WMX SSD Final Report

Myyk Seok
Xiao Lin
Will Constable

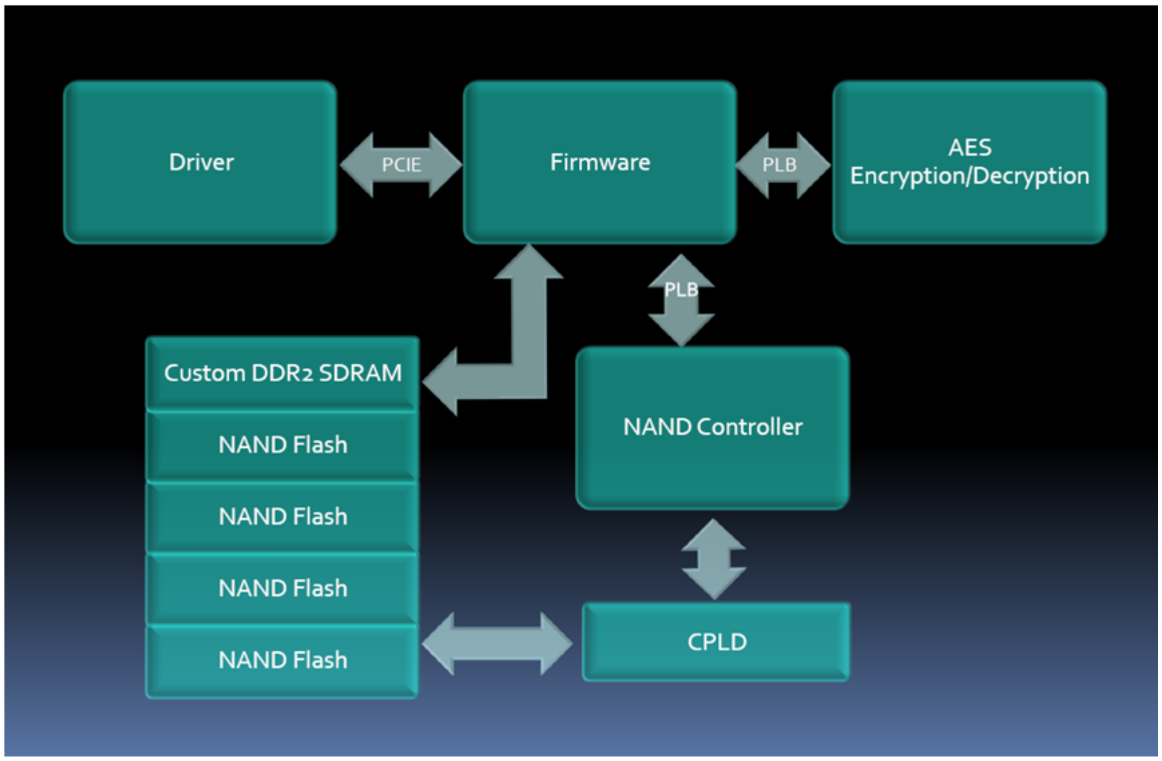
Table of Contents

Project Description

Solid State Disk (SSD) is a new technology that uses nonvolatile memory (i.e. NAND Flash) instead of traditional magnetic media to store data for a computer system. Commercial SSD products attempt to abstract away the solutions for these fundamental limitations in NAND technology using a flash translation layer (FTL), which is generally a closed, proprietary solution.

Our SSD project involves building a custom secure solid-state drive. SSD provides several advantages over HDD, including faster read and write performances, lower power, and no moving parts. In general, SSD controller comprises of custom flash DIMM, DRAM cache, and flash controller. The desired functionality of the SSD controller includes build-in PCIE interface in FPGA, 128-bit AES encryption and decryption. The end product would potentially be deployable in real systems via PCIE interface.

SSD will be implemented on LX110T XUP board with MicroBlaze. Our general design is as follows:



Major components include PCIE, DIMM, MTD Driver, NAND Controller, DDR2 SDRAM Controller, and AES Encryption/Decryption. The PCIE will be built upon existing code provided to us by Eric Chung. A MTD driver on the PC side will be implemented. For the DIMM, we have decided on four 2 GB NAND Flash chip part MT29F16G08DAAWP-ET, giving us a total of 8 GB of storage on NAND Flash. Our group implemented the NAND Controller for the NAND Flash in hardware. In addition, we are using MT47H64M8 for DDR2 SDRAM and the Multi-port Memory Controller (MPMC) controls the DDR2 SDRAM. We have updated AES source code from opencore.org to meet our design needs. The key for AES will be loaded from CF card at device initialization.

Hardware / Software Detailed Overview

We will first present the overall project layout, and the hardware / software integration in some detail, and examine the individual pieces in more detail afterwards.

On the hardware side, we have a custom DIMM module fabricated in real hardware, with 4 NAND chips, 1 CPLD for level shifting, and 1 DDR2 SDRAM chip. In FPGA synthesized hardware, we have a NAND controller, DDR controller, AES encryption and decryption cores, microblaze processor, and PCI Express endpoint (interfacing to real PCI express hardware on FPGA), CF-card reader, RS-232 communication.

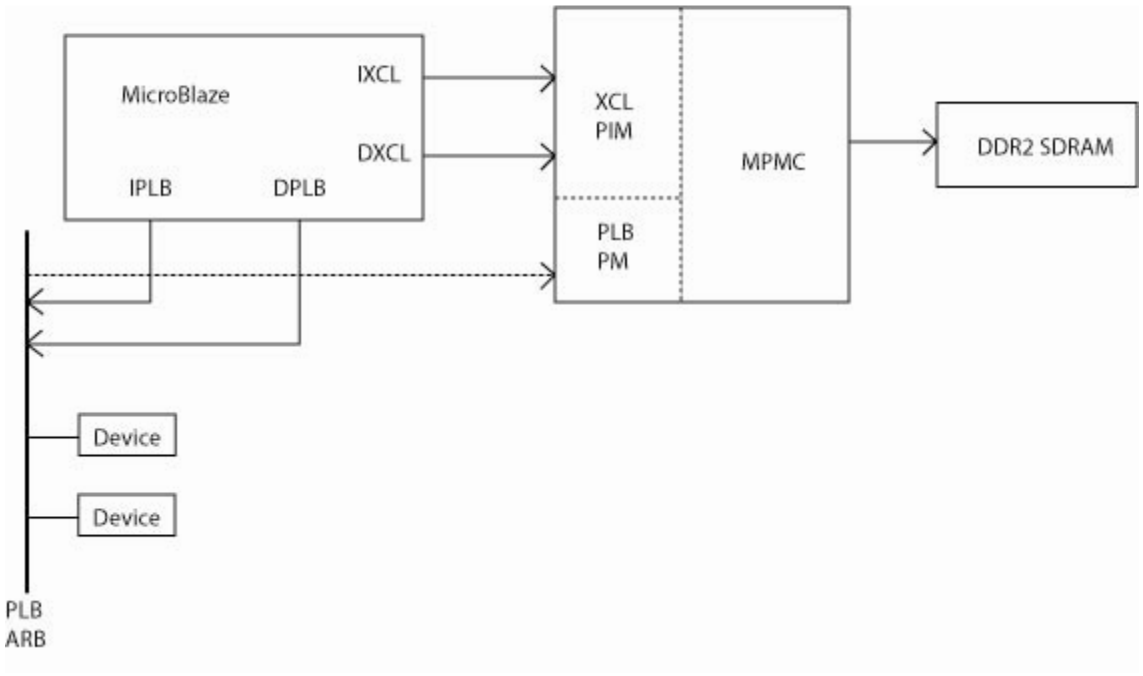
On the software side, we have firmware running on the Microblaze, which governs the data transactions between the hardware blocks. Specifically, it interprets commands and handles data to and from the host PC using the PCI express link, and completes necessary transactions with the hardware modules such as encryption and storage. On the host PC, which must run a linux 2.6.31 or newer kernel with MTD compiled in, we have a custom MTD device driver that communicates over PCI express with the Microblaze firmware, and sends read, write, and erase commands to our device, returning the appropriate information through the MTD interface to the user, using the JFFS2 file system.

Once assembled, our system is capable of presenting a working drive interface to the linux user. The user can first copy a JFFS2 filesystem image to our device, and then mount the drive just like any disk on linux. At this point, all the I/O to the mount point is directed over PCI express to the microblaze, which processes the commands and moves the data through the hardware blocks.

DDR2 SDRAM Controller

DDR2 SDRAM is a double data rate synchronous dynamic random access memory interface. It transfers data on the rising and falling edges of the bus clock signal. In DDR2 SDRAM, the bus is clocked at twice the speed of the memory clock. This leads to a total of four data transfer per memory clock cycle.

The specific DDR2 SDRAM part for our Solid State Drive is MT47H64M8. We want to have a DDR2 SDRAM controller for DDR2 SDRAM and directly connect it to Microblaze. Xilinx provides Multi Port Memory Port Controller (MPMC) in XPS. MPMC is a fully parameterizable memory controller that supports DDR2 SDRAM. Below is an overall diagram between MPMC and Microblaze. The MPMC provides direct memory access to the processor IXCL and DXCL interfaces. Xilinx CacheLink (XCL) PIM translates XCL commands to NPI transactions to perform reads and writes to memory. XCL uses FSL bus interface protocol with FIFO interface. The MircroBlaze IXCL and DXCL ports connect to one MPMC port for two XCL connections. In addition, the PLB PIM connects the PLB bus to the MPMC.



The MPMC Physical Interface (PHY) is the interface between memory and the MPMC address path, control path, and data path. We are using Memory Interface Generator (MIG) based PHY interface from Core Generator.

We need to configure the MPMC for DDR2 SDRAM part MT47H64M8. MIG customizes PHY for our specific part and generates MIG UCF file. A script converts MIG UCF file to MPMC UCF file with signals names corresponding to our design. We can then update system.ucf from our design with MPMC UCF. We then configure the correct memory part and memory configuration in XPS using MPMC Gui. Ideally, now MPMC would be configured for our specific DDR SDRAM part.

Configuration on the mpmc is as follows:

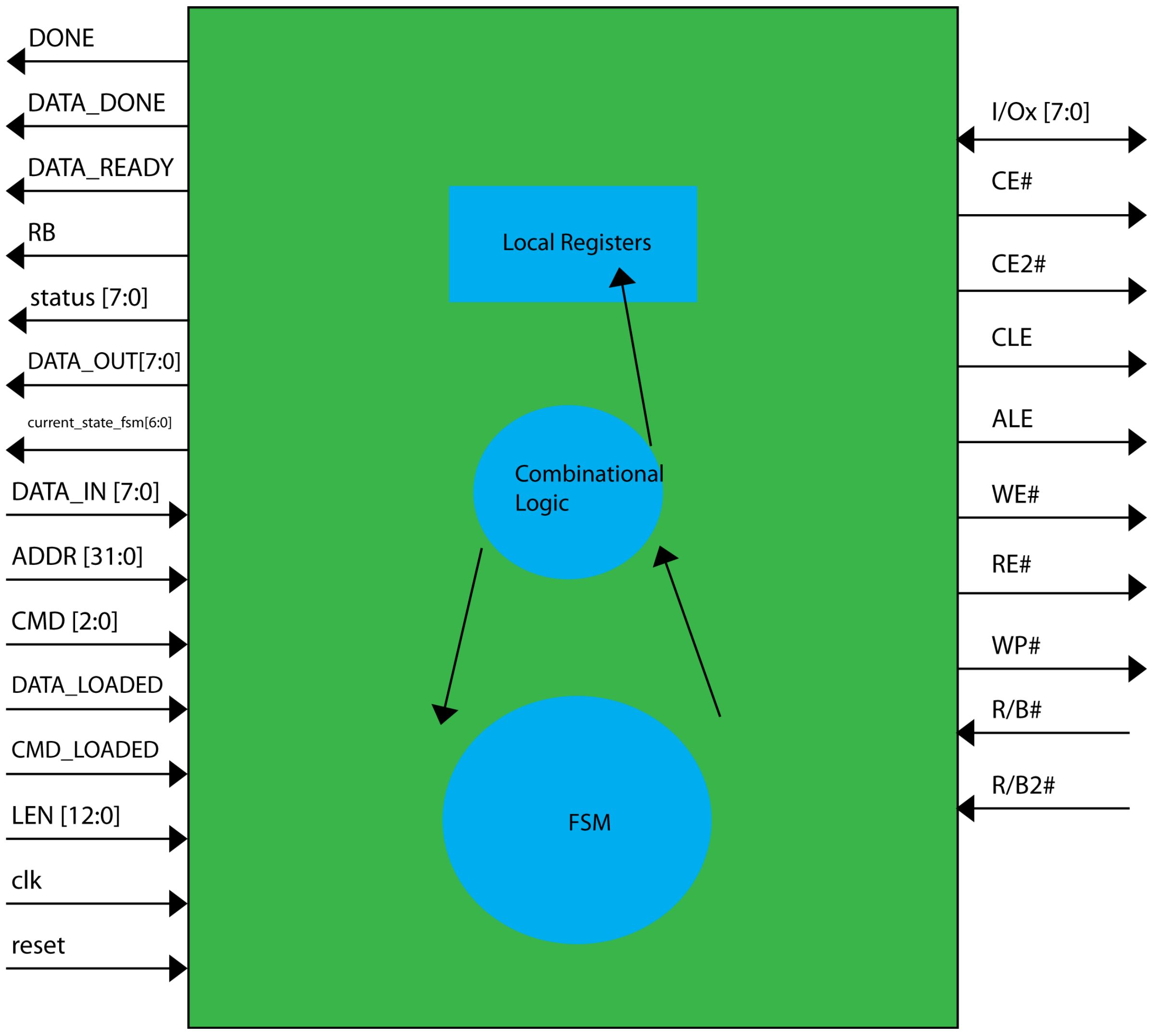
- **Part No:** MT47H32M8-37E
- **Memory Data Width:** 8
- **Native Data Width of PLB:** 32
- **Number of IDELAYCTRL Elements:** 1
- **IDELAYCTRL Constraints Locations:** IDELAYCTRL_X06

We used TestApp_Memory as a starting project to test the DDR2 SDRAM. We wrote more verification test on writing data to SDRAM and reading back data from SDRAM. The tests passed for all 4 bytes aligned addresses.

NAND Controller

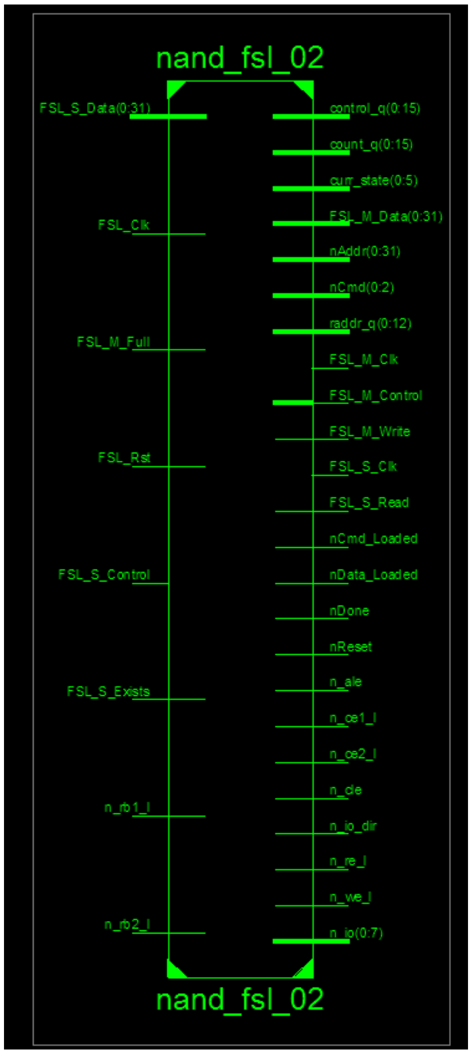
For each of our four 2 GB NAND Flash chip (part number: MT29F16G08DAAWP-ET), we needed a flash controller so that our firmware could easily interface with it and meet all necessary hardware timing constraints.

This NAND controller is similar in design to the one implemented by Microsoft Research (FPR.pdf). The NAND controller implements a subset of the standard flash commands (read, program, erase, etc.). We designed this NAND controller as a finite state machine (FSM), but the flash command sequences could also have been implemented using a simple microcontroller and associated program, as described by the Microsoft Research document. We chose not to go that route because we think it would be more complex than we had time for, while not giving us any enough benefits to be worth the time.



Our NAND controller bible was the document “Micron - 8gb_nand_slc.pdf”. The approach was to read through this end to end. This also helped for informing out decisions on how we would have used FTL and why we chose not to go that route. Since we had a good idea of what the FSM would look like, we were able to start out by writing a skeleton for it in Verilog. The skeleton defined inputs, outputs and an empty always block. Next step was to write the states allowing for a hard reset, into the state that accepts command, and enters into that command’s states. Then, we wrote all the states to allow for the READ STATUS command. The first portion of this command was the Command Latch sequence that was reused in each command. We wrote a test bench that hooked our controller to a model nand chip provided by Micron. By talking to the model we could find out if our timing constraints were correct; it would tell us if we didn’t wait too long. Once we had READ STATUS debugged, we wrote PROGRAM PAGE and soon found out we had to issue RESET before anything was guaranteed to work, so instead of waiting on the other team to debug their RESET, we wrote this too. They wrote READ PAGE, but we ended up reorganizing that to fit the format of the rest of the design to better share logic. We developed the NAND controller on Linux with the help of KDE’s Kate editor and simulation through ModelSim.

We had trouble getting everything to synthesize in ISE. We learned that to synthesize, we could only ever modify a signal in only one always block. That issue was easily conquered. Then we realize we were getting a lot of inferred latches from our FSM. So we decided it would be best to eliminate as many of them as we could. We did so by moving as many signals as made sense, out of the FSM’s always block, and into assign statements. And then, we made the rest of the signals fully specified in each state. We got rid of all inferred latches in our implementation in the end.



Next, we were ready for building the NAND peripheral. Since we had made most of the NAND Controller, and we were sharing this component with the other team. We decided to let them drive development on the peripheral. The peripheral communicates with the microblaze over PLB and FSL. The FSL component forced us to need a peripheral FSM that synchronized communication with our NAND controller. This component was driven to code completeness but was not debugged even with our joint effort.

CPLD

A complex programmable logic device (CPLD) is a programmable logic device. The building block of a CPLD is the macro cell, which contains logic implementing disjunctive normal form expressions and more specialized logic operations.

The NAND Flash operates on 3.3 V while the DIMM operates on 1.8V. The function of CPLD is for voltage translation between 3.3V NAND Flash and 1.8V DIMM. Our CPLD component is XC2C256 CoolRunner-II, which include two I/O banks that permit easy interfacing to 3.3V and 1.8V. We implemented the CPLD logic with tri state buffers and ucf file in ISE. We downloaded CPLD logic and ucf file to the CPLD hardware through JTAG chain with IMPACT software.

The download of CPLD code was successful. However verification of CPLD would be hard without the NAND controller. As the result, we did not fully tested CPLD.

Documentation on XC2C256 CoolRunner-II can be found at:
http://www.xilinx.com/support/documentation/data_sheets/ds094.pdf

Firmware

The purpose of our firmware is to route data between PCIE, AES, DDR2 SDRAM, and NAND Controller. Our firmware is implemented in Microblaze. We managed to implement the firmware to route data between PCIE, AES, and DDR2 SDRAM.

Firmware waits for a synchronization byte from the PCIE. Once the synchronization byte is received, the firmware enters a while loop that run constantly until the Microblaze is stopped. In the while loop, the firmware gets the header from the PCIE. The header includes COMMAND, OFFSET, and SIZE. Data is then sent across the PCIE from the PC and the firmware processes the data according to the COMMAND. The data is always sent in 4KB blocks regardless of the actual size. The erase block for DDR2 SDRAM is 2KB.

Commands that are supported include READ, WRITE, and ERASE. The ideal implementation of the firmware is as follows:

ERASE: Calculate the address of DDR2 SDRAM according the OFFSET. Use memset to set 2KB of memory to 0xff starting at the calculated address. Encrypt the erased blocks.

WRITE: Get data from PCIE. Calculate the address of DDR2 SDRAM according the OFFSET. Encrypt the data using AES with the actual size of the data. Write data to DDR2 SDRAM according to the address.

READ: Calculate the address of DDR2 SDRAM according the OFFSET. Write data to DDR2 SDRAM according to the address. Decrypt the data using AES with the actual size of the data. Send data across PCIE.

Testing and verification of firmware involves reading and writing data from the PC and verifying the data. For the most part, print statements were used extensively in our debugging. We first made sure the MTD driver was sending the correct data. We then made sure the firmware receives and processes the correct data by testing with Eric's original driver code. After our verification of each individual part, we then proceed to the integration testing. Our integration testing mostly dealt with in-depth print statements. We debugged the bugs according to the print statements.

AES

We used a 128-bit AES algorithm, with encryption and decryption cores implemented in Verilog, published open source on opencores.org. The encryption and decryption cores are separate entities in our project, each built into a PLB-enabled wrapper, allowing communication between the cores and the Microblaze firmware.

The encryption hardware takes a 128-bit key and a 128-bit plaintext block, and after a start signal, takes 12 clock cycles before outputting a done signal and the 128-bit cipher text block. The FSM control for encryption is thus quite simple, just 3 states- one for waiting on a an encrypt command from the host, one to signal the start of encryption, and one to wait for encryption to finish and return to start.

The Microblaze firmware is in control of the AES hardware, by pushing command bits and 32 bit data words across the PLB into PLB registers in the Encryption peripheral. Since it takes multiple cycles to transfer 2 128-bit words across a 32-bit bus, there are control bits used by the firmware to signal to the hardware that the key and plaintext have been transferred, and used by the hardware to signal the firmware that an encryption process has completed and the ciphertext are ready to be read from the PLB registers.

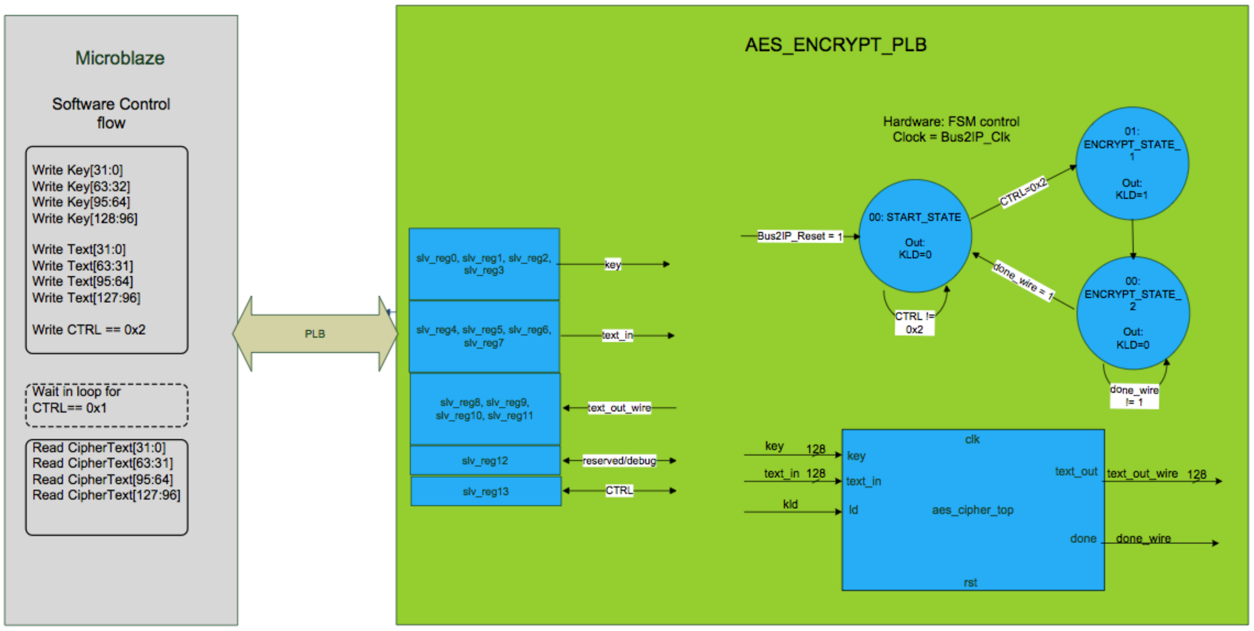


Figure 1: AES Encryption Block Diagram

Decryption is more complex. The key needs to be loaded before the ciphertext, but the timing must also line up so that the entire process takes 24 cycles, with 12 cycles inbetween the beginning of key load and the beginning of ciphertext load. It is theoretically possible to do subsequent decryptions using the same key in only 12 cycles, but this requires the same level of precise synchronization between one ciphertext block and the next, and that would be difficult to achieve over the PLB.

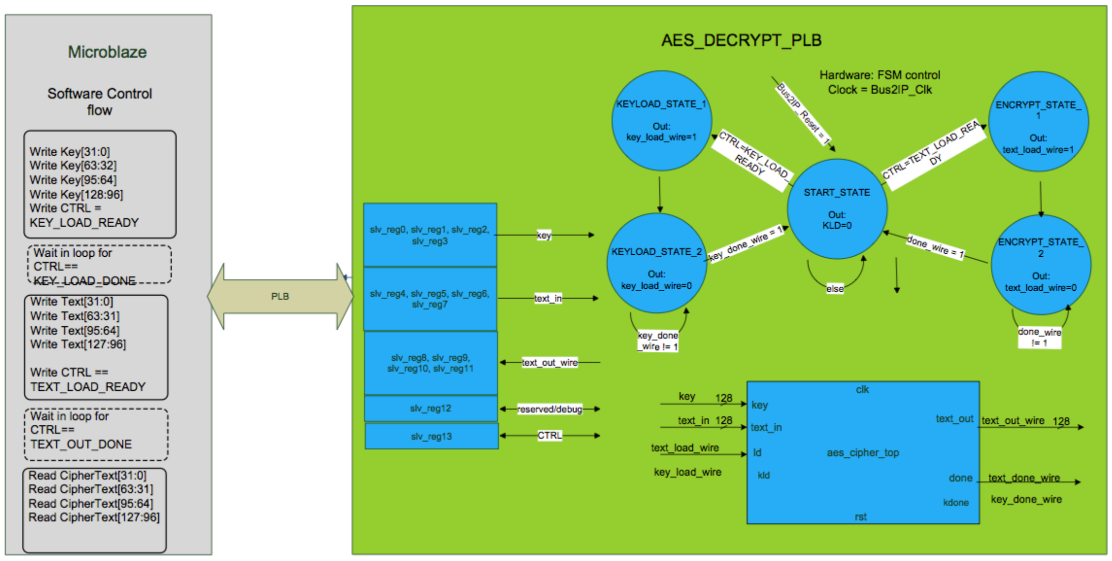


Figure 2: AES decryption block diagram

DIMM

Background / need

We wanted to use NAND Flash as our storage medium, and we needed a way to interface several NAND chips to our FPGA. The DIMM slot and the FPGA expansion headers were two obvious possibilities for connecting a homebuilt module, and the CF card slot was backup possibility. The DIMM slot offered far more pins than the others, allowing for more NAND chips, meaning both a greater capacity and theoretical max read/write speed.

Architecture (bus width, etc), Component choice

The DIMM slot was unfortunately powered at 1.8V for DDR, and most available NAND chips were 3.3 V instead. Once we ruled out 1.8V NAND due to availability, capacity, and price, we chose a Micron 3.3V NAND part and focused on finding the best way to do 1.8V to 3.3V level shifting. Bus level shifters tended to come in smaller packages (requiring more chips on our DIMM to reach the required pincount), so we chose a Xilinx Coolrunner II CPLD, offering high IO pin density at a low price and low power consumption. We also wanted to provide DDR memory for the system to use as cache, since we were evicting the standard DDR2 DIMM normally present in the FPGA. We picked a single 32MB Micron DDR2 chip, which runs at 1.8V and can interface directly to the DIMM pins.

Excel Schematic (clippings)

The first step towards creating a real schematic for this DIMM was to plan the wiring using and Excel sheet (a clipping of it is pictured below). From this, we intended to enter the schematic into Cadence tools ourselves, but in the end, struggled to get Cadence tools to run on the CMU network. We produced a full featured schematic in excel, depicting all nets, all power connectors, jumpers, decoupling caps, and interconnections.

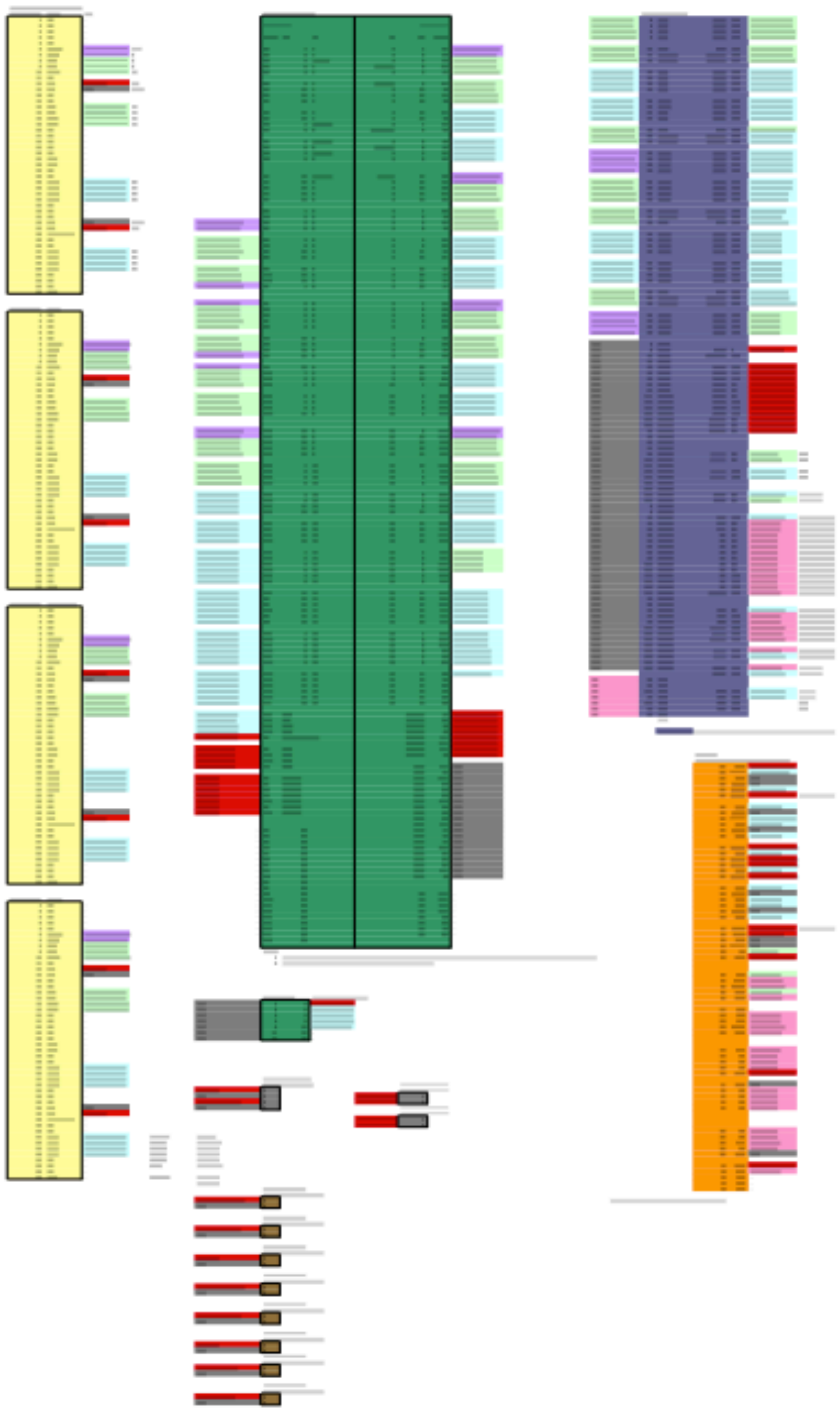


Figure 3 Excel Schematic for our DIMM module. See DIMM_pinout+BOM_Final(11-9-2009).xls

Real Schematic / Layout (sample)

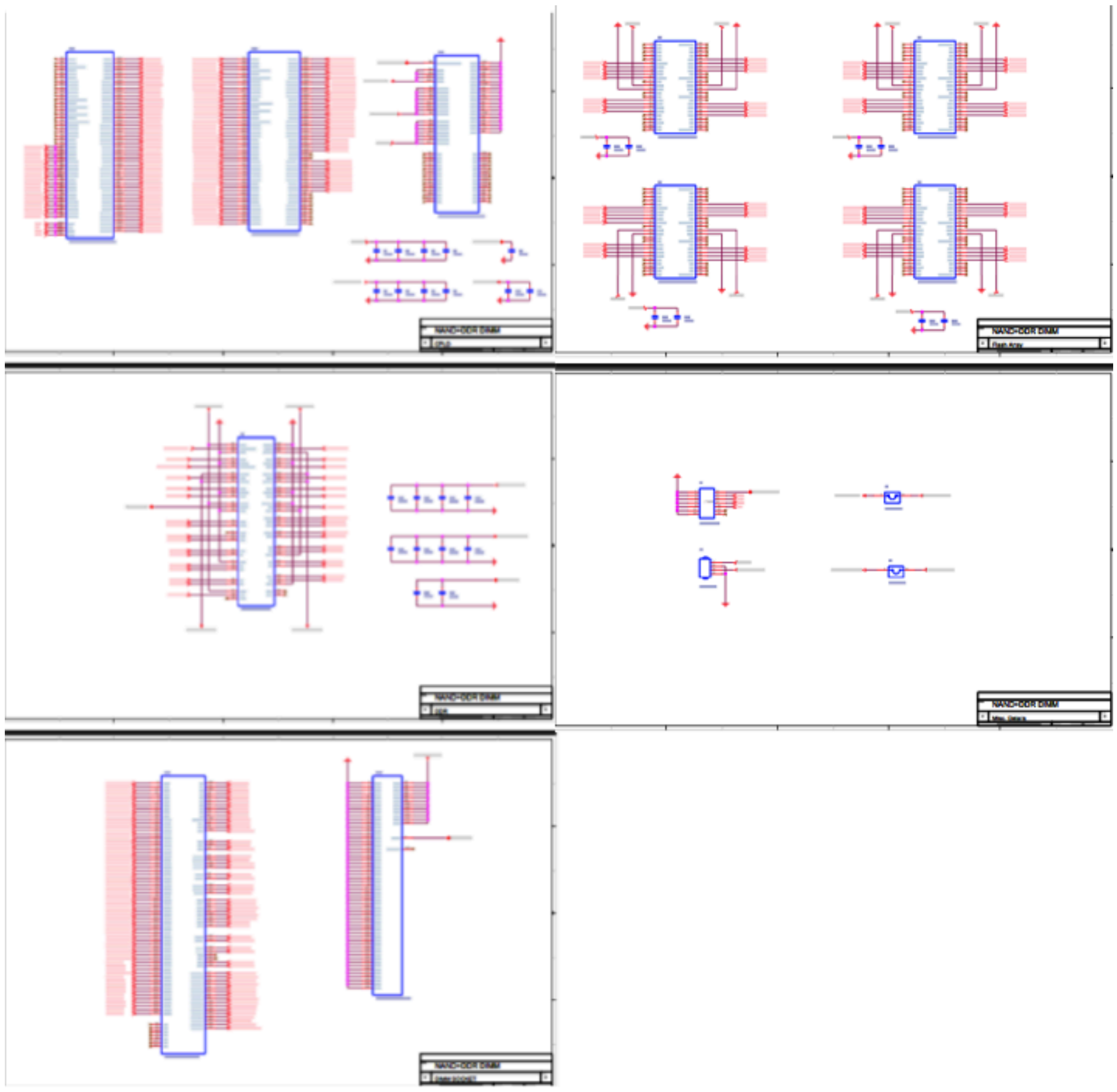


Figure 4 Final DIMM Schematic from CIREXX. See official schematic NAND_DIMM_REVA.pdf in our project directory for closeup

The official schematic was done by CIREXX contractors, with our constant input and feedback. We reviewed and approved the schematic, and then CIREXX produced the layout files, also interactively with us, adhering to the physical constraints of the DIMM socket and the FPGA board. Finally, CIREXX printed PCBs and populated them with components.

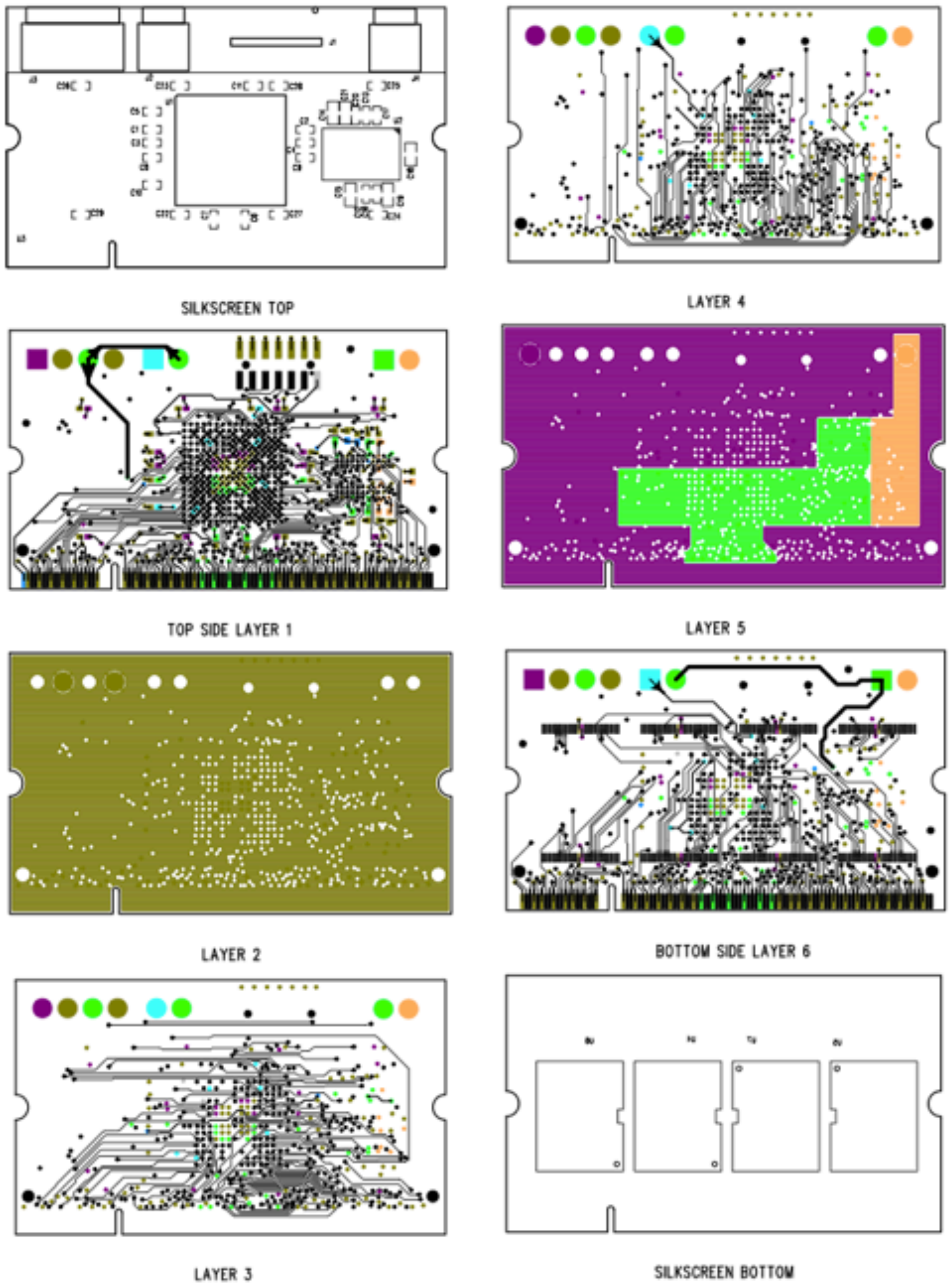


Figure 5 CIREXX Final DIMM layout files, showing 6 PCB layers and 2 Silkscreen layers. See "83232-A CHECK FILES" folder

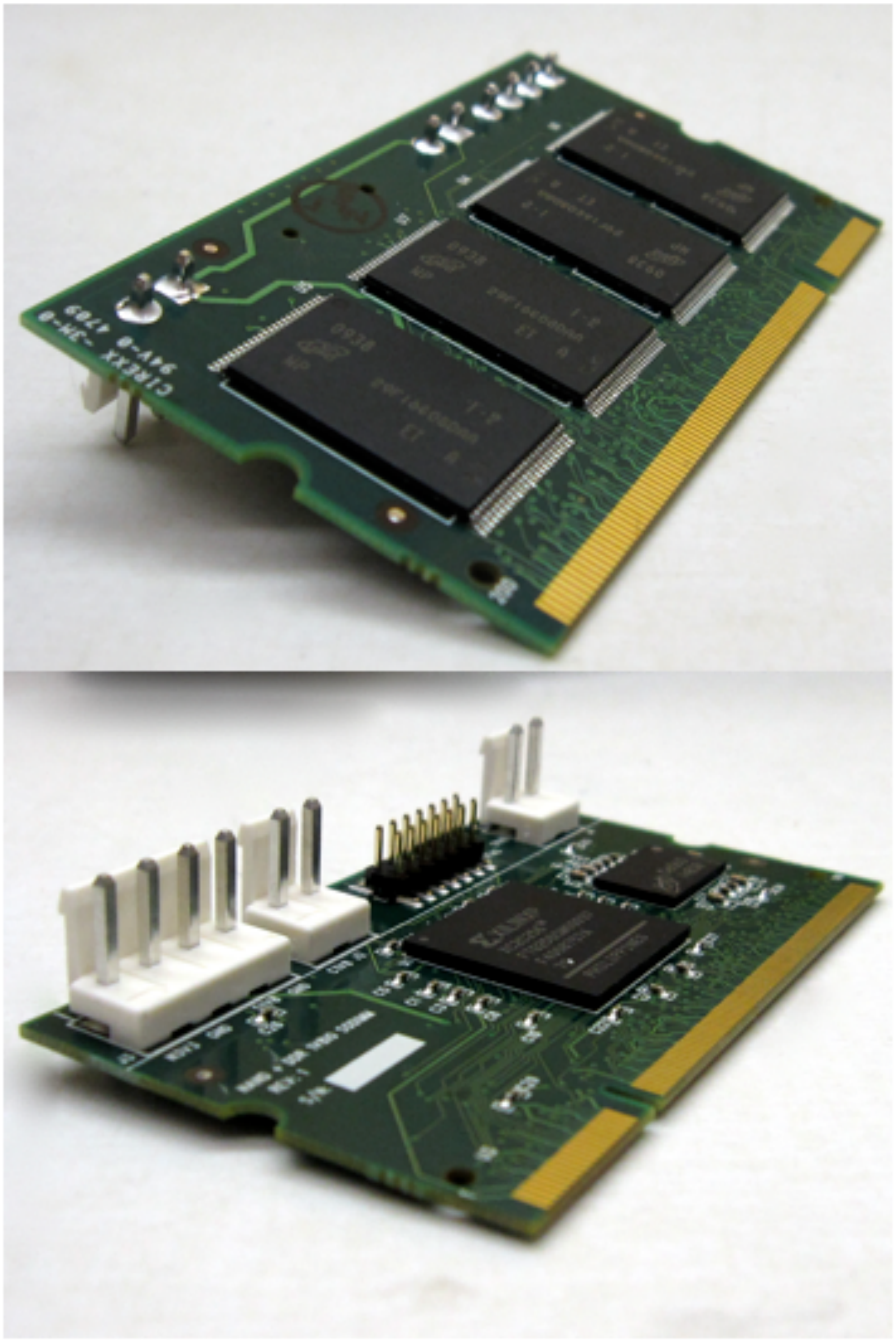


Figure 6: Photos of both sides of our finished DIMM

Building and Running Our Project

First, extract our project archive wmxssd.zip. You will need to take portions of this archive to the linux host computer, and portions to the Xilinx synthesis computer.

Linux Side – initial Preparation
First, copy the linux_dev folder onto the linux machine. You need to set up the proper linux kernel (2.6.31.5 is what we used, newer ones may work) and also set up the MTD source tree to compile into the kernel. Follow our instructions, posted on our wiki page for this:

http://wmxssd.wikispaces.com/MTD+driver

Hardware Synthesis
Open the Xilinx XPS project in the integration_proj folder, and synthesize the hardware. Make sure our DIMM is inserted in the Virtex5-LX board you are about to program, and the DIMM power is connected properly. Download the bitstream to the board. Launch XMD, stop the microblaze, download executable.elf, and wait until after you power off the linux host PC before you start execution of the microblaze firmware.

Boot it up
By now, the linux (host) PC should be setup as per the instructions above. Power the host PC down, insert the PCIe card into the PCIe slot (if you didn't already), start the microblaze firmware, and then Power on the host PC. Upon boot, you can immediately load the appropriate kernel modules and start using the drive, by following these instructions:

```
modprobe –a mtdblock mtdchar mtd wmxssd_mod jffs2
cp linux_dev/wmxssd/willbaby.jffs2.image /dev/mtd0
mount –t jffs2 /dev/mtdblock0 /<mountpoint of your choice>
```

Now you can use the mounted drive to read / write / erase files!

Tools and design methodology

As part of our project management, we have both SVN and wikispace setup for SSD. Our wikispace was made mostly for internal use. Our wmxssd wikispace is located at http://wmxssd.wikispaces.com/

Our wikispace includes access to our svn directory, directions and commands for MTD driver, and setup for SysAce.

Our team has been using SVN extensively in managing our XPS projects, source code, diagrams, and documents. In addition, we setup VNC such that each of our team members can work from home.

Most of our project was done in Xilinx 11 XPS. However we also used ModelSim in simulating NAND Controller and ISE in synthesizing NAND Controller. Kate was used in writing NAND Controller. ISE was also used in CPLD coding.

Status and future work

Features Supported

- Custom 32MB DDR2 DRAM as ramdisk storage media on FPGA
- PCIE provides communication between PC and FPGA
- WMXSSD mounts to a directory and works as a normal disk in Linux

- AES provides 128-bit data encryption/decryption, but the glue to encrypt/decrypt blocks of file data has a bug which makes the mounted filesystem unreliable, so we disabled this feature in our final demo

Xiao and Will plan to continue working on this project next semester for Ken Mai, and completing the NAND controller in order to achieve a totally functional SSD

Lessons Learned

Will

Involve teammates on your ‘parts’ to improve arch. decisions and speed up debugging; counterintuitive, seems like it would be a better idea to totally split off and work separately in parallel.

Planning more thoroughly (breaking into smaller milestones) seems difficult (tempting to say ‘impossible’) at the beginning, encouraging you to proceed with a vague plan. But without these micro-milestones, its really tough to get a realistic plan formed, but it takes quite a bit of work to get those small details planned out, and its even tougher when not everyone on the team is dedicated to achieving that goal.

Building hardware to interface with the FPGA isn’t necessarily a bad idea for this course, although it comes close. With any more schedule slip, we could have ended up passing the final deadline with no hardware in hand. We managed to hit some marks, in that we use parts of the hardware in the final demo, which I consider a significant success. Starting early and aiming for hardware complete in the first 1/3 of project was crucial for the level of success we achieved.

We made some good calls along the line, to pull the plug on certain ideas and move with something else. One was SATA - discovering that it presented a roadblock and changing to PCIe early on was a good move. Abandoning Cadence tools and getting a contractor involved was the right choice for the DIMM production, but we took too long to do that. That burned more than a month and takes sole responsibility for the majority of DIMM schedule slip,

Xiao

Wisdom Gained

- Knowledge of Solid State Drive
- In-depth understanding of peripherals, structures and files associated with Xilinx XPS project
- DDR2 SDRAM, NAND Flash, PCIE, MTD, CPLD

Knowledge I wish I possessed at the beginning of the project

- Implementation of SATA is too complicated
- Ins and outs of Xilinx 11 and Virtex-5 XUP

Good/Bad decision

- Bad decision to try to implement SATA

Words of wisdom for future generations

- Knowledge of Virtex boards and Xilinx XPS before class begins
- Do research on the project before class begins

Individual Pages – Myyk

I (Myyk) started out by knocking out some of the labs for the team. Then I moved on to AES. I didn't realize that we were using Xilinx incorrectly, and my peripheral wasn't actually rebuilding. Then I let others handle AES while I really dove into designing the overall system. I started out by trying to understand PCIe. Then I read as much as I could about our NAND chip and Flash Transition Layers. Will helped in the areas he knew better. Together we came up with plans for how the flash transition layer would do wear leveling, bad block managing, aes, ecc, ddr2, and communicate to the computer through PCIe. I also designed the NAND flash controller.

Through the process of designing the flash transition layer, a new option emerged. I found we could develop an MTD interface to our NAND device to handle the flash transition layer, thus reducing the total firmware we would have. We pitched the idea to Ken Mai and he gave us the go. The NAND controller we had from Micron was not going to work for our model. So instead of hitting it with a hammer, I started a new one from scratch. Tao, from the other team, had started working on a way to communicate to the NAND controller and we were going to work on it together. But I ended up starting the project, implementing all but one of the commands, debugging, and making it synthesizable mostly on my own. This consumed many hours of my life. Days were blurring together as I cranked this out, since this was of high importance to both teams. And I was done by the date we had set for this task.

The NAND controller peripheral was a joint effort of my team and the other team. I would help them through emails. And I helped them try to debug NAND controller in the last week. When they weren't around, I was helping Will, and together we got the MTD driver with the firmware to use the device as a RAM disk. I then wrote another MTD driver that would work as a NAND device disk, but we unfortunately never got to try it out. I don't know if I could quantify accurately the amount of time I put into this class, but I somehow managed to keep giving it more each week.

I liked the course. I would have liked to have had much better preparation to use Xilinx. I would have liked to have been able to only use ISE. I still don't know much about the other Xilinx software available.

Individual Pages – Xiao

Accomplishments

Virtex-5

- Tinkered with Virtex-5 to work with Xilinx 11 (~ 3 weeks)

Firmware: routes data between DDR2 SDRAM, AES, and PCIE.

- Wrote, debug the firmware to test between AES Encryption/Decryption and the original PCIE driver on the PC's side. The communication between firmware, PCIE driver and AES works perfectly when the input is multiples of 16 bytes. (~1 week)
- Debugged with Will and Myyk on the communication between firmware and MTD driver. Works perfectly for the demo (~60 hours).
- Debugged with Will and Myyk on the communication between firmware, MTD Driver and AES. Since the size of data coming into firmware is not always multiples of 16 bytes, we had to rewrite and debug the original code to integrate the size issue. Almost working (~12 hours).

DDR2 SDRAM Controller

- Updated the UCF file using CoreGen, configured and debugged mpmc for our DDR2 SDRAM component (~30 hours).
- Configured, integrated and debugged Eric's PCIE project to work with our DDR2 SDRAM component (~4 hours).

AES Encryption/Decryption:

- Debugged AES encryption and decryption from Will's code with Chipscope. AES works perfectly (~30 hours).

CPLD

- Wrote CPLD code for our CPLD component (~5 hours).
- Wrote the UCF for our CPLD component (~3 hours).

NAND UCF File

- Wrote the UCF for NAND Controller (~2 hours)

SATA

- Read about SATA and synthesized some source code. However SATA was never used in our project in the end. (~ 5 weeks)

General Comments

In general, the class was pretty interesting and challenging. If you want us to really read the debugging book, make it worth some points. The Pentium Chronicle didn't really help that much in terms of time management. I would say the professor should instill more fear and stress upon us. Probably more in-class demos than presentations would be more helpful in getting work out of us.

Individual Pages - Will

What I did

I took much of the responsibility for the DIMM- choosing parts, creating the excel schematic, wrestling with the two semi-functioning versions of Xilinx we have access too, acting as the liaison to CIREXX (contractors) researched decoupling, layout issues, guided the contractor to complete the schematic and layout. The entire process from start to finish lasted about 12 weeks. I would estimate an average of 10 hours per week dedicated to DIMM over the course of that time. (i.e. 120 hours)

For AES, I started with verilog code from opencores and wrapped it in a PLB-connected xilinx peripheral. I wrote the FSM in the peripheral responsible for keeping hardware signals to AES in sync with the commands and data coming over the PLB from Microblaze. I wrote Microblaze firmware to communicate with the FSM-glued AES peripheral, and wrote a thorough and extensible testbench to verify its accuracy against an array of input vectors. Then, I added CF and FAT support to the project, and modified my tests to read the keys off of CF, but the rest of the vector locally. I worked on AES for about 4 or 5 weeks, about 10 hours per week. (i.e. 50 hours)

For DDR, I debugged the coregen implementation Xiao created, and got it (the DDR on our DIMM) up and running. I wrote a testbench (in microblaze firmware) to test the full range of addresses, as well as some strange access patterns. I spent about 10 hours on this.

For the linux side, I downloaded and attempted to understand the poorly documented MTD code. I built a kernel, booted it, attached MTD and rebuilt, rebooted and got an MTD driver for a ramdisk on the host side to mount as a JFFS2 disk. (days of grief abbreviated by commas) I then copied the ramdisk driver, took out the guts and rerouted function pointers to my own code. I wrote an FSM style loop in the Microblaze firmware to handle PCIe transactions, and created a common ‘PCIe header‘ to precede each transaction and define commands and parameters. I built Eric Chung’s PCIe driver into the same kernel module, and tunneled the necessary read, write, erase, etc. commands through PCIe to the microblaze. To understand and build the MTD code and the linux kernel took me about 30 hours. Cloning the ramdisk driver and modifying it to work through PCIE with our project took about 50 hours, many of which fell on thanksgiving break :(

For a stab at integration, I built a project with PCIe, DDR2 and AES hardware. I first tried to get a JFFS2 filesystem to mount from the linux side, using our DDR as a storage media.

Course Suggestions

It might be helpful for each team to have a one on one with the professor at the start and really delve into the expected difficulty of all the parts of the proposed design, and the expected time commitment necessary to get it done.

Resources

http://www.xilinx.com/support/documentation/data_sheets/ds094.pdf

http://www.opencores.org/

http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf

http://www.ece.cmu.edu/~protoflex/doku.php?id=documentation:userguide

Micron - 8gb_nand_slc.pdf

FRP.pdf