18-545: Final Report – Solid State Drive

Winston Wan, Tao Yang, and Timothy Tan
Carnegie Mellon University
{wwan1, taoy, tgt}@andrew.cmu.edu

December 9, 2009

Contents

List of Figures

List of Tables

1 Introduction

This final report covers the progress that was made on the FPGA-based Solid State Drive (SSD) project during the Fall 2009 semester. We outline the details of our system, document the decisions that have been made, provide directions understanding and using the project, and give insights to future developments that may be made in continuation.

2 Purpose and Goals

The purpose of this project is to create a flexible, extensible platform by adapting an FPGA as a SSD controller. We hope that the system will facilitate rapid SSD controller research and testing in the future.

The initial primary goal for the end of this semester was to be able to connect the FPGA to a host computer via Serial ATA, mount the SSD on a computer and verify reads and writes to the drive. Midway through the project, due to time constraints, the implementation of the Serial ATA interface was dropped in favor of one using the PCI Express interface and a custom-written block device driver.

As of the end of the Fall 2009 semester, the PCI Express and block device driver host interface, the custom SODIMM, and a rudimentary FTL was complete. The NAND chip controller and FSL connection to the FTL was fully implemented and simulated, but encountered issues with deployment to real hardware. However, a demostration of the drive capabilities from the host perspective was possible by using DRAM on the custom SODIMM as temporary storage in place of the permanent Flash storage.

3 System Overview

The FPGA-based SSD can be viewed as four major subsystems:

1. the PCIe host interface with the PC and a customized block device driver

2. the Flash Translation Layer (FTL) that provides the hard disk abstraction and management of the flash NAND chips

3. the DDR2 DRAM memory chip on the custom-fabricated SODIMM and the modified controller required to access it

4. the NAND Flash storage on the custom-fabricated SODIMM and the peripheral and controller needed to access it

Figure 1 below shows a diagram of these subsystems. These four major subsystems are implemented in a combination of three methods - physical hardware, programmed hardware (FPGA) and software.

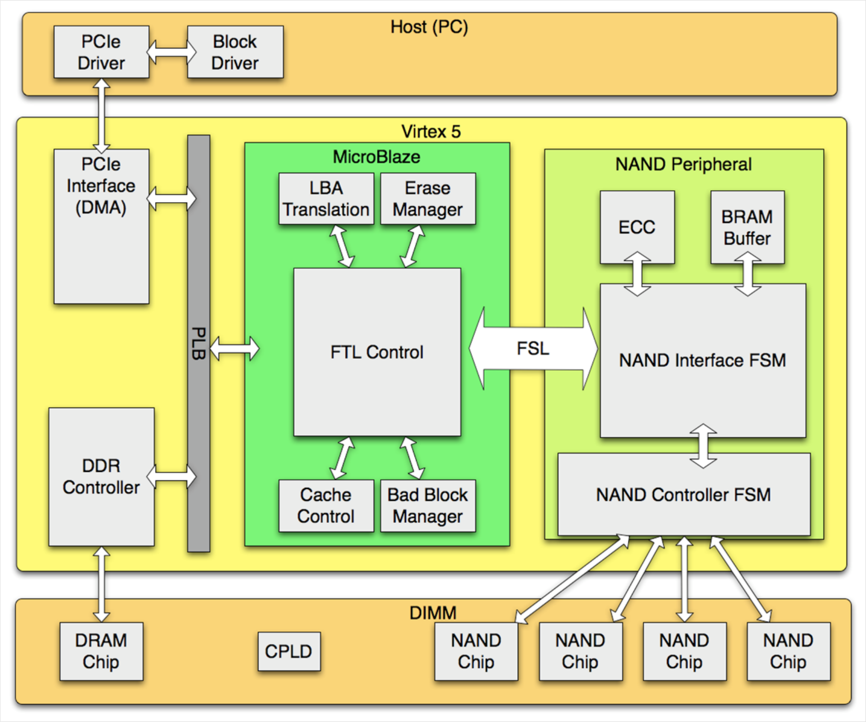


Figure 1: Top level system diagram

4 System Setup

4.1 Hardware for SSD components

- Xilinx Virtex-5 LX110T FPGA
- Gigabyte GA-G31M-ES2L motherboard
- Intel Core 2 Duo processor (LGA775)
- 2GB DDR2 SDRAM
- Custom-fabricated SODIMM (with NAND chips and DDR2 SDRAM)
- (optional) 48-pin TSOP socket for testing a single NAND chip.

4.2 Software for SSD components

- Ubuntu Linux running kernel version 2.6.28.16
- Modified PCI Express DMA kernel device driver
- Custom block device driver

4.3 Software for Virtex-5 Programming

- Windows XP (downloading a bitstream in Windows Vista/7 doesn't seem to work right)
- Xilinx ISE, EDK and XPS 11.3

Figure 2: The entire setup - 11 computers and counting

5 Custom-fabricated SODIMM

The SODIMM that was custom-fabricated for this project consisted of an array of four 16 Gb Micron NAND Flash chips for storage, one 256 Mb DDR2 DRAM chip, and a Complex Programmable Logic Device (CPLD) for voltage translation between the flash chips and the FPGA. All these elements share the 122 usable signal pins on the SODIMM.

5.1 Why use the SODIMM slot?

The SODIMM slot was chosen to be used as expansion header for our NAND and DDR2 SDRAM chips due to the large number of pins available for data and signal transfer, and because the signal routing for the SODIMM pins are known to be able to tolerate a 266 MHz clock. Unfortunately the XGI expansion headers on the LX110T have no published maximum signal speed rating, as far as we could tell. As our NAND chips can be clocked as fast as 50 MHz, signal integrity is important and we thus went with the safer option of using the SODIMM slot.

5.2 Components of the SODIMM

- NAND Flash - 4 x 16 Gbit Micron MT29F16G08DAAWP
- DDR2 SDRAM - 256 Mbit Micron MT47H32M8-37E
- CPLD - Xilinx CoolRunner-II XC2C256 (FT256)
- JTAG header - Digikey: S6009-07-ND
- 4-pin power header - Digikey: A30513-ND
- 2-pin power header - 2 x Digikey: A30511-ND
- Several sets of capacitors for decoupling, according to the Qimonda DDR2 Design Guide (refer to Appendix A.1 for more information)

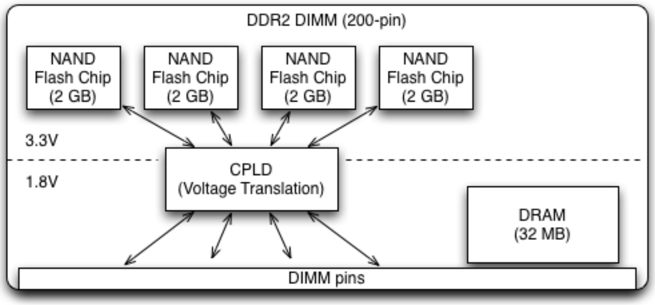


Figure 3: SODIMM components block diagram

Our design called for each NAND chip to be wired up through the CPLD to the FPGA independently (as opposed to chaining them together like a RAID-0 setup). This allows for the greatest flexibility in routing signals, at the expense of being able to squeeze one or two more chips into the custom SODIMM for a larger overall capacity (we trade more control signals for fewer data buses). Also, the DDR2 SDRAM chip had to be appropriately wired up, as there were some restrictions on the FPGA pins (and thus SODIMM pins) it can connect to.

The CPLD has two main banks - they are configured to run at 1.8V (LVCMOS18 IOSTANDARD) and 3.3V (LVCMOS33 IOSTANDARD) respectively. Signals were carefully routed to the banks running at their respective voltage levels. Also, all remaining signal pins on the SODIMM that did not have a signal assigned were still connected to the CPLD as spares.

Last but not least, as the SODIMM slot did not provide 3.3V rails, power headers were mounted on the custom SODIMM, allowing us to connect an external 3.3V source to it. Additional power headers were also provided for the various 1.8V rails, just in case the need arose.

5.3 Voltage issues

Since DDR2 specifies a voltage of 1.8V but the NAND flash chips operate at 3.3V, voltage translation is required. Instead of going with traditional voltage translation devices, a CPLD was used to perform the said translation as it was of a much more compact size and suited our space restrictions well. The DDR2 DRAM chip runs natively at 1.8V, and so does not pass through the CPLD.

5.4 Schematic, Layout and other details

The schematic, layout, pictures, and other details of the Custom-fabricated SODIMM can be found in Appendix A.

6 Host Interface

The host interface that was decided to be the most effective and simple to implement was the direct PCIe connection between the PC host and the Virtex 5 FPGA board. Since read/write commands sent to the drive must be translated to a format that can be sent and interpreted over a PCIe interface, a block device driver was developed to handle the abstraction for the operating system, a protocol for data transfer was defined for communication over PCIe, and code was adapted to facilitate the use of the PCIe port. The figure below shows the layered structure of the host interface, where the block device abstraction is built on top of the character device abstraction of the PCI device. The physical data transfer is implemented by the operating system through Direct Memory Access (DMA).

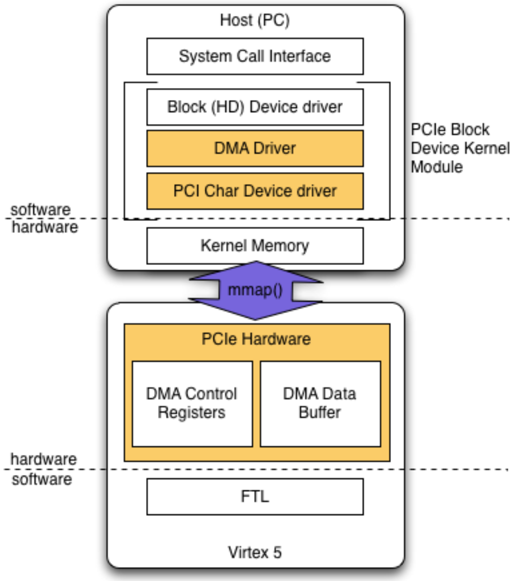


Figure 4: PCIe Interface block diagram

6.1 Host Components

The host, commonly the PC, side of the PCIe connection has all of the hardware in place, and much of the software stack is provided by the operating system. In Linux, PCI devices are character devices that stream data, whereas a disk is a randomly addressable block device. A block device driver is needed to provide a block layer on top of the existing PCI functionality. Code was adapted from a previous project by Eric Chung that implemented the PCI drivers for the Virtex 5 LX110T FPGA. These existing components are shown as the highlighted blocks in the figure above. To adapt the code for our purposes, the “DMA Driver” code that was implemented in user space by Chung was moved into kernel space by removing the memory mappings to user space and removing the need to open the PCI device with a file descriptor. In the first implementation of the new kernel module, we attempted to merge the block device driver into the existing code to create a single module. We ran inconsistent crashing of the kernel using this design, which may have been caused by the fact that the module was trying to act as a PCI driver and a block device driver at the same time. After exporting the read/write function symbols in the PCI portion of the module, the block device driver was split into a separate module which called the PCI read/writes.

It is important to note that these drivers are highly dependent on the kernel version 2.6.28. Between Linux 2.6.28 and 2.6.31, the block subsystem went through many changes in respect to the elevator and request architectures. Currently, we use the old and simplest form of the request system. A point of future progress would be to remove the use of the request and directly use the block input/output structure, the *bio*. This will reduce overhead because a solid state drive has no need for queuing and consolidating requests like a physical disk does, and efficiency optimizations should be primarily placed in the Flash Transition Layer, or any specialized solid state drive management system, whether onboard or on the host.

6.2 Board Components

The hardware required on the FPGA side of the PCI interface consists of a large 8 KB data buffer and a set of control registers. This memory space allows DMA transactions between the PC and the board. A controller that is part of the FTL on the FPGA board manages the signaling. The data buffer is treated like an address mapped queue by the MicroBlaze code, which can make read and writes to the queue. It is important that the size of each data transfer match both on the MicroBlaze and the host.

The code on the MicroBlaze simply waits to read one synchronization byte, which then lets a constant loop of receiving and responding to requests. The command protocol that was defined is described below. The code also performs any necessary initialization of the MicroBlaze caches pre-synchronization and runs the FTL initialization of the drive.

6.3 Command Flow

A read or write command is issued by the kernel driver when reads are needed or when the kernel data caches run out of space. These calls are mapped in a device struct to functions in the kernel module. These functions then complete the action by setting the DMA control registers to signal the device. Table 1 describes these steps in detail.

Step	Read command	Write command
1	Reset the device control register (DCR)	Reset the DCR
2	Determine values for size and count registers	Copy data into data buffer
3	Store address into read address register	Determine values for size and count registers
4	Signal read command register	Store address into read address register
5	Wait for response in status register	Signal write command register
6	Copy data out of data buffer	Wait for count register to empty

Table 1: Host DMA communication protocol over PCIe

6.4 PCI Command Protocol

The simple protocol currently implemented for communication over PCIe is as follows:

1. Send 8-byte header with logical block address and command type

2. Send or read a full page of data associated with that logical block address

7 Flash Translation Layer

The Flash Translation Layer receives commands and data through the PCIe interface, and outputs translated commands with the data to the NAND chip controller. At the host interface, the FTL must provide the logical block abstraction for the disk drivers to treat the device as a disk. At the storage interface, the FTL must maintain the health of the NAND chips. Lastly, the FTL implements caching to improve performance. Each operation in the FTL must be transactionally safe, which means that power loss to the system at any point in any data transfer should not corrupt the system.

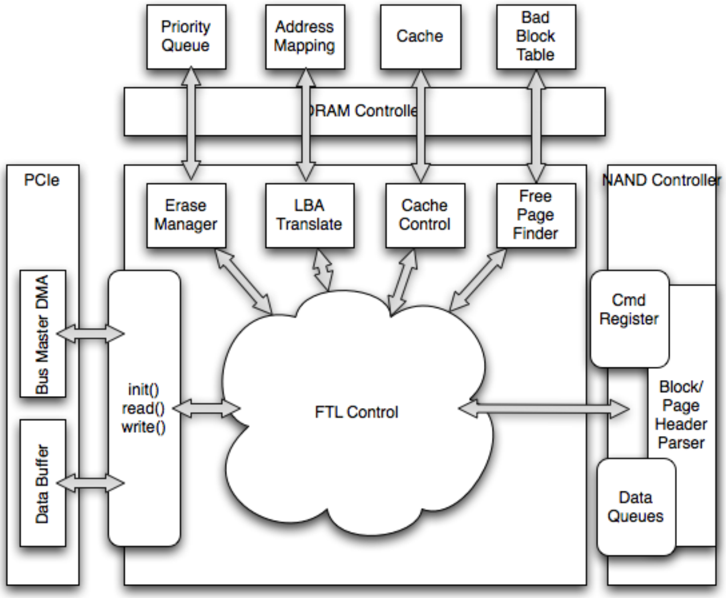


Figure 5: FTL components block diagram

7.1 Initialization

When the drive is started, the FTL immediately begins to probe the header block of each page in order to set up the necessary data structures in memory. Although initialization is slow, storing all drive metadata on each page allows simpler algorithms and eliminates the concern about certain table pages receiving high write loads.

If two pages are discovered to contain the same Logical Block Address (this can only occur due to an improper shutdown), the page with the higher physical address will be invalidated. Pages found to have invalid LBAs will also be likewise invalidated.

7.2 Command Flow

The FTL listens to the DMA registers for new commands. When a command is received, the logical block address known by the host must be translated into the physical address of a page on one of the Flash chips using the mapping tables. In the case of a read, data is forwarded from the chips to the DMA data buffer. In the case of a write, a free page must be found to write the data to. The new page is assigned the same logical block address as the old page that used to hold the page being written. Finally, the old page is invalidated and updates to the erase priority queue are made. With a cache-suppported system, which is not currently implemented, the logical address will be searched for in the cache, thus eleiminating the need to access the Flash chips.

7.3 Address Mapping

A Logical Block Address to Physical Address mapping is maintained by the FTL through a one-level page table. During each read or write, these tables will be accessed to achieve the translation. The tables are loaded into memory during initialization of the drive by reading the headers for each page. When a new page is written, first all the data is copied to the new page. Then, the headers are set with the same logical block address as the old page. Finally, the old page can be invalidated and the in-memory table can be updated.

7.4 Block Management

Blocks and pages on the storage chips are subject to a number of states, such as free vs. used, valid vs. invalid, or bad. This information is used by the Block Management system to determine the location of the next free page. Block management is maintained in memory and implemented as a software algorithm, but the backing metadata is transactionally persistent in the headers for each page. Wear leveling is a block management algorithm that determines when to erase pages, and which pages to erase. Repeated use of the same pages is avoided by neglecting to erase pages with heavy use.

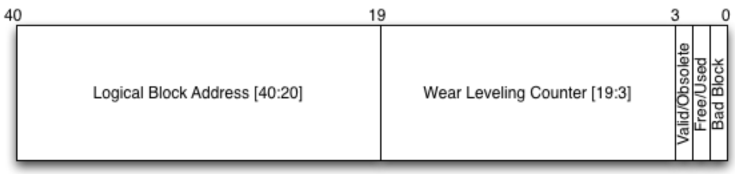


Figure 6: Flash page header definition (in undefined bits of page)

7.4.1 Handling Bad Blocks

During initialization, a block is found to be bad if the bad block bit in any page is set to 1. These bits will be stored in a bit array that acts as a bad block table in memory. When searching for free pages, this bit array will be indexed into by the block number in the physical page address. This feature is not currently implemented since the demonstration version runs on DRAM storage, which has no risk for bad blocks.

7.4.2 Finding Free Pages

Free physical blocks will be maintained in an implicit list. A new free block is not needed until the previous one is completely allocated. Free pages in storage will be allocated sequentially from a physical block, since pages are mapped to logical blocks anyway. The current implementation of this feature is to consecutively issue free pages, without consideration for the block number, since erases have not been completed. Clearly, should the target of the next free page were to extend past the end of the capacity of the drive, a fatal error will arise.

7.4.3 Erasing Obsolete Pages

When a page is marked obsolete, or found as obsolete during initialization, the block it belongs to is stored into a priority queue sorted by the priority of the block being erased. This priority will be determined by a formula that accounts for the block's wear count and number of obsolete pages it contains. The more obsolete pages in a block, the more effecient the block erasure will be to free pages. The higher the wear count, the less advantangeous it is to further progress that count, since at some point we risk losing blocks due to lifetime. This operation is the most critical operation for an effective solid state drive, and therefore deserves additional attention and consideration. In the future, many strategies for optimizing erases should be compared.

7.5 Data Caching

Caching pages as they are read is a common way to drastically improve performance in transaction-intensive systems. There is currently no caching implemented in the system, but there are many ways in which our design supports caching. Function prototypes for cached reads and writes have been included in the FTL code, and the DDR2 controller is already in place. By caching data into RAM, better performance is expected, since the DDR2 reads are slightly faster than the NAND reads. However, caching introduces many transaction and consistency issues that will need to be designed against.

8 DDR2 SDRAM Controller

The default DDR2 SDRAM controller that comes with the Xilinx LX110T board that we are using is meant to be used with a 256MB DDR2 SDRAM SODIMM (with four 512 Mb chips on it). However, since we are interfacing our custom-fabricated SODIMM which only contains one 256 Mb DDR2 SDRAM chip, there is thus has a very different routing and signalling scheme compared to the original SODIMM module, and so the controller must be modified.

With reference to the Xilinx Multi-Port Memory Controller (MPMC) Data Sheet, there are steps to be followed to use Xilinx's Core Generator (Coregen) tool to prepare a new MPMC core that is tailored to our single DDR2 SDRAM chip. Once a new core is generated, the Universal Constraints File (UCF) has to be modified to reflect the specific SODIMM pins (and thus FPGA pins) that we will be routing our signals through. Once this is done, the UCF file is verified to be correct using Coregen, and converted to work with the existing PCIe project we started with. Unfortunately things (as usual) didn't go as per Xilinx's plan, and there were many perplexing synthesis error messages that resulted from the drop from 256MB to 32MB of DDR2 SDRAM - errors that consumed yet more time to guess and attempt to resolve.

Nonetheless, a working DDR2 SDRAM controller was eventually synthesized correctly, and tests for accuracy passed.

9 NAND Peripheral

The NAND Peripheral is connected to the MicroBlaze via bi-directional Fast Simplex Links (FSL). The peripheral also contains a Block RAM (BRAM) buffer, an Error Correcting Code (ECC) module and the NAND Controller module. The peripheral is driven by an "interface FSM" as shown in Figure 7.

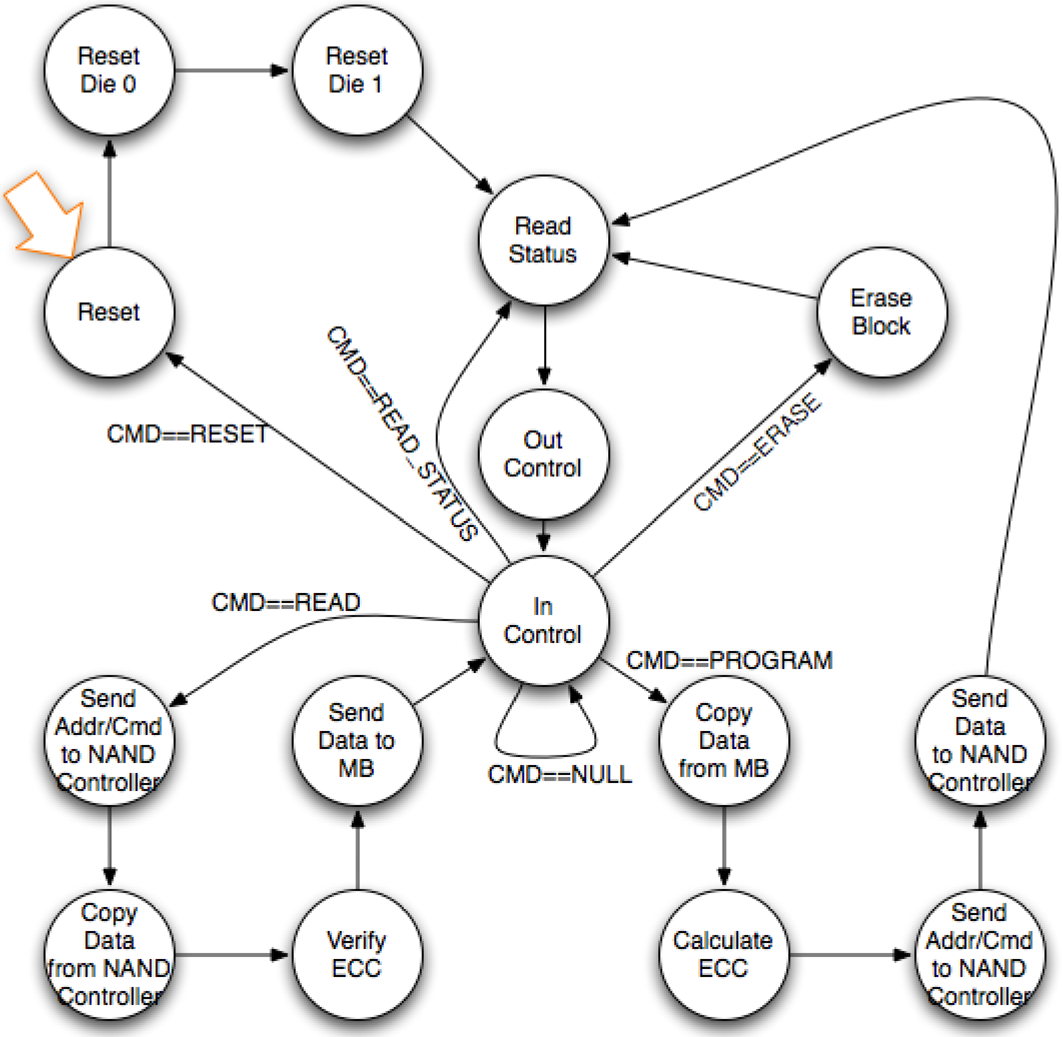


Figure 7: “Interface FSM” for the NAND Peripheral

9.1 FSL Communication

The FSL was selected over the PLB due to its higher data transfer rate achieved during characterization tests.

Any request is always enacted from the Microblaze, and so the interface FSM in the Nand Peripheral idles in the “In_Control” state. When the Microblaze sends a request to the peripheral, it first sends a two-dword long header (each transfer is 1 dword in width), giving the command, size of the request (if any) and the address to perform the request on. In the case of a Write, the interface FSM will continue reading in the specified number of bytes into a local BRAM page buffer. For Reads and Erases, control is passed to the NAND controller.

9.2 Page Buffering

Data is buffered in the BRAM page buffer in order to facilitate the calculation of ECC checksums (in the case of a write) and the verification of data read from the NAND controller (in the case of a read). Since the data is buffered, corrections to bit errors are easy.

9.3 ECC Calculation and Verification

A ECC module was written to do process 32 bits at an instance, to perform realtime calculation of the checksum as the data is streamed in. The ECC algorithm allows for Single Error Correction, Double Error Detection (SECDED). The interface FSM calculates and verifies ECC during each read or write of 4096 bytes (one logical block), and tags it to the ECC checksum to the end of the page (pages are 4314 bytes). As such, this whole process and the extra bytes for ECC are totally transparent to the host system.

9.4 NAND Controller Communication

To initiate a transaction, the host will first signal the address and command to the NAND controller, and give a start signal in the status register. The interface FSM will also be watching the status registers for signs from the NAND controller that it is sending data out, so that the data can be buffered and then verified.

10 NAND Controller

The interface FSM and the controller will communicate through a custom designed protocol. On the other end the nand controller will be issuing commands to the flash chip through the nand interface, which will be mapped to the pins on the SODIMM slot.

10.1 Signal Description

Table 2 contains the descriptions for all the top level ports of the NAND controller module handshaking with the NAND peripheral FSM, and Table 3 contains the descriptions of the the top level ports connected to the physical NAND chips.

Note: Signals *i, o, dir* are merged into a single bidirectional IO bus at the top level. The reason they are separate here is because the Virtex 5 does not support tri-state buffers except at the top level driving external ports.

10.2 Initialization sequence

According to the Micron datasheet, the NAND chip requires at least 100 ns after Vdd reaches 2.5V to stabilize. Therefore, upon receiving a reset from the interface logic, the controller will count up to 16384 before start accepting commands (assuming the clock cycle is at least 10ns). Once that is completed, the controller will signal DONE so the interface FSM can start issuing commands. Though not explicitly designed, the first two commands from the interface FSM must be RESET commands to both chips, according to the datasheets. After both resets are done, the NAND chips are available for normal usage.

Signal	Type	Description
clk	I	Clock, should be between 20-100 Mhz
reset	I	On posedge, resets the NAND controller
DONE	O	Signals the end of a transaction to the interface FSM
DATA_DONE	O	Signals the byte has been written, ready to receive new data
DATA_READY	O	Signals the byte read from the NAND chip is in DATA_OUT
Status[7:0]	O	Stores the results of a read status commands
DATA_OUT[7:0]	O	Stores the byte read from the NAND chip, only valid after DATA_READY
DATA_IN[7:0]	I	On a write, interface FSM will put the data into DATA_IN
ADDR[31:0]	I	Indicates the target address ADDR[31]: chip select within the device ADDR[30:19]: block address within a chip ADDR[18:13]:page address within a block ADDR[12:0]: column address within a page
CMD[2:0]	I	Indicates the CMD 0: Reset 1:Read status 2:Block erase 3: Program page 5:Read Page
DATA_LOADED	I	Indicate the data to be written is in DATA_IN
CMD_LOADED	I	Indicate the next command is in CMD
LEN[12:0]	I	Indicate the number of bytes to be read/written

Table 2: NAND Controller - Interface signals

Signal	Type	Description
n_ce1_l	O	Chip enable for the first chip
n_ce2_l	O	Chip enable for the second chip
n_ale	O	Address latch enable, indicate the address is being latched
n_cle	O	Command latch enable, indicate commands are being latched
n_re_l	O	Read enable, strobe to transfer data from NAND chip to controller
n_we_l	O	Write enable, strobe to transfer data from controller to NAND chip
n_o[7:0]	O	Data output from FPGA to NAND chip
n_i[7:0]	I	Data input from NAND to FPGA
dir	O	Indicate direction of CPLD data,tied directly to re_l
n_rb1_l	I	Ready/Busy from chip 1
n_rb2_l	I	Ready/Busy from chip 2

Table 3: NAND Controller - NAND signals

10.3 Protocols

To initiate a transaction, the host will first put commands into CMD, and assert CMD_LOADED. For a read/write/erase operation, address should also be put in ADDR, length into LEN before CMD_LOADED is asserted. CMD_LOADED and CMD can be de-asserted in the following cycle. However LEN and ADDR [31] need to be held throughout the transaction. Once the CMD is received, the controller will move to different states depending on the command.

Read:

The controller will move to the read state, issue the instruction to the nand chip, and start reading data from the NAND chip by strobing re_l. Then it will put the data in DATA_OUT and signals DATA_READY. Note the controller does not require another explicit handshake at this point, it will move on to read the next byte, therefore the controller must pick up the data in DATA_OUT within 2 cycles of DATA_READY being asserted. The controller will continue to read data until its internal count reaches LEN. Note the NAND controller does not implement page bounds checking. In which case it will signal DONE after the last DATA_READY to conclude the transaction.

Write:
Similar to a read, the interface FSM will first put out CMD, ADDR, LEN and signals CMD_LOADED. Note that CMD and LEN can be de-asserted after 12 cycles. Then the interface FSM can start putting out data to be written into DATA_IN. After DATA_LOADED is asserted and de-asserted in the following cycle, the interface FSM will wait for DATA_DONE to be asserted by the controller. After the last DATA_DONE, the controller will assert DONE to conclude the transaction. Note similar to READ, bounds checking are not implemented.

Other commands:
The controller will put command into CMD and assert CMD_LOADED and then wait for DONE. In the case of Read status, the result of status will be put into the status port when DONE is asserted.

10.4 Controller Logic

The controller will be waiting in the start states until it sees rising edge of CMD_LOADED. As shown below:

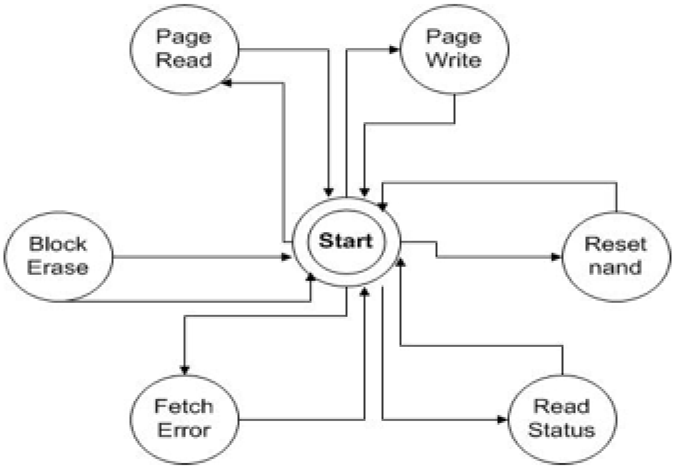


Figure 8: FSM for NAND commands

10.4.1 Read Logic

To execute a read command, the controller first latches the command, followed by 5 cycles of address. After another cycle of command, it will wait for the nand chip to toggle the R/B line. After that data can be read out sequentially by toggling re_l line.

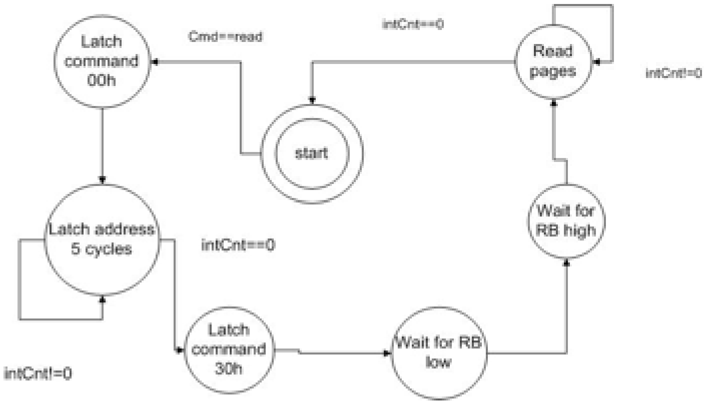


Figure 9: FSM for NAND read command

10.4.2 Write Logic

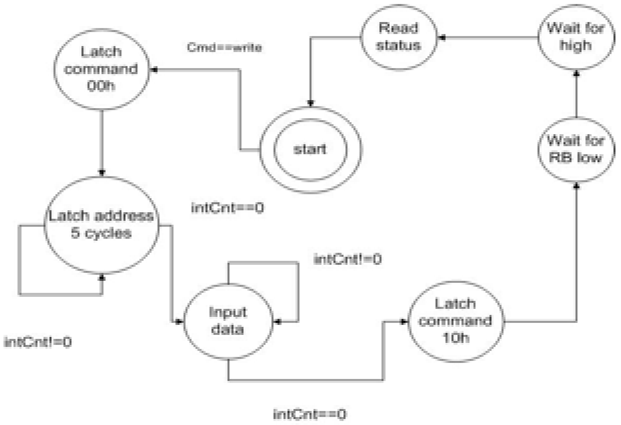


Figure 10: FSM for NAND write command

The write logic is similar, after the address is latched in, the controller will start sending out the data to the data registers on the nand chip by toggling the we_l lines. After a command cycle to signal the data transfer is done, the controller will wait for the nand chip to finish programming (250 us). A read status command is needed at the end of verify the integrity of the page written to. The write command will fail (status [0] == 0) if the controller is writing to a used page.

10.4.3 Erase Logic

Since erase can only be done at the block granularity, only 3 address cycles will be needed. Upon sending the address, the controller will wait for the nand chip to signal done by toggling R/B lines (700 us). Like write, a read status command is needed at the end to verify the integrity of the page.

10.4.4 Other Commands

The other commands (reset, read status) do not require address or data cycles. The controller simply issues the command and wait for the reply from the nand chips. Read status is mostly used

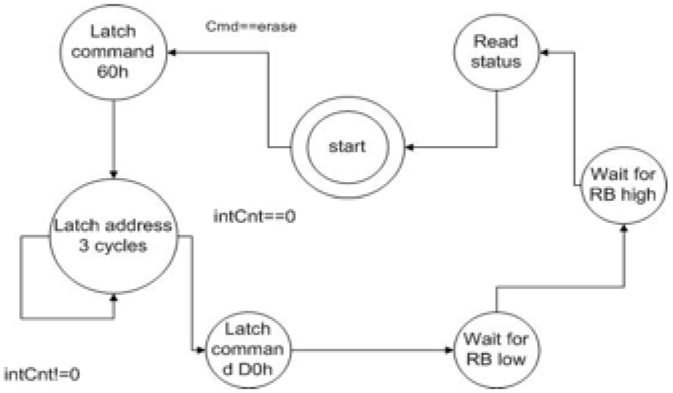


Figure 11: FSM for NAND erase command

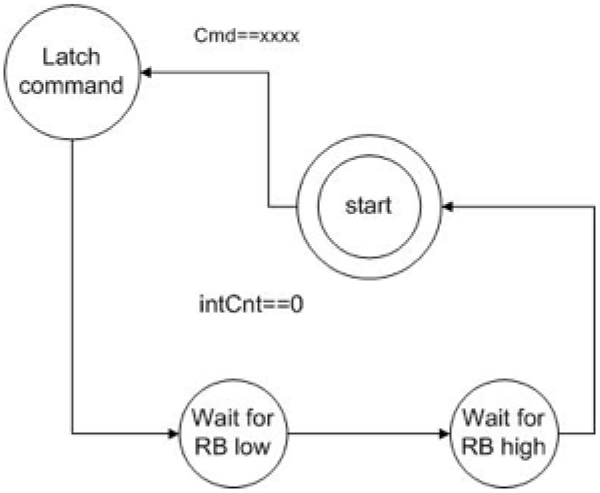


Figure 12: FSM for other NAND commands

to verify if a transaction is successful.

10.5 NAND Chips

Each NAND chip constains 16Gb of storage and is operated through a 17 pin interface, driven by the NAND controller. Each device consists of 2 dies, each with 8192 blocks, and each block contains 64 page. Each page can store 4314 bytes of data. Typically, this is segmented into the first 4096 bytes for actual data, 194 bytes for book keeping and as spare space, and the remaining 24 bytes for storing a Single Error Correct, Double Error Detect (SECDED) Error Correcting Code (ECC).

The details of the pins and bus timing diagram will not be shown due to non-disclosure agreements.

10.6 Verification

The design is verified through simulation. Courtesy of Micron, we have an unsynthesizable verilog model for the NAND chip that we are using. Through simulation, we have verified the protocols and more importantly, that all the timing constraints are met. We have also modeled a CPLD module after the NAND controller but before the NAND chip verilog model, to ensure that the extra delay through the CPLD will not cause our design to fail. We are still in the process of verifying the NAND controller with real NAND chip.

11 State of the System

PCI Express DMA Kernel Module:
This module was modified and is fully functional, and provided to the other SSD team to aid their MTD development.

Block Device Driver, Basic FTL, DDR2 Controller, 32MB DDR2 SDRAM:
These subsystems are fully operational and tested. The lab demonstration showed the capabilities of these parts to appear as a block drive and write data to temporary storage.

CPLD:
The CPLD module was written and the RTL schematic generated by ISE matches our intent for the device. However, more concrete tests should be run. This module was provided to the other SSD team.

NAND Peripheral FSM, Fast Simplex Link, Buffer:
These subsystems, while tested to work in simulation, have worked in hardware when standing by itself. However, hooking the NAND controller to it has not been successful yet. These subsystems are provided to the other SSD team.

NAND Controller:
The NAND controller was tested with the specific Micron simulation model for the chip we are utilizing, and it was successful. However, the controller has not been successfully integrated into hardware at this point. Currently, if the controller is run at a 50MHz clock, we do see the NAND chip responding to the reset commands given to both dies (ready/busy line goes down, then back up, deterministically). However, the NAND chip does not seem to be responding to any additional commands. One possibly to be explored is that the CPLD may not be passing signals to the NAND controller truthfully, especially on the bidirectional data bus. This is work in progress.

This component was worked on by both SSD teams.

ECC module:
The ECC module was verified to be working in simulation, but has not been integrated into actual hardware yet.

Custom SODIMM:
The custom SODIMM was completely designed, fabricated, and tested. There were two identical parts produced, to be shared between the SSD teams.

12 Running the System

12.1 Block Device Driver, PCI Express DMA Kernel Module, Basic FTL

1. Load the Xilinx XPS project “pcie04”.
2. Insert the Xilinx LX110T FPGA board onto the Gigabyte motherboard. Connect the serial cable to a computer to monitor debug messages.
3. Power on the Xilinx LX110T board and download the bitstream to the board before turning the computer on. Once downloaded and the processor started, there should be some output on the serial port regarding the initialization of the drive.
4. Turn on the computer.
5. If the driver is not compiled, run “make”.
6. Install the PCI Express DMA kernel device driver. (“sudo insmod ssd_dma.ko”)
7. Load the custom block device driver. (“sudo ./ssd_block_load”)
8. The debug console should show that a synchronization byte was received.
9. Build a filesystem on the drive (“mkfs -t ext2 /dev/ssd_blocka”)
10. Mount device to a mount point (“sudo mount /dev/ssd_blocka drive”);
11. Create files on mounted device.
12. One good test is to reboot the system while the FPGA is still running. When the drive is remounted after reboot, the files created earlier should still exist. This ensures that you will not be reading data cached by the OS.

13 Major Decisions and Issues

At this point in the project, there have been some major design decisions that have been made, as well as some tricky issues of note.

13.1 SATA vs. PCIe

The decision of how the drive interfaced with the host computer was a choice between a direct PCIe x1 connection and a SATA connection. The benefit of SATA was that it would not require any special driver code and should work on any computer and operating system with SATA support. Unfortunately, the SATA network layers were deemed to complex to implement a hardware driver for, especially since we did not have access to any suitable debugging tools, like a SATA protocol analyser.

However, the rest of the system was nonetheless designed with the possibility of swapping the PCI express controller with a SATA controller in the future.

Option	Pros	Cons
SATA	Plugs into any computer Implements error checking Communication protocol predefined	Complex design Hardware implementation No suitable debug tools
PCIe	Existing DMA code Mostly software	Must write custom block device driver Must find/define documentation

Table 4: Decision Matrix for Host Interface Protocol

13.2 Software vs. Hardware FTL

The decision to implement FTL logic in software was due to ease in development and debugging. System performance of this design is not dependant on the FTL, so the slower software FTL implementation is irrelevant. Also, since the algorithms behind the FTL management have not been completely decided upon, it is important to maintain some flexibility in the implementation.

Option	Pros	Cons
Software	Easier to implement Can change algorithms quickly	Processor and Bus are slow
Hardware	Faster performance FSM is appropriate	Harder to implement

Table 5: Decision Matrix for FTL Implementation

13.3 FTL Design and Algorithms

The design details of the FTL are the least finalized portion of the overall system. Initial implementation will begin with the design and algorithms outlined in Section 4. However, more research will be conducted in this area and the FTL may change as we discover or design more possible solutions.

13.4 FTL vs Memory Technology Device (MTD)

The MTD is a generic Linux subsystem created specifically for Flash memory devices that was very recently introduced. It is neither a block nor a character device. Although going the route of implementing the MTD is easier, as all the necessary Flash management features are already implemented in the software stack, we decided to stick with implementing a traditional FTL for our SSD as that would give us the flexibility to swap out the PCI express controller and replace it with a SATA controller in the future, as per the original plan.

13.5 FPGA Board Choice

The decision to use the Virtex 5 LX110T FPGA over the Virtex 5 FX was primarily based on the fact that the existing PCIe interface code was written for the LX. Other considerations were that the MicroBlaze processor on the LX board was fast enough to power the FTL, rendering the PowerPC on the FX unnecessary. The MicroBlaze offers better support for faster connections to memory and the NAND controller via the Fast Simplex Link.

13.6 Motherboard Compatibility

Inserting the the Virtex 5 LX110T board into the x16 lane slot on the lab's Dell Precision's motherboard caused it to fail to boot, after the FPGA and MicroBlaze processor was initialized. When inserted into the Dell's x8 lane slot on the motherboard, the Virtex 5 LX110T was detected as an invalid Network Interface Card. One suspected cause of this issue is that lane width negotiation on the PCIe bus did not succeed. When we moved to the motherboard (Gigabyte GA-G31M-ES2L) that Eric Chung used for development of the PCIe Interface code, the Virtex 5 LX was detected in the PCIe x1 slot.

13.7 Processor Local Bus vs Fast Simplex Link

Initially we were planning to use Processor Local Bus (PLB) to communicate between PowerPC core and our controller logic. However after further study and characterization experiments, we realized that the PLB is very slow. Both DRAM and SRAM requests go through the PLB, and the maximum observed throughput under 15MB/s. Characterization tests for the Fast Simplex Link (FSL) that connects the Microblaze directly to the peripheral show maximum throughput at approximately 35MB/s.

Since the PLB is likely to be a performance bottleneck, we chose to use the FSL instead. The downside is by that time, we already designed the protocols for the controller module using PLB and memory mapped registers (probably no longer feasible given the speed of PLB anyway). Rather, we need to write a interface FSM to send/receive data packets from the FSL, decode them and issue commands to the controller. This unexpected increase in design complexity set our schedule back considerably. However, this being an ongoing research project, we decided to optimize for better performance(we believe this will become the performance bottleneck), at the cost of ease of implementation.

13.8 NAND controller implementation

We have access to another NAND controller with ECC integrated into it, from Micron. While it was designed for an older generation of NAND chips, for the Spartan III board, we were hoping to be able to reuse the code there for our newer chips. Unfortunately, after further study we found out that there were pretty significant differences between the two chips. While the protocols were similar, the timing constraints are significantly different. It did not help that the NAND controller came with relatively limited documentation. Although we know that the NAND controller implemented an SRAM interface to the Microblaze host, we could not find any examples or documentation on how the embedded core is supposed to communicate via the interface. As such, in order to make it work, significant modifications are needed. As the module is also written in VHDL, none of us had any experience in it. In order to reduce the risk of introducing new bugs due to the general lack of familiarity with VHDL constructs, we decided to rewrite the NAND controller from scratch in Verilog.

14 Unexpected Issues

1. Nobody had expected that the Xilinx provided template file used by the Base System Builder for the Xilinx LX110T boards that we were using had errors and generated a project that did not synthesize. It took some time to debug before the root cause of the issues were found. Xilinx code can't always be trusted.

2. We had severely underestimated the amount of time needed to get the custom-fabricated SODIMM design correct - selecting the right components and verifying all the connections took ages. Sending the SODIMM out for layout and fabrication also took longer than expected, due to the company requiring us to verify their work after every stage.

3. After following all of Xilinx's steps in their specifications sheet on how to modify and replace the Multi-Port Memory Controller (MPMC) very closely, it was a major surprise that there were countless error messages that appeared on synthesis, requiring plenty

4. The standard 14-pin Xilinx JTAG cable configuration is defined as having 7 pins connected to ground, 1 pin connected to Vref (typically 1.5V - 5V), 4 pins connected to signals, and 2 pins unconnected. Being the first time wiring up a JTAG connector to a device, we did not realise that the Vref line is an input to both the CPLD as well as the USB programming dongle - neither drove a voltage on the line. It took us a while to figure out what was wrong, and then come up with a quick hack to supply a voltage on the Vref line.

5. In the latest versions of the Linux Kernel (¿ 2.6.28), the block device subsystem in the kernel was rewritten and the APIs were completely changed. Unfortunately, none of the online documentation for writing a block device driver explained any of the new changes as they were still based on the older kernels. As such, much time was spent on trying to figure out the differences in the API and how they are used.

6. "ERROR:MapLib:824 - Tri-state buffers are not supported in Virtex5. Block nand_01_0/nand_01_0/n_io must be removed from the design." This error drove us nuts for a good week, seemingly indicating that there is no way we could interface our FPGA with the NAND flash chips, as the data bus are bi-directional. Google wasn't too kind either, as most responses from people were just as cryptic. Turns out that specifying the port as "inout" in the port list in the module is not sufficient - in fact, in addition to the original inout port, there must also be a corresponding in, out, and direction port(s) of the same width. Only by editing the MPD file of the peripheral can you specify that in, our and direction ports actually drive an inout port, and tell XST to use a tri-state buffer.

7. Many other issues too numerous to remember - however, one thing is for sure - with Xilinx, something that is unexpected will happen, always.

15 Words of Wisdom

1. While ChipScope may seem utterly terrible to use initially, it is an excellent tool for debugging. Learn it very early in the semester. Learn to use the triggers judiciously.

2. Xilinx's example code often isn't written in the best way (especially if it's a Verilog example). If you think there's a better way, there probably is.

3. Know that Xilinx's code and instructions often cannot be taken for granted. It is highly likely that you will encounter bugs.

4. Verilog and VHDL are treated very differently by Xilinx, even though they're both popular hardware description languages - something that works on VHDL may not be supported in Verilog, or vice versa.

5. You can never delay something by one day from your schedule and think you can make up for it later.
6. When the going gets tough, it might be a good idea to consider pushing your fellow teammates along.
7. One can very readily get burnt out - so keep a look out for it coming!
8. Do not change the (SO)DIMM on the FPGA - doing so may cause the memory controller, and thus the entire project, to fail.
9. If you are intending to use the DDR2 SDRAM, make sure you are not overwriting memory containing your program. The default address to store a program when initializing the Microblaze can be set under the Application's settings.

16 Future Work

There are plans to carry on this project as a part of a research project under Professor Ken Mai. The main issues to work out is to firstly verify that the CPLD is working as we expect it to, and secondly, to find out why the NAND flash chips are not responding to the commands and signals sent to it. After basic functionality is achieved, the remaining components, like the ECC and AES modules should be put into place.

The current implementation of the FTL is also somewhat inefficient and not highly robust. Fixes for these shortcomings would be necessary for the drive to be a long lasting device. Caching by the FTL has not been done, and could provide a speed-up. As always, new algorithms for wear leveling and erase management are welcome updates.

The block device driver could be updated to not use the request and elevator systems, thus making it more flexible across kernel versions, as well as removing unhelpful overheads.

17 Individual Contributions

17.1 Winston Wan

During the first weeks of the project, I primarily focused on learning about the SATA protocol and the requirements to making a SATA compliant drive. In the meanwhile, I tried to organize the top level design of the project and create a wiki for our own note-taking purposes. We worked closely with the other group and Professor Mai to determine that SATA was going to be too major of an undertaking, and decided on using the PCIe port. In the following weeks, I worked on familiarizing myself with Eric Chung's existing codebase, and running his basic tests on his machine with much help from Tim and Will. Also during this time, I tried to solidify all the interfaces between each of the subsystems. The majority of the semester was spent trying to adapt Eric's user space code into the kernel, as well as adapting the generic block device driver into the system. Due to major misunderstandings of the capabilities of the loadable kernel modules and trouble with different kernel versions on different computers of ours, this work took very long. In parallel, I began developing a simple FTL in user space. This rudimentary FTL was very easily inserted for use on the Microblaze. After completing the block driver and PCI driver as kernel modules with Tim's help, I spent some time aiding the other group with similar struggles that plagued us earlier as they attempted to build their MTD module. In the last week of class, I prepared the demo code and the demo poster to allow Tim and Tao to focus on the NAND controller issues. I think each member of our group did a great job of balancing out the amount of work and helping each other when needed. Tim was especially helpful in all parts of the project.

17.2 Tao Yang

During the first two months, I helped Will and Tim designed the DIMM (3 weeks). We created the initial schematics (in excel) using the flash chip and CPLD we picked. We also finished the labs (2 weeks), collectively came up with the overall architecture, refined our design, and made various design decisions such as PCI Express/Serial ATA, FX boards/LX boards etc (a month). Around mid to late October, I started working on the NAND controller (joined by Myyk 2 weeks later). To verify my understanding of the bus protocols, I first modeled and tested the controller through unsynthesizable verilog, and later converted it into a FSM design. After the FSM was fully implemented and simulated against the test bench I wrote, we ran into some synthesizability issues, which set us back for about a week. However, even though we finished testing our module around mid November, we could not test it against real chip since we need to interface it through micro blaze. Though we decided the protocols a while ago, implementing and testing the FSL interface module (by Tim) turned out to be a much more difficult job then we envisioned. While helping Tim writing and debugging the interface logic, I also implemented and tested the ECC module (though it was never integrated into the system due to lack of time). Around thanksgiving, we (we and Tim) finally finished our initial implementation of the FSL interface module and started testing our controller against the real chip (note the DIMM arrived the day before Thanksgiving, surface mount sockets and spare flash chips a few days before that). We spent the rest of the time debugging and eventually got some response and the NAND chip (the reset commands works, at least we think).

17.3 Timothy Tan

The initial few weeks of the project was spent trying to study various interface options for the various chips we needed connected to the FPGA (didn't want to settle for the SODIMM slot initially), and to figure out what devices and chips we needed, and the differences between all of them. I also jumped into picking up the SATA specification and protocols, to get an understand of the type of FSM we would need to write - it being my second task at that stage of the project. At the same time, one of the immediate issues faced was that in order to use the Virtex-5 LX110T boards provided by Professor Ken Mai, the Xilinx software had to be reinstalled, and licenses located for version 11.

Unfortunately, just as I had created a SATA RocketIO core and convinced myself that writing the SATA controller FSM was doable, I received a dose of reality, that it was very unlikely that the SATA controller could be completed without the use of more sophisticated analysis tools. PCI Express had much greater odds of success, and so it was - dropping everything SATA and picking up books on kernel device drivers. The next stage of the SODIMM design was to create a schematic that contained our design, which could be sent out for a company to do the layout and fabrication. Unfortunately we ran into major issues with the design tools available in the school and wasted a great deal of time working things out with the department IT admins. While the schematic, layout and fabrication work was eventually outsourced to Cirexx to do, it still involved many rounds of checking and rechecking by Will, Tao and I to make sure we did not misread any datasheets or leave out a crucial wire between components. And it was a good thing we labored on, as we soon discovered several problems which required some rethinking.

Besides working with Winston on the kernel drivers, I worked on characterizing the speeds of the various buses and interfaces available on the FPGA - turns out that the FSL was the fastest, and as such was selected to interface our MicroBlaze and the NAND controller. I initially started with Xilinx's example code with an implicit FSM for FSL data transfer, and wrote a transfer test between the Microblaze and a 4KB buffer - it worked beautifully, and we called it the "interface FSM". Not giving it too much though I worked on extending the "interface FSM" to communicate with the NAND controller. Unfortunately, that is where things went wrong. With the larger "interface FSM", XST became confused and took 12 hours to synthesize the design! There was no choice but to immediately rewrite the "interface FSM" explicitly. This cost a few days, but thankfully, with Tao's help, the Modelsim simulations eventually gave a green light again. Unfortunately a green light in simulation translated to no-go in hardware when we used both the surface mount and the actual SODIMM. Tao and I spent the entire Thanksgiving and most of the days before the public demo trying to follow every clue, trying every method of debugging and correcting potential problems before the chip started showing signs of life.

This course has been one of the most memorable one of all in CMU, and I believe that none of us in the group have ever spent so much time working so intently on a project. It was a thrill and I certainly learnt a lot from the experience.

18 Class Impressions/Improvements

We thought this class was a unique experience to do a highly independent research and design project at the undergraduate level. Being immersed in the project and being able to struggle with problems was educational, but some more guidance would probably have resulted in a more successful final demonstration. The tools were difficult to use and the labs were too guided and too short to provide appropriate preparation for the course project. We believe the chance to do a different project like the Solid State Drive instead of the prescribed video game platform was a great opportunity, and we hope that there will be more of these alternative project options in the future.

19 References

- Chung, Eric "PCI Express Notes" ProtoFlex Project 2009. Retrieved: Oct 1, 2009.
⟨ http://www.ece.cmu.edu/ protoflex/doku.php?id=internal:pci_express:pci_express_notes ⟩

- Chung, Eric "ProtoFlex User Guide" ProtoFlex Project 2009. Retrieved: Oct 1, 2009.
⟨ http://www.ece.cmu.edu/ protoflex/doku.php?id=documentation:userguide ⟩

- Corbet, Jonathan, Rubini, Alessandro and Kroah-Hartman, Greg Linux Device Drivers CA: O.Reilly Media, Inc. 2005.

- Gal, Eran "Algorithms and Data Structures for Flash Memories" 2004. Retrieved: Sep 15, 2009.
⟨ http://www.tau.ac.il/ stoledo/Pubs/flash-survey.pdf ⟩

- Micron Technology, Inc. "256Mb: x4, x8, x16 DDR2 SDRAM" Product Specification 2009. Retrieved: Sep 29, 2009.
⟨ http://download.micron.com/pdf/datasheets/dram/ddr2/256MbDDR2.pdf ⟩

- Micron Technology, Inc. "8, 16, 32, 64Gb NAND Flash Memory" 2008. Retrieved: Sep 3, 2009.
⟨ URL not available publicly ⟩

- Micron Technology, Inc. "ECC Module for Xilinx Spartan-3" Technical Note TN-29-05 2007. Retrieved: Oct 2, 2009.
⟨ http://download.micron.com/pdf/technotes/nand/tn2905.pdf ⟩

- Micron Technology, Inc. "Hamming Codes for NAND Flash Memory Devices" Technical Note TN-29-08 2007. Retrieved: Sep 15, 2009
⟨ http://download.micron.com/pdf/technotes/nand/tn2908.pdf ⟩

- Micron Technology, Inc. "Increasing NAND Flash Performance" Technical Note TN-29-14 2007. Retrieved: Oct 31, 2009.
⟨ http://download.micron.com/pdf/technotes/nand/tn2914.pdf ⟩

- Micron Technology, Inc. "ML505/6/7 Block Diagram" ML505/6/7 Virtex-5 Evaluation Platform 2008. Retrieved: Sep 2, 2009.
⟨ http://www.xilinx.com/support/documentation/boards_and_kits/ml50x_schematics.pdf ⟩

- Micron Technology, Inc. "NAND Flash 101" Technical Note TN-29-19 2006. Retrieved: Oct 1, 2009.
⟨ http://download.micron.com/pdf/technotes/nand/tn2919.pdf ⟩

- Micron Technology, Inc. "Wear-Leveling Techniques in NAND Flash Devices" Technical Note TN-29-42 2008. Retrieved Sep 15, 2009.
⟨ http://download.micron.com/pdf/technotes/nand/tn2942_nand_wear_leveling.pdf ⟩

- Ng, Mark. "Supporting Multiple SD Devices with CPLDs" Xcell Journal First Quarter 2008. 37-39.

- Numonyx, B.V. "Error correction code in single level cell NAND flash memories" Application Note AN1823 2008. Retrieved: Nov 18, 2009.
⟨ http://www.numonyx.com/Documents/Application

- Petazzoni, Thomas. "Block Device Drivers" Free Electrons, Embedded Linux Developers 2009. Retrieved: Nov 25, 2009.
⟨ http://free-electrons.com/doc/block_drivers.pdf ⟩

- Qimonda AG. "Design Guide for DDR2 Memory Products" Application Note AN_CD035 2008. Retrieved: Sep 30, 2009.
⟨ http://www.qimonda.com/static/download/promopages/AN_CD036_Design_Guide_for_DDR2_rev100_x2x.
⟩

- Rajimwale, Abhishek. "Block Management in Solid-State Devices" USENIX '09 2009. Retrieved: Nov 26, 2009.
⟨ http://www.usenix.org/events/usenix09/tech/full_papers/rajimwale/rajimwale.pdf ⟩

- Xilinx, Inc. “Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions” Application Note XAPP1052 2009. Retrieved: Nov 15, 2009.
  ⟨ http://www.xilinx.com/support/documentation/application_notes/xapp1052.pdf ⟩

- Xilinx, Inc. “Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel” Application Note XAPP529 2004. Retrieved: Oct 7, 2009.
  ⟨ http://www.xilinx.com/support/documentation/application_notes/xapp529.pdf ⟩

- Xilinx, Inc. “DDR2 Memory Controller for PowerPC 440 Processors” Product Specification DS567 2009. Retrieved: Oct 18, 2009.
  ⟨ ftp://ftp.xilinx.com/pub/documentation/misc/ppc440mc_ddr2.pdf ⟩

- Xilinx, Inc. “High-Performance DDR2 SDRAM Interface in Virtex-5 Devices” Application Note XAPP858 2008. Retrieved: Oct 2, 2009.
  ⟨ http://www.xilinx.com/support/documentation/application_notes/xapp858.pdf ⟩

- Xilinx, Inc. “Level Translation Using Xilinx CoolRunner-II CPLDs” Application Note XAPP785 2005. Retrieved: Sep 18, 2009.
  ⟨ http://www.xilinx.com/support/documentation/application_notes/xapp785.pdf ⟩

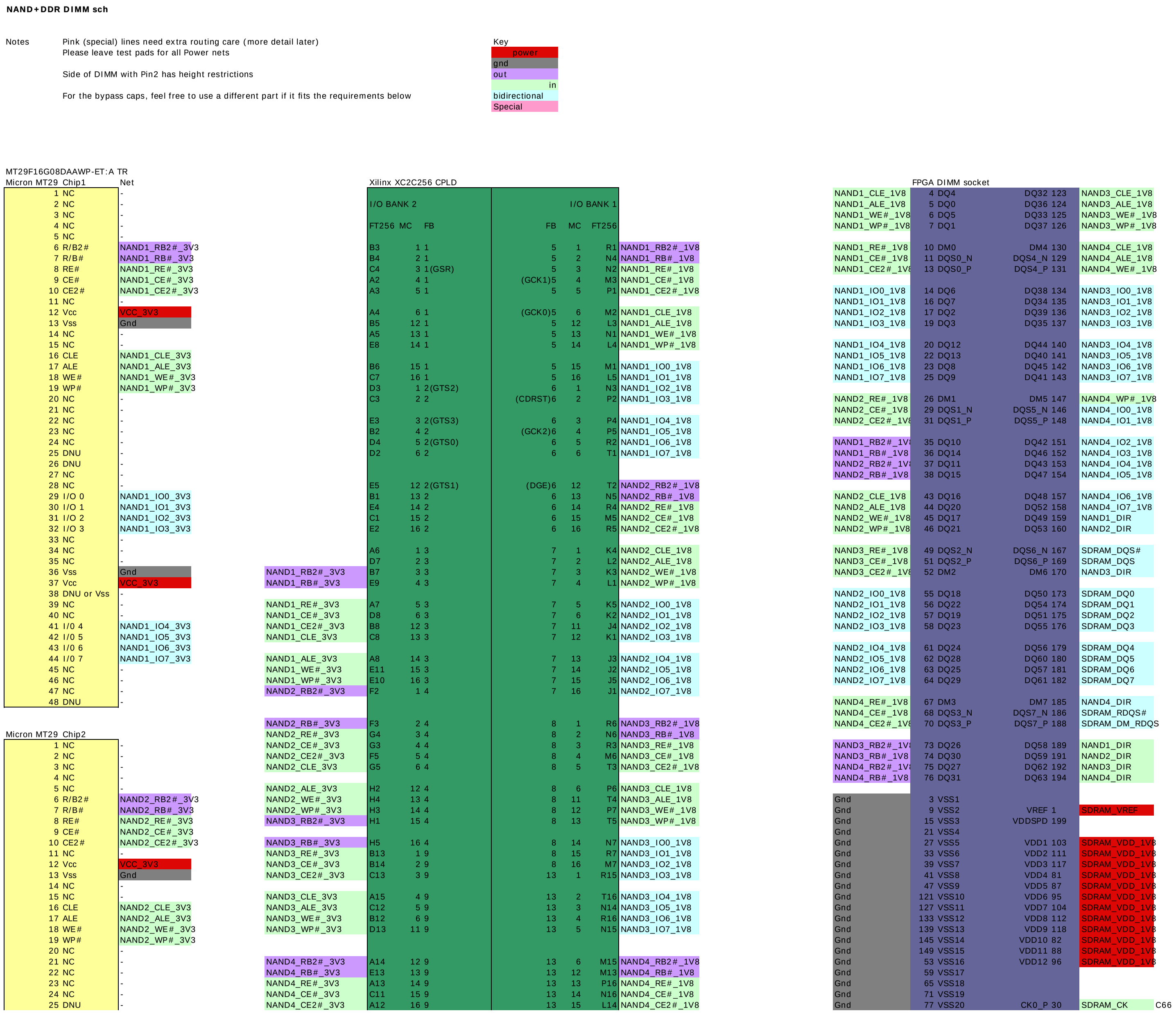
- Xilinx, Inc. “Multi-Port Memory Controller (MPMC) (v4.02.a)” Product Specification DS643 2008. Retrieved: Oct 7, 2009.
  ⟨ http://china.xilinx.com/support/documentation/ip_documentation/mpmc.pdf ⟩

- Xilinx, Inc. “XC2C256 CoolRunner-II CPLD” Product Specification DS094 2007. Retrieved: Sep 28, 2009.
  ⟨ http://www.xilinx.com/support/documentation/data_sheets/ds094.pdf ⟩

- Xilinx, Inc. “Xilinx Synthesis Technology (XST) User Guide” Xilinx Development System Retrieved: Nov 29, 2009.
  ⟨ http://www.xilinx.com/itp/xilinx9/books/docs/xst/xst.pdf ⟩

A Custom-fabricated SODIMM Design Details

A.1 Original Design



NAND+DDR DIMM sch

Notes

Pink (special) lines need extra routing care (more detail later)
Please leave test pads for all Power nets

Side of DIMM with Pin2 has height restrictions

For the bypass caps, feel free to use a different part if it fits the requirements below

Key
power
gnd
out
in
bidirectional
Special

MT29F16G08DAAWP-ET:A TR

Micron MT29 Chip1		Net
1	NC	-
2	NC	-
3	NC	-
4	NC	-
5	NC	-
6	R/B2#	NAND1_RB2#_3V3
7	R/B#	NAND1_RB#_3V3
8	RE#	NAND1_RE#_3V3
9	CE#	NAND1_CE#_3V3
10	CE2#	NAND1_CE2#_3V3
11	NC	-
12	Vcc	VCC_3V3
13	Vss	Gnd
14	NC	-
15	NC	-
16	CLE	NAND1_CLE_3V3
17	ALE	NAND1_ALE_3V3
18	WE#	NAND1_WE#_3V3
19	WP#	NAND1_WP#_3V3
20	NC	-
21	NC	-
22	NC	-
23	NC	-
24	NC	-
25	DNU	-
26	DNU	-
27	NC	-
28	NC	-
29	I/O 0	NAND1_IO0_3V3
30	I/O 1	NAND1_IO1_3V3
31	I/O 2	NAND1_IO2_3V3
32	I/O 3	NAND1_IO3_3V3
33	NC	-
34	NC	-
35	NC	-
36	Vss	Gnd
37	Vcc	VCC_3V3
38	DNU or Vss	-
39	NC	-
40	NC	-
41	I/0 4	NAND1_IO4_3V3
42	I/0 5	NAND1_IO5_3V3
43	I/0 6	NAND1_IO6_3V3
44	I/0 7	NAND1_IO7_3V3
45	NC	-
46	NC	-
47	NC	-
48	DNU	-

Micron MT29 Chip2		
1	NC	-
2	NC	-
3	NC	-
4	NC	-
5	NC	-
6	R/B2#	NAND2_RB2#_3V3
7	R/B#	NAND2_RB#_3V3
8	RE#	NAND2_RE#_3V3
9	CE#	NAND2_CE#_3V3
10	CE2#	NAND2_CE2#_3V3
11	NC	-
12	Vcc	VCC_3V3
13	Vss	Gnd
14	NC	-
15	NC	-
16	CLE	NAND2_CLE_3V3
17	ALE	NAND2_ALE_3V3
18	WE#	NAND2_WE#_3V3
19	WP#	NAND2_WP#_3V3
20	NC	-
21	NC	-
22	NC	-
23	NC	-
24	NC	-
25	DNU	-

Xilinx XC2C256 CPLD

	I/O BANK 2					I/O BANK 1	
	FT256	MC	FB	FB	MC	FT256	
	B3	1	1	5	1	R1	NAND1_RB2#_1V8
	B4	2	1	5	2	N4	NAND1_RB#_1V8
	C4	3	1(GSR)	5	3	N2	NAND1_RE#_1V8
	A2	4	1	(GCK1)5	4	M3	NAND1_CE#_1V8
	A3	5	1	5	5	P1	NAND1_CE2#_1V8
	A4	6	1	(GCK0)5	6	M2	NAND1_CLE_1V8
	B5	12	1	5	12	L3	NAND1_ALE_1V8
	A5	13	1	5	13	N1	NAND1_WE#_1V8
	E8	14	1	5	14	L4	NAND1_WP#_1V8
	B6	15	1	5	15	M1	NAND1_IO0_1V8
	C7	16	1	5	16	L5	NAND1_IO1_1V8
	D3	1	2(GTS2)	6	1	N3	NAND1_IO2_1V8
	C3	2	2	(CDRST)6	2	P2	NAND1_IO3_1V8
	E3	3	2(GTS3)	6	3	P4	NAND1_IO4_1V8
	B2	4	2	(GCK2)6	4	P5	NAND1_IO5_1V8
	D4	5	2(GTS0)	6	5	R2	NAND1_IO6_1V8
	D2	6	2	6	6	T1	NAND1_IO7_1V8
	E5	12	2(GTS1)	(DGE)6	12	T2	NAND2_RB2#_1V8
	B1	13	2	6	13	N5	NAND2_RB#_1V8
	E4	14	2	6	14	R4	NAND2_RE#_1V8
	C1	15	2	6	15	M5	NAND2_CE#_1V8
	E2	16	2	6	16	R5	NAND2_CE2#_1V8
	A6	1	3	7	1	K4	NAND2_CLE_1V8
	D7	2	3	7	2	L2	NAND2_ALE_1V8
NAND1_RB2#_3V3	B7	3	3	7	3	K3	NAND2_WE#_1V8
NAND1_RB#_3V3	E9	4	3	7	4	L1	NAND2_WP#_1V8
NAND1_RE#_3V3	A7	5	3	7	5	K5	NAND2_IO0_1V8
NAND1_CE#_3V3	D8	6	3	7	6	K2	NAND2_IO1_1V8
NAND1_CE2#_3V3	B8	12	3	7	11	J4	NAND2_IO2_1V8
NAND1_CLE_3V3	C8	13	3	7	12	K1	NAND2_IO3_1V8
NAND1_ALE_3V3	A8	14	3	7	13	J3	NAND2_IO4_1V8
NAND1_WE#_3V3	E11	15	3	7	14	J2	NAND2_IO5_1V8
NAND1_WP#_3V3	E10	16	3	7	15	J5	NAND2_IO6_1V8
NAND2_RB2#_3V3	F2	1	4	7	16	J1	NAND2_IO7_1V8
NAND2_RB#_3V3	F3	2	4	8	1	R6	NAND3_RB2#_1V8
NAND2_RE#_3V3	G4	3	4	8	2	N6	NAND3_RB#_1V8
NAND2_CE#_3V3	G3	4	4	8	3	R3	NAND3_RE#_1V8
NAND2_CE2#_3V3	F5	5	4	8	4	M6	NAND3_CE#_1V8
NAND2_CLE_3V3	G5	6	4	8	5	T3	NAND3_CE2#_1V8
NAND2_ALE_3V3	H2	12	4	8	6	P6	NAND3_CLE_1V8
NAND2_WE#_3V3	H4	13	4	8	11	T4	NAND3_ALE_1V8
NAND2_WP#_3V3	H3	14	4	8	12	P7	NAND3_WE#_1V8
NAND3_RB2#_3V3	H1	15	4	8	13	T5	NAND3_WP#_1V8
NAND3_RB#_3V3	H5	16	4	8	14	N7	NAND3_IO0_1V8
NAND3_RE#_3V3	B13	1	9	8	15	R7	NAND3_IO1_1V8
NAND3_CE#_3V3	B14	2	9	8	16	M7	NAND3_IO2_1V8
NAND3_CE2#_3V3	C13	3	9	13	1	R15	NAND3_IO3_1V8
NAND3_CLE_3V3	A15	4	9	13	2	T16	NAND3_IO4_1V8
NAND3_ALE_3V3	C12	5	9	13	3	N14	NAND3_IO5_1V8
NAND3_WE#_3V3	B12	6	9	13	4	R16	NAND3_IO6_1V8
NAND3_WP#_3V3	D13	11	9	13	5	N15	NAND3_IO7_1V8
NAND4_RB2#_3V3	A14	12	9	13	6	M15	NAND4_RB2#_1V8
NAND4_RB#_3V3	E13	13	9	13	12	M13	NAND4_RB#_1V8
NAND4_RE#_3V3	A13	14	9	13	13	P16	NAND4_RE#_1V8
NAND4_CE#_3V3	C11	15	9	13	14	N16	NAND4_CE#_1V8
NAND4_CE2#_3V3	A12	16	9	13	15	L14	NAND4_CE2#_1V8

FPGA DIMM socket

NAND1_CLE_1V8	4	DQ4	DQ32	123	NAND3_CLE_1V8
NAND1_ALE_1V8	5	DQ0	DQ36	124	NAND3_ALE_1V8
NAND1_WE#_1V8	6	DQ5	DQ33	125	NAND3_WE#_1V8
NAND1_WP#_1V8	7	DQ1	DQ37	126	NAND3_WP#_1V8
NAND1_RE#_1V8	10	DM0	DM4	130	NAND4_CLE_1V8
NAND1_CE#_1V8	11	DQS0_N	DQS4_N	129	NAND4_ALE_1V8
NAND1_CE2#_1V8	13	DQS0_P	DQS4_P	131	NAND4_WE#_1V8
NAND1_IO0_1V8	14	DQ6	DQ38	134	NAND3_IO0_1V8
NAND1_IO1_1V8	16	DQ7	DQ34	135	NAND3_IO1_1V8
NAND1_IO2_1V8	17	DQ2	DQ39	136	NAND3_IO2_1V8
NAND1_IO3_1V8	19	DQ3	DQ35	137	NAND3_IO3_1V8
NAND1_IO4_1V8	20	DQ12	DQ44	140	NAND3_IO4_1V8
NAND1_IO5_1V8	22	DQ13	DQ40	141	NAND3_IO5_1V8
NAND1_IO6_1V8	23	DQ8	DQ45	142	NAND3_IO6_1V8
NAND1_IO7_1V8	25	DQ9	DQ41	143	NAND3_IO7_1V8
NAND2_RE#_1V8	26	DM1	DM5	147	NAND4_WP#_1V8
NAND2_CE#_1V8	29	DQS1_N	DQS5_N	146	NAND4_IO0_1V8
NAND2_CE2#_1V8	31	DQS1_P	DQS5_P	148	NAND4_IO1_1V8
NAND1_RB2#_1V8	35	DQ10	DQ42	151	NAND4_IO2_1V8
NAND1_RB#_1V8	36	DQ14	DQ46	152	NAND4_IO3_1V8
NAND2_RB2#_1V8	37	DQ11	DQ43	153	NAND4_IO4_1V8
NAND2_RB#_1V8	38	DQ15	DQ47	154	NAND4_IO5_1V8
NAND2_CLE_1V8	43	DQ16	DQ48	157	NAND4_IO6_1V8
NAND2_ALE_1V8	44	DQ20	DQ52	158	NAND4_IO7_1V8
NAND2_WE#_1V8	45	DQ17	DQ49	159	NAND1_DIR
NAND2_WP#_1V8	46	DQ21	DQ53	160	NAND2_DIR
NAND3_RE#_1V8	49	DQS2_N	DQS6_N	167	SDRAM_DQS#
NAND3_CE#_1V8	51	DQS2_P	DQS6_P	169	SDRAM_DQS
NAND3_CE2#_1V8	52	DM2	DM6	170	NAND3_DIR
NAND2_IO0_1V8	55	DQ18	DQ50	173	SDRAM_DQ0
NAND2_IO1_1V8	56	DQ22	DQ54	174	SDRAM_DQ1
NAND2_IO2_1V8	57	DQ19	DQ51	175	SDRAM_DQ2
NAND2_IO3_1V8	58	DQ23	DQ55	176	SDRAM_DQ3
NAND2_IO4_1V8	61	DQ24	DQ56	179	SDRAM_DQ4
NAND2_IO5_1V8	62	DQ28	DQ60	180	SDRAM_DQ5
NAND2_IO6_1V8	63	DQ25	DQ57	181	SDRAM_DQ6
NAND2_IO7_1V8	64	DQ29	DQ61	182	SDRAM_DQ7
NAND4_RE#_1V8	67	DM3	DM7	185	NAND4_DIR
NAND4_CE#_1V8	68	DQS3_N	DQS7_N	186	SDRAM_RDQS#
NAND4_CE2#_1V8	70	DQS3_P	DQS7_P	188	SDRAM_DM_RDQS
NAND3_RB2#_1V8	73	DQ26	DQ58	189	NAND1_DIR
NAND3_RB#_1V8	74	DQ30	DQ59	191	NAND2_DIR
NAND4_RB2#_1V8	75	DQ27	DQ62	192	NAND3_DIR
NAND4_RB#_1V8	76	DQ31	DQ63	194	NAND4_DIR
Gnd	3	VSS1			
Gnd	9	VSS2	VREF	1	SDRAM_VREF
Gnd	15	VSS3	VDDSPD	199	
Gnd	21	VSS4			
Gnd	27	VSS5	VDD1	103	SDRAM_VDD_1V8
Gnd	33	VSS6	VDD2	111	SDRAM_VDD_1V8
Gnd	39	VSS7	VDD3	117	SDRAM_VDD_1V8
Gnd	41	VSS8	VDD4	81	SDRAM_VDD_1V8
Gnd	47	VSS9	VDD5	87	SDRAM_VDD_1V8
Gnd	121	VSS10	VDD6	95	SDRAM_VDD_1V8
Gnd	127	VSS11	VDD7	104	SDRAM_VDD_1V8
Gnd	133	VSS12	VDD8	112	SDRAM_VDD_1V8
Gnd	139	VSS13	VDD9	118	SDRAM_VDD_1V8
Gnd	145	VSS14	VDD10	82	SDRAM_VDD_1V8
Gnd	149	VSS15	VDD11	88	SDRAM_VDD_1V8
Gnd	53	VSS16	VDD12	96	SDRAM_VDD_1V8
Gnd	59	VSS17			
Gnd	65	VSS18			
Gnd	71	VSS19			
Gnd	77	VSS20	CK0_P	30	SDRAM_CK C66

26 DNU	-
27 NC	-
28 NC	-
29 I/O 0	NAND2_IO0_3V3
30 I/O 1	NAND2_IO1_3V3
31 I/O 2	NAND2_IO2_3V3
32 I/O 3	NAND2_IO3_3V3
33 NC	-
34 NC	-
35 NC	-
36 Vss	Gnd
37 Vcc	VCC_3V3
38 DNU or Vss	-
39 NC	-
40 NC	-
41 I/0 4	NAND2_IO4_3V3
42 I/0 5	NAND2_IO5_3V3
43 I/0 6	NAND2_IO6_3V3
44 I/0 7	NAND2_IO7_3V3
45 NC	-
46 NC	-
47 NC	-
48 DNU	-

Micron MT29 NAND Chip3

1 NC	-
2 NC	-
3 NC	-
4 NC	-
5 NC	-
6 R/B2#	NAND3_RB2#_3V3
7 R/B#	NAND3_RB#_3V3
8 RE#	NAND3_RE#_3V3
9 CE#	NAND3_CE#_3V3
10 CE2#	NAND3_CE2#_3V3
11 NC	-
12 Vcc	VCC_3V3
13 Vss	Gnd
14 NC	-
15 NC	-
16 CLE	NAND3_CLE_3V3
17 ALE	NAND3_ALE_3V3
18 WE#	NAND3_WE#_3V3
19 WP#	NAND3_WP#_3V3
20 NC	-
21 NC	-
22 NC	-
23 NC	-
24 NC	-
25 DNU	-
26 DNU	-
27 NC	-
28 NC	-
29 I/O 0	NAND3_IO0_3V3
30 I/O 1	NAND3_IO1_3V3
31 I/O 2	NAND3_IO2_3V3
32 I/O 3	NAND3_IO3_3V3
33 NC	-
34 NC	-
35 NC	-
36 Vss	Gnd
37 Vcc	VCC_3V3
38 DNU or Vss	-
39 NC	-
40 NC	-
41 I/0 4	NAND3_IO4_3V3
42 I/0 5	NAND3_IO5_3V3
43 I/0 6	NAND3_IO6_3V3
44 I/0 7	NAND3_IO7_3V3
45 NC	-
46 NC	-
47 NC	-
48 DNU	-

Micron MT29 Chip 4

1 NC	-
2 NC	-
3 NC	-
4 NC	-
5 NC	-
6 R/B2#	NAND4_RB2#_3V3
7 R/B#	NAND4_RB#_3V3
8 RE#	NAND4_RE#_3V3
9 CE#	NAND4_CE#_3V3
10 CE2#	NAND4_CE2#_3V3
11 NC	-
12 Vcc	VCC_3V3
13 Vss	Gnd
14 NC	-
15 NC	-
16 CLE	NAND4_CLE_3V3
17 ALE	NAND4_ALE_3V3

NAND4_CLE_3V3	A16	1	10	13	16	M14	NAND4_CLE_1V8
NAND4_ALE_3V3	B15	2	10	14	1	P15	NAND4_ALE_1V8
NAND4_WE#_3V3	C14	3	10	14	2	P14	NAND4_WE#_1V8
NAND4_WP#_3V3	G11	4	10	14	3	P13	NAND4_WP#_1V8
NAND2_IO0_3V3	B16	5	10	14	4	R13	NAND4_IO0_1V8
NAND2_IO1_3V3	D15	6	10	14	5	N13	NAND4_IO1_1V8
NAND2_IO2_3V3	E14	11	10	14	6	R14	NAND4_IO2_1V8
NAND2_IO3_3V3	C16	12	10	14	12	T15	NAND4_IO3_1V8
NAND2_IO4_3V3	F14	13	10	14	13	R12	NAND4_IO4_1V8
NAND2_IO5_3V3	F13	14	10	14	14	N11	NAND4_IO5_1V8
NAND2_IO6_3V3	E15	15	10	14	15	M11	NAND4_IO6_1V8
NAND2_IO7_3V3	G13	16	10	14	16	N10	NAND4_IO7_1V8
NAND1_IO0_3V3	B11	1	11	15	1	L15	NAND1_DIR
NAND1_IO1_3V3	D11	2	11	15	2	L13	NAND2_DIR
NAND1_IO2_3V3	A11	3	11	15	3	M12	NAND3_DIR
NAND1_IO3_3V3	D10	4	11	15	4	M16	NAND4_DIR
NAND1_IO4_3V3	B10	5	11	15	5	K14	
NAND1_IO5_3V3	E12	6	11	15	6	L16	
NAND1_IO6_3V3	F12	11	11	15	11	K15	CPLD_RESV_1
NAND1_IO7_3V3	B9	12	11	15	12	L12	CPLD_RESV_2
NAND3_IO0_3V3	C9	13	11	15	13	K16	CPLD_RESV_3
NAND3_IO1_3V3	C10	14	11	15	14	J14	CPLD_RESV_4
NAND3_IO2_3V3	A9	15	11	15	15	J15	CPLD_RESV_5
NAND3_IO3_3V3	D9	16	11	15	16	J13	CPLD_RESV_6
NAND3_IO4_3V3	F15	1	12	16	1	P10	CPLD_RESV_7
NAND3_IO5_3V3	G14	2	12	16	2	R10	CPLD_RESV_8
NAND3_IO6_3V3	E16	3	12	16	3	T10	CPLD_RESV_9
NAND3_IO7_3V3	H12	4	12	16	4	R9	CPLD_RESV_10
NAND4_IO0_3V3	F16	5	12	16	5	N9	CPLD_RESV_11
NAND4_IO1_3V3	H16	6	12	16	6	M8	
NAND4_IO2_3V3	G15	11	12	16	11	T8	
NAND4_IO3_3V3	H13	12	12	16	12	P8	
NAND4_IO4_3V3	G16	13	12	16	13	R8	
NAND4_IO5_3V3	H14	14	12	16	14	T7	
NAND4_IO6_3V3	H15	15	12	16	15	N8	
NAND4_IO7_3V3	J12	16	12	16	16	T6	
CPLD_JTAG_TCK	P12	TCK			VCCIO1	J6	SDRAM_VDD_1V8
CPLD_JTAG_TDI	R11	TDI			VCCIO1	K6	SDRAM_VDD_1V8
CPLD_JTAG_TDO	A10	TDO			VCCIO1	L7	SDRAM_VDD_1V8
CPLD_JTAG_TMS	N12	TMS			VCCIO1	L8	SDRAM_VDD_1V8
CPLD_JTAG_VCC	F4	VCCAUX(JTAG)			VCCIO1	J11	SDRAM_VDD_1V8
					VCCIO1	K11	SDRAM_VDD_1V8
CPLD_VCC_1V8	P3	VCC			VCCIO1	L10	SDRAM_VDD_1V8
CPLD_VCC_1V8	K13	VCC			VCCIO1	L9	SDRAM_VDD_1V8
CPLD_VCC_1V8	D12	VCC					
CPLD_VCC_1V8	D5	VCC			GND	F11	Gnd
					GND	F6	Gnd
VCC_3V3	F7	VCCIO2			GND	G10	Gnd
VCC_3V3	F8	VCCIO2			GND	G7	Gnd
VCC_3V3	G6	VCCIO2			GND	G8	Gnd
VCC_3V3	H6	VCCIO2			GND	G9	Gnd
VCC_3V3	F10	VCCIO2			GND	H10	Gnd
VCC_3V3	F9	VCCIO2			GND	H7	Gnd
VCC_3V3	H11	VCCIO2			GND	H8	Gnd
					GND	H9	Gnd
					GND	J10	Gnd
-	A1	NC			GND	J7	Gnd
-	C2	NC			GND	J8	Gnd
-	E6	NC			GND	J9	Gnd
-	D1	NC			GND	K10	Gnd
-	E1	NC			GND	K7	Gnd
-	G2	NC			GND	K8	Gnd
-	F1	NC			GND	K9	Gnd
-	G1	NC			GND	L11	Gnd
-	M4	NC			GND	L6	Gnd
-	T9	NC					
-	P9	NC					
-	M9	NC			NC	D16	-
-	M10	NC			NC	G12	-
-	T11	NC			NC	C15	-
-	T12	NC			NC	D14	-
-	T13	NC			NC	D6	-
-	P11	NC			NC	C6	-
-	T14	NC			NC	E7	-
-	J16	NC			NC	C5	-
	K12	NC					

Notes:
1 GTS = global output enable, GSR = global reset/set, GCK = global clock, CDRST = clock divide reset, DGE = DataGATE enable.
2 GTS, GSR and GCK pins can be used for general purpose I/O.

JTAG Header Digikey: S6009-07-ND

Gnd	155 VSS21	CK0_N 32	SDRAM_CK#	C66
Gnd	161 VSS22			
Gnd	165 VSS23	CK1_P 164	CPLD_RESV_2	C67
Gnd	171 VSS24	CK1_N 166	CPLD_RESV_3	C67
Gnd	177 VSS25			
Gnd	183 VSS26			
Gnd	187 VSS27	CKE1 80	CPLD_RESV_4	Pull Down
Gnd	193 VSS28	CKE0 79	SDRAM_CKE	Pull Down
Gnd	2 VSS29			
Gnd	8 VSS30			
Gnd	12 VSS31	A13 116	CPLD_RESV_5	VTTDDR Swing
Gnd	18 VSS32	A12 89	SDRAM_A12	VTTDDR Swing
Gnd	24 VSS33	A11 90	SDRAM_A11	VTTDDR Swing
Gnd	28 VSS34	A9 91	SDRAM_A9	VTTDDR Swing
Gnd	34 VSS35	A8 93	SDRAM_A8	VTTDDR Swing
Gnd	40 VSS36	A7 92	SDRAM_A7	VTTDDR Swing
Gnd	42 VSS37	A6 94	SDRAM_A6	VTTDDR Swing
Gnd	48 VSS38	A5 97	SDRAM_A5	VTTDDR Swing
Gnd	122 VSS39	A4 98	SDRAM_A4	VTTDDR Swing
Gnd	128 VSS40	A3 99	SDRAM_A3	VTTDDR Swing
Gnd	132 VSS41	A2 100	SDRAM_A2	VTTDDR Swing
Gnd	138 VSS42	A1 101	SDRAM_A1	VTTDDR Swing
Gnd	144 VSS43	A0 102	SDRAM_A0	VTTDDR Swing
Gnd	150 VSS44	A10/AP 105	SDRAM_A10	VTTDDR Swing
Gnd	54 VSS45			
Gnd	60 VSS46			
Gnd	66 VSS47	BA2 85	CPLD_RESV_6	VTTDDR Swing
Gnd	72 VSS48	BA1 106	SDRAM_BA1	VTTDDR Swing
Gnd	78 VSS49	BA0 107	SDRAM_BA0	VTTDDR Swing
Gnd	156 VSS50	RAS_B 108	SDRAM_RAS#	VTTDDR Swing
Gnd	162 VSS51	WE_B 109	SDRAM_WE#	VTTDDR Swing
Gnd	168 VSS52	CAS_B 113	SDRAM_CAS#	VTTDDR Swing
Gnd	172 VSS53			
Gnd	178 VSS54	S0_B 110	SDRAM_CS#	VTTDDR Swing
Gnd	184 VSS55	S1_B 115	CPLD_RESV_7	VTTDDR Swing
Gnd	190 VSS56			
Gnd	196 VSS57	ODT0 114	SDRAM_ODT	Pull Down
		ODT1 119	CPLD_RESV_8	Pull Down
NC	69 NC8			
NC	83 NC3			
NC	163 NC7	SDA 195	CPLD_RESV_9	Pull Up
NC	50 NC6	SCL 197	CPLD_RESV_1	Pull Up
NC	120 NC4	SA0 198		Gnd
NC	84 NC2	SA1 200		Gnd
NC	86 NC1			

"P2"

Note-- the DIMM slot is being repurposed so the signal names can be confusing. Be careful!

SDRAM 1
Micron 60 FBGA x4 MT47H64M8 DDR

A1	VDD	SDRAM_VDD_1V8	
A2	S#/NF	SDRAM_RDQS#	
A3	VSS	Gnd	
A7	VSSQ	Gnd	
A8	DQS#	SDRAM_DQS#	
A9	VDDQ	SDRAM_VDDQ_1	DQ power supply: 1.8V ±0.1V. Isolated o
B1	DQ6	SDRAM_DQ6	
B2	VSSQ	Gnd	
B3	/RDQS	SDRAM_DM_RDQS	
B7	DQS	SDRAM_DQS	
B8	VSSQ	Gnd	
B9	DQ7	SDRAM_DQ7	
C1	VDDQ	SDRAM_VDDQ_1V8	
C2	DQ1	SDRAM_DQ1	
C3	VDDQ	SDRAM_VDDQ_1V8	
C7	VDDQ	SDRAM_VDDQ_1V8	
C8	DQ0	SDRAM_DQ0	
C9	VDDQ	SDRAM_VDDQ_1V8	
D1	DQ4	SDRAM_DQ4	
D2	VSSQ	Gnd	
D3	DQ3	SDRAM_DQ3	
D7	DQ2	SDRAM_DQ2	
D8	VSSQ	Gnd	
D9	DQ5	SDRAM_DQ5	
E1	VDDL	SDRAM_VDD_1V	DLL power supply: 1.8V ±0.1V.
E2	VREF	SDRAM_VREF	
E3	VSS	Gnd	
E7	VSSDL	Gnd	
E8	CK	SDRAM_CK	
E9	VDD	SDRAM_VDD_1V8	
F2	CKE	SDRAM_CKE	
F3	WE#	SDRAM_WE#	
F7	RAS#	SDRAM_RAS#	
F8	CK#	SDRAM_CK#	
F9	ODT	SDRAM_ODT	

18	WE#	NAND4_WE#_3V3
19	WP#	NAND4_WP#_3V3
20	NC	-
21	NC	-
22	NC	-
23	NC	-
24	NC	-
25	DNU	-
26	DNU	-
27	NC	-
28	NC	-
29	I/O 0	NAND4_IO0_3V3
30	I/O 1	NAND4_IO1_3V3
31	I/O 2	NAND4_IO2_3V3
32	I/O 3	NAND4_IO3_3V3
33	NC	-
34	NC	-
35	NC	-
36	Vss	Gnd
37	Vcc	VCC_3V3
38	DNU or Vss	-
39	NC	-
40	NC	-
41	I/0 4	NAND4_IO4_3V3
42	I/0 5	NAND4_IO5_3V3
43	I/0 6	NAND4_IO6_3V3
44	I/0 7	NAND4_IO7_3V3
45	NC	-
46	NC	-
47	NC	-
48	DNU	-

Gnd	1	2	CPLD_JTAG_VCC
Gnd	3	4	CPLD_JTAG_TMS
Gnd	5	6	CPLD_JTAG_TCK
Gnd	7	8	CPLD_JTAG_TDO
Gnd	9	10	CPLD_JTAG_TDI
Gnd	11	12	-
Gnd	13	14	-

4 pin Power Header
Digikey: A30513-ND

VCC_3V3	1
Gnd	2
SDRAM_VDD_1V8	3
Gnd	4

2 Pin HDR (jumper)
Digikey: A30511-nd

CPLD_VCC_1V8	1
SDRAM_VDD_1	2

2 Pin HDR (jumper)
Digikey: A30511-nd

SDRAM_VDDQ_	1
SDRAM_VDD_1	2

470pF 0402 X7R
Digikey: 709-1085-1-ND

SDRAM_VDDQ_1V8	1
Gnd	2

470pF 0402 X7R
Digikey: 709-1085-1-ND

SDRAM_VDD_1V8	1
Gnd	2

470pF 0402 X7R
Digikey: 709-1085-1-ND

VCC_3V3	1
Gnd	2

100nF 0402 X5R
Digikey: 399-3027-1-ND

SDRAM_VDDQ_1V8	1
Gnd	2

100nF 0402 X5R
Digikey: 399-3027-1-ND

SDRAM_VDD_1V8	1
Gnd	2

100nF 0402 X5R
Digikey: 399-3027-1-ND

VCC_3V3	1
Gnd	2

Power
http://www.qimonda.com/static/download/promopages/AN_CD036_Design_Guide_for_DDR2_rev100_x2x.pdf

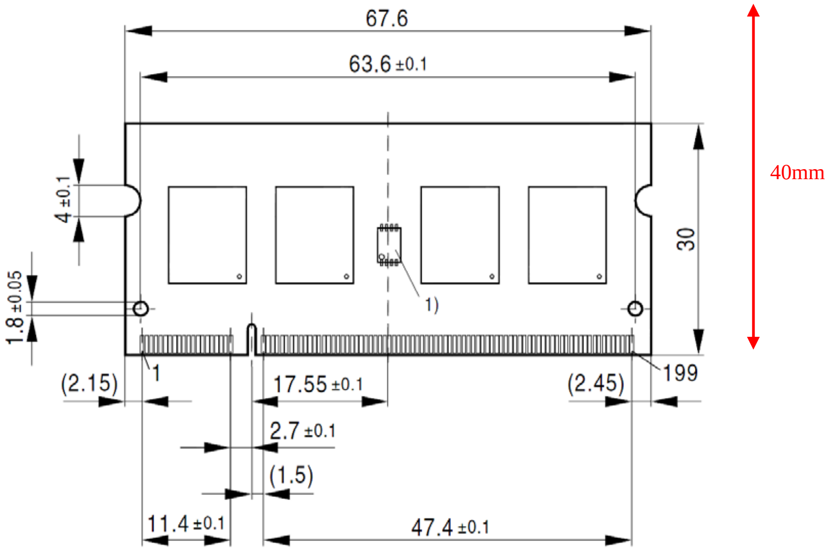
A typical selection would consist of a 470 pF, 2.7nF, 10 nF, 100nF, 470 nF, and a 10uF bypass capacitor from VDD and VDDQ to GND, along with a 470 pF and 100 nF placed from VREF to GND. This sequence of decoupling should be duplicated for each DDR2 chip on the board. If board space or cost is a concern, a simplified array of decoupling capacitors should include, of **at least two caps (470 pF, 100 nF) should be used for each supply voltage and VRE. All capacitors should be placed close to the device as possible and between the via and the power pin. If placed after the power pin, the effect of the decoupling cap is strongly reduced. All capacitors should be ceramic and should have an XR5 material rating. Package types are typically SMD 0402 or 0603.**

G1	RFU	-
G2	BA0	SDRAM_BA0
G3	BA1	SDRAM_BA1
G7	CAS#	SDRAM_CAS#
G8	CS#	SDRAM_CS#
H2	A10	SDRAM_A10
H3	A1	SDRAM_A1
H7	A2	SDRAM_A2
H8	A0	SDRAM_A0
H9	VDD	SDRAM_VDD_1V8
J1	VSS	Gnd
J2	A3	SDRAM_A3
J3	A5	SDRAM_A5
J7	A6	SDRAM_A6
J8	A4	SDRAM_A4
K2	A7	SDRAM_A7
K3	A9	SDRAM_A9
K7	A11	SDRAM_A11
K8	A8	SDRAM_A8
K9	VSS	Gnd
L1	VDD	SDRAM_VDD_1V8
L2	A12	SDRAM_A12
L3	RFU	-
L7	RFU	-
L8	RFU	-

Note-- make sure not to confuse DQ5 and DQS

DIMM Package Outline

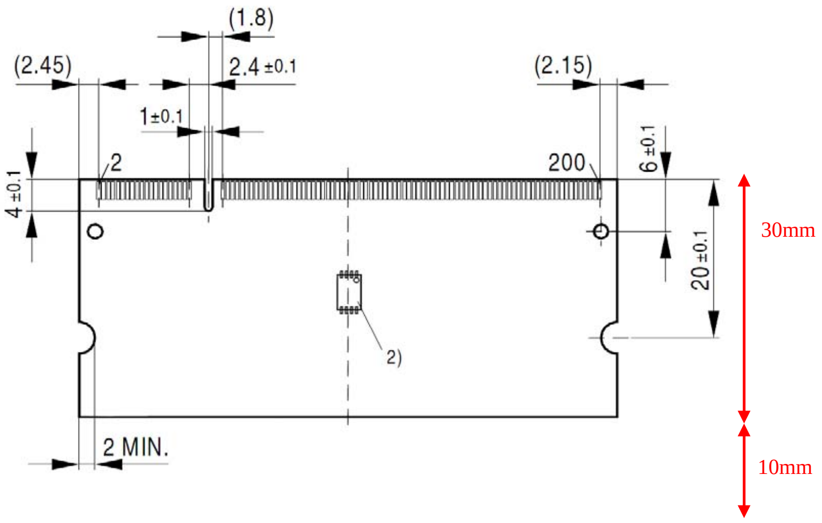
NOTE: Images not drawn to same scale.



FRONT (pin 1)

- DDR
- CPLD
- Connector
- Caps

Front clearance
(component thickness allowable)
- 20mm throughout



REAR (pin 2)

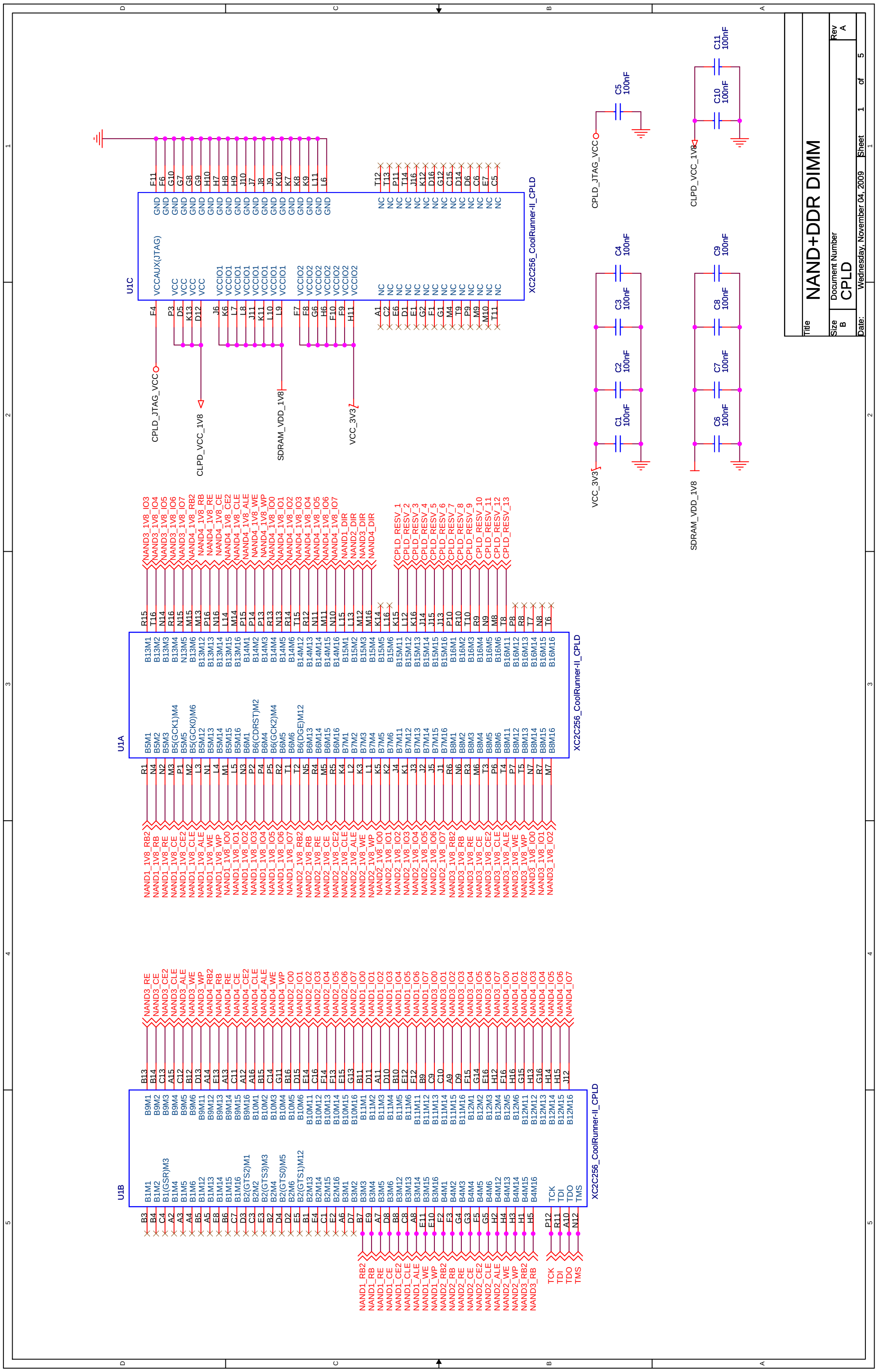
- NAND x 4
- Caps

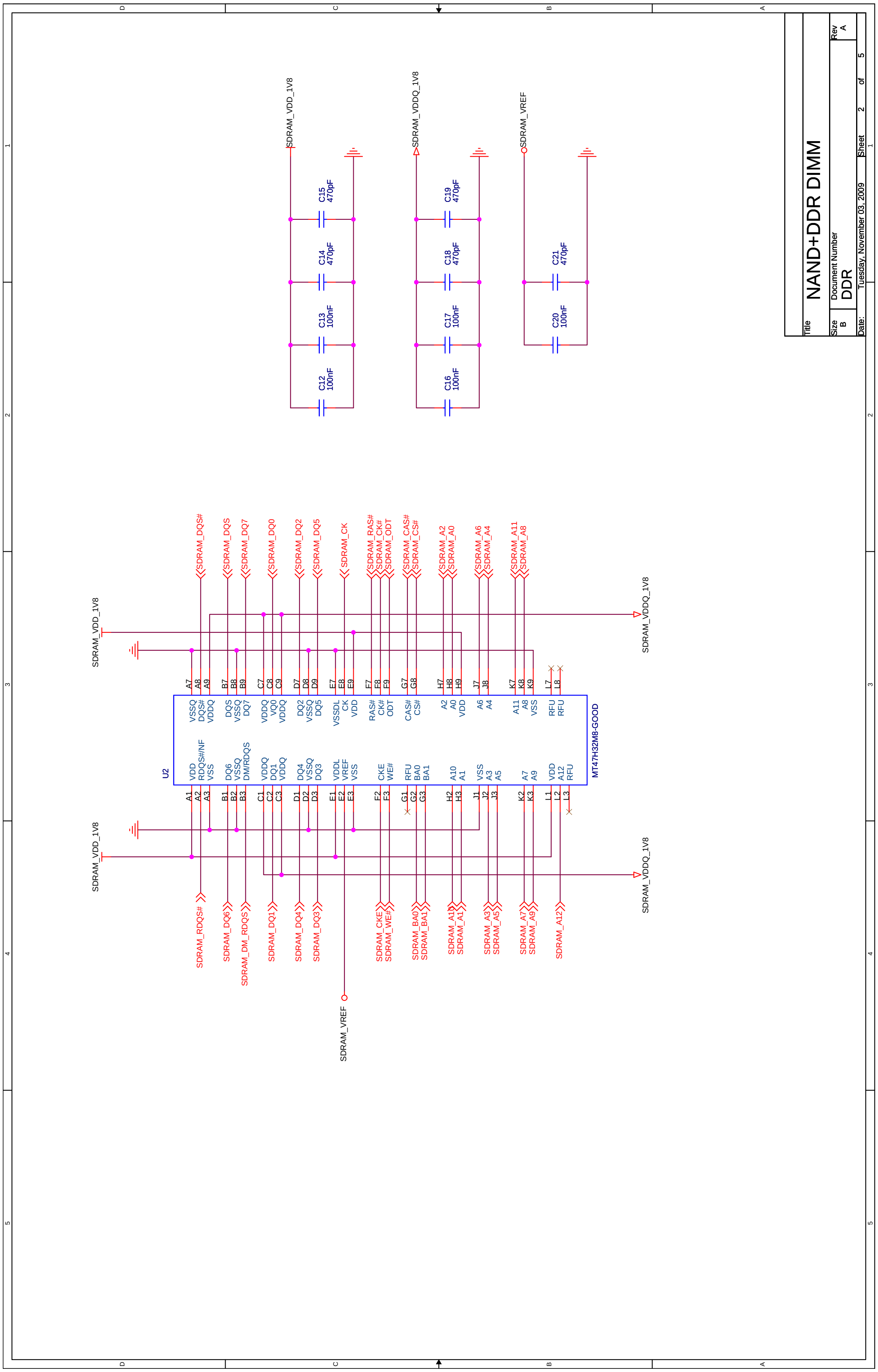
Rear clearance
(component thickness allowable)
- 1.2mm for first 30mm height
- 0mm for remaining 10mm height

Silkscreen:

- power and jumper pin nets
- serial number of DIMM starting at 0
- label nand1 through nand4

A.2 Schematic

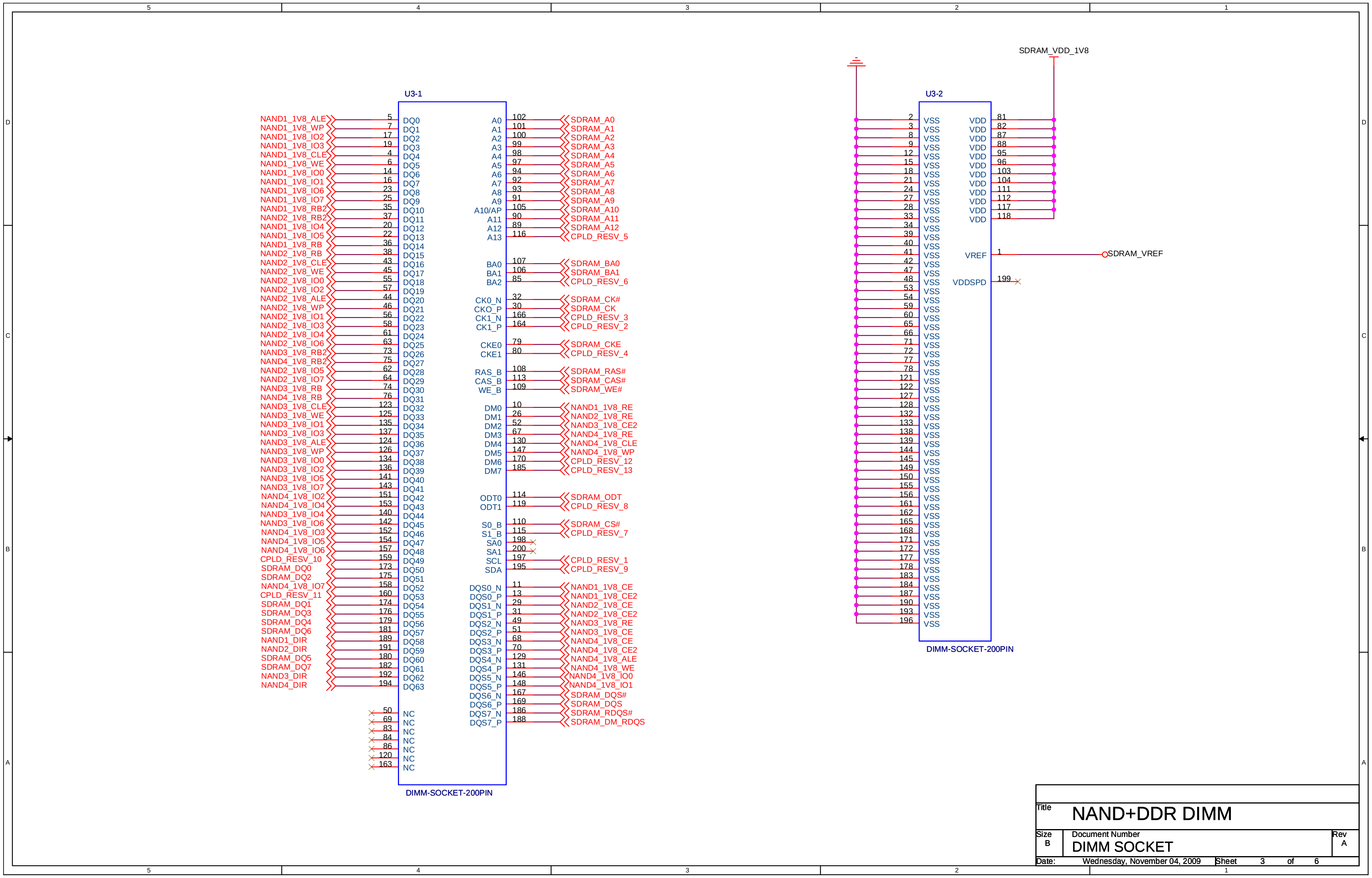




Title
NAND+DDR DIMM

Size B | Document Number DDR | Rev A

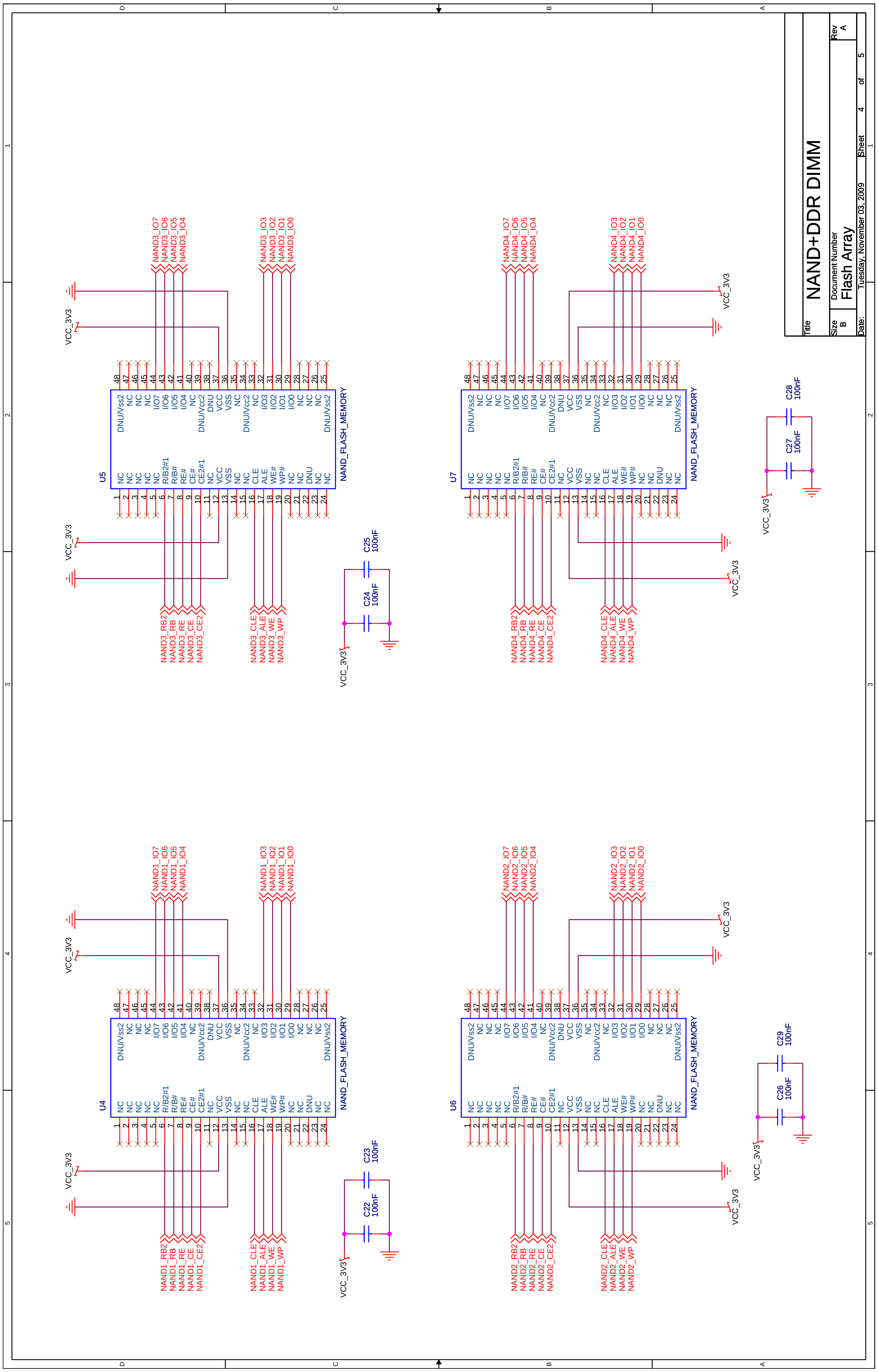
Date: Tuesday, November 03, 2009 | Sheet 2 of 5

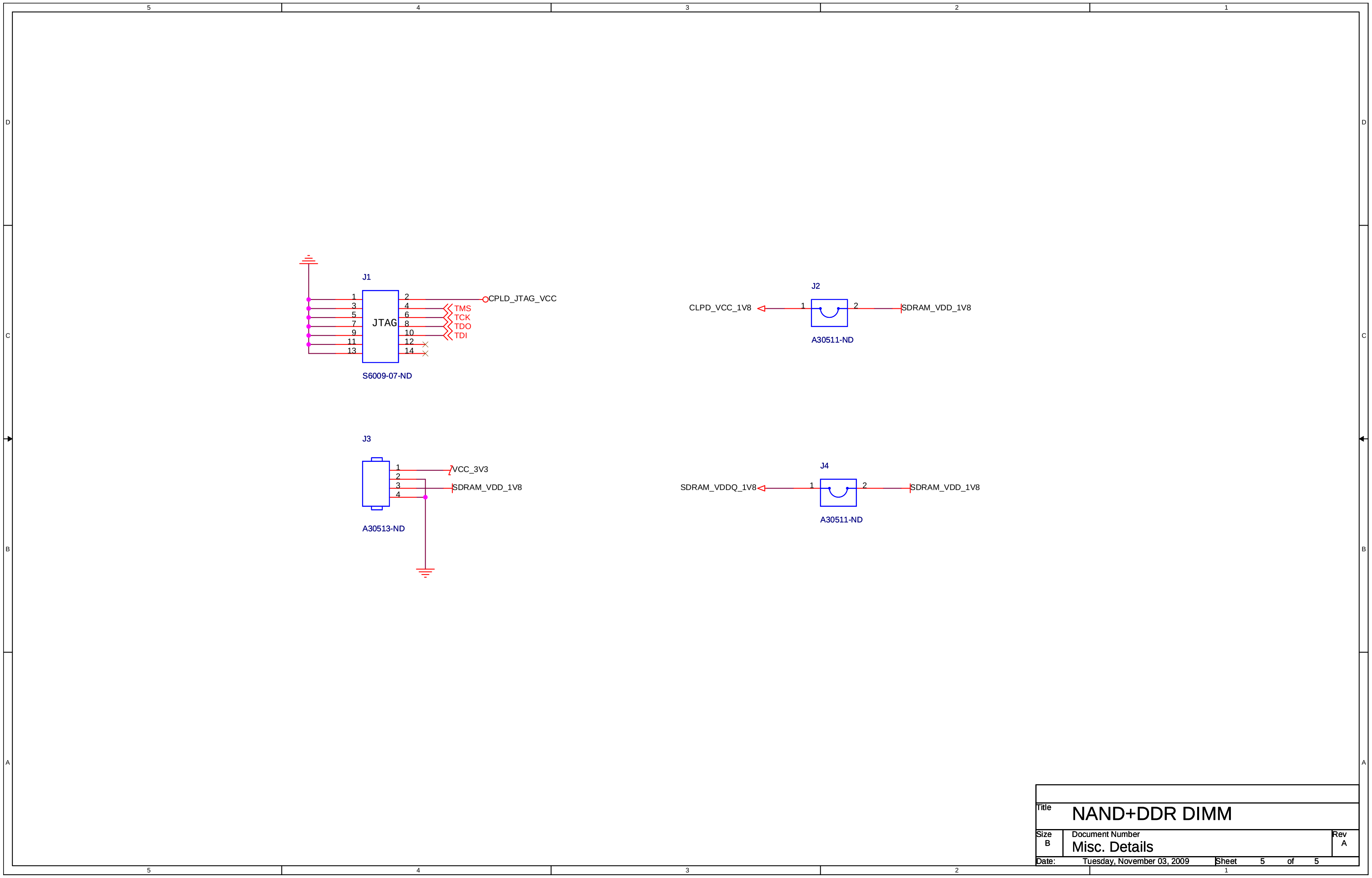


Title
NAND+DDR DIMM

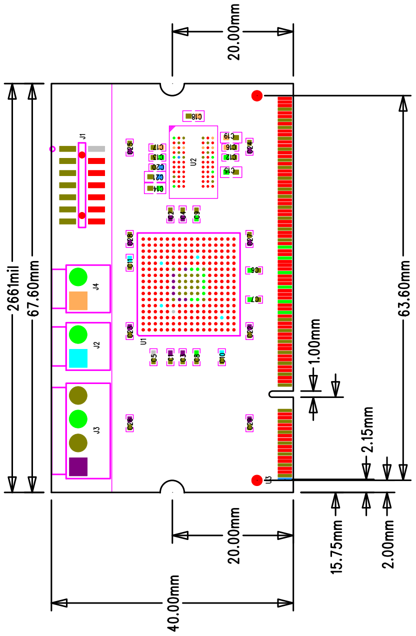
Size B | Document Number DIMM SOCKET | Rev A

Date: Wednesday, November 04, 2009 | Sheet 3 of 6

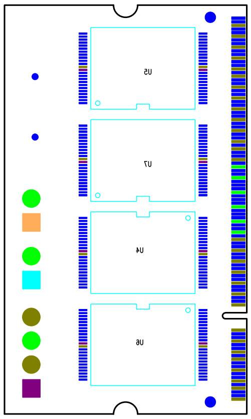

NAND+DDR DIMM
Title
Size
B
Document Number
Flash Array
Rev
A
Date: Tuesday, November 03, 2009
Sheet 4 of 5
U4
U5
U6
U7
NAND_FLASH_MEMORY
NC
NC
NC
NC
NC
R/B2#1
R/B#
RE#
CE#
CE2#1
NC
VCC
VSS
NC
NC
CLE
ALE
WE#
WP#
NC
NC
DNU
NC
NC
DNU/Vss2
NC
NC
NC
I/O7
I/O6
I/O5
I/O4
NC
DNU/Vcc2
DNU
VCC
VSS
NC
NC
DNU/Vcc2
I/O3
I/O2
I/O1
I/O0
NC
NC
NC
DNU/Vss2
NAND1_IO7
NAND1_IO6
NAND1_IO5
NAND1_IO4
NAND1_IO3
NAND1_IO2
NAND1_IO1
NAND1_IO0
NAND1_RB2
NAND1_RB
NAND1_RE
NAND1_CE
NAND1_CE2
NAND1_CLE
NAND1_ALE
NAND1_WE
NAND1_WP
NAND2_IO7
NAND2_IO6
NAND2_IO5
NAND2_IO4
NAND2_IO3
NAND2_IO2
NAND2_IO1
NAND2_IO0
NAND2_RB2
NAND2_RB
NAND2_RE
NAND2_CE
NAND2_CE2
NAND2_CLE
NAND2_ALE
NAND2_WE
NAND2_WP
NAND3_IO7
NAND3_IO6
NAND3_IO5
NAND3_IO4
NAND3_IO3
NAND3_IO2
NAND3_IO1
NAND3_IO0
NAND3_RB2
NAND3_RB
NAND3_RE
NAND3_CE
NAND3_CE2
NAND3_CLE
NAND3_ALE
NAND3_WE
NAND3_WP
NAND4_IO7
NAND4_IO6
NAND4_IO5
NAND4_IO4
NAND4_IO3
NAND4_IO2
NAND4_IO1
NAND4_IO0
NAND4_RB2
NAND4_RB
NAND4_RE
NAND4_CE
NAND4_CE2
NAND4_CLE
NAND4_ALE
NAND4_WE
NAND4_WP
VCC_3V3
C22 100nF
C23 100nF
C24 100nF
C25 100nF
C26 100nF
C27 100nF
C28 100nF
C29 100nF
A
B
C
D
1
2
3
4
5


J1
JTAG
1
3
5
7
9
11
13
2
4
6
8
10
12
14
CPLD_JTAG_VCC
TMS
TCK
TDO
TDI
S6009-07-ND
J2
CLPD_VCC_1V8
1
2
SDRAM_VDD_1V8
A30511-ND
J3
1
2
3
4
VCC_3V3
SDRAM_VDD_1V8
A30513-ND
J4
SDRAM_VDDQ_1V8
1
2
SDRAM_VDD_1V8
A30511-ND
Title
NAND+DDR DIMM
Size
B
Document Number
Misc. Details
Rev
A
Date:
Tuesday, November 03, 2009
Sheet
5
of
5
5
4
3
2
1
D
C
B
A

A.3 Placement

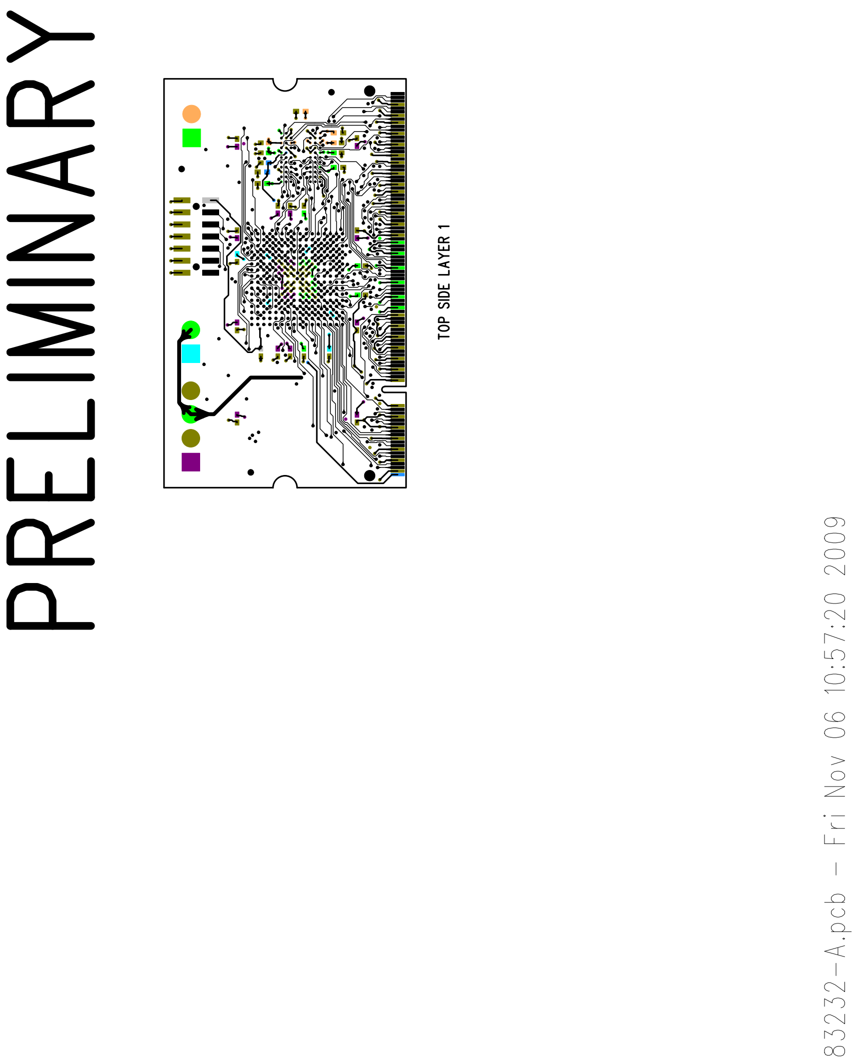


83232-01.pcb - Wed Nov 04 16:16:05 2009

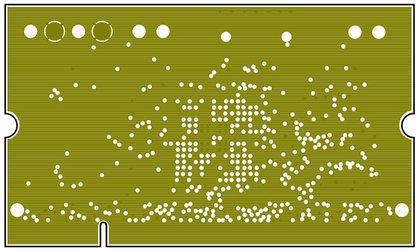


83232-01.pcb – Wed Nov 04 16:16:08 2009

A.4 Layers


PRELIMINARY
TOP SIDE LAYER 1
83232-A.pcb - Fri Nov 06 10:57:20 2009

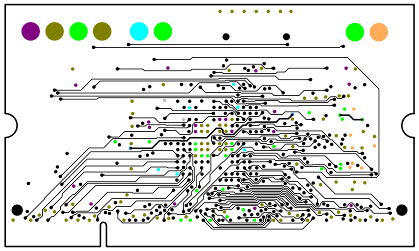
PRELIMINARY



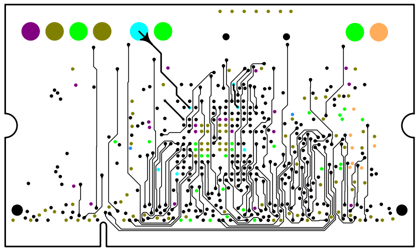
LAYER 2

83232-A.pcb - Fri Nov 06 10:57:25 2009

PRELIMINARY



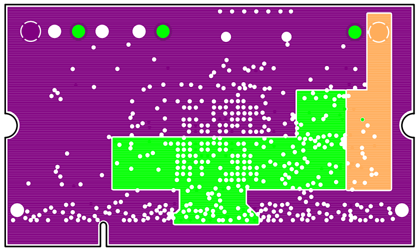
LAYER 3



LAYER 4

83232-A.pcb - Fri Nov 06 10:57:30 2009

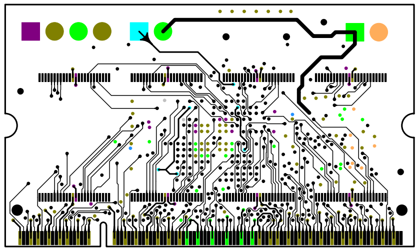
PRELIMINARY



LAYER 5

83232-A.pcb - Fri Nov 06 10:57:32 2009

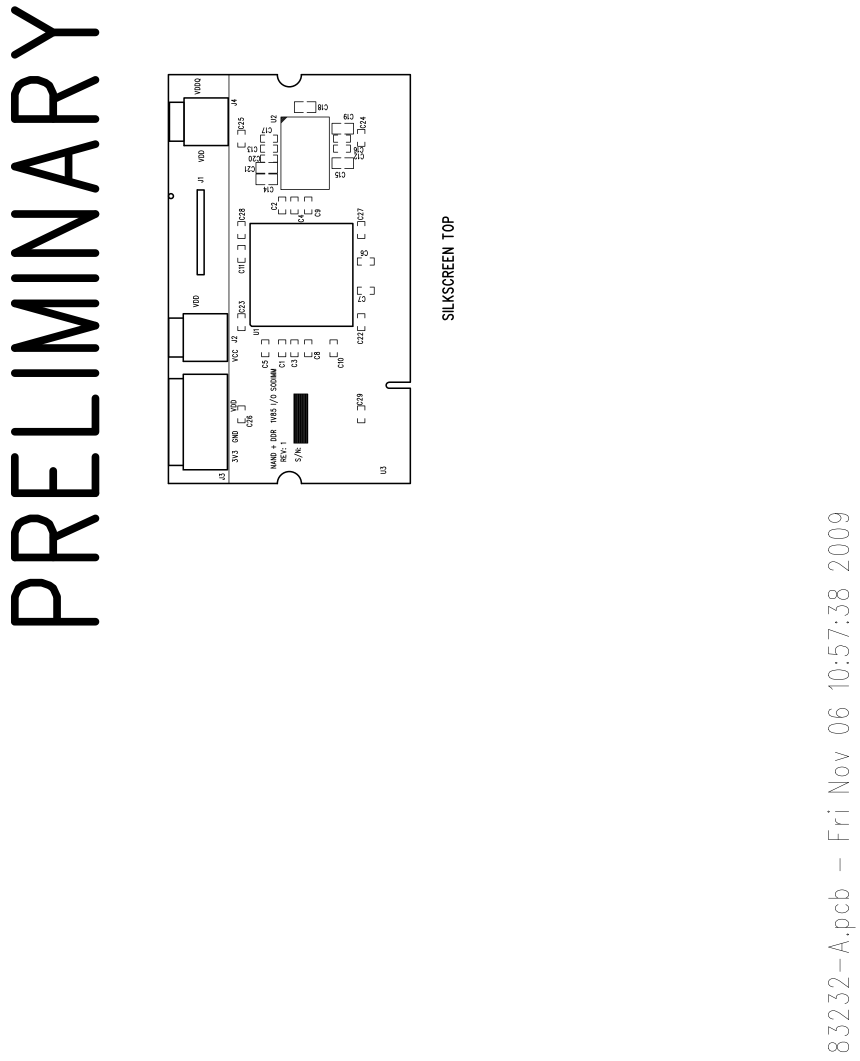
PRELIMINARY



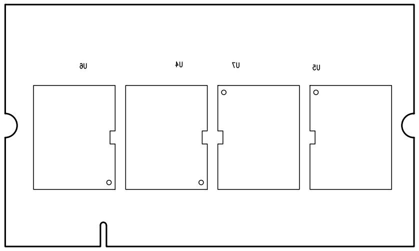
BOTTOM SIDE LAYER 6

83232-A.pcb - Fri Nov 06 10:57:34 2009

A.5 Silk


PRELIMINARY
SILKSCREEN TOP
83232-A.pcb - Fri Nov 06 10:57:38 2009
VDDQ
J4
VDD
J1
VDD
J2
VCC
J3
3V3
GND
VDD
NAND + DDR 1V85 I/O SODIMM
REV: 1
S/N:
U1
U2
U3
C1
C2
C3
C4
C5
C6
C7
C8
C9
C10
C11
C12
C13
C14
C15
C17
C18
C19
C20
C21
C22
C23
C24
C25
C26
C27
C28
C29

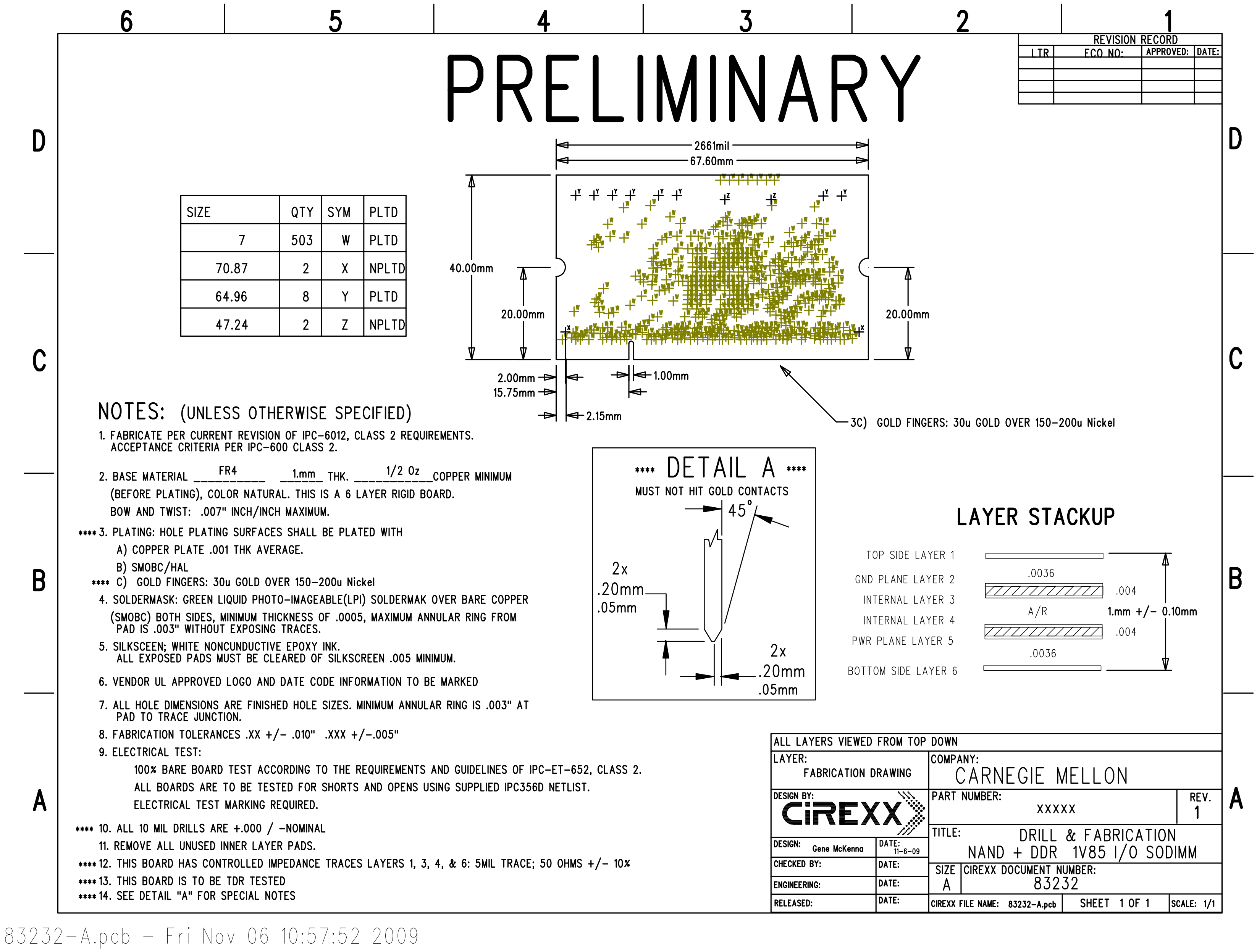
PRELIMINARY



SILKSCREEN BOTTOM

83232-A.pcb - Fri Nov 06 10:57:40 2009

A.6 Final Assembly



REVISION RECORD

LTR	ECO NO:	APPROVED:	DATE:

SIZE	QTY	SYM	PLTD
7	503	W	PLTD
70.87	2	X	NPLTD
64.96	8	Y	PLTD
47.24	2	Z	NPLTD

NOTES: (UNLESS OTHERWISE SPECIFIED)

1. FABRICATE PER CURRENT REVISION OF IPC-6012, CLASS 2 REQUIREMENTS. ACCEPTANCE CRITERIA PER IPC-600 CLASS 2.

2. BASE MATERIAL __FR4__ __1.mm__ THK. __1/2 Oz__COPPER MINIMUM (BEFORE PLATING), COLOR NATURAL. THIS IS A 6 LAYER RIGID BOARD. BOW AND TWIST: .007" INCH/INCH MAXIMUM.

**** 3. PLATING: HOLE PLATING SURFACES SHALL BE PLATED WITH
A) COPPER PLATE .001 THK AVERAGE.
B) SMOBC/HAL
**** C) GOLD FINGERS: 30u GOLD OVER 150-200u Nickel

4. SOLDERMASK: GREEN LIQUID PHOTO-IMAGEABLE(LPI) SOLDERMAK OVER BARE COPPER (SMOBC) BOTH SIDES, MINIMUM THICKNESS OF .0005, MAXIMUM ANNULAR RING FROM PAD IS .003" WITHOUT EXPOSING TRACES.

5. SILKSCEEN; WHITE NONCUNDUCTIVE EPOXY INK. ALL EXPOSED PADS MUST BE CLEARED OF SILKSCREEN .005 MINIMUM.

6. VENDOR UL APPROVED LOGO AND DATE CODE INFORMATION TO BE MARKED

7. ALL HOLE DIMENSIONS ARE FINISHED HOLE SIZES. MINIMUM ANNULAR RING IS .003" AT PAD TO TRACE JUNCTION.

8. FABRICATION TOLERANCES .XX +/- .010" .XXX +/-.005"

9. ELECTRICAL TEST:
100% BARE BOARD TEST ACCORDING TO THE REQUIREMENTS AND GUIDELINES OF IPC-ET-652, CLASS 2.
ALL BOARDS ARE TO BE TESTED FOR SHORTS AND OPENS USING SUPPLIED IPC356D NETLIST.
ELECTRICAL TEST MARKING REQUIRED.

**** 10. ALL 10 MIL DRILLS ARE +.000 / -NOMINAL
11. REMOVE ALL UNUSED INNER LAYER PADS.
**** 12. THIS BOARD HAS CONTROLLED IMPEDANCE TRACES LAYERS 1, 3, 4, & 6: 5MIL TRACE; 50 OHMS +/- 10%
**** 13. THIS BOARD IS TO BE TDR TESTED
**** 14. SEE DETAIL "A" FOR SPECIAL NOTES

ALL LAYERS VIEWED FROM TOP DOWN
LAYER: FABRICATION DRAWING
COMPANY: CARNEGIE MELLON
DESIGN BY: CiREXX
PART NUMBER: XXXXX
REV. 1
DESIGN: Gene McKenna DATE: 11-6-09
CHECKED BY: DATE:
ENGINEERING: DATE:
RELEASED: DATE:
TITLE: DRILL & FABRICATION NAND + DDR 1V85 I/O SODIMM
SIZE A
CIREXX DOCUMENT NUMBER: 83232
CIREXX FILE NAME: 83232-A.pcb
SHEET 1 OF 1
SCALE: 1/1

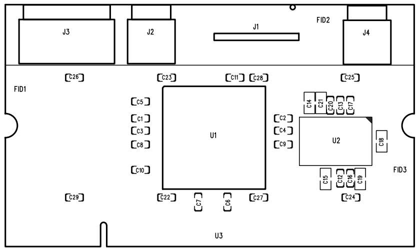
83232-A.pcb - Fri Nov 06 10:57:52 2009

6 5 4 3 2 1

D C B A

REVISION RECORD			
LTR	ECO NO:	APPROVED:	DATE:

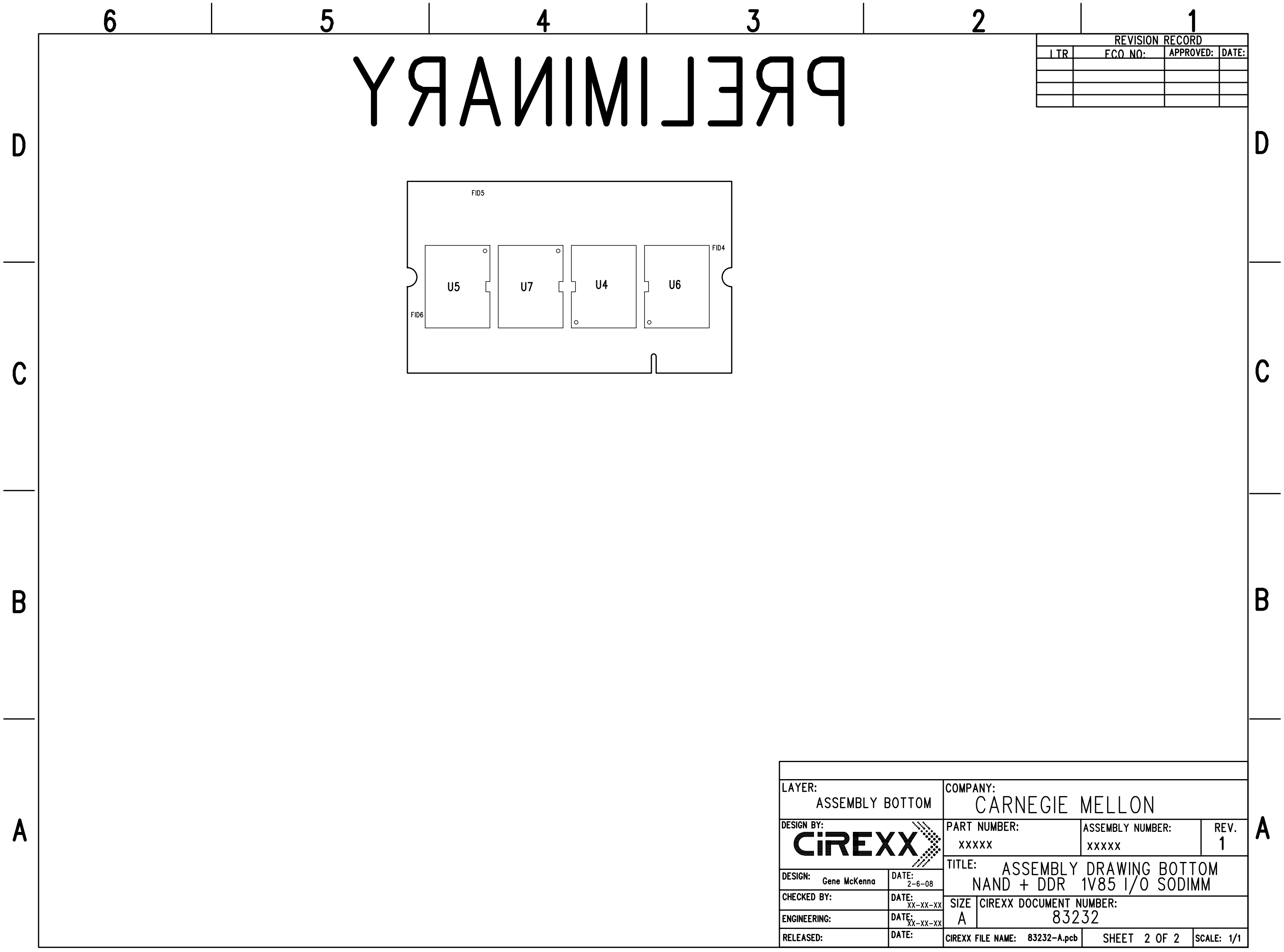
PRELIMINARY



ALL LAYERS VIEWED FROM TOP DOWN				
LAYER: ASSEMBLY TOP		COMPANY: CARNEGIE MELLON		
DESIGN BY: CiREXX		PART NUMBER: xxxxx	ASSEMBLY NUMBER: xxxxx	REV. 1
DESIGN: Gene McKenna	DATE: 11-6-09	TITLE: ASSEMBLY DRAWING TOP NAND + DDR 1V85 I/O SODIMM		
CHECKED BY:	DATE:	SIZE A	CIREXX DOCUMENT NUMBER: 83232	
ENGINEERING:	DATE:			
RELEASED:	DATE:	CIREXX FILE NAME: 83232-A.pcb	SHEET 1 OF 2	SCALE: 1/1

D C B A

83232-A.pcb - Fri Nov 06 10:57:43 2009



REVISION RECORD			
LTR	ECO NO:	APPROVED:	DATE:

LAYER: ASSEMBLY BOTTOM		COMPANY: CARNEGIE MELLON		
DESIGN BY: CiREXX		PART NUMBER: xxxxx	ASSEMBLY NUMBER: xxxxx	REV. 1
DESIGN: Gene McKenna	DATE: 2-6-08	TITLE: ASSEMBLY DRAWING BOTTOM NAND + DDR 1V85 I/O SODIMM		
CHECKED BY:	DATE: XX-XX-XX	SIZE A	CIREXX DOCUMENT NUMBER: 83232	
ENGINEERING:	DATE: XX-XX-XX			
RELEASED:	DATE:	CIREXX FILE NAME: 83232-A.pcb	SHEET 2 OF 2	SCALE: 1/1

83232-A.pcb - Fri Nov 06 10:57:46 2009

A.7 End Product/Photos

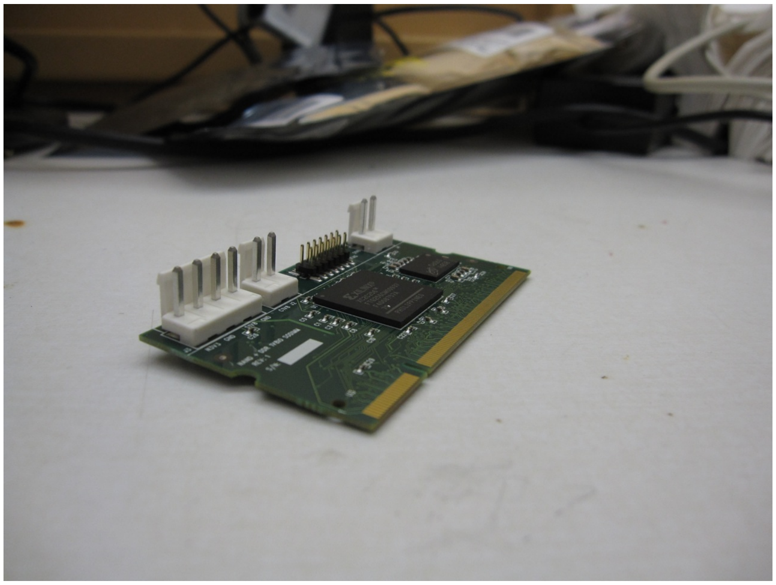


Figure 13: Frontal view of the custom-fabricated SODIMM

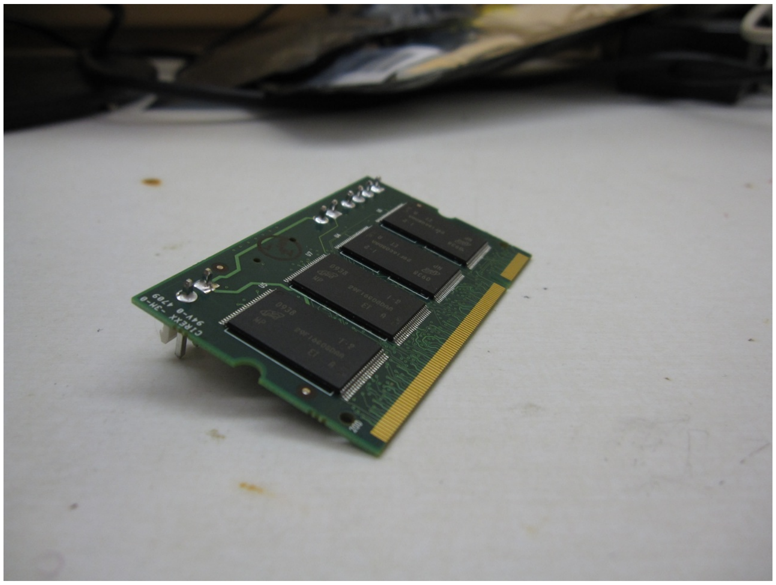


Figure 14: Rear view of the custom-fabricated SODIMM

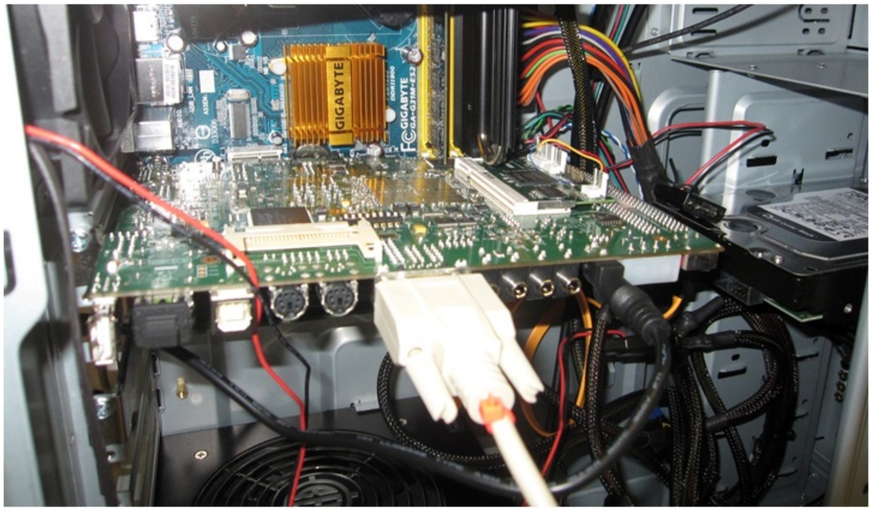


Figure 15: Xilinx LX110T board with the custom-fabricated SODIMM installed, inserted into a PCI Express x1 lane on the Gigabyte GA-G31M-ES2L motherboard