Team Sidekick

Written Report

18-545: Fall 2011

December 11, 2011

Alexander Soto
Curtis Layton
Rohit Banerjee

Table of Contents

General Overview of Project ……... 3

PCI-express Endpoint Device …….. 4

PCI-express/DMA Driver ……..… 13

Instruction Register File ……... 23

Personal Statements ……... 29

Acknowledgements …….. 34

RELEASABILITY STATEMENT
All our code is open-source and free for use with proper reference of origin – please consult William Nace for access to our source

General Overview of Project

Our original objective of the project was to implement a standalone system that would execute a normalized cross-correlation algorithm in hardware with the template image being one taken, presumably, by the Moon rover of the surface of the moon and the database image (being correlated against) being maps of the moon's surface. In order to build this system, a few component pieces would have to be built – these included a PC-side application that would be able to transport maps to the FPGA, a PCI-express endpoint device on the FPGA that would be capable of reading and writing data to and from the PC, a DDR controller which would interface with an on-board DIMM to store the maps (and other data), the actual correlation algorithm, as well as the control and register file for the actual algorithm. It was hoped that after implementing the vision algorithm, it would be possible to easily extend the framework to inject other vision algorithms into the ecosystem and develop a computer vision/general image-processing co-processor which would be integrated with the moon rover to perform hardware acceleration of these algorithms. **At a more macro-scale – this framework is motivated by the shift towards heterogeneous computing due to power constraints imposed by smaller processes as well as the plateauing of performance from the integration many, single-thread optimized cores for specialized applications. Heterogeneous computing such as AMD's APU and Intel's integration of CPU and GPU on one chip signifies a shift towards mixing application-specific and general-purpose processors on the same die. Our project is a more specialized application (computer vision/image processing) based system.**

Ultimately, we were unsuccessful in integrating the correlation IP into the system but the general framework for adding computer vision/image processing blocks was successfully developed. The

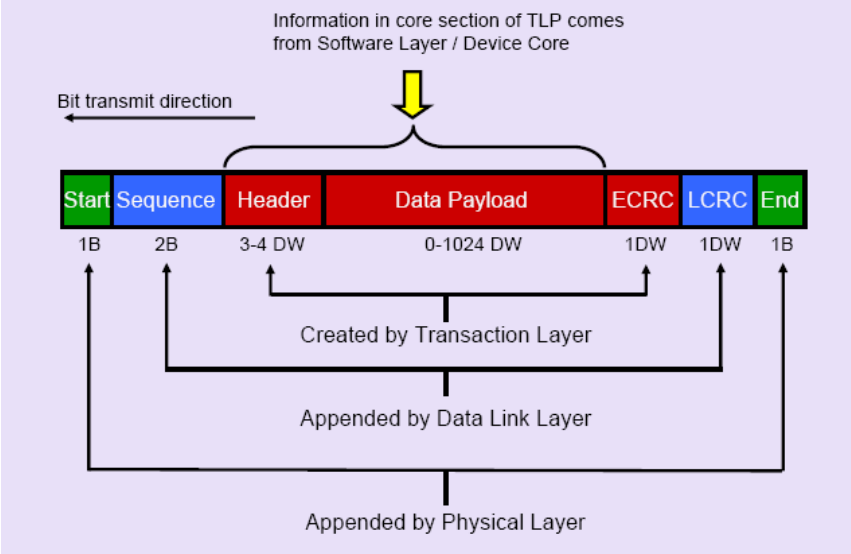
framework of system components that were developed was successfully built and their functionality demonstrated using a common image processing algorithm, Sobel Edge detection. **Our chief contribution in this course is the PC to FPGA link via a working PCI-express link as well as the supporting driver which manages the communication.** This report contains detailed description of the developed blocks as well as the personal contributions and thoughts on the course by the members of team Sidekick. For a full listing of source code and how to run the project, please consult the accompanying CD.

PCI-Express

Overview of PCI-Express Specification

The PCI-E specification is extremely complex as it is an industry standard with many companies involved in its creation. Fortunately, in order to work with it via a Xilinx FPGA, most of those details can be abstracted away. In fact, once you get an FPGA connected with a working interface and driver, PCI-E can be (almost) ignored and hardware design can proceed as if your FPGA now had a simple (but extremely fast) bus to the PC. This is the point that opens doors into the world of heterogeneous computing, and what we wished to exploit for our project.

Most of what will be said briefly can be found on the PCI-E Wikipedia page, and if problems come up this report should not serve as a PCI-E specification guide. However, there are some important things to note. First of all, the lane width dictates the performance of the bus. Scaling from x1 to x8 within the Xilinx tool set is extremely simple as the Verilog is parameterized (the tools even fill in the parameters for you!). The ML505 board we worked with was a fixed width of x1. The longer the bus, the faster it is, and this scaling is linear.



The next important point is packet generation. This is the portion of the specification that will be most valuable in understanding to get a good idea in how the Xilinx blocks work as they do. PCI-E is a layered protocol, consisting of 3 layers.

In this picture we see that there is a header and footer for each layer. Luckily, the physical and data link layer are not important for the purposes of working with the Xilinx PCI-E block. The layer of importance is the transaction layer. It contains a header and optional error correction.

Memory Write 3DW Header Overview

+0								+1								+2								+3							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0	0	TC			0	0	0	0	TD	EP	Attr		0	0	Length									
Requester ID																Tag								Last DW BE				First DW BE			
Address [31:2]																														00	

The header contains the information on to where to route the information. There are different packet types for reads and writes of different sizes. Depending on what level of abstraction you choose, you may be forced to work with these packets directly to route information to the appropriate place on the FPGA. These are followed by a payload, which is what is sent from the driver software as useful information. Packet generation between the driver and the FPGA is a

complex process handled by the OS and motherboard, or more specifically, the PCI-E root complex on the motherboard. Note: if I mention a process is “complex”, that’s me saying that I never had to worry about it while working on this project, I do not mean to sound mysterious. Hopefully, you won’t have to go down a path that worries about those complex interactions either.

Additionally, the memory addressing of the PCI-E device is vital to understand, if you have further questions on this topic look up “PCI-E Configuration Space”. Actually, before I go on let me talk about PCI-E configuration, as this is important to FPGA designs. In general, the PCI-E bus has a very short enumeration time, that is the time it takes for the motherboard to configure the device. Once this window is done, you will have to restart the motherboard for the device to be enumerated. Our solution was simply to program the card, leave it powered on with the AC adapter, and then turn on the computer. There is a more elegant solution that involves Platform Flash on the ML505 card, but we did not attempt configuration in that way. Basically, you can’t configure the card while the computer is on and expect the motherboard to see the device.

Back to memory addressing. PCI-E contains these things called Base Address Registers – each device has 6, 32-bit registers that signify the start of a new address space. Therefore, any one device can have 6 32-bit spaces, or 3 64 bit spaces (or any other combination). In general, it is recommended that a BAR signifies a new function/device on the PCI-E card itself. You wouldn’t use the same BAR for the same task (these are just best practices). These address registers are the offset’s that the OS sees of the device. The OS itself remaps this to a virtual address, which is in turn written and read to by the driver. Yet each BAR needs to be configured by the driver

separately. In addition, when memory enters the FPGA, there is a 6 bit signal that tells the device which BAR was being polled.

PCI-E under Xilinx

So we've covered the barebones basics of PCI-E specification. Now, we go onto Xilinx's implementation of PCI-E and how we can use their tools to quickly get up and running. Xilinx Virtex-5 FPGAs have a PCI-E endpoint device, which is a portion of the chip that is non-configurable, specifically designed to interface with PCI-E. Communicating with this block requires knowledge of Xilinx specific link interfaces (like LocalLink). This is tedious, but allows direct access to those packets I was talking about. Great for when we want to get down to the guts of what's going on. Information on this block can be found in the *Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs – User Guide*. In general, if you're at this level you'll have to worry about things like packet completion signals, and you have to construct the packets to and from the PCI-E device yourself. While this is doable, it is not recommended for a 545 project, as mastering that portion of PCI-E would take a significant portion of the semester.

Fortunately, Xilinx provides a convenient wrapper around this interface called the PCI-E endpoint block plus wrapper. This is a set of reconfigurable logic that wraps around the hard coded block in order to create a friendlier experience in communicating with the spec. Generating this logic is done through CoreGen, a Xilinx tool that creates frequently used blocks of logic. Information on how this is done can be found in the *LogiCORE ™ IP Endpoint Block Plus v1.14 for PCI Express.* This guide is straightforward, but the one important thing that is vital to understand is how the BARs work. Here you get to set how many BARs your PCI-E

device has, as well as what types they are. These don't create actual logic on the chip. They only set parameters that the OS can read. That is, setting a 1024MB BAR2 space does not create memory on the chip for that space, only the ability for the OS to traverse those addresses.

The abstractions don't end just yet! In addition, CoreGen creates an example design that is a basic set of memories that can be read and written to. This is a great starting point to see if you can at least recognize the PCI-E device. On top of this, there are a variety of Xilinx application notes that attempt to help you get started. Depending on what your goals are, selecting the correct app note can be tremendously valuable. Some of the ones we looked at were XAPP 1052 which deals with DMA via PCI-E, and XAPP 859 which merges the example design from CoreGen with XAPP 1052 to create a platform to read and write to the DDR2 memory on the ML505 board. Our design is heavily based on the XAPP 859.

XAPP 859 has 3 components – a windows driver, a PCI-E interface, and a DDR2 DMA interface. Since we needed to edit the driver ourselves, that was done outside of the application note. The PCI-E interface is a glorified wrapper around what is normally output from CoreGen. Finally, the DDR2 DMA interface consists of a register file that can be controlled by a driver. These registers in turn control a DMA engine that is able to speak to the motherboard's root complex, and transfer data without intervention of the CPU. It is through these DMA transactions that our card can reach a throughput of ~100 MB/s. This DMA interface was mainly kept in tack, with some rerouting of the wires so that we could use the DDR2 chip when an increase in bandwidth was necessary.

Base Address Register 2

In our project, our goal was to create a reconfigurable PCI-E plugin card that could be used to accelerate any algorithm that would run faster on dedicated hardware as opposed to software. To meet that goal, we created a second BAR that would hold our custom logic. That is, we would solely use BAR0 to communicate with the DMA controller, and BAR2 to communicate to our custom hardware. This abstraction proved useful, as it allowed us to change one space independent of the other.

Modification of the app note began with modifying receive and transmit engines, which are a core part of XAPP 859. These are portions of hardware that deal with packet generation and decoding. In essence, they can be thought of as giant FSMs which allow you easy access to PCI-E packet information, such as what address a packet was going to, what its payload was, and what BAR it was meant for. They were adjusted to take in information from a second bar, as well as present information into the packet creator to modify multiple BARs.

Additionally, under instruction_reg_file.v, we created a place to work with our custom logic. By this point, we have removed all parts of PCI-E away. We are left with 5 simple ports – read and write data, read and write addresses, as well as a write enable (and clock / reset of course). Interaction with this block is simple. If you write to the PCI-E BAR2 space, it will show up on the write data lines, with the correct write address and a write enable asserted. Similarly, if you receive a read address, you are expected to output data that will be read back from the PC.

So long as these guidelines are followed, then you can treat the PCI-E device as a memory addressable block that can be configured to do anything useful. You can write to block rams. Or register files. Or you can write straight into an FSM you wrote. The possibilities are endless, and quite powerful.

Xilinx Tools

One of the biggest lessons we learned during this project, was that selecting the right tools is invaluable. Since testing a PCI-E device is hard to do in simulation, we did most of our debugging on the board itself. However, this meant that code changes had to go through an entire implementation run – which was a nightmare. In fact, we had a bug that caused our implementation time to sky rocket to 3 hours, because timing constraints were impossible to meet. It turned out that we were using an incorrect UCF file, but throughout this time (about 4 weeks!) we were forced to debug in 3 hour windows. Needless to say, this was not the most productive portion of our semester.

Yet, it led to one of our most valuable lessons – selection of your tools early on has a big impact on what you can do later on throughout the course. We began using Xilinx ISE, assuming most Xilinx features would be supported. We quickly realized that this was not the case, and had to port our project over to PlanAhead, in hopes that they would support partitioning. The end goal was to reduce synthesis time to a point that we could work on the project consistently. However, there was a small detail in that PlanAhead supported partitioning for most – but not all – Xilinx devices. Unfortunately for us, the Virtex-5 LX110T was one such device. Don't make the mistake, and think your tool chain out carefully, it has a lot more bearing than you may think at

first. Yet, in the end it turned out for the best. That's because PlanAhead has much nicer timing analysis tools, and we found quite a few bugs using those tools. In fact, most of those bugs had gone unnoticed in the ISE tool chain.

Out of all the tools we used, we have to say that ChipScope was the most invaluable. Had it not been for the ability to import a logic analyzer into the design, we would have never successfully completed this project. In particular, we had a few bugs that would have been near impossible to catch in simulation because we were working with PCI-E details. The best thing for us to do most of the time was to plug in an ILA into our problematic module and see what was going on.

It's all in the Details

I wanted to write a small section about some details that were trivial, but in the end cost us a lot of trouble and time. First, the switches on the board disengage and connect the PCI-E bus, and they must be configured appropriately for the device to work. I spent way too much time debugging code when in reality I had the wrong hardware switches flipped.

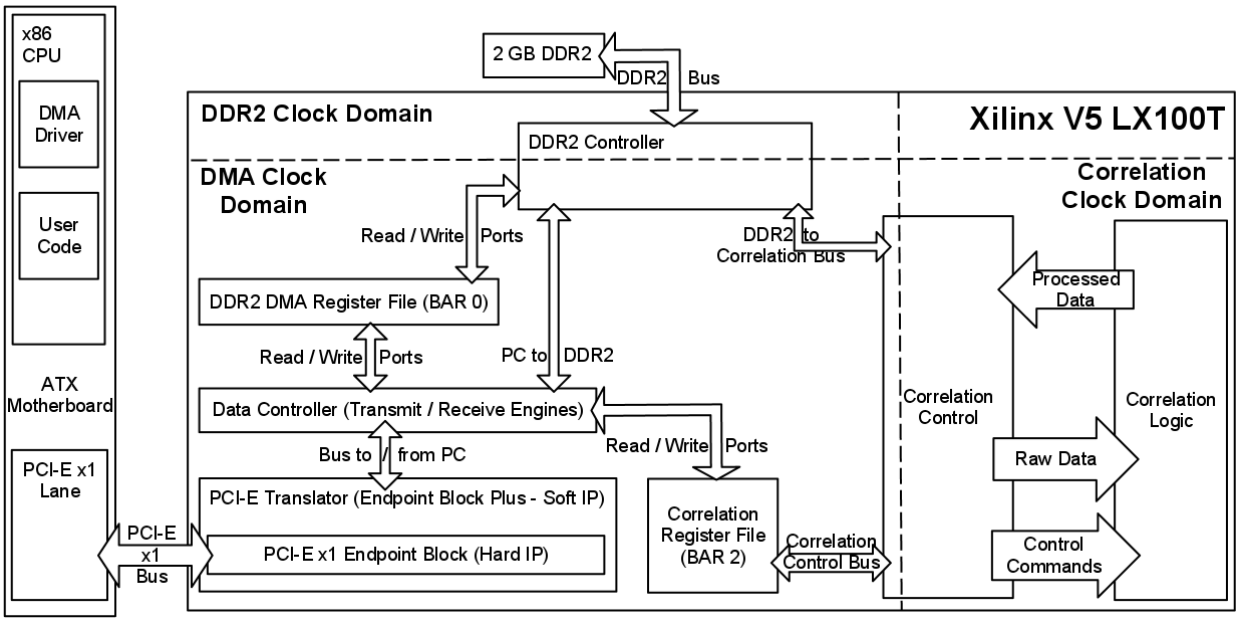
Also, when using a UCF file from an application note or elsewhere read the details. We were using an almost perfect UCF file that seemed to work fine. However, it was meant for the LX50T and had block RAM placements that didn't match. Therefore, we failed timing and were stuck with 3+ hour implementation time. In the end, it turned out being such a silly mistake.

Speaking of details, if you can't find the answer to one of your questions, my advice is to read the *User Guides*. These are like the golden file within Xilinx and always seem detailed enough.

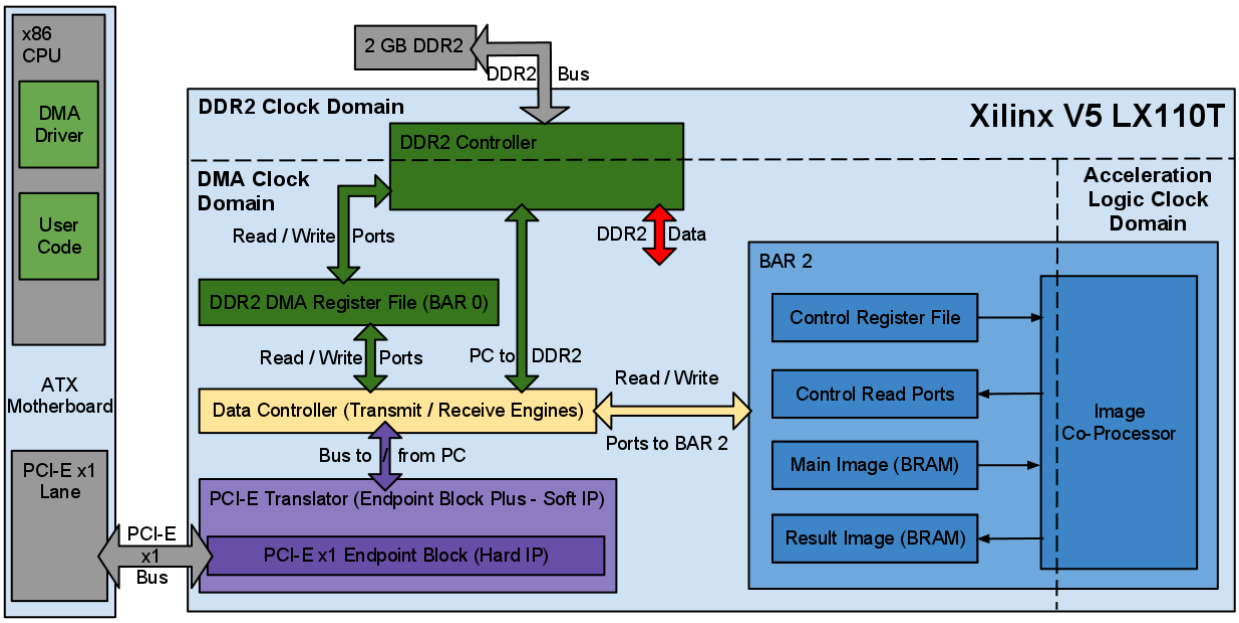
While they may seem intimidating in size, they have yet to fail us throughout this course. In particular bot the Virtex 5 and XST user guides were invaluable.

Design Process

We had a very organic design process, as we were not sure what to expect from the PCI-E side of things. In addition, we were designing an organic product. The end result would depend on the module that we plugged into it. We ended up doing an image processing algorithm, but during our initial design phases we wanted to be as flexible as possible. Below is a diagram with about 1/4th of the time remaining:



As you can see, most of the design was fairly thought out at this point. We knew what the major blocks would be, as we had implemented a fair number of them. However, even late into the class we changed the entire right side of the diagram – we decided to change which algorithm we would accelerate. Our final design is below:



Device Driver

Introduction

This section will explain the design and use of the Linux device driver. Because the XAPP859 application note only includes a windows driver, the Linux driver required for our project needed to be designed from the ground up. The application note did include descriptions of the DMA registers, and their purposes and desired behaviors. This made it fairly easy to test out the driver functionality. If we were constructing both the software and hardware from scratch, it would have been much harder to design and debug the driver. One very important resource for this process was the book, *Linux Device Drivers*, which is an O'Reilly book that is available free of charge on the internet. Given some knowledge about computer systems (i.e. have you taken 15-213?), you should be able to construct a device driver using the information from this book. In the following sections, I will describe both my thoughts on designing a PCI-E driver as well as parts of the driver specific to our project.

Development Environment

The development environment I used for creating the driver was a standard install of Ubuntu 10.04. I did not run into any version issues. It was nice to work with Ubuntu because it allowed me to focus less on getting the system running and more on the actual development. The driver is compiled using the standard gcc compiler. In order to properly compile, I needed kernel source code which is readily available online. I also used a custom Makefile (which is included) to make the driver compile properly, which needed to be run as root. I did all of my coding using gedit, mostly because I am kind of a n00b. I would recommend using vim or emacs!

General PCI-E Device and Driver Information

When the PCI express interface was introduced, it was very important to preserve backwards compatibility within software. In that way, Linux device drivers for a PCI-E device are identical to drivers for PCI devices, even though PCI-E is a serial protocol and PCI is a bus protocol. If you are a designer that has had experience in the past with PCI drivers, you should have no trouble with PCI-E, as the differences will be in the hardware. To this end, I will describe the basics for design a PCI or PCI-E driver, they are the same.

PCI uses a specific addressing scheme to identify devices on the bus, which is split up into bus number, device number and function number. There is also a vendor ID, which is specific to the hardware device. I am not going to go into specifics other than that when designing the driver you need to know the vendor ID and device ID of the PCI device you want to talk to in order for the driver to properly recognize it. As far as designing a system with the Xilinx tools, the vendor ID and device ID are set in CoreGen when creating the PCI Express Endpoint. As long as the

numbers match in hardware and software, there shouldn't be a problem, as long as you don't have two devices with the exact same identifiers. One useful command in Linux for discovering the devices attached to the computer and various parameters associated with those devices is *lspci*. If your device is working (at least somewhat) properly, it will show up after *lspci* is run.

When the host computer is powered on, there is a short amount of time where the PCI device (in the case the FPGA board) needs to power up and set itself in a ready state. The PCI Express specification states that the device must be ready 100ms after the host computer has been powered on. This is problematic if your design is large and must be read onto the FPGA using slow flash memory. We got around this problem by powering the FPGA board separately and programming it before powering on the host. There are ways of getting it to configure in time, one of which is called partial reconfiguration, which lets the device configure the PCI-E endpoint, and then the rest of the application can follow. This means a large design does not have to be ready immediately, however the PCI-E controller can still happily find the device.

The meat of the interface, and the most important for starting the driver design, are the Base Address Registers (BAR). There are six available per device in the specification, BAR0-BAR6. When a specific BAR is configured to be used on the PCI device, it will get mapped into the memory of the host computer when the device is enumerated. This provides an easy communication channel from the PC to the device. You set up the size of each BAR in the CoreGen in Xilinx (or in the hardware configuration of your specific device). The quantity, sizes and addresses of the BARs in PCI devices can be easily viewed using the command *lspci –v*. On the FPGA side, you are on your own to implement the registers. As far as the PCI-E interface

goes, everything is done with packets. The PCI-E endpoint decodes the packets, and if they correspond to reads and writes to the BARs, they will get handled in a certain way. As far as the XAPP859 goes, there are configuration registers that map to BAR0, which are fully described in the documentation.

The Anatomy of a PCI Driver

If you are new to writing device drivers, as I was when I started working on this project, the driver file will look a bit strange. There is no main function, and there may not seem to be a logical flow. This is because the driver file is actually a set of functions, with corresponding markers, that the system will call when it needs them. The functions are marked using flags such as *__devinit* in order for the system to know which function does what. In designing the driver, you must provide certain functions for the system to call. These include functions that run device initialization and removal. There are other (optional) functions that relate to interrupts, and power management. In our driver, I chose to implement the bare minimum.

Init and Probe Functions

The first important function is the init function. This is called sk_init(void) and is labeled with the __init marker. This function gets called when the module is loaded, and contains critical setup information. In this function, a function is first called to request device information given that the vendor id and device id are known. If the system can find a device matching the information, it returns a device_id struct containing useful information about the device (such as the quantity and size of the base address registers). After this step, the device driver is registered with the system, where it passes a pci_driver struct, which contains the name of the device

(important for using the device later) as well as function pointers to a couple of other important functions.

In this section you can also set up interrupts, which I have done but not tested due to omission of interrupt support in XAPP859. The last steps that are performed in the init function are registering the device type and creating DMA buffers. For simplicity, I register the device as a character device. This allows for the use of read, write and ioctl commands in the user space, and generally seemed to be a logical choice. For more information, take a look at the Linux Device Drivers book. For a device that is designed a memory controller, it may be more intelligent to use a block driver. When registering the device as a character device, you pass a struct that contains function pointers to the functions that you want implemented in user space. For our device, the commands needed were read, write, ioctl, open and release. The name of the struct containing the pointers is sk_fops. Also, when registering the character device, you give it a device number. I used 239. This is a number that you will need when you actually go and use the device.

To create the DMA read and write buffers, I used the command *pci_alloc_consistent()*. This reserves a section of kernel space memory that is contiguous and safe to access from hardware. This is very important for DMA, as the device needs to be able to read from the memory independent of any software. The function returns a pointer that the kernel can use to read and write to the memory, as well as a physical address (which is passed back through a pointer). The physical address is used by the PCI device as the target for a DMA transfer.

The probe function is the second step in getting the device driver up and running. In our driver the function is called sk_probe, and is labeled with the __devinit marker. A pointer to this function was passed to the kernel in the init function. The probe function gets called when the pci core thinks it has a device that matches this driver (i.e. after the init function has passed the pci_driver struct.) This function contains some more setup steps, the first of which is to enable the device using the *pci_enable_device()* function. There are also more steps in the probe function for setting up interrupts, but again I have not tested this functionality.

The most important part of the probe function is setting up the base address registers. Using the functions *pci_resource_len()* and *pci_resource_start(),* you can identify the size and address of specific BARs on the device. In the case of XAPP859, the length of BAR0 is 128 and the address is set by the system. It is important to store the address as a pointer (and to keep track of the BAR size, so you do not end up corrupting parts of memory you shouldn't be touching). Before you can use the pointer, it must be passed through the function *ioremap_nocahe.* This function tells the system to not cache any reads and writes, as you are talking to an actual device and not a chunk of memory.

Exit, Remove and Other Functions

The exit function is basically the opposite of the init function. It is called *sk_exit()* and is labeled with the __exit marker. In this function, the device structure is removed, driver unregistered, buffers freed, etcetera. The remove function is called during the exit function. The remove function disables the device.

File Access Commands and DMA

The Actual file access is handled by the read, write and ioctl commands. These functions are implemented in the driver as *sk_read(), sk_write()* and *sk_ioctl().* All three of these commands get linked in using the *sk_fops* that was passed to the kernel during the setup steps. This means that once the device is opened in a user space program, the standard readp, writep and ioctl functions can be used.

The read and write functions set up and execute the actual DMA transfers, using the buffers created during the setup steps. One important note at this point is that for a normal character device, the driver is required to keep track of the file location which is currently being read. This means that if you do two reads, the second one will pick up where the first left off. This was not the ideal case for our implementation, and I chose to design the read and write functions to assume that they will get explicitly passed the file location, which would contain the desired offset. This means that the user space program must use the *readp()* and *writep()* commands as opposed to the standard *read()* and *write()* functions. The functions are the same except that another argument is added for specifying the file location. Also, the read and write functions assume that the file offset that is provided is actually an address into the memory on the FPGA board. So if you specify a 0 for the file location, it will write to the first address of the ram on the FPGA board.

The first operation that the read function performs is to make sure that the address and size that are received are divisible by the DMA transfer size. The XAPP859 design only supports specific transfer sizes, and to make my life easier I chose to use a transfer size of 512 bytes. This means

that the data must be aligned to a 512 byte boundary. This may seem limiting, however our original plan was to transfer large (~2GB) files over in one transfer. After the size and address are properly calculated, a loop is entered that reads the data back in 512 byte chunks.

The actual DMA transfer is governed by the hardware on the FPGA board. All the user does is to write the correct setup parameters and addresses, and signal the device to start the transfer. Note that the complexity of the design is in the DMA hardware, which is a part of the XAPP859. Although it may seem very easy to perform DMA transfers by glancing at the driver, all of the complexity is hidden in the device.

The processes that the read function goes through during each iteration of the loop goes as follows. The BAR0 on the FPGA contains setup registers that are described in detail in XAPP859. The first step is to write to the register that holds the source address on the FPGA side. This corresponds to the offset into the ram on the FPGA board. The second part is to write to the register that holds the destination address on the host machine. This address was given during the setup phase as the physical memory address of one of the DMA buffers. The next step is to write the register that holds the size of the transfer with the transfer size. After all that has been done, all that the driver needs to do is write one bit to the control register on the FPGA board to start the transfer. Because XAPP859 does not support interrupts, the read function polls the FPGA control register which will signal when the transfer has completed. After successfully reading back a chunk of data, memcpy is used to transfer is back to user space.

The write function works almost identically to the read function, except that the data gets copied into the DMA buffer first, and different registers are set for the source and destination of the transfers. XAPP859 supports duplex DMA transfers, and includes different registers for the source and destination addresses for both the FPGA memory and the host memory. I tried to add at least basic support for this in by using separate read and write buffers, however it would be hard to get this to work properly without implementing interrupts.

The ioctl function provides a much simpler interface than either the read or write functions. It allows you to read and write a word of data at a time from within the user program. I set up the function so that the registers from BAR0 can be read and written from user space. It is easy to reconfigure this function, as the most important parts are the *arg* and *cmd* arguments. To make the process easier, I defined an enumeration to keep track of the functions which exists in both the driver and the user program. Although the ioctl function isn't really used in our design, it was very useful for debugging and would be an integral part of a different device driver.

Device Setup in User Code

After the driver has been compiled, it is inserted into the Kernel using the *insmod* command, and likewise is removed by the *rmmod* command. Also, after the character device has been registered, the number that was used before (239) to identify the device is now used to assign it a name. To make this processes easier, I created a script that will set up the correct driver correctly and insert it. After the setup script has completed, all the user has to do is open /dev/sidekick as a file, and use pread, pwrite and ioctl.

Important Notes

To the best of my knowledge, all of the function calls are the in the correct places. There are some choices about which functions actually need to be included, and where those functions are, which will vary wildly depending on the application. What I have provided works, but it may not be 100% correct in the sense of being intelligently designed. For instance, there was not a good sense of exactly how initialization functions should be distributed between the init function and the probe function. Similarly, it wasn't clear what should go into the remove function and what should go into the exit function. I came up with my design by reading through a bunch of other driver source code, and following the book as closely as I could.

DMA Driver vs. Demo Driver

For our demo, we did not use the DMA portion of the driver. Instead, the read and write functions were modified to read and write to BAR2. This is the only major change.

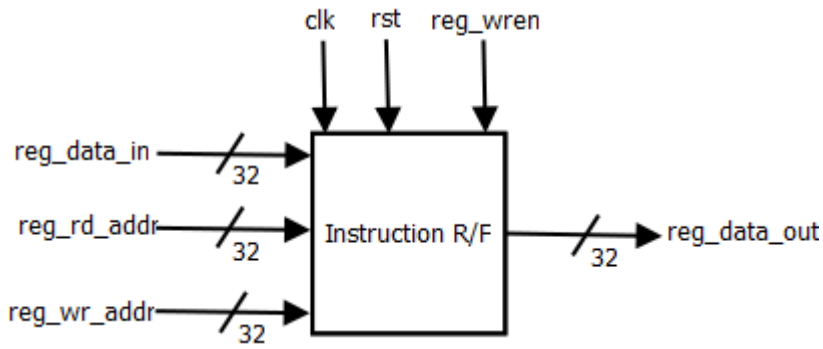
Future Work

My goal in designing this driver is that it would provide a starting point for creating other Linux PCI-E drivers. There are a couple of caveats with the driver I created, mostly because there are quite a few parts of the driver that are specifically tailored to be used with XAPP859. If you design a character mode device like I did, I would recommend properly implementing the file location parameter (i.e. the driver should keep track of it). Also, the DMA system would work better if interrupts were used, however this would require some extensive modifications to XAPP859 and make the driver more complicated.

The best advice I have in designing a device driver is to read *Linux Device Drivers.* It is full of useful information, and it is fairly easy to find which parts are needed for your specific project.

Instruction Register File

The initial plan for our project was to integrate the correlation.vhd HDL source from the Robotics Institute into an ecosystem which would allow the algorithm to run as a peripheral device. Our objectives were to build the PCI-express endpoint device, a DDR controller, and a Register File with which to communicate to the integrated algorithm -- the PCI-e was constructed because the maps we planned on passing to the correlation algorithm were somewhere between 300 and 400 MB in size meaning we needed the throughput of a PCI-e link as well as the storage capabilities of a DIMM chip. However, since the project reached more of a proof-of-concept phase, we ended up not using the DDR control or RAM chip (though fully verified it) and not needing the full throughput capacity of the PCI-e link. Thus, all the data that was communicated to and from the x86 CPU to the FPGA board was stored in an internal logic management/storage control structure known as the Instruction Register File.



Two of the internal block RAMs are main_mem and avg_mem; main_mem is capable of holding 76800 bytes, one for each of the 320x240 byte-wide chars which make up the input image from

the webcam. The other block RAM, avg_mem is capable of holding the same number of shorts; the width is double the size of the input because the output of the Sobel edge detection algorithm is 12 bits. The R/F is tied to the PCI-e clock:

On every positive edge of the clock, if the register file's write enable is asserted and the control bits of the R/F write address indicate that something needs to be written to the main_mem block RAM, then the data is written into the block RAM. This is the logic for loading the image into the internal structure of the FPGA. Once the entire image has been written, a register which maps into the address space designated as the control_mem block RAM is asserted high indicating the data is ready for reading; this signal is referred to as edgeReset.

Alternately, the logic for reading data out of the R/F is very similar to the write logic -- at every positive edge of the clock, reg_data_out is set to the data present at the address specified at reg_rd_addr. The addressing is all taken care of at the driver level so no further decoding is performed at the RTL level.

In order to meet timing, the Sobel Edge detection logic is run at a clock which has twice the period of the PCI-e clock -- this is accomplished using a simple clock divider circuit which uses a 2 bit counter which increments on the positive edge of the PCI-e clock and outputs a 0 when the counter value is 0 and a 1 when the counter value is 2 ensuring that the period is twice that of the PCI-e clock but the duty cycle of the previous clock is maintained. All the control logic for sending and receiving data from the Sobel Edge Detection block is run on this clock which shall be referred to as ourClk.

The registers and control signals used to communicate data to and from the Sobel Edge Detection block are as follows -

2D register array of 9, 32-bit values known as matrix into which the 9 pixels representing a 3x3 grid are loaded; Two 32-bit values known as row and col which are used to calculate the address of the pixel within the image; One 4-bit register known as registerCount which counts up to 9 to indicate that a full grid of pixels has been read; One 1-bit register known as isDone indicating that the entire image has been processed; The 32-bit edgeReset value which is set by the driver indicating that the actual image processing should begin; One 1-bit register called writePixel which is an internal control signal indicating that the output value from the SED block should be written into memory; One 32-bit wire sobel_out which is tied to the output port of the SED block and holds the result of the algorithm for one grid of pixels, One 32-bit register known as matrix_write_addr which holds the value of the address in the main_mem block RAM where the pixel we are currently looking at resides. Some local parameters are MAX_HEIGHT and MAX_WIDTH which are set to 320 and 240 respectively (webcam image dimensions).

At the positive edge of ourClk, if edgeReset is asserted high then all the above signals are simply set to 0. If edgeReset is not asserted however, then we can be in any one of 3 states.

State A: (registerCount > 9) indicating that we have fully loaded matrix with the 9 32-bit pixels. In this state, registerCount is reset to 0 and writePixel is asserted. Furthermore, if we have reached the end of the image (row = MAX_HEIGHT - 2 AND col = MAX_WIDTH-2) then we assert the isDone signal. Since we iterate over the image in row-major order, the other case is that we have reached the end of the row (col = MAX_WIDTH-2) and we need to reset col to 0

and increment the row variable. Otherwise, the only other case is that we are in the middle of a row in which case col is incremented.

State B: (registerCount < 9) in this situation, all we need to do is increment registerCount indicating that we will be reading the next pixel;

Depending on the value of registerCount, the matrix_write_addr is set to the appropriate address using the row and col offsets and the matrix 2D array is loaded with the pixel.

State C: (writePixel is asserted) in this situation, the avg_mem block RAM is updated with the output of the SED block.

Please consult the code for a complete understanding -- essentially, we are iterating over an image pixel by pixel in row-major order and filling a grid of 9 pixels based on our current location. These 9 pixels are sent to the SED block for processing and the output is written into the avg_mem block RAM which represents the pixel array for the greyscale edge-detected image. This is a fairly inefficient algorithm -- it amortizes out to each pixel being separately read approximately 3 times due to the overlapping of the grids; furthermore, we could technically reuse the same block RAM that the image is being stored in and save memory that way. Once the entire image has been processed, the isDone signal is asserted which signals the driver to read the contents of avg_mem into the output file from which the image is reconstructed by the demo program (explained below).

Sobel Edge Detection logic

The Sobel edge detection logic is open source and was obtained from http://edge.kitiyo.com/ with some modifications. There are 8 32-bit input values representing the 8 pixels surrounding

the central pixel in a 3x3 grid of pixels -- it should be noted that only 8 pixels are required for the convolution because a pixel cannot have an edge against itself. The output is a 32-bit value which represents the result of the convolution

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A}$$

.

Essentially what the block does is convolve the Sobel operator matrix and the transpose of the operator matrix with the grid of pixels to determine the presence of vertical and horizontal edges. This value is output back to the instruction register file to be written into the avg_mem block RAM to be read out by the PC.

Demo Program:

The demo program was designed to show the full system in action and integrate it with an interesting application in order to show the system in use -- for the purposes of the class demo, we decided it would be interesting to take an image using the Logitech webcam, invoke the edge detection algorithm housed within the V5 FPGA, and then display the modified greyscale and normalized images to the user.

The driver for the webcam was installed on the Linux (Ubuntu) box that we were using to interface with the FPGA during development -- it was our central code repository for the DMA driver and became the working location for the demo executable. The first call in the demo program is to actually take the picture -- the USB webcam is detected as a device on the Linux machine and is mounted as /dev/video0. Using the streamer UNIX command, a ppm format

image is captured from the webcam and stored in a pre-determined location. The next step is to convert this PPM image into a usable image which can be transported to the FPGA - this was done in the PPMReader class where the PPM image is opened for binary/reading as a file; the header information is stripped and stored - the only useful information in the header is the width and depth of the picture as this affects the logic managing the actual sobel edge detection algorithm. Next, each of the pixels in the image is read into a struct whose member variables are three unsigned chars representing the red, green, and blue components which make up each pixel.

The Sobel edge detection algorithm implemented on the board is only able to use greyscale images so the next step is to convert the RGB pixels into greyscale pixels -- this was accomplishing by effectively creating an array of chars which were a weighted sum of the RGB chars. The formula is:

0.2989 * R + 0.5870 * G + 0.1140 * B

Once this array of chars is constructed, it is written to a file which will be transported via PCI-e to the FPGA for processing. The internals of the DMA driver are discussed in detail in a separate section but at a high-level, certain registers signifying transport parameters are set and then large chunks of data representing the greyscale image are passed to the internal PCI-e endpoint device instantiated in our hardware. The driver is invoked from the demo program and the transport occurs -- a very small time later, another output file is generated in which is contained the output data from the Sobel edge detection algorithm. Each of these is a greyscale pixel value -- a new 320x240 image is constructed in which each of the RGB pixel components is set to the corresponding greyscale value. Furthermore, a histogram of pixel values is constructed from

which a median pixel value is ascertained - a second pass is conducted in each of the pixel values where values above the median are set to the black pixel value and those below are set to the white pixel values. This normalization step is used to filter out the noise which may have been contained in the previous image in order to really define the edges. The source files for the demo program is contained in the ~/../cam directory - they can be built using the included Makefile. The program run_sidekick can be executed by moving the built executable to the ~/../demo directory and then running it. The Logitech webcam has to be attached via USB, the appropriate bit file has to be loaded into the V5-LX, and all connections and power sources have to be set in order for the program to run.

Personal Section for Curtis

What I Did

I took the device driver as my part of the project, and wrote the entirety of it. I learned a great deal from the whole processes, and I feel confident that I could do it again. It was an incremental process and involved a great deal of reading, and researching other drivers.

Once the driver was mostly completed, I moved on to help my teammates with hardware debugging. I worked a great deal on resolving some of the device mismatch issues we were having. During that processes, I was able to learn a great deal about how constraints work, and why they are important. I also learned more about how a design goes from Verilog all the way to being put on an FPGA. Before taking this class, I was unsure of the process, but I feel like I have emerged feeling much more confident. Although I did not create any Verilog, I spent a great deal of time staring at it, and figuring out how it worked.

Words of Wisdom

If you are to embark upon designing a device driver, make sure that you have good access to the hardware that you are writing the driver for. If you are not 100% sure of what the hardware is doing, you will never be able to properly design and debug the driver. This is mostly because it becomes impossible to distinguish hardware and software bugs.

Along the same line, it is great to use an application note where a lot of the hardware design is "finished and tested". However, this can also lull you into a false sense of security, where you are quick to blame mistakes on your software instead of someone else's "tested" hardware. This was the case with our project, where the DMA transfers were incorrect unless a short wait statement was added in before polling for a transfer to complete.

Another note for designing device drivers is that there seems to be quite a few ways to do the same thing. If you look through the code for other PCI-E drivers, there is a chance that things will look quite different. The same basic framework should be the same, however.

As far as choosing your group, I have one important suggestion. If you are not comfortable with computer hardware/Verilog and you still had the nerve to take this class, make sure that the skills that you do have match up with your team. However, I also think it would be bad to only have hardware experts on a team, but it would be way worse to not at least one teammate who has taken a higher level courses involving Verilog. This should also play a big role in choosing a project. Make sure very early on that your project lends itself to the skills of you and your teammates.

Class Impressions

I came into this class without having touched Verilog since 18-240. I had previously worked with FPGAs, however it was on more of a block-level design basis. I have more experience with embedded systems and programming. I knew that it was going to be a challenge. It was not impossible; however this class would have been useless without teammates who have had more hardware experience than I have.

Personal Section for Alex

What I Did

I had an elementary understanding of the PCI-E interface with Xilinx tools from previous work experience. Therefore, I was tasked on setting up the PCI-E hardware and modifying it such that it would be simple for us to use throughout the semester. I took the Xilinx application note 859 and changed some of the engines within so that our driver could communicate to two distinct address spaces, and so that it could treat them as two different devices. Additionally, I did some basic debugging on the driver code while we were trying to get it to work on the hardware to initiate DMA transactions.

Words of Wisdom

I've stated most of my technical advice within this report. However, I'll take the time now to speak on team dynamics. Make sure you are in constant communication and that everyone knows what they are tasked to do. We failed at that, and decided to work on portions of the project autonomously. This led to very late realizations on what was possible, and caused our

project to pivot very quickly. We pulled through, but would advise you against that situation if at all possible.

Class Impressions

Overall, the class was well structured. I thoroughly enjoyed the ability to select our own project. In fact, that was our sole task for the first two weeks – we wanted to do something different and that made an impact. We're hoping we've accomplished that here by creating a framework for others to easily build upon. Additionally, I think the status reports were a good form of keeping us going. However, I will mention that mandatory in lab time was not too helpful for our group. In general, I am most productive when I work on one task for an extended period of time and a lab in between classes was not suitable for that schedule. Additionally, I don't think many of my group mates made productive use of in-lab time. I understand why they exists, just giving my personal opinion.

Personal Section for Rohit

What I Did

My primary job was to work on the actual correlation software which was the initial objective for our project. At first, the plan was to simply port those parts which required hardware acceleration to hardware and leave the rest of the software unchanged -- this turned out to be infeasible however as the use of several object-orient programming concepts, containers etc. as well as the use of non-standard libraries ensured that the overhead of making the data hardware-ready and then software-ready after the FPGA resources had been invoked would make any hardware-acceleration useless. Furthermore, the software harness provided by the Robotics Institute turned out to be missing several key elements and extremely underdeveloped; for example, the Makefile

had no specific rules for building the project and I had to do some research to redefine the Makefile to correctly link and build the entire source. Once the project had been built and run, I discovered that several script utilities were missing which had to be brought into the codebase. These scripts were not available in the repository provided by the RI liaison and I ended up having to communicate with three separate graduate students (one who was in another country and one who was at another college) as well as our RI liaison to try to get the scripts together which I was ultimately unsuccessful in doing. My next step was to scrap the software part altogether and utilize the HDL source (written in VHDL) provided by one of the graduate students -- I made the assumption that the IP was functionally correct *and began making modifications within our ecosystem to integrate the logic block which included removing the block rams from within the correlation IP, changing everything to streaming, synchronous input and output as well as writing all the logic to feed data from the internal block RAMs to the correlation logic.* Ultimately, my teammates told me that they had tried the testbench provided for the correlation block and not only did the block not do what it was supposed to, the testbench had not been written for that block as the ports did not match.

Ultimately, I ended up helping Alex write the Instruction R/F from creating the block RAMs to the logic for loading data to the SED block as well as making modifications to the SED block to help it integrate better into our system. This included writing all the logic for the clock manipulation, FSMs for reading and writing the internal block RAMs and control logic for invoking and storing data to and from the SED block. I also wrote the demo program which demonstrated the functionality of our system as well as a generic application which could be run using our framework. This included writing the interface to the webcam driver, generating

images, writing the software to convert the image format into usable pixel data, transporting the pixel data to the fpga and reconstructing the image from the FPGA output.

Words of Wisdom

The biggest piece of advice I would give after my personal experience with the course is to be extremely weary of source from sources external to yourself – if you do you code from sources other than yourself, make verifying them your top priority before moving forward with your project – if I had made the decision to not assume that it had been simulated and verified, then we would have found our problem a lot earlier in the semester. Furthermore, if you get into trouble, let your teammates know – my teammates' biggest criticism of me was that I didn't let them know how much trouble the correlation side of the equation was in because I stubbornly believed that I would be able to figure it out.

Class Impressions

Coming in, I was very excited to take the class but ultimately I was not able to give it the time it required which was fully my fault. I would recommend that people really do consider this class one of their top priorities in order to have a successful project.

Acknowledgments

We would like to first and foremost acknowledge and thank Professor Bill Nace and our class TA Lincoln Roop for their help and guidance during the semester as well as being understanding of our shifting deliverables. We would also like to acknowledge the help of Kevin Peterson and Professors Don Thomas and James Hoe of guiding our project in the initial stages.