Memo

To: Professor Pisano

From: Benjamin Havey

Team: Team 19 – Ultra Fast ADC

Date: 5/2/14

Subject:` Personal Progress Report

1.0 Introduction

2.0 Design

2.1 Overview

2.2 Board Design

2.3 Linux

2.4 Boot Loader

2.5 Drivers

1.0 Introduction

My portion of the project was defined as the firmware. This included firmware design, ADC and DDS controllers, and data collection at a rate that, at worst case, matched the ethernet throughput (1 Gbps). Our team got together and went over multiple potential chip choices to meet these constraints.

All high speed ADCs use LVDS pairs for their data lines. The only real devices with many high speed LVDS pairs are FPGAs. The issue is there isn't a lot of buffering space on FPGAs without attaching them to DDR RAM (which presented a difficult routing challenge for the hardware team). Gigabit ethernet is also difficult to implement on a FPGA.

Another choice that was considered was an ARM chip. This would allow for larger buffering and easy ethernet implementation, but did not have the necessary LVDS pairs. High speed GPIO on Linux is also not super easy.

Having worked with the Zedboard development board before I suggested we use the Zynq SoC for our system. This combined an ARM chip and an FPGA on the same die and had enough RAM to buffer data at high speeds.

Our customer also desired to be able to select a dynamic sample size. This was implemented by buffering different amounts of samples into block RAM (BRAM) on the FPGA then transferring it to the actual RAM.

2.0 Design

2.1 Overview

Once we chose the Zynq my portion of the project was slated to included the following aspects: Linux, bootloader, board design, DDS and ADC controls, data collection, FPGA/ARM interconnect, and high level drivers to interact with the different components. In the end I completed all these parts except for the DDS and ADC controls, and also heavily contributed to implementing faster communications.

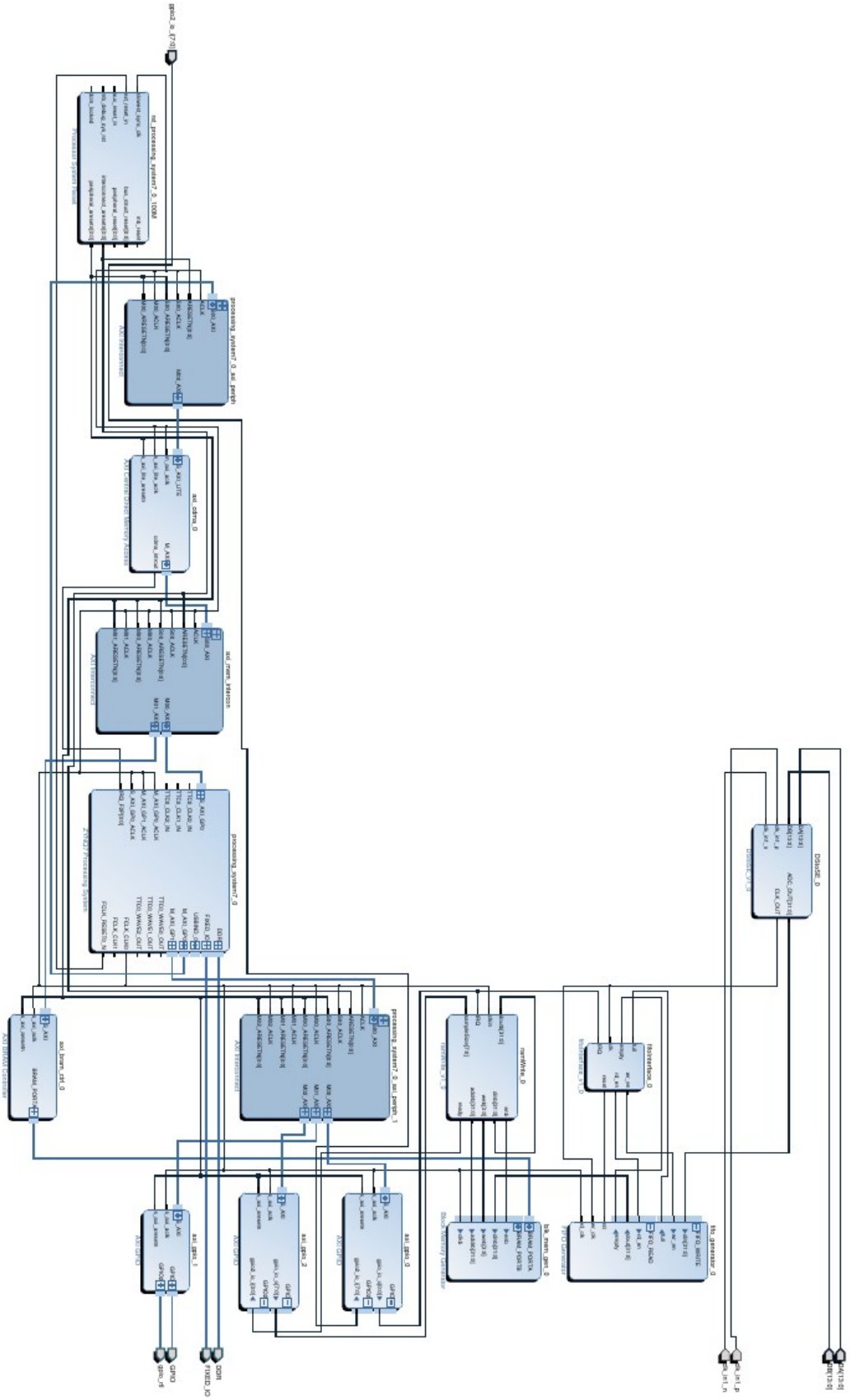
For Linux I needed to construct a root file system (rootfs) as well as build a kernel with the drivers necessary to access the special devices on the FPGA. A bootloader was necessary, called a u-boot, to use all the features on the SoC in Linux.

The board design required learning how to interact with Xilinx IPs and interface with the AMBA bus attached to the Zynq. The DDS and ADC were ultimately controlled outside of the firmware (through bit-banging the DDS and using the default hardware configurations for the ADC).

The data collection was managed through a Xilinx DMA IP called CDMA (which stands for central DMA). This was hooked into a general purpose slave port on the ARM core's AMBA bus. For higher speeds a specialty streaming AXI protocol could be used on an accelerated coherency port (ACP), but this required lots of interfacing and the general purpose slave port ended up allowing 3 Gbps throughput, well within our ethernet constraints.

2.2 Board Design

The figure (Fig 1) on the following page shows the final board design for the firmware:



There are a lot of aspects here so I will break up the design into three chunks: the lower left, the lower right and the upper right portions of the board design.

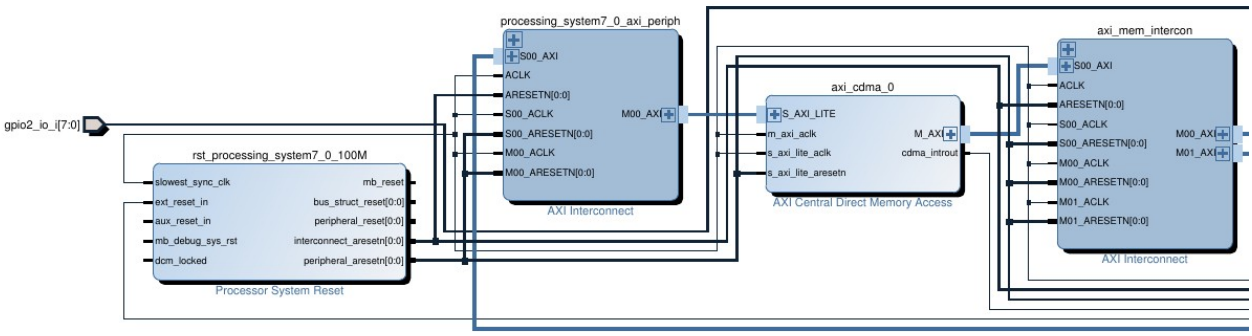


Fig 2: Lower Left Part of Board Design

gpio2_io_i: These are signals that control the ADC. Because we decided to control the ADC via hardware these are set as inputs so they will synthesize as high impedance buffers.

rst_processing_system7_0_100M: This was automatically generated with the creation of a Zynq board design. It synchronizes the resets between the different AXI modules and ports.

processing_system7_0_axi_periph: This is a Xilinx IP. This (and other AXI interconnects) allow for various AXI modules to be put on the AMBA bus. It controls the clocks and allows for bus scheduling if multiple modules share a single port. This particular one allows the user to control the CDMA module and set it's configurations.

axi_cdma_0: A Xilinx IP for doing DMA transfers. It can transfer up to 32K from a destination pointer to a source pointer. The source address is the address of the BRAM on the board design while the destination address is on a desired part of the microZed's external RAM. The axi_cdma_0 is interfaced with a device driver on the Linux side which writes to the address of the CDMA's S_AXI port.

axi_mem_intercon: Another AXI Interconnect module this one allows the CDMA to act as a master to the BRAM controllers and to the RAM on the microZed.

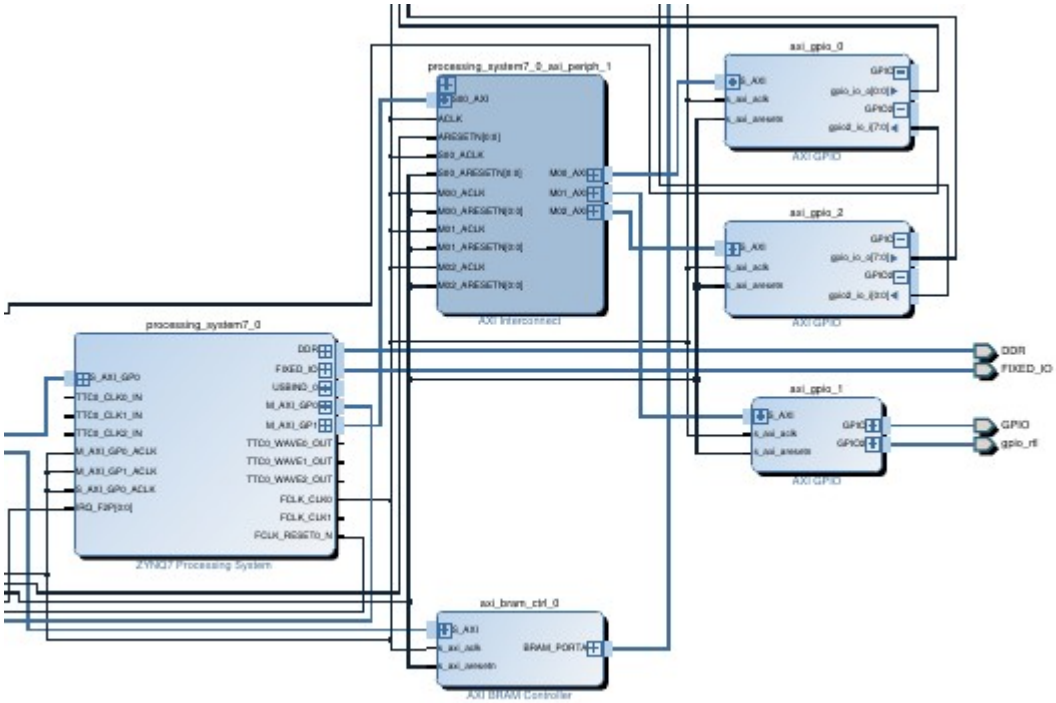


Fig 3: Bottom Right Part of Board Design

processing_system7_0: The most important module. This is the ZYNQ configuration IP provided by Xilinx. It can be configured with up to 2 master general purpose ports (MGP), 4 slave general purpose ports (SGP), 2 slave higher performance ports (SHP), and 1 slave accelerated coherency port (ACP). It also allows for configurable DDR timings (left as default), fixed_io ports (left as default) and up to 4 clocks set by a mmc (divided/multiplied from the ARM clock). Only one clock is used, FCLK_CLK0. It is set to 125MHz and drives the BRAM controller along with the read half of the asynchronous FIFO. M_AXI_GP0 is used exclusively to control the CDMA (due to the heavy timing requirements of the CDMA). M_AXI_GP1 handles all other controls. The IRQ_F2P input is an interrupt that gets set by the CDMA module every time a DMA transfer is complete to let the processor know the data is now usable.

axi_bram_ctrl_0: Another Xilinx IP. This controls the BRAM (shown in Fig 4). The BRAM is 2 channel but this controller can only access the read channel. The ADC data is wrote to the other channel by another module. This device is a slave to the CDMA module and the valid source pointers for the CDMA are within the address range of this module – any source address selected outside of the BRAM range will return a segmentation fault in Linux.

axi_gpio_0: A Xilinx IP, this two channel GPIO module allows the user to strobe a bit on channel 1 to initiate the next step (and in doing so reset the custom modules). Channel 2 is the high impedance input from the ADC and as such is unused from the user side.

axi_gpio_1: Another GPIO module. Channel 1 is hooked up to the DDS controls to allow for DDS configuration in user land on Linux and Channel 2 allows the user to read the DDS status lines.

axi_gpio_2: Channel 1 of this GPIO module can be set to change the current sample size for a step. Channel 2 is a “complete” signal that tells the top-level Linux driver that the current step's samples are collected. The user can then initiate a CDMA transfer to get the samples into RAM.

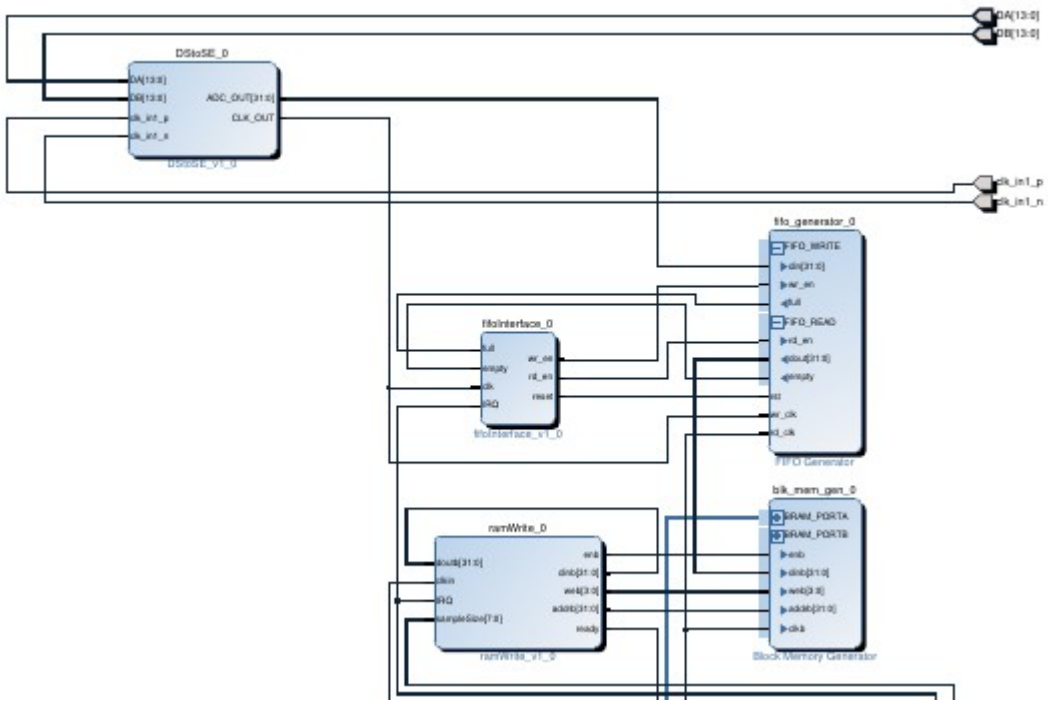


Fig 3: Top Right Part of Board Design

DA[13:0]/DB[13:0]: These inputs are where the data comes in from the ADC. At this point they are still double data rate (DDR) low voltage differential signals (LVDS). They are fed into the DStoSE_0 module where they are converted into single data rate single ended signals.

clk_in1_p/n: The positive and negative portions of the LVDS ADC clock. This is single data rate.

DStoSE_0: This custom module takes in the differential DA/DB and clk_in signals and converts them to single ended wires using LVDS intrinsics. The new single ended ADC data wires are then fed into a parallel edge FIFO which converts the double data rate signals into single data rate.

fifo_generator_0: Another Xilinx IP, this is an asynchronous FIFO which utilizes block RAM intrinsics. This is a necessary buffer between the BRAM and the ADC data stream because the BRAM can only handle data at a top clock rate of 150MHz. The write clock is driven by the ADC clock and the read clock is driven by the FCLK_CLK0 clock from the ARM core. FCLK_CLK0 was chosen to be 125MHz because at half speed the FIFO only needs to be the same size as the BRAM to not loose any samples. Decreasing the speed would require a larger FIFO and increasing the speed would not lower the FIFO size and would create tighter timing constraints.

f**ifoInterface_0:** This custom IP allows for writing to occur when the FIFO is not full and for reading to occur when the FIFO is not empty. It also does a safe reset on the FIFO when a new step is requested so new ADC data can fill the FIFO.

blk_mem_gen_0: This Xilinx IP allows for an interface to the BRAM intrinsics that matches the Xilinx BRAM controller – something necessary for the CDMA module. BRAM_PORTA is the read port and is attached to the CDMA module through the BRAM controller while BRAM_PORTB is the read port which is controlled by the custom ramWrite module. Block memory can be a maximum of 32KB due to physical FPGA constraints.

ramWrite_0: This custom IP controls the write port on the block memory. This takes in the sample size from the user then feeds in the ADC data from the FIFO at 125MHz. Every negative edge of a cycle it increments the address for the new data. When the amount of data written matches the sample size this module sets the write enable signal high, resets the write address to 0, and sets the “ready” signal high to let the CPU know that a step has been collected and is ready to be transferred via DMA.

2.3 Linux

The baseline Linux distro that was used was Zynq Linux, the one provided by Xilinx. This had to be modified in a few ways to meet our designs. First a Linux kernel had to be created. This was simply a matter of getting the Zynq kernel source from Xilinx and cross compiling it to an ARM executable. This had to be made into an image file and then a micro header had to be added to be read by our u-boot bootloader (more on this in Section 2.4).

At this point a root file system had to be made. For the sake of size constraints (future developers may wish to put a Zynq chip directly on a motherboard, in which case they probably will not need a separate SD card slot because of the small size), the root file system was chosen to be a compressed image. For added speed (so the programs wouldn't execute off a class 10 SD card) a ramfs was chosen as the rootfs. A rootfs was taken from the Xilinx website as a baseline. From here I made three major changes to the rootfs:

First the SD card was partitioned to a 1GB FAT32 OS partition and a 3GB FAT32 DATA partition. This had to be added to the /etc/fstab file so the OS would know where the data was. I also removed the OS partition as an entry from the fstab file so future users would not accidentally delete the kernel. The OS partition is mounted at boot to /mnt and contains all of the FPGA interface programs and drivers.

Secondly the ethernet had to change. By default the ethernet is hard coded to a single static IP. To make this more flexible a /etc/network/interfaces file was created which redirected the static IP to whatever was desired. /etc/network/ifup and ifdown files also needed to be added to properly use ifconfig.

The last modification to the rootfs was a system.d entry to start a script “/mnt/init.sh” after the DATA partition has been mounted. The rootfs is difficult to reconfigure and requires multiple steps to make non-volatile changes because it is a ramfs. This is the purpose of the DATA partition in the first place. By putting a start script in the non-volatile partition future users can make small changes to the rootfs using the start script without needing to go through the painful process of stripping headers, uncompressing, re-imaging, re-compressing, and adding headers to the rootfs.

2.4 Bootloader

The bootloader for the Zynq had special constraints that needed to be taken into consideration. The suggested bootloader for a Zynq is a u-boot. At startup a boot.bin file is read from the starting sectors of the SD card (the start of the OS partition must also be at the start of the SD card). A boot.bin is comprised of at least three items. The first stage boot loader (FSBL), the FPGA bit stream, and the uImage.

The FSBL is the initial bootloader. It must be extremely small and is only in charge of setting up basic networking (for network boots) and for setting up the devices that will be necessary to run the full boot process. The FSBL only needs to be programmed once as long as the ethernet , SD card, and USB pins remain in a fixed location. (This was the case for me as we were using the microZed) This is done using the Xilinx SDK.

In the case of the Zynq the FSBL also programs the FPGA with the bit stream. This bit stream contains all of the custom FPGA logic (which needs to be programmed at every power cycle as FPGAs are volatile). The bit stream is generated at the end of a successful synthesis and implementation in Vivado and needs to be re-generated every time a change is made to the board design.

Once this is complete the uImage is loaded. The uImage is a compressed version of the Linux kernel. I went over how this is made in section 2.3. This only has to be remade if you wish to update or change your Linux kernel.

With all three files created a boot.bif file was made to put them in the correct order then the Xilinx bootgen file was used to convert the boot.bif to a boot.bin. At this point I had a fully working bootloader, but this only allows for a kernel to boot. Once the kernel is booted it expects two files to be in the same partition as the bootloader: a devicetree and a uramdisk.

The devicetree is what tells Linux the addresses of everything that is attached to it's AMBA bus from the FPGA. This includes things that are hooked into the AXI ports as well as the fixed IO devices. The fixed IO devices include ethernet, USB, VGA, uart, SD, etc. The devicetree needs to be reconstructed every time there is a change to the connections on the ARM, generally when a new module is put on a new IO port. To update the devicetree the current board design must be implemented in Vivado, then the hardware should be exported to the SDK. In the SDK a device tree needs to be made using the board support package tool provided by Xilinx. Our particular device trees needed to have certain registers manually

configued so that the ethernet PHY will accept a dynamic IP addresses instead of the default static IP address. This was done by editing the ps7_ethernet entry in the devicetree.dts file created by the SDK.

A uramdisk is the Linux rootfs image that has been compressed and has a micro header added to it. The process for this was explained in Section 2.3. This is the last thing to be loaded by the kernel. The ramdisk is mounted onto the starting sector of ram and can then be treated as if it were a rootfs on a standard spinning disk or SD card. Because the rootfs is loaded onto RAM any changes made to it are volatile.

With all the above things constructed in the above order I was able to boot into linux from an SD card on the Zynq.

2.5 Drivers

To simplify the development process all drivers were built in user land space (otherwise a new uImage and boot.bin would have to be constructed for every small change made to the code). The drivers all ended up integrated with the server code. I constructed a few different functions which acted as drivers to the FPGA. There was a “reset” driver, a DMA driver, and a set sample size driver. Chris handled the DDS driver logic to which I added the necessary AXI connect code to communicate with the FPGA.

A normal sweep goes as follows: the user send configurations from the client. The server then takes these configurations and places them in variables associated with the different sweep parameters. First the desired sample size is sent to the ramWrite FPGA module. Then the DDS is written to to set the desired starting frequency. Once this is complete a reset signal is sent and the FIFO/RAM all reset, allowing for a step to be recorded. Once a full step has been collected (as defined by sample size) the ramWrite module sends a “ready” signal to the processor. At this point the ARM core will probe the ready channel and should start a CDMA transfer the size of the step. The program then probes the CDMA module's status register for a “complete” signal. When it sees this it increments the destination address register on the CDMA for the next step, then sets the DDS to the next frequency. A toggle up and down of the reset signal then starts the next step all over again. When a full sweep has complete, the server starts to send the sweep data which is now all buffered in RAM.

All ports were accessed by creating a void pointer to the memory map of their relevant addresses using mmap. This pointer could then be assigned to values (to write to the port) or referenced (to read from the port). To write/read for a specific register on a module the pointer was offset by the register offset, ex: *(port + OFFSET);

The RAM that was being written to was also protected using mmap. While mapping memory you must do so with increments of your minimum page size. For our RAM this was 4K so I mapped 8*4K ram, which allows for the “worst case” step size of 8K samples. A new portion of ram is mapped at every step. The size of the RAM on the microZed allows for full buffering of a sweep.