18-545 Advanced Digital Design

Digital Logic Analyzer

Fall 2014

Doci Mou, Julian Binder, Tom Mullins, Brent Strysko
December 8, 2014

TABLE OF CONTENTS

OVERVIEW

PROJECT DESCRIPTION/MOTIVATION

Our goal for this project was to create a logic analyzer capable of performing basic functions an electronics hobbyist would be interested in using. As hobbyists ourselves, we wanted to create a logic analyzer that would be both fast and customizable. We looked at consumer-grade products that were on the market, and aimed to improve upon them. Using a Xilinx Zynq FPGA board, we wanted to design a logic analyzer that would be both useful and affordable for our intended audience.

We determined that two of the most useful features of a logic analyzer were the ability to use many channels at a time and the ability to decode protocols. With that in mind, we decided that our logic analyzer would be capable of supporting up to 32 channels and a maximum sampling rate of 100MHz. Thirty-two channels allows for a much greater bandwidth than most products we looked at, and 100MHz is faster than all reasonably priced consumer products we could find. With these settings the user can choose when to begin sampling data, either manually or on certain trigger conditions. We also support the ability to decode features of certain protocols, specifically SPI, UART, I2C, and XMEM.

All of the data gathered by our device was to be displayed by a web interface onto an external screen, managed by a web interface running Linux off the ARM core on the Zynq board. This interface both displayed data and allowed the user to set configuration settings. The data displayed included the raw data sampled by the device as well as decoded protocols. In choosing configuration settings, the user sends data to the driver, which then sends it to the memory module, which forwards it to the relevant part of the system hardware.

TOOLS & PLATFORMS

We chose to break the project up into three levels of abstraction: low-level hardware, kernel-level driver, and higher-level software. Our hardware consisted of the Xilinx Zynq board, an expansion board for the GPIO pins, and an external Arduino microprocessors we used for testing. We used the Vivado Design Suite to simulate and synthesize our design onto the FPGA. Some of us also utilized Synopsys VCS to simulate when not in the lab or when someone else is holding the license to Vivado's IP cores, which were essential to our project. The Zynq FPGA also included an on-board ARM dual-core Cortex, which allowed us to boot into PetaLinux and to use our driver to connect to the web interface. The ARM chip interacted with the programmable logic through AXI ports, which we utilized primarily for memory.

COMPETITIVE ANALYSIS

As engineering students, we were very familiar with the basic functions of an oscilloscope and have all tinkered with a logic analyzer when making our own projects. When coming up with the specifications for our project, we wanted to make sure that our logic analyzer would be comparable to commercial

products marketed for hobbyists. We did not have the opportunity to personally test many products, but we did have the opportunity to become familiar with the Saleae Logic.

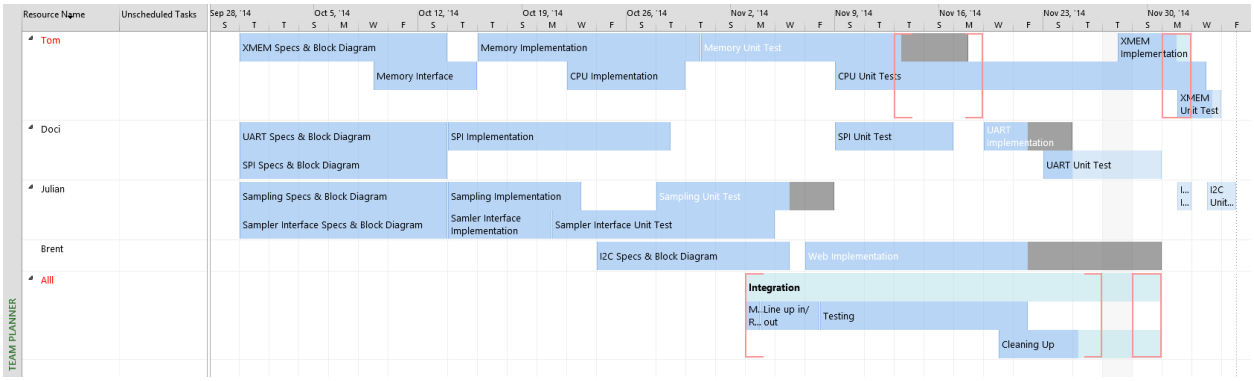
The Saleae Logic is a logic analyzer marketed for “embedded developers” and often used by hobbyists. It is marketed to run at 24MHz and has 8 channels, and can decode up to 17 different protocols (though 9 of these are currently in beta). It is a very popular device and was fairly affordable at about $150. The Saleae Logic offered free software separate from the hardware device and meant to be run on another computer, a feature common to low-end logic analyzers. Because of the feature set included in this product, we decided to implement a subset of the features and improving upon them. For example, we can run 32 channels where they can only run 8. We can sample at up to 100MHz whereas they are marketed to run at 24MHz. Our software features were very much inspired by those included in the Saleae Logic, especially the interactivity of the graph. We did not include as many protocols as the Saleae did, but instead focused on the technical specifications that would hopefully put our logic analyzer ahead.

Other products we looked at were high-end oscilloscope/logic analyzer combinations. They were all scopes that ran at high speeds and had very large bandwidth and also cost five or six figures. Some examples were the Agilent 16900 Series, the Tektronix MSO70000 series, and the Tektronix TLA7000 series. The purpose of exploring these options was to determine the tradeoffs the large companies made in terms of price, bandwidth, sample rate, and speed. The conclusions drawn from comparing these would help us determine which combination of those settings would be most applicable when we encountered the situation ourselves.

In the end, we decided on our final numbers as a result of considering the ease of implementation, the ability to integrate well with our layers of abstraction, our audience, and most importantly, the scope of our project. As impressive a feat as it would have been to optimize our design for as ridiculously fast a speed as possible, we only had a semester to do it.

SCHEDULE

An approximate schedule for our group was as follows:



As is obvious from the graph, only small amounts of work were done early on, and they were done at a very slow pace. The work began to pick up about halfway through the semester, but for the rest of the semester there was a lot of work to be done by everyone. We did not start integrating our modules until about November, at which point there was a significant amount more to do. More and more work was crammed into the last few weeks of the semester, and although some of us were on track according to our own schedules we were rarely ever in the green as a group.

We encountered the most problems toward the last three weeks, time we had originally planned to be a buffer for unexpected bugs and integration errors. Because we were so behind on work, we never got the chance to use this buffer time and in the end went through the public demonstrations with bugs in our code. That said, we did achieve basic functionality and many of our features were buggy but still implemented.

TECHNICAL SPECIFICATIONS – DESIGN

SAMPLER MODULE

OVERVIEW

The sampler module controls the sampling on all thirty two channels, triggering of all channels, and compression of the samples into a bandwidth friendly format. The sampler module contains all of the registers for controlling sampling and triggering. It is split into three functional units. The overall sampling module contains mostly status and control registers. Inside of the sampling module is the triggering module which controls triggering. Finally there are 32 separate sampling channels which can be configured to sample and send their data to any of the protocol engines.

INTERFACE

The sampler module primarily interfaces with the interface module. Each of the sampling channels has its own input port. The sampling module as a whole also has an input port from the configuration bus. The sampling module interfaces with the router using a 32 bit packet where the most significant 31 bits specify the time that the sample occurred at and the least significant bit is whether the edge is rising or falling.

SAMPLER MODULE

The sampler module includes several status registers. The sampler module has a register which of the sampling channels are enabled. It also includes registers for storing the maximum sample age, the maximum number of samples, and for keeping track of which channels are running. The sampler module also has a register which can be used to trigger a set of channels from the external software interface. In addition to these registers the sampler module also interfaces with the configuration bus in order to allow the CPU module to write to the sampling registers.

The sampling module features a configurable sampling rate. This is achieved through a configurable sample clock. The clock is set by registers which can be controlled through the configuration bus. A separate FSM controls the Xilinx MMCM which is able to set a clock divider and multiplier. While the clock is being set and not yet stable the sampler is held in reset via a synchronized reset module.

SAMPLING CHANNELS

There are 32 sampling channels integrated into the sampling module. Each of the channels are identical. Each sampling channel has two status registers. The first keeps track of the sample count. It starts counting on startup and is reset when each sampling run is complete. The second register keeps track of the sample count when the trigger is engaged. This allows the sampling module to calculate the age of each sample and determine when to stop sampling.

In order to capture the IO signal the sampler uses a two register synchronizer. Once the data progresses through the synchronizer it is compared against the previous sample's result. If the two samples are different then an edge was detected and the module calculates whether it is a rising or falling edge. These

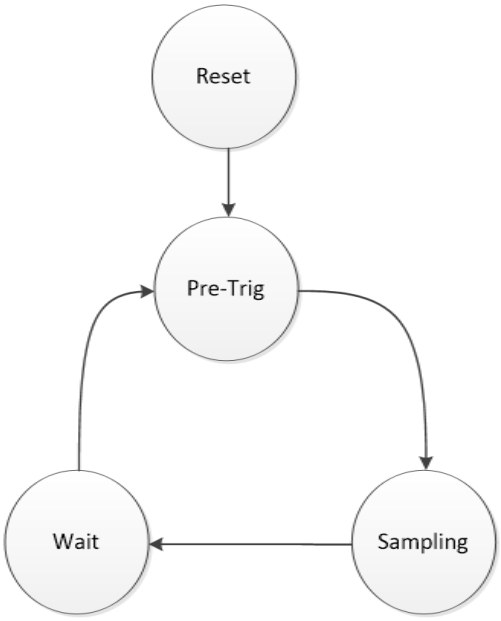
edges are then paired with the sampling time to create a time-edge packet. The first 31 bits are the sample number and the last bit represents the edge type. Zero is used for a falling edge and one for a rising edge.

Each of the sampling channels also features pretriggering. Pretriggering is implemented using two FIFO queues. The first FIFO is used when the channel has not been triggered. It stores the first n samples where n is the depth of the fifo. Based upon the maximum sample age which is a sampler module level setting it will discard old samples. It will also discard samples as the FIFO becomes full.

When a trigger event occurs all triggered channels output the oldest value that they have. This is the initial value. Then the pretriggering buffers are cleared. These values are stored in time edge format and the signal is reconstructed based on the initial value. Finally, the sampled values are outputted as they enter the sampling channel. Two FIFO queues are used as buffers to allow for only one sample to be outputted each clock edge. An additional requirement is that the sampling channels all output their data in order with respect to the remaining sampling channels. In order to achieve this a counter is used so that the sample that is outputted is no newer than the samples being outputted on any of the other channels. Once sampling is over, the sampler waits until its buffers have been cleared before switching to a reset state.

Each sampling channel interfaces with the interface module through a node. The node handles all of the FIFO interface signals. The interface uses a valid ready handshake. In this method the sampling channel asserts valid whenever there is a valid signal to be output. The interface module reads these signals and asserts ready in order to read the data. Such a fashion ensures that no data is lost.

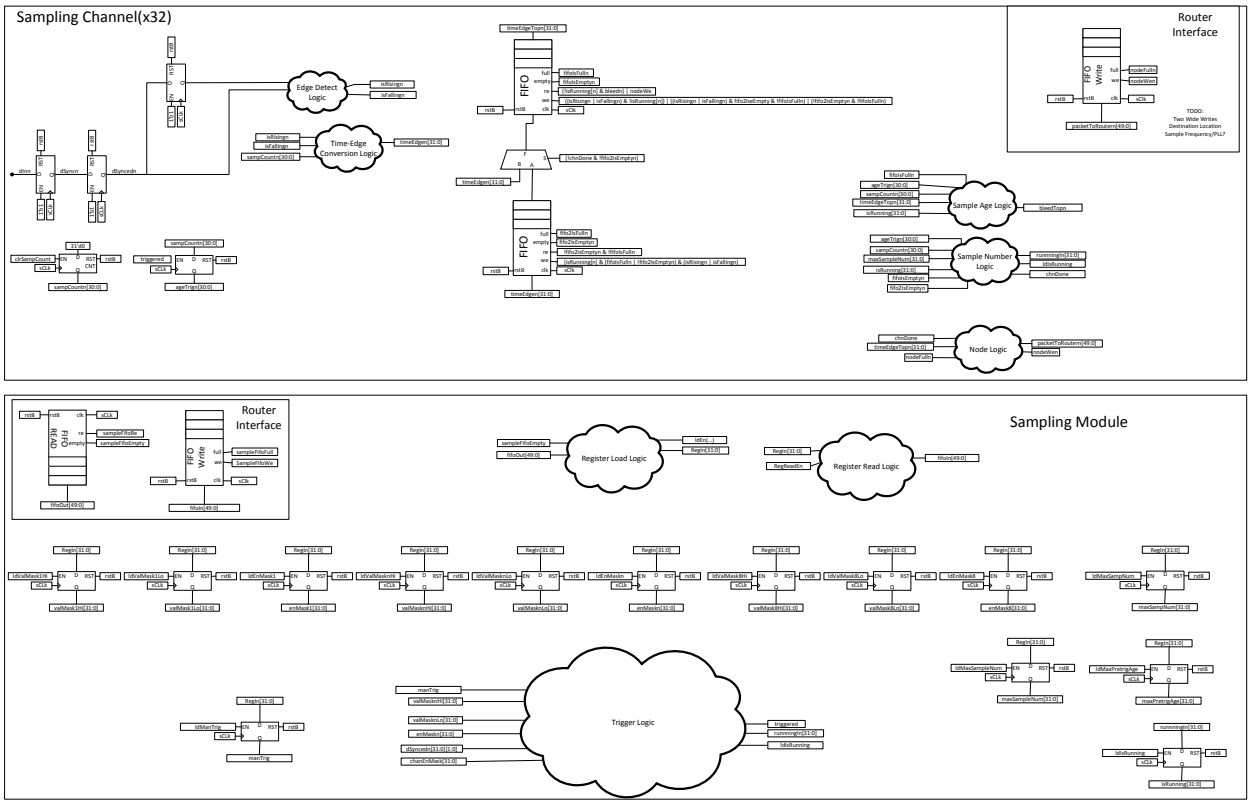
Here is a diagram of the sampling channel states:



TRIGGER MODULE

The trigger module is part of the overall sampler module. The trigger module consists of eight sets of three thirty two bit registers. Each set is called a trigger group. Each trigger group contains one enable register which masks the thirty two channels. The remaining two registers combine to allow the user to set a value for each of the thirty two channels. A group triggers when the enabled channel value registers match the sampled value. A trigger is generated on the same cycle that any of the trigger groups find a match. This trigger setup is useful because it allows users to configure the trigger for their custom application. For example this trigger setup can be used to monitor an address bus and trigger when the address is a certain value.

BLOCK DIAGRAM



SAMPLER INTERFACE

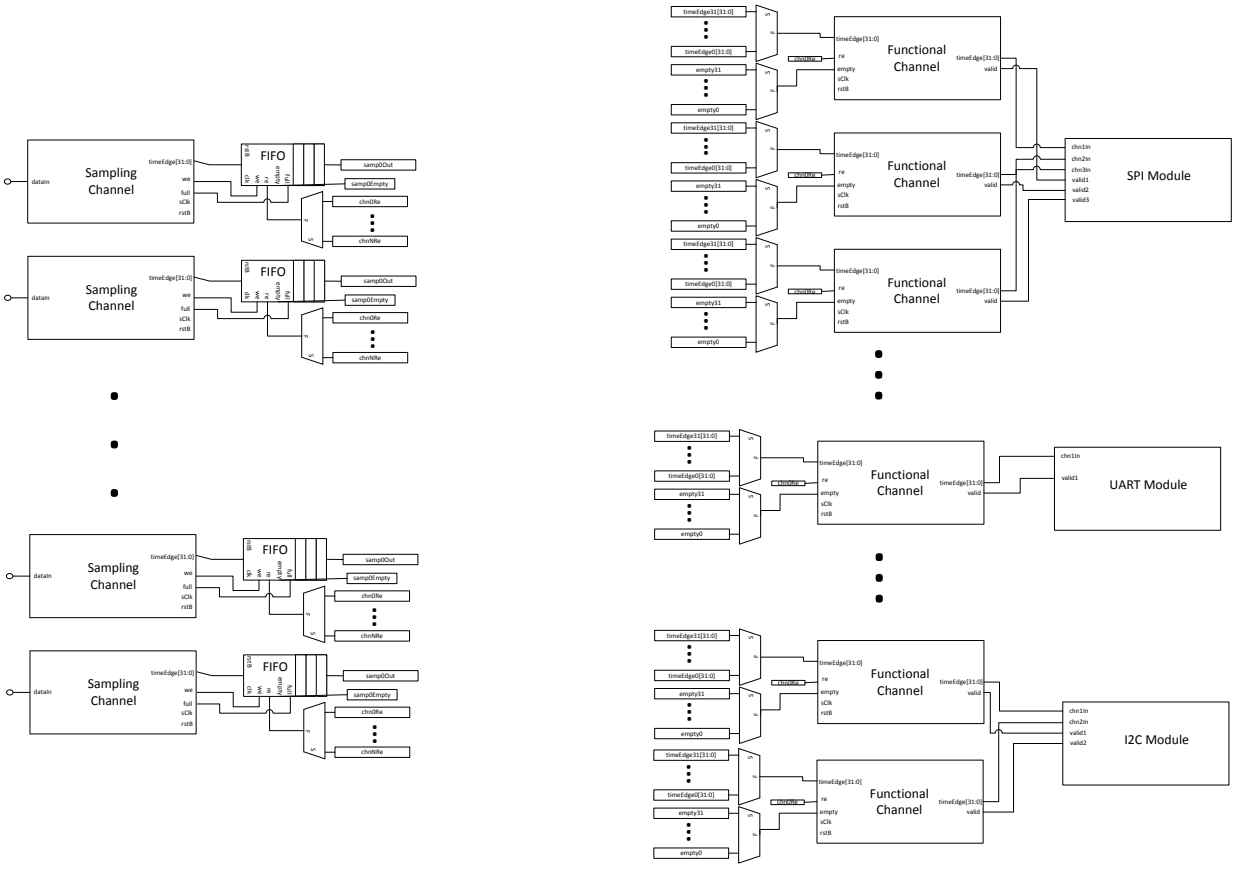
OVERVIEW

The interface module routes packets between the sampling channels, the functional modules, and memory. The interface module is configurable and allows for any sampling channel to be routed to any functional module. The interface module interfaces to the sampling channel, the functional modules, and memory module via the valid/ready handshake.

SAMPLE ROUTING

Samples enter the interface module through an asynchronous FIFO queue. This FIFO forms the clock barrier between the sampling module, which can have a clock frequency between 3 and 100 MHZ and the interface module which has a set clock. From this point the samples travel through multiplexers to another set of FIFOs. There are thirty two multiplexers that route the input signals to a total of 47 different functional channel inputs. There are 47 multiplexers that route the valid/ready signals back to the input FIFOS. The memory module latches onto the output of the input FIFOs. If there is a functional channel reading from the FIFO it will catch the data as it is output. If not, then the memory channel itself will pull the data out of the FIFO.

BLOCK DIAGRAM



SPI

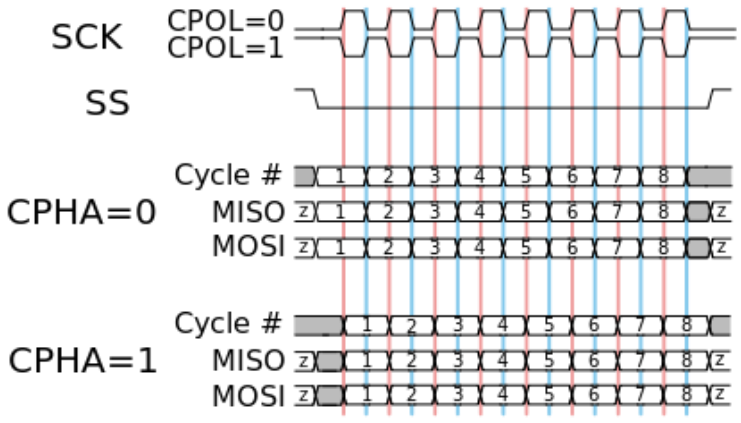
The serial peripheral interface (SPI) protocol is one of the four protocols we chose to implement a decoder module for. It is a common protocol that is a synchronous serial data link by Motorola. It usually operates in full duplex mode and communicates in a master/slave medium. It is sometimes called four-wire serial bus or synchronous serial interface (SSI).

This protocol requires either three or four lines of data which are provided to it by the sampler interface. A configuration bus provides additional inputs. This is summarized in the following table:

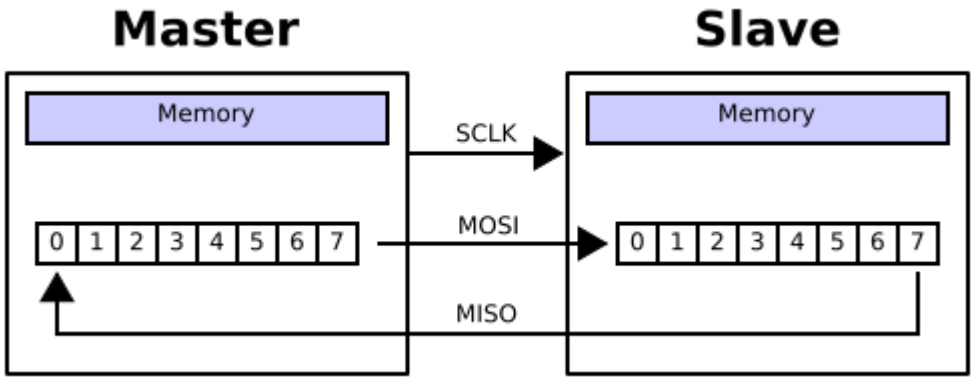
SPI Data Inputs	SPI Configuration Inputs
SCLK (clock)	spiMode (type)
MOSI (master-out, slave-in)	cpol (clock polarity)
MISO (master-in, slave-out)	cpha (clock phase)
SS (slave select)	wordSize (size in bits)

For our module, configuration must be sent before samples can be taken, as the data collected would be meaningless otherwise. The register spiMode refers to whether the sample will be traditional four-wire, half-duplex three-wire, or no-slave three-wire. In four-wire mode, sclk is sent with the data lines, mosi and miso, and slave select is an active low line indicating which slave is to receive the data from the master line. In half-duplex three-wire, the mosi and miso lines are combined. In no-slave three-wire mode, the signal on the slave select line is ignored and the device is assumed to always be the active slave. The cpol refers to the polarity of the clock, whether the first edge will be rising or falling. The cpha is the clock phase, which determines whether data is taken on the first edge of the clock signal or the second edge per clock tick. Since our system uses time-edge packets rather than signals themselves, the current status of the signals must be kept track of internally.

The following image provided by WikiMedia illustrates cpol and cpha. The red line represents (!cpha) and the blue line represents (cpha):



Data communication occurs when the master configures the clock, then transmits a low signal along the slave select line to the intended slave device, and a bit is shifted out on the MOSI line one clock cycle at a time until the entire block of data has been read out. As this is occurring, the slave device shifts out a bit on the MISO line, creating a ring of data that has effectively caused the master and slave devices data bits to change places. The bits transmitted are usually 8-bit words, but other word sizes are allowed and common. Our device allows for up to 32-bits of transfer at a time. The following diagram provided by WikiMedia illustrates this for an 8-bit transfer:



When data transmission has ended, the sclk line is returned to its original state.

At that point, we take the mosi/miso data we collected in shift registers and output them along with the times during which they were received. Two packets are sent out, one after the other according to the module clock, and the first packet appends a 1 to signify that it is not an error packet to the 31-bit time during which the packet data was first received, which is appended to the 32-bits of data, and then sends it to memory. The second packet appends a 1 to the 32-bit time during which the last data packet was received which is turn is appended to the 32-bits of data as well, and then sends that to memory.

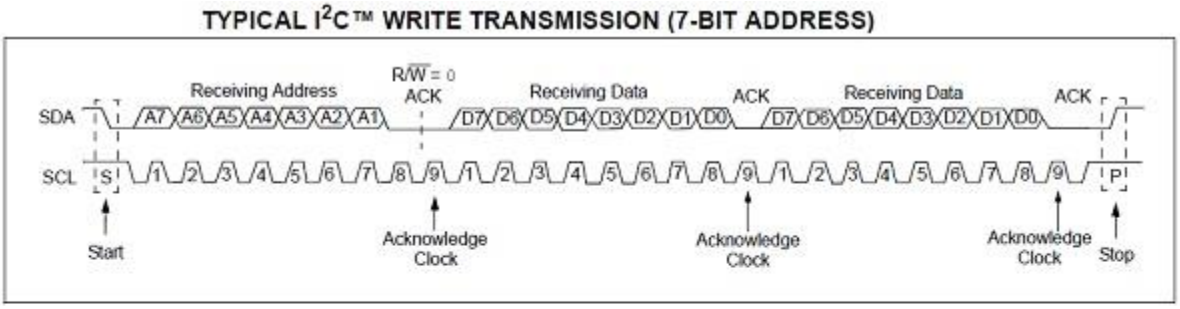
The SPI module in our design is completely implemented for all three modes, all four clock modes, and least significant bit shifting. Making modifications to the code to make it shift in most significant bit shifting as well will not be difficult, but because we chose one and stuck to it, it will likely not be supported by the driver, memory, or software interface and that will require additional modifications. The biggest error we ran into was that it worked completely in simulation and synthesized on the board, but failed to run correctly onward. Data was being transmitted in the first iteration of sampling, but either got stuck somewhere in the pipeline or simply wasn't going through to memory when on the board. This was not solved in time for the end of the class, but the code for SPI is still included in the files.

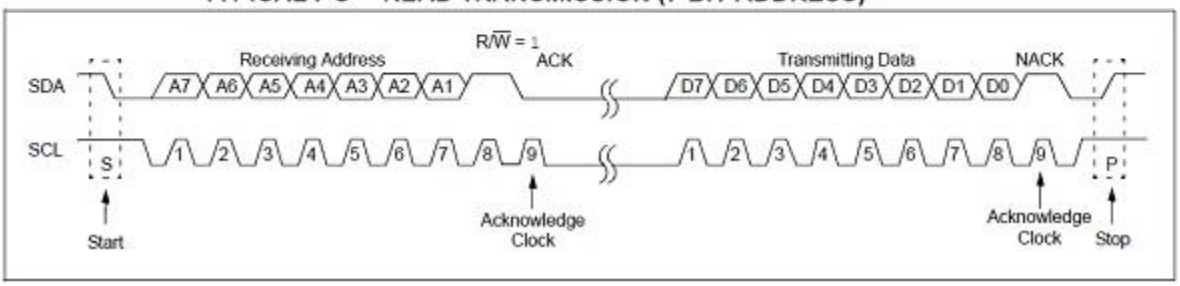
I2C

The Digital Logic Analyzer also features four I2C functional modules. I2C, also known as TWI, is a two wire serial protocol developed by Philips Semiconductor that is commonly used for chip to chip communication. The protocol is often used by hobbyists to communicate with off chip resources such as EEPROM, RAM, ADCs/DACs, and a large variety of sensors.

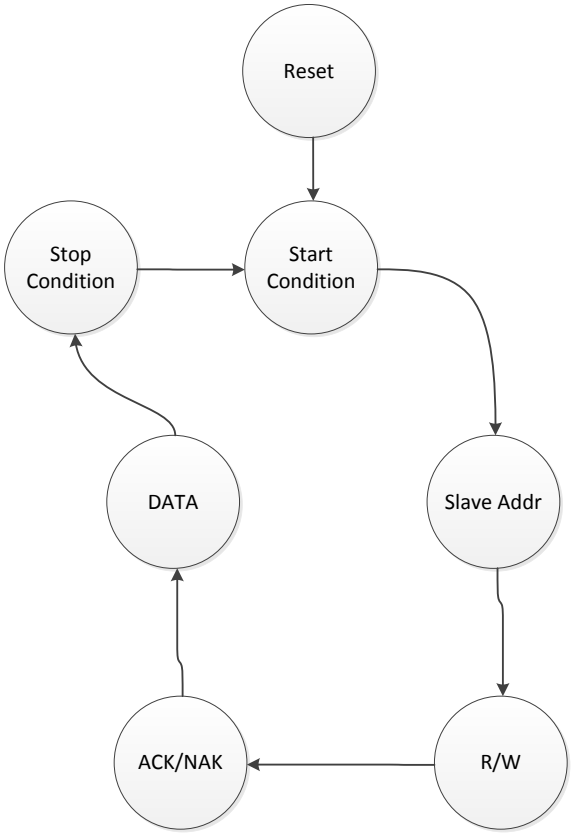
The I2C protocol features a master with multiple slaves. Each of the salves on the bus is given a unique salve address. In order to communicate the master first issues a start bit. After the start bit the master transmits the slave address over the bus followed by a read/write. If the slave recognizes its own address, it will assert an ACK signal on the bus. Otherwise the lack of signal is called a NAK. If a read bit was asserted then the slave will transmit over the bus. Otherwise the master will write. After the transaction is completed the master will assert a stop bit and the communication is over.

Here is an example bus waveform courtesy of Microchip.





The DLA uses a finite state machine to keep track of the state of the bus. The state machine is capable of determining what part of the transaction is occurring. It is also possible for the FSM to determine the data that was sent as well as any errors that occur during transmission.



The I2C module must also keep track of the two sampling inputs. It must be able to determine which input is older and then update its own internal model of the bus. From this internal model edges are detected and used to drive the finite state machine.

In order to manage the large amount of output two FIFOs are used. The data is split based upon type between these FIFOs in a fashion that ensures that if two data packets are generated in the same clock cycle they will go to different FIFOS. After the data is written to the FIFOs it is read out as soon as the memory channel is ready using the standard valid/ready interface.

UART

The Universal Asynchronous Receiver/Transmitter (UART) protocol is a simple protocol that shifts out individual bits of data to another device, least significant bit first. The protocol requires a single start bit, five to eight data bits, an optional parity bit, and one or two stop bits. Since the idle state of the line is low, the start bit must be logic low and the stop bits must be logic high. An example of the format is as follows:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
START	5 – 8 DATA BITS								PARITY	STOP BIT(S)	

The way through which UART transmitters and receivers obtain information require that both devices be set to the same settings. In our module, this requires that configuration data be set before sampling begins. The configuration data needed for UART is summarized in the following table:

UART Configuration Inputs
stopBits (number of stop bits)
bitRate (bit rate)
bits (size of data)
parityBits (parity)

While UART is a very simple protocol, it was one we could not get working in the end. It output the correct data for certain inputs in simulation, but because of existing errors was put on hold so that focus could shift on the other modules. The biggest issue was that the module was designed under the assumption that the first data bit would be logic high, for whatever reason. This allowed for the module to internally calculate the time difference between the 0^{th} and 1^{st} bits, therefore also allowing it to know how many bits had been transmitted between edges. The implementation also did not take the parity bit into account, ignoring it entirely when settings dictated that it existed.

The reason this method of calculating time and bits did not work is the fact that two consecutive edges are not guaranteed prior to the start of receiving data. A proposed solution to this was to read in sampling speed and mathematically compute the time between packets, as baud rate is a given with configuration. However, given that at this point the final demo was a few days away, we chose to focus on other portions of the system so that we would at least have a full end-to-end chain of functionality for some protocol. As a result, the new system was half-implemented and not even working in simulation, so it was scrapped entirely at the end and did not make it into our code base.

XMEM

The XMEM analysis module was designed to trace external memory transactions on an AVR microcontroller. The protocol which AVRs use for external memory is a pretty simple parallel protocol designed to support a 16-bit address space with limited I/O pins. There are 8 pins specifying the high byte of the address, and 8 bidirectional pins which are multiplexed between the low byte of the address and the 8-bit data. The AVR requires an external 8-bit latch, which latches the low byte of the address when address-latch-enable (ALE) is asserted. Then either read-enable (RDN) or write-enable (WRN) is asserted low, and the external memory or the AVR drive the data pins accordingly.

The analysis module simply keeps track of the current state of all 16 address bits and all 8 data bits, and then generates information packets whenever a falling edge is seen on RDN or WRN. It does this by watching all input channels and pulling in the valid one with the earliest timestamp. Because of the ordering guaranteed by the sampler interconnect, this guarantees it won't later receive a sample from an earlier time. Between a rising and falling edge of ALE, it will continuously update current known state of the low byte of the address with the current known state of the data lines. If it sees both WRN and RDN asserted, or if it sees ALE asserted during a WRN or RDN, or if it sees data lines change while WRN is asserted, it will generate an appropriate error packet, since these are invalid in the protocol.

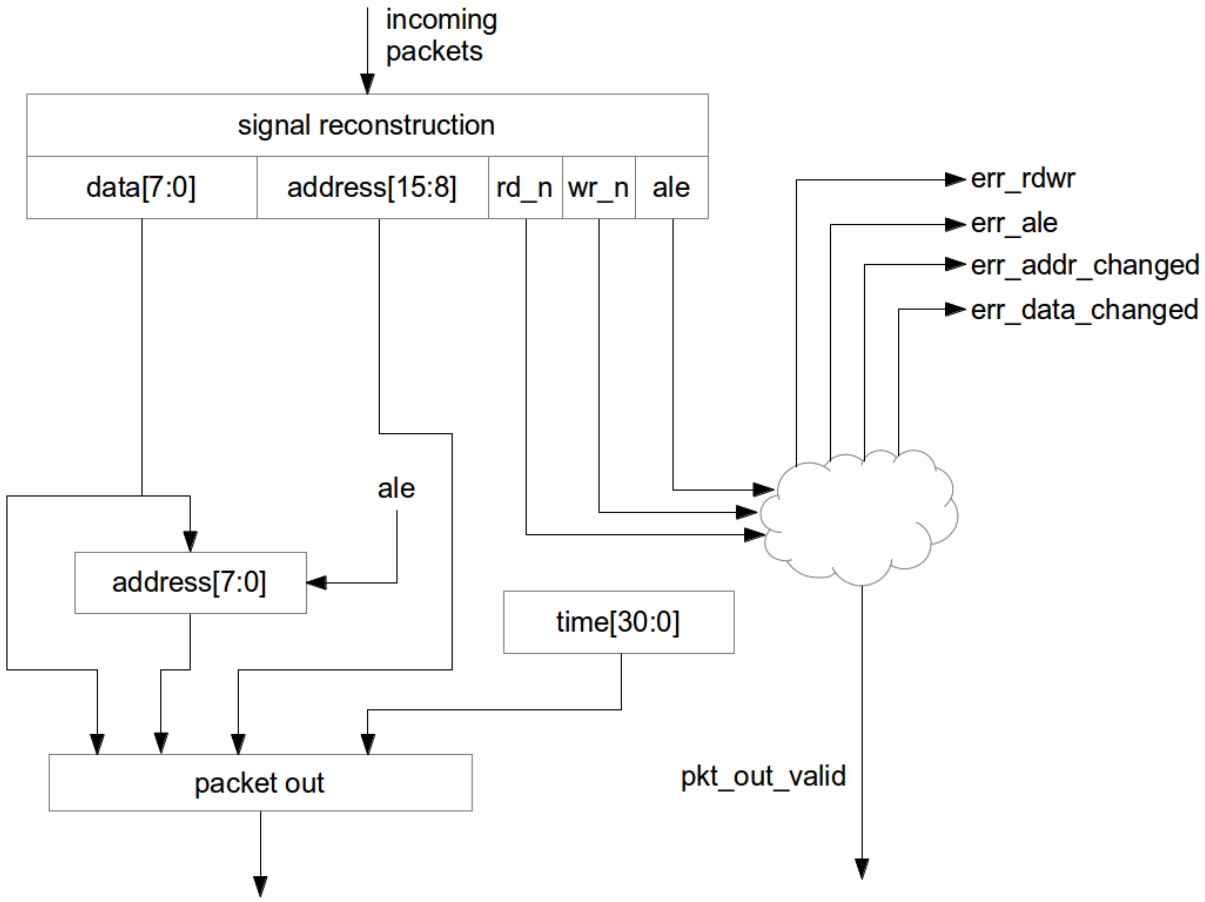


Figure 4: XMEM analysis module

MEMORY

The sample data and analysis data is sent to memory through an AXI port accessible to the FPGA. This port, the accelerator coherency port (ACP), is connected to the cache coherency unit of the CPUs, so that all DRAM accesses done by the FPGA are coherent.

There are fourteen memory channels, one for the sample data and one for each analysis module. Each channel has an input of 64-bit values, with a valid/ready interface. The memory system does not need to know about the format of this data; it simply writes 64-bit double-words into the memory buffers. Each memory channel has an internal buffer so that it can collect double-words into larger bursts of data, up to 128 bytes of data. This is because AXI3 can have address-incrementing burst transfers of up to 16 beats, with 8 bytes per beat. Collecting data into bursts allows greater bandwidth due to the overhead of each transfer.

The memory channels output a custom BurstIf interface, which is a pretty simple interface which transfers bursts of up to 16 64-bit data elements accompanied by a base address and a response signal for when the write is finished. It was designed so that the AxiMaster module, which accepts a BurstIf and issues AXI writes, would have minimal buffering or extra logic. The memory channels themselves keep track of the head and tail pointers for the circular buffers in memory and send the base address for each transfer, so that the AxiMaster modules don't need to worry about buffer addresses or offsets.

One problem encountered was that the AXI crossbar IP statically assigns each of its slave inputs to an ID on its master output. Because the ACP AXI interface has only 3-bit IDs, this meant there could be at most 8 AXI masters connected to the interconnect. The solution was the BurstSwitch module, which merges two BurstIf buses into one by selecting whichever has available data. BurstIf allows only one outstanding transaction, so this could cause bandwidth problems, but it is actually not much of a performance penalty. The ACP AXI interface only accepts three outstanding writes at once so that is the limiting factor in the system. That said, the sampler channel was deliberately not put on a BurstSwitch, since it has the largest bandwidth requirements.

The 32 sampler channels are sent into the SampleToMemInterconnect module. This has internal buffers for each channel at the input. It then outputs 64-bit data to a memory module just like the analysis modules with a valid/ready interface. Each 32-bit time/edge input on a channel is appended to the 5-bit channel number, and the remaining 27 bits are zero. In order to ensure fairness amongst the channels, the sample with the earliest time is selected each cycle, which has the added benefit that samples are written to memory in order of sample time.

Testing of the memory interface was largely done in simulation using the AXI bus functional model (BFM) available in the processing_system7 IP. There was actually a simulation bug with the ACP interface where it would drop every fourth write, which was very inconvenient. The ACP interface in hardware accepts only 3 outstanding write transactions at once, so the BFM was configured to allow only 3. However, the BFM did not work with non-powers-of-two outstanding transaction limits, so every fourth write would go out of bounds of an array and provide X's. Even though the simulation files for the IP were supposed to

be read-only, I was able to change the parameters to allow 4 outstanding writes and that fixed the problem. As of now, the bug has not been reported to Xilinx.

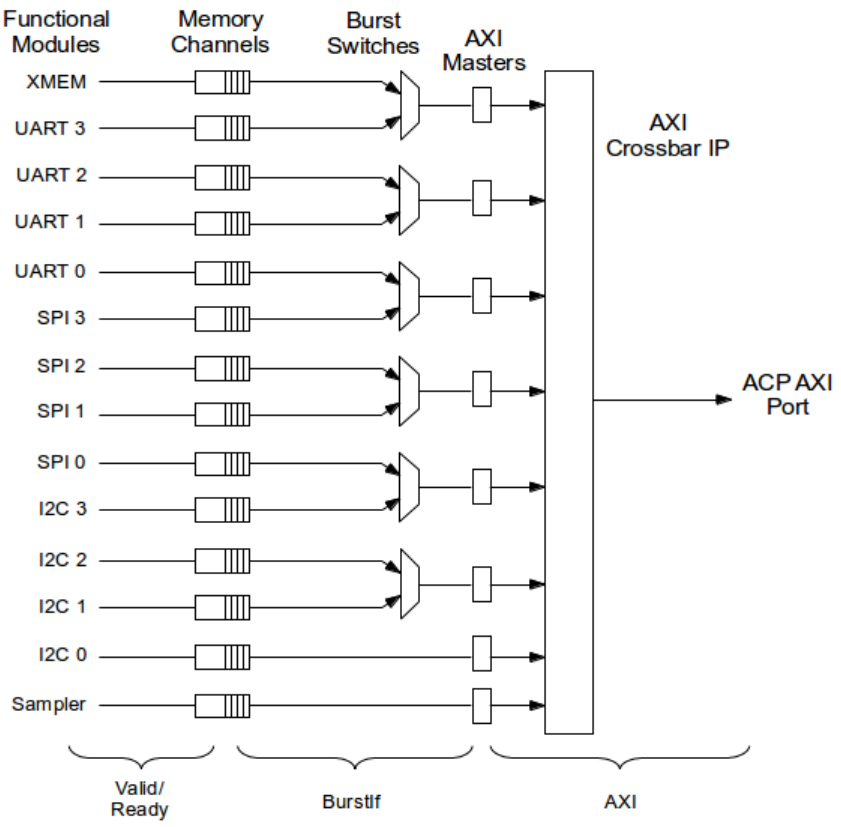


Figure 1: Memory Interface

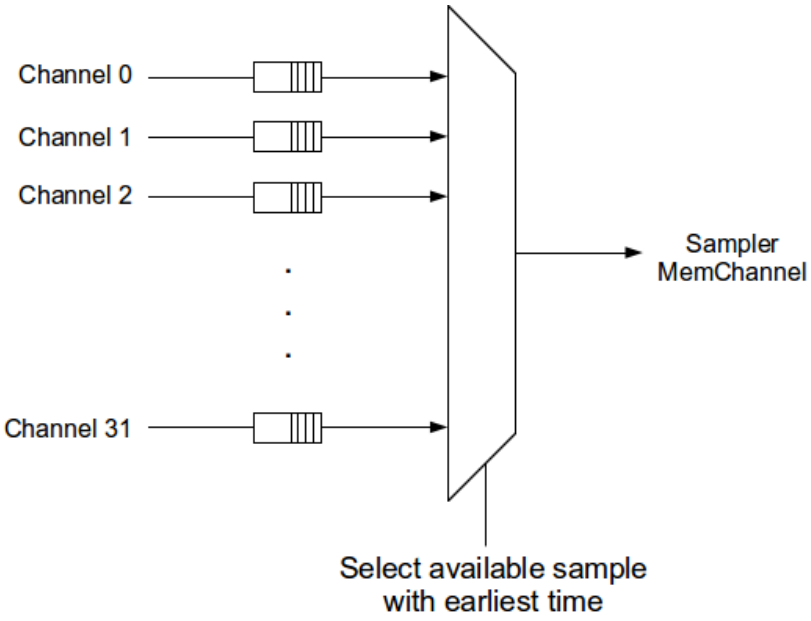


Figure 2: Sampler to Memory Interconnect

CPU INTERFACE

Configuration parameters are set by the CPU through memory-mapped registers. Accesses to the memory region 0x40000000-0x7FFFFFFF are sent to the FPGA through AXI port GP0. All write transactions are then broadcast on the global configuration bus, which consists of an 8-bit register number derived from the accessed address and a 32-bit data payload. All modules in the system watch the data bus for writes to their internal configuration registers. The bus does not support reads so most registers are write-only, and those which are readable are handled directly by the AXI slave. This includes the tail pointer for each memory channel, and one interrupt status register. If any bits in the status register are set, the first of the 16 shared interrupts IRQ_F2P[0] is asserted. There was a bug in the IP generation tools in Vivado which caused the processing_system7 Verilog wrapper to give the IRQ_F2P input a width of 1 instead of 16, so I only felt safe using the first interrupt, but only one was really needed. The interrupt handler can read the interrupt status register for more information. To acknowledge receipt of the particular interrupts, the driver writes back a 1 to each flag it wants to clear. This eliminates races where a flag gets set while the driver is already in the interrupt handler, so when it clears all of the flags it clobbers that flag without ever seeing it.

Testing of the CPU interface was done in simulation using the AXI bus functional model (BFM) available in the processing_system7 IP. It had an internal task write_data() which would send AXI writes on the GP0 interface, just like driver writes in the real system. Testing on hardware was initially done by setting up a small FSM that would write a bunch of consecutive values into memory, and configuring it using the actual CPU interface.

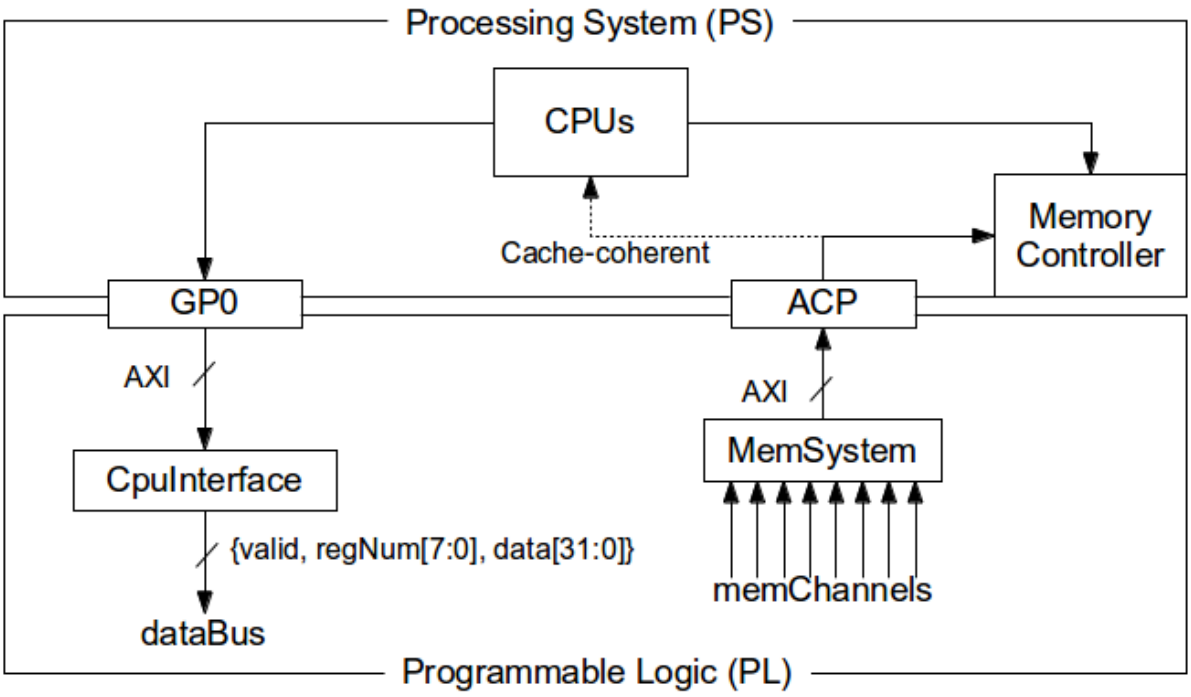


Figure 3: System-level Diagram of CPU and Memory Interfaces

DRIVER

The Linux driver is responsible for two different tasks. First, it exposes all of the configuration options for different parts of the design to userspace via sysfs, and writes that configuration to the design. Second, it provides sample and analysis data to userspace through special character devices by providing target memory buffers to the design and reading data from them.

Configuration is handled through sysfs, a Linux kernel mechanism for exposing internal kernel data structures. Linux has a tree structure of kobjects, which automatically generate a directory structure inside of /sys, and each kobject can have a number of attributes, each of which has a file in the directory of its kobject. For example, the dla-sample character device created by the driver has an embedded kobject, and I added the “clock_cfg” attribute, so this made a file called /sys/devices/virtual/dla-sample/dla-sample/clock_cfg. Each attribute has a store() function, which is called when the file is written to, and a show() function, which is called when the file is read. Initially, upon store() being called, the driver would save the value locally and write it to the design immediately. Because none of the configuration registers in our design were readable by the CPU interface, saving it locally allowed the driver to use the local value in show(). However, later on the driver design changed so that most of the registers would be written all at once when triggering was armed, not immediately inside of store(). This way, when the sample clock changed, the entire sampler could be reset and its configuration could be rewritten next time the user wanted to collect data. Also, due to some lack of communication, the analysis modules were written to accept configuration exactly once before operating, so that configuration was also delayed until triggering was actually armed. Arming the trigger is done through the “trigger_arm” or “manual_trig” attributes of dla-sample, which each have a store() function which rewrites sampler and analysis module configuration then arms the trigger.

The sampler configuration was a little challenging to handle, because after its clock is reconfigured there is a delay before the clock becomes stable during which it is unavailable. In order to avoid overflowing the asynchronous FIFO holding configuration writes, the driver avoids any configuration writes during that time. The sample clock stable interrupt was added to the memory interface's interrupt status register, so that the driver would know when it became stable. It keeps a local flag, clock_stable, and clears it whenever writing to the clock configuration register. If it is about to write to any sampler registers and the flag is 0, the writing process will sleep in a waitq. When the interrupt occurs, the flag is set to 1, and any processes sleeping in the clock waitq are awoken.

Result data is provided to the user through character devices in /dev. Each character device corresponds to one memory channel, which corresponds to one buffer in memory. The driver allocates these using kmalloc(), which guarantees contiguous physical pages, then the physical base addresses and lengths are written to the memory interface configuration registers. Whenever a process uses the read() syscall on one of the character devices, the driver reads the tail pointer from the memory interface and compares it to its saved head pointer. If they match, there is no new data, so it sleeps in a waitq. On a memory write done interrupt, all processes sleeping in that waitq are awoken. This way read() will block until data is available, which is the behavior that is usually expected of the syscall. Finally, the driver copies all

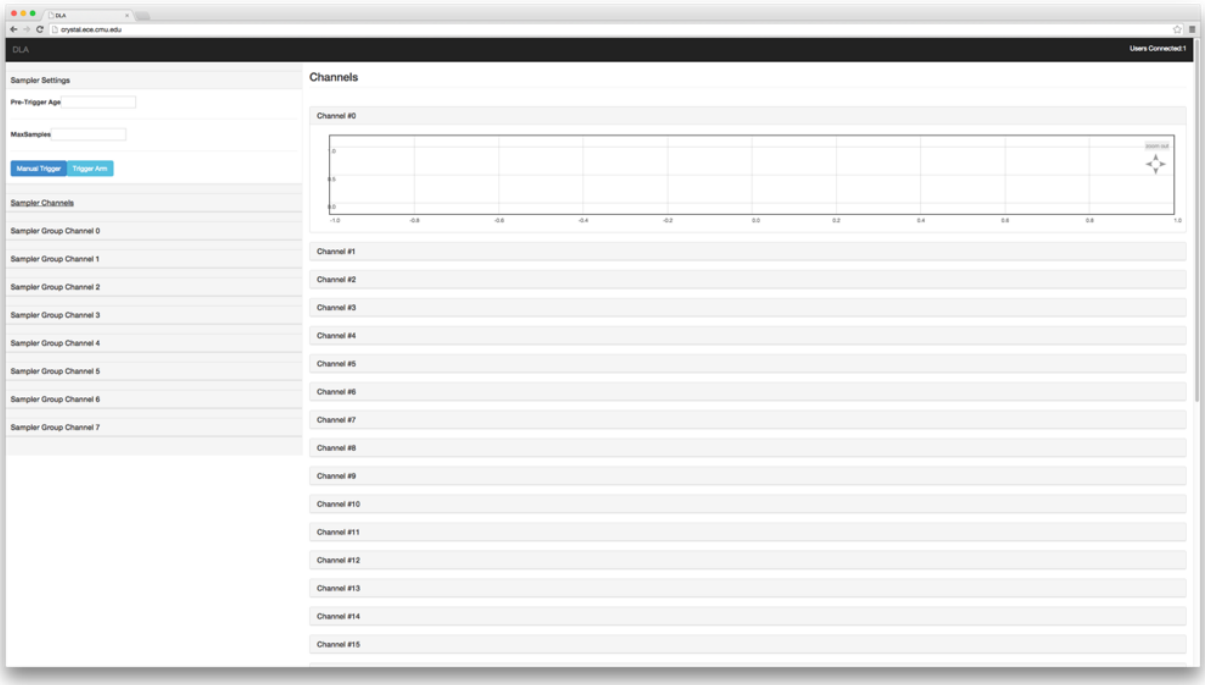
available data into the userspace buffer, and writes head back to the memory interface to let it know there is more space in the memory buffer.

The Linux kernel used was the PetaLinux kernel provided by Xilinx, and the driver code and Makefile were written based on the kernel module template provided by the PetaLinux SDK. An Ubuntu Linaro filesystem image was used for the rest of the Linux system instead of the usual PetaLinux ramfs, because we wanted a writable filesystem on the SD card which we could install node.js and other tools on.

WEBSITE

OVERVIEW

The Web Interface is the interface through which a user interacts with the DLA. The interface's design goals were ease of use, full functionality of all registers exposed through the driver, and the ability for more than one user to use it at a time. Through a combination of several technologies any device connected to the Internet can access the interface (when DLA is powered on) by visiting crystal.ece.cmu.edu. As other users make changes to the registers and the data changes, the user's interface is updated in real-time to reflect those changes.



CLIENT SIDE

The client side of the interface was constructed using Bootstrap, jQuery, Socket.IO, and Flot Charts. When a user visits the url crystal.ece.cmu.edu the interface is loaded. The only pieces of data that must be known at page load are the number of channels and triggering group in order to create the HTML DOM. Once all resources are loaded, the client opens a Socket.IO connection to the server where the server sends the status of all the registers, current data, and # of users connected. Whenever any of this information changes on the server each client will receive a new message from the server.

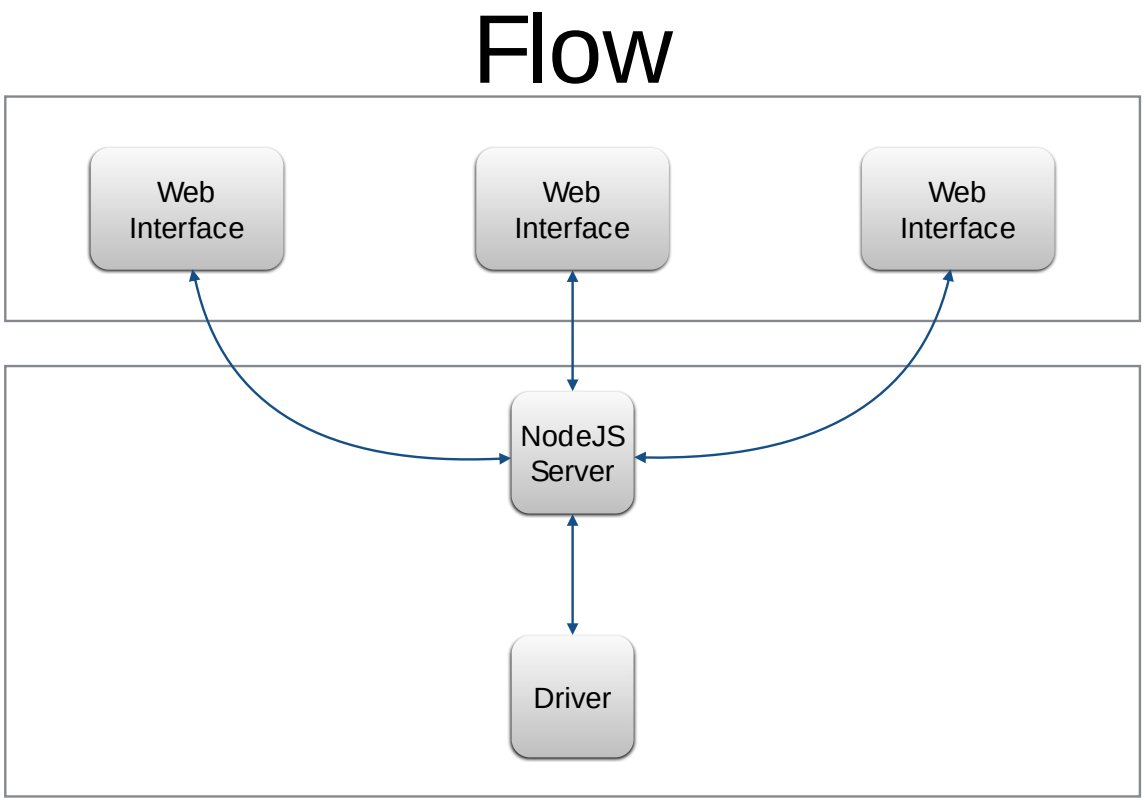
The registers are configured on the left-hand side of the interface. Each menu group is collapsible. When each input is changed the client emits a message to the server over the Socket.IO connection. If another instance of the interface changes a register all other clients will receive that change and update the HTML element through JQuery.

Each channel has its own (collapsible) graph allowing for easy viewing of data. Each channel can be individually panned and zoomed. Moving one's cursor over the data points display their value, timestamp, sample #, and functional module information (if pertaining).

SERVER SIDE

The server side of the interface was created using NodeJS and Jade. Whenever a user makes a request for the interface the server is responsible for serving the respective files. The server also listens for Socket.IO connections and communicates the state of the DLA by reading the files exposed through the driver. Whenever a sampler or functional module would have more data available on its respective dev device the server would detect the change and broadcast this information to all connected clients.

The functional flow of the Web Interface is shown below:



TECHNICAL SPECIFICATIONS – TESTING

UNIT TESTING

Unit testing was done by the author of each module. The minimum requirements were to make sure that data went through correctly for at least some subset of settings, and to make sure that the inputs and outputs were hooked up accurately enough to be able to be placed in the full testbench during integration testing. For most of us, unit testing was done almost entirely in simulation. We each had our own preferred setup- one of us preferred to run simulations in Vivado in the lab, another preferred running it at home after VNC-ing into the lab computer, and the other preferred to run Synopsys VCS at home. While our methods differed, the end result was the same for preparing our modules for integration.

INTEGRATION TESTING

Integration testing was split into two parts. The first part focused on testing in simulation. For this a full test bench was created. This test bench was capable of writing to registers via simulated memory writes as well as reading from memory via simulated memory reads. The testbench first set all of the configuration registers for the memory interface, the interface module and all of the functional modules. After doing so it would set the clock and wait for the clock to become stable again. Following this it would write the registers to configure the sampling channels. At this point several tasks were written to create input data for the DLA. One task was written to simulate a random waveform across many samples. This was used primarily to test triggering mechanisms and to make sure that samples reach memory properly. A second task was written to test SPI. This task recreated an SPI waveform and was capable of sending any arbitrary eight bit packet. This task was incredibly useful in verifying SPI functionality and integration. A third task was created to verify I2C. This task was capable of sending any arbitrary packet over I2C. It was used to verify that all created I2C packets reached memory.

The second part of integration testing was performed on the Xilinx Zync ZC706 development board that the DLA used. A set of scripts were created in order to perform this type of testing. These scripts activated certain registers in the design using the driver. Based on the configuration various parts of the design were tested. One script was created which could configure the sampler to preform sampling and deposit the results directly into memory. Another was created to allow for sampling at different clock frequencies. Separate scripts were created to test SPI and I2C. Final integration testing involved placing the web framework on top of the driver and using it to configure the DLA. This was done in a fashion that allowed for a wide variety of systems tests to be run.

RESULTS

COMPLETE

While the DLA does not meet the specifications that were laid out at the beginning of the project, many key portions are complete. The entire sampling system including the triggering system functions exactly as planned. In testing we were able to configure the trigger module to engage when a number counting up across eight different channels reached a certain value. We were able to sample a square wave at frequencies ranging from 1KHz up to 15Mhz. Pre-triggering was also tested in a simulated real world example and found to work properly. The interface module is also fully functional. It is capable and tested on board to route signals from any of the sampling channels to any of the functional modules. It is also capable of bleeding off packets from these lines and delivering them to memory. The memory interface also worked exactly as planned. It is capable of pulling samples out of the interface and delivering them to the driver. It is also capable of pulling data out of the functional modules as well. Additionally the CPU side of the interface is also capable of writing to the configuration bus as well as reading from the memory module. The Linux driver written to interface between the website and the FPGA fabric also works as planned. It exposes all of the data coming out and the configuration inputs while managing all functional requirements. A large portion of the web interface is complete. It is capable of reading data from the sampling channels and displaying it as a waveform. It is also capable of setting up the triggering module, initiating a manual trigger, and setting the clock. Finally the website is also accessible to multiple users simultaneously.

INCOMPLETE

Some parts of the project remain incomplete. The main pieces missing are the full integration of each of the analysis modules at each level of the design. I2C is the furthest along, working on a basic level in hardware, and showing up on the interface. However, it is not really displayed on the user interface, only listed in a text format. SPI works in simulation, but when integrated into the hardware design it outputs two packets and no more until the FPGA is reset. XMEM was shown to work on a basic level in simulation, and was synthesized and put onto the hardware, but due to time we did not test it with our hardware test platform. It also would not be displayed in the user interface anyway. UART was not finished due to time. One other thing missing from the user interface is the ability to map input channels to analysis module inputs, which must be done manually by writing to files in /sys exposed by the driver.

CHALLENGES

The project faced a variety of technical and logistical challenges in design, implementation, and testing. Many of the challenges arise from interfacing many different types of systems together across multiple clock domains and platforms. One key problem was how to guarantee that samples from different sampling channels would arrive in order to the functional modules. This is important to guarantee that the waveforms would be reconstructed correctly. While this problem was eventually solved it proved to be a less than optimal solution that limited the maximum pretrigger age to the size of the pretriggering buffer. Another key challenge was handling the large amount of data streaming across the clock barrier between the sampling region and the interface region. This required the use of asynchronous queues and careful planning of how each module would enter and exit reset. By far the biggest challenge during design was to integrate each level of the stack. There was a major bug in this process which caused the entire board to crash requiring a hard reset. In the end this bug was traced to an issue in which driver writes to the configuration bus did not work correctly. However finding this issue was troublesome because of the complex integration of each of the three levels. The project also faced a significant logistical problem. Because of group size it was difficult to find times where all four members were available. As a result meetings did not occur with the frequency that they needed to occur at. Because of the infrequent meetings certain members of the group got off track and did not meet the schedule. This caused their parts of the project to slip significantly and require other group members to either complete them or they were not completed at all.

LESSONS LEARNED

We chose this particular project because it enabled us to design a system from the ground up instead of implement an existing design. This helped teach us to think like designers of a system instead of implementers of a system, meaning that we had to take responsibility for each of our design choices and thoroughly think through each decision. In addition to this, we had to think through which choice made the most sense to implement both in terms of time and complexity. We had to design for not only hardware constraints and ease of implementation but also for the constraints of the course, in particular the fact that this is a semester-long project and not a multi-year one. There was more space on the board than we needed, but we also tried to design so that we would keep utilization low in order to synthesize faster.

As a result, we learned a lot about the consequences of design decisions early on, and how to remedy them later on when we decided they were no longer the best design. Specifically, we re-designed the router several weeks in, choosing to instead move data through a set of interfaces as explained in previous pages. We chose this because it would guarantee order of outgoing packets, simplifying the protocol analysis modules. This also allowed us to not be subject to the constraints of having a system-wide data packet and allowed us to run at a faster speed, an important component of a logic analyzer.

We also learned quite a bit about the specific tools we used. None of us were familiar with Vivado before embarking on this project, so we struggled with a lot of its reasoning at times. We spent a few weeks discovering (and rediscovering) useful features, sometimes only to forget them later. The individual who worked on the website portion of the project was also introduced to new applications.

As a group, we learned a lot about time management and project management as well. We went through several iterations of how we would keep each other in check and on track before finding that none of us worked in quite the same ways or on the same schedules. We gave each other an arguably excessive amount of leeway in the beginning for missing our deadlines, and reeled that in significantly as we got more and more behind. Keeping the entire group up to date on work done and status of individual portions proved difficult when some members refused to report it. Sometimes the status reported was inaccurate, and this set a precedent of not being able to trust that work was done without having proof presented. The lesson learned from this was to find a method or set of methods that worked for everyone, and to designate someone to keep track of everybody's responsibilities.

PERSONAL STATEMENTS

JULIAN

I worked on a large variety of tasks for the Digital Logic Analyzer. In the beginning I worked on overall architecture including what the different modules would be and how they would communicate with each other. At this point we assigned specific tasks to each member. I was initially assigned to complete both the sampler module and the interface module. I designed the entirety of the sampler module. I first worked on a specification in order to determine what the sampler module needed to do, how it would interface with the rest of the project, and how the internals of the module would work. After this I created a very detailed block diagram and data path. These documents went through several iterations based upon feedback from the group and the course staff. After these documents were finalized I wrote the initial design in System Verilog. Based on this design I wrote unit tests and verified functionality. In parallel with writing the Samper module I also began work on the interface module. I went through several iterations of the interface module due to design requirements changes. The initial version only expected eight sampling channels and as such was configured as a router. This would have allowed for significant space savings, easier integration, and two way register communication. This task however became significantly more complicated when the course staff asked us to increase the number of sampling channels. This change required a significantly more complicated router. As such I designed and implemented a most-full-fifo-first router. This router routed the most congested channels. This would have worked had the functional modules not required both a latency guarantee and an order guarantee. I thought that these would be solved through reordering but it was determined that this problem was unbounded and thus unsolvable for streaming applications. I finally rewrote the interface to use a point to point routing scheme that guaranteed in order correctness. After writing this I completed unit tests and then integrated with the sampling channel. At this point I created an overall top level design and preformed a full sampler to driver test with Tom's help. After this was successful I went back to a couple other areas including configuring the sampler clock. This concluded all of the portions that I was assigned. However since the group was behind I also worked on helping Doci complete her SPI module. This involved significant debugging and logistical help with the tools. I also wrote a full testbench that included the functional modules. I spent a significant portion of Thanksgiving working on debugging various bugs in other parts of the design. I also developed the DUT that would be used for our demo. Then on the final night of the project I wrote the majority of the I2C module as well as tested it since the person who was supposed to write it did not do it for the entire semester.

There are several improvement that I think could be made to the class. The first of them is that I still felt that there was too much logistical overhead and that the overhead was not in the right place. I felt that the weekly individual and group status reports were rather useless. Rather than having to present to the entire class I think it would have been more useful if each team met with the professor for 15-20 minutes each week. This would have had a greater effect on accountability and thus allowed teams to remain on schedule. I also think that the class could have benefited from more involvement by the teaching assistants. While I understand the difficulty of TAing a capstone class I feel that the teaching assistants

could have at least provided some sort of design review feedback based upon their experience (which should be considerable if they are TAing a capstone.) Finally, I really think that the focus on recreating videogames, while fun, does not serve the best interests of the class. Actually creating a design from scratch, close to what is done in the real world, provided immense value to me that would not have been offered had I simply copied somebody else's work.

DOCI

I very much appreciated the fact that our project was one we designed entirely. While it may have been simpler or easier to not do any system-level designing at all and just have to build sets of black boxes that interacted with one another, I learned a lot more being on a project that required design. Whether I was giving my own input or listening to the thought processes of others, designing places you in a different mindset than taking a set of specs and implementing it does. Part of the design process was research, during which I learned a lot about the low-level specifics on the FPGA of how dynamic clocking works, particularly how clock dividers and clock regions came into play.

My roles for the group were primarily to work on two protocol analysis modules, SPI and UART, and later included managing our tasks. While the latter did not take any time during the first half of the project, I spent almost the same amount of time managing and writing toward the end of the semester.

The biggest issue for me during the first half of the semester was simply understanding all the tools, protocols, and what requirements for the project were. We as a group spent quite some time narrowing down our design, as feasibility issues and additionally requested features kept coming up. I personally barely touched the Vivado toolchain until late in the semester, at which point I had to ask Tom and Julian how to operate everything in the rhythm with which they had set up. I'd been primarily using VCS as my go-to simulator, as it was a platform I was already familiar with and took less time to load and run. I also misunderstood a particular part of the SPI protocol, the clock phase, for a week or so before catching my mistake. As a result, most of this first half was spent writing and rewriting FSMs and datapaths, implementing a basic version in simulation, catching an error, and redesigning parts of the module or scrapping it entirely and starting from scratch.

The biggest issue for me during the second half of the semester was undoubtedly that of team responsibilities. At this point, we were working on integrating the hardware modules with one another using the testbench Julian had written. I was not as familiar with Vivado so often required his help to sort through all the obscure error messages and critical warnings. The memory system and Linux driver occasionally ran into errors, and it was incredibly difficult to trace and determine whether it was the fault of the protocol module, the memory interface, the generated IP interconnects, or the driver itself. These were the errors that eventually killed the SPI module, rendering it unfit for demonstration. We found it functional during simulation, able to synthesize, but unable to run on the board. While I'm sure long nights of debugging would have eventually found the issue, we were very behind schedule and did not have time. Since we did not have a working analysis module at this point, Julian magically wrote an I2C module in a few hours' time, and that was the version we were able to demo with.

But the issue at hand really is team responsibilities, and more specifically, personal responsibilities. While we all started slow during the first few weeks, we were still on top of things and completed our labs on time. As the semester went by though, this pace intermittently sped up and slowed down. The person we were annoyed at for being late or absent to every meeting for the first month became the person we were annoyed at for promising work they did not and would not do. The rest of the team's

attitude varied wildly between excessive complaints and excessive complacency, and confronting the one seemed to have the uncanny ability to bring out the other.

That said, I'm disappointed but just barely satisfied with the state of our project by the end. Incomplete work, my UART module included, was largely to blame for this but the root causes of such lackluster performance was really a lack of interest for this project, a lack of motivation for the class, and a lack of respect for other people's work and time. Perhaps this would not have occurred if everyone had agreed on a specific end-goal of the course, or perhaps the TAs could have caught this earlier if they were more present.

Regardless of our team's issues, I think there are several aspects of this course that can be improved upon. We were required to submit online status reports, but nothing was ever done as a result of or because of them. When issues were raised, which would be a huge red flag if the teaching staff were verbally told of them, they were left alone and never addressed. I understand it is a very independent and hands-off course, but having to meet with the course staff regularly (and without the rest of the class) to report things would have provided a pressure to complete tasks that team meetings and status report presentations could not. In addition, the detachment of the course staff from the process of our work felt like passive encouragement to do the bare minimum. The equal grading of mid-semester conveyed the same message, which was disappointing since we were already facing responsibility issues. So all in all, this course was one I definitely enjoyed and learned a fair amount from but wish it was more structurally organized so that students could be held accountable for their work, or lack of it.

TOM

My contributions to the design include the CPU interface, the memory interface, the kernel driver, the XMEM analysis module, and various aspects of system-level design. Initially we had team discussions about the system architecture, and I did some research on how to interact with the ARM CPUs. Later on when we decided the sampler interconnect needed an overhaul, I contributed to discussions of how to solve the problems that had come up. I also helped write the clock reconfiguration FSM early on when we were playing around with the reconfigurable MMCM provided by Xilinx. When we divided up work among team members, I was given the CPU interface, the memory interface, the driver, and XMEM. I designed the complete system for delivering arbitrary data output by the analysis modules and sampler channels to userspace programs running on the CPU. This included writing the CPU interface and testing it in simulation, and then writing a single memory channel and testing it in simulation, and then putting the two together to do a sort of loopback test in hardware. A base address and length were written via the CPU interface, and a small FSM wrote some values to those locations in memory, which could then be verified. Once that was working, I significantly expanded the memory system, adding the burst writes, AXI interconnect, full configuration support, and interrupts. This was in close conjunction with the CPU interface, which supported the configuration and interrupts. I tested the full system with a single memory channel in simulation using the AXI bus functional model (BFM) from Xilinx, and then tested the full system with all memory channels in simulation. This testing was not as thorough as it should have been, but I was trying to get everything off the ground as quickly as possible since the project depended on a working memory subsystem. That bit me later when I was finding bugs during full system testing that should have been found much earlier during unit testing. Anyway, once the sampler was ready, Julian and I worked on a top-level integration of our parts, and tried it out on the board. I also had to get a basic driver up and running for this testing. The driver was ultimately not too complex, but had a significant amount of code due to the sheer number of configuration registers and channels and triggering options it needed to support. Finally in the last two weeks, I had the memory subsystem and driver in a place where they were basically working and needed little more work, so I could work on the XMEM analysis module. The implementation and testing in simulation of that took only a few days over Thanksgiving break, since I had already made a basic design during our initial design review. Every day of the final week was a large push to get all of the final features and polish into the memory interface and driver, and to chase down bugs that came up. Julian and I both worked very hard tracking down a particular bug where the kernel would crash horrifically if certain parameters were set wrong, which eventually turned out to be an address decode bug in the memory module. We got the basic system working very reliably in the end.

There are a few things that might improve the class. Firstly, often lecture content did not fill the full time allotted, which seems like wasted time that could be spent meeting with the team in the lab. These lectures could either have more content, or be condensed to allow for more lab sessions. Also, the weekly team presentations were not very useful. The weekly status reports were useful communication to the course staff, but the presentations were kind of redundant and, again, that time could be better spent working on the project. Overall, the parts of the project I implemented were fun.

BRENT

This semester I worked primarily on the Web Interface. This semester our team accomplished the most of the main functionality we set out to accomplish however we never fully finished any one particular functional module. Although one might attribute this outcome to fact that our project was self-designed from the ground up, opposed to building something already designed such as a video game, this was not the case.

Our biggest challenge was team motivation and dynamics. Originally, our entire team started out extremely motivated to do this project, however after several rounds of certain members asking for help and being criticized for it repeatedly, overall team motivation dropped. This project design was situated to some team member's strengths much more than others and that should have been acknowledged more clearly.

The Web Interface took a majority of my time unfortunately this semester. In the beginning of the semester the interface was projected to take a small amount of time. As other modules changed in design and features were added and removed this proved to be a very time intensive task. Since other team members were strongly suggesting that the interface be used to help debug I was told to place a higher priority on it than the I2C module. I learned a lot from working on this, although I would have preferred to spend more time on the I2C module.

In regards to the course structure itself I felt that the weekly status updates were not worth it, as we did not receive any feedback on them. I feel that bi-monthly team meetings with the course staff would have better helped our team and the course staff's understanding of our progress.

TABLES & APPENDICES

REGISTER MAP

Base Address	Count	Module
0x40000000	96	Memory Interface
0x40000180	32	Sampler
0x40000200	96	Sampler Interface
0x40000380	2	I2C 0
0x40000388	2	I2C 1
0x40000390	2	I2C 2
0x40000398	2	I2C 3
0x400003A0	2	SPI 0
0x400003A8	2	SPI 1
0x400003B0	2	SPI 2
0x400003B8	2	SPI 3
0x400003C0	2	UART 0
0x400003C8	2	UART 1
0x400003D0	2	UART 2
0x400003D8	2	UART 3
0x400003E0	2	XMEM
0x400003E8	1	Sample Clock

Table 1: System-level Register Map

Offset	Count	Register Group
0x00	4	Sampler Channel
0x10	4	I2C 0 Channel
0x20	4	I2C 1 Channel
0x30	4	I2C 2 Channel
0x40	4	I2C 3 Channel
0x50	4	SPI 0 Channel
0x60	4	SPI 1 Channel
0x70	4	SPI 2 Channel
0x80	4	SPI 3 Channel
0x90	4	UART 0 Channel
0xA0	4	UART 1 Channel
0xB0	4	UART 2 Channel
0xC0	4	UART 3 Channel
0xD0	4	XMEM Channel
0xE0	1	Interrupt Status Register

Table 2: Memory Interface Register Groups

Offset	Type	Register	Notes
0x0	wo	Buffer Base Address	8-byte aligned
0x4	wo	Buffer Length	8-byte aligned
0x8	wo	Head Pointer Offset	8-byte aligned
0xC	ro	Tail Pointer Offset	8-byte aligned

Table 3: Memory Channel Registers

Bits	Type	Field
0	w1c	Sampler to Memory Overflow
13:1	ro	Reserved
14	w1c	Memory Write Done
15	w1c	Button 1 Pressed
16	w1c	Sample Clock Stable
31:17	ro	Reserved

Table 4: Interrupt Status Register

Offset	Count	Register
0x00	8	Value Mask High for Group *n*
0x20	8	Value Mask Low for Group *n*
0x40	8	Enable Mask for Group *n*
0x60	1	Trigger Control
0x64	1	Max Sample Number
0x68	1	Max Pretrigger Age
0x6C	1	Channel Enable Mask

Table 5: Sampler Registers

Offset	Count	Register
0x000	47	Input Channel for Signal *n*
0x0BC	32	Destination Signal for Channel *n*

Table 6: Sampler Interface Registers

Bits	Field
15:0	Bit Rate
16	Parity
17	Stop Bits
20:18	Data Bits
31:21	Reserved

Table 7: UART Configuration Register

Bits	Field
4:0	Word Size
5	Clock Phase
6	Clock Polarity
7	Three-wire Type (1 no SS, 0 half-duplex)
8	Four-wire (1 four-wire, 0 three-wire)
31:9	Reserved

Table 8: SPI Configuration Register

Bits	Field
4:0	Data Width
31:5	Reserved

Table 9: I2C Configuration Register

Bits	Field
5:0	Half of Multiplier
11:6	Half of Divider
31:12	Reserved

Table 10: Sample Clock Configuration Register

STATEMENT OF USE

The members of this team, Julian Binder, Doci Mou, Thomas Mullins, and Brent Strysko, hereby give permission to any members of a non-profit or educational group to use our documents and code in their projects, provided that no members of the non-profit or educational group benefit financially from these documents and code, and that the original authors of the works are credited. We are not responsible for any generated intellectual property cores of Xilinx, which were provided to us as part of the end user license agreement from Vivado, and are subject to Xilinx's permissions.