debug the hardware system by checking the signals coming from the VTC, as well as checking the status register of the VDMA. Both of these performed correctly. The sync signals were at the correct frequency and the VDMA register shows that the VMDA is being started without any errors. This has led our team to the decision of calling it quits for this lab. We will get a good start on MP-1 and we will try to get ahead of the problems that are sure to arise. Maybe, just maybe we will not have the same tool chain issues.