Final Report

Team IBM PC

Patrick Brown (pmbrown)
Aalique Grahame (agrahame)
Meghan Kaffine (mkaffine)
Eliot Wong (ecwong)

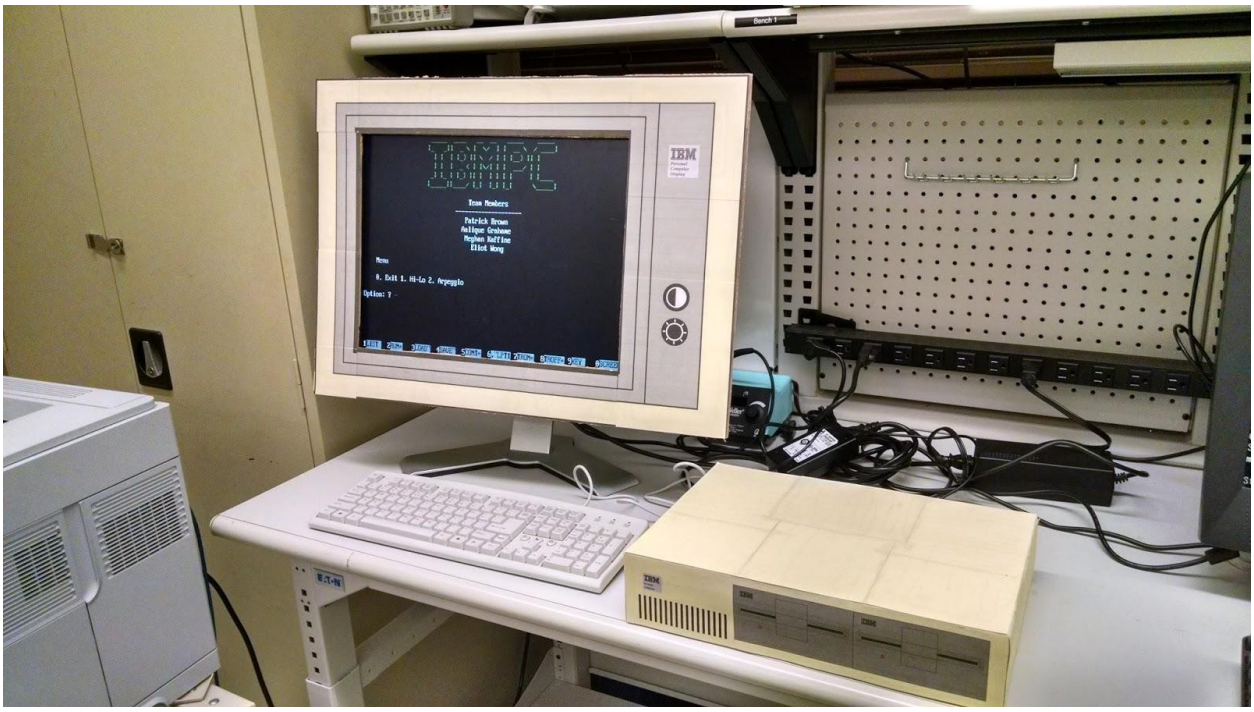
Project: IBM PC

Table of Contents

What We Built

Introduction

Our project for this semester was the development of a Verilog implementation of an IBM 5150, commonly referred to as the IBM PC. The IBM 5150 isn't commonly thought of as a video gaming machine. It was more commonly used as a business computer. However, there were a large number of games written for or ported to the IBM PC. The IBM PC has all the attributes we commonly associate with video game machines, including video, sound, and user input. Sharing these attributes with video game machines made the IBM PC a good project choice for 18-545.



Motivation

Our motivation in choosing the IBM PC for this project, instead of another machine from the 1980's, was its place in computer history. The IBM PC was not the first personal computer - there were other personal computers that predated it, including the Altair 8800, Commodore PET, and Apple II. However, the IBM PC was revolutionary because it set a standard, known as "IBM PC Compatible", for personal computers that still exists today. By setting this standard, the IBM PC popularized personal computing, kickstarting the computer revolution of the 1980's.

Historical Background

IBM's entry into the personal computer market is considered a watershed event in computer history. This was the moment in which personal computing became a part of mainstream life. Yes, there were many successful personal computers before it, but IBM had something that no other manufacturer had - the IBM brand. IBM was the market leader in the computing industry at the time and was one of the most respected companies in the United States. IBM's entry into the market sent a clear signal across the world - personal computing is the way of the future.

IBM's computer would be massively successful and help IBM keep its place as the market leader. However, it wasn't only the IBM name that made the IBM PC a standard for computing. The IBM PC became a standard because it was easy for other manufacturers to duplicate. Historically, IBM made computers all on its own - hardware and software. However, IBM wanted to move the IBM PC to market as soon as possible.

IBM's solution was to look outside the company for products they could buy off-the-shelf. They bought most of their integrated circuits, including the 8088 CPU, from Intel. The disk drives and monitor were imported from Japan. For an operating system, they turned to two companies - the market leader for software, Digital Research, and a small company out of Albuquerque, Microsoft. Digital Research was reluctant to do business with IBM, but Microsoft, eager for sales, agreed to bundle DOS with every IBM PC in exchange for royalties. The only thing on the PC that was IBM-owned was the BIOS chip.

IBM's reliance on outside vendors made it easy for other manufacturers to clone its PC. All an outside company needed to do was to buy components from the same vendors and produce their own BIOS chip. Companies such as Compaq became highly successful by selling computers that were "100% IBM PC Compatible". This meant that software and hardware developed for the IBM PC would also work on their machines. This turned "IBM PC Compatible" into an industry standard. With the establishment of an industry standard, the personal computer market grew exponentially. In addition, it popularized the processors made by Intel and the software written by Microsoft. The IBM PC left an indelible mark on the industry by popularizing the personal computer.

Game Description

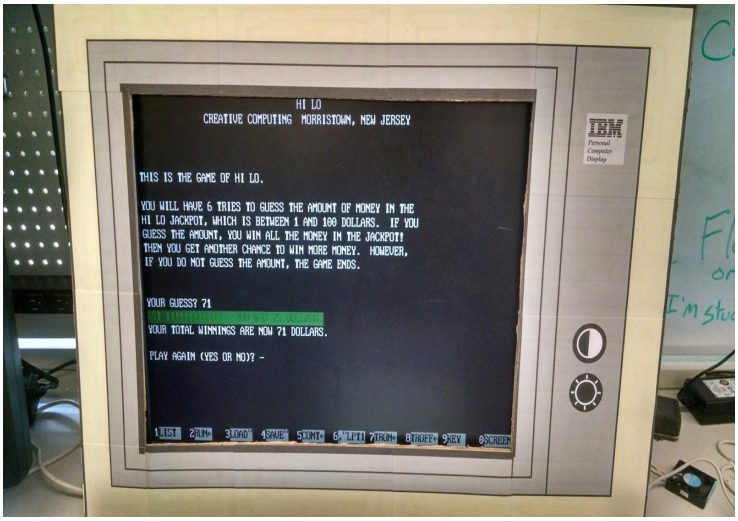
In our original plan, we had endeavored to have a functioning floppy disk drive for demo. This would have enabled us to load in programs from simulated floppy disks. However, the floppy subsystem was far too complex for the amount of time we had available to us. Instead, we developed a keyboard loader system that stored the keystrokes of a BASIC program that we could show off for demo.

The BASIC program we developed is comprised of four separate segments, weaved together using a custom developed start screen. Two of these are games and two of these are audio demos. The start screen allows the user to choose which demo he wants to view. The first demo is Hi-Lo, which is a simple one player game. The second demo is Arpeggio, which is a simple demonstration of our audio capabilities. The third demo is Game of Life, which is a more

complex two player game. The fourth demo is labeled as “???” and is a complex audio demo that plays “The Imperial March” by John Williams.

One of the requirements of this course was that we did not develop the games that we play on our hardware. This requirement was obeyed - both Hi-Lo and the Game of Life come from a book of BASIC games that came out in 1978. A link to where we got these games from is located in the Appendix under “Code Sources”. However, these games did not showcase the audio capabilities of the IBM PC. Therefore we added the Arpeggio and Imperial March sections to the demo.

Hi-Lo



The game of Hi-Lo is relatively simple. The user’s goal is to guess X, where X is an integer between 1 and 100, inclusive. The user has six guesses. The computer will tell him if his guess is too high or too low. If the user guesses correctly, he wins $X. If he guesses incorrectly six times, he loses all his winnings. He can then play again to win more money or quit.

Most of the code for this game was taken from the book of BASIC games where we found the Game of Life. However, we did make a few small modifications to it. We have the capability for colored text, so we chose to change the text attributes for the text that is displayed at the conclusion of a game. The text changes to blinking green if the user wins. The text changes to solid red if the user loses.

Arpeggio

Both Hi-Lo and the Game of Life are neat games, but they lack sound or color capability. We modified Hi-Lo to have color and added a few beeps at the end of Game of Life, but we felt that we needed a way to show off the sound capabilities of the IBM PC. The IBM PC does not have especially advanced sound capabilities, but we felt a demonstration of sound was necessary for two reasons. First, one of the requirements of the course was to have a system with sound. Second, the speaker is modulated by the Intel 8253 Programmable Interval Timer.

The 8253 turned out to be one of the most difficult parts of this project to get working correctly. Therefore, we wanted to show off how well it runs.

The Arpeggio is a simple demonstration of our sound capabilities. It plays tones ranging from 440 Hz to 1000 Hz in 5 Hz steps. It plays each tone for 27.5 ms. It then plays tones ranging from 1000 Hz to 440 Hz in -5 Hz steps. It plays these tones for 27.5 ms as well. The entire sequence takes approximately 8 seconds to play. This demo shows off our sound capabilities and provided us with a way to ensure that the 8253 works as expected.

Game of Life

The Game of Life is a two player game based on mathematician John Horton Conway's famous mathematical simulation. The goal of the game is to make your species survive. There are two players who must ensure the survival of their respective species. Player 1 gets "*" pieces and Player 2 gets "#" pieces. The game is played on a 5x5 grid. The game starts by Player 1 and Player 2 placing their 3 initial pieces on the grid.

The game abides by the same basic mechanics as the original Game of Life simulation. Live cells with fewer than two neighbors dies. Live cells with two or three neighbors continue to the next turn. Live cells with more than three neighbors dies. Dead cells with three live neighbors come to life in the next turn due to reproduction.

However, each player gets the chance to place a new piece at the beginning of every turn. If they place in the same location, nothing happens to that cell. No player can place a piece at an occupied location. If a player runs out of pieces, the other player wins the game.

The motivation for adding a game in addition to Hi-Lo was the course requirement to have multiplayer capability. Even though previous groups had gotten by with single player games, we chose to play it safe and add in a multiplayer game for demo.

The Imperial March

Even though the Arpeggio demonstrates that we have sound capability, it seemed like a bland way of doing so. Meghan suggested adding “The Imperial March” by John Williams to our demo. This piece of music features prominently in the Star Wars films as the theme of Darth Vader.

This song fits with IBM’s place in the computer market in the 1980’s. In the decades prior, IBM was the Galactic Empire of computer companies. However, 1977 would mark the release of “A New Hope” and the release of the Apple II. The introduction of the Apple II prompted IBM to start development of a personal computer of its own. 1980 would mark the release of “The Empire Strikes Back”. IBM would strike back in 1981 with the release of the IBM PC. Thus, we decided to make our IBM PC play “The Imperial March”.

To make our dream of playing this song a reality, we tracked down different pieces of sheet music for the song. We used the frequency table in the IBM BASIC guide to translate the notes into frequencies and determine the correct durations for each note. We then converted the frequencies and durations into “SOUND f,d” commands. We added these to our demo program and used our keyboard loader to put them into the IBM PC.

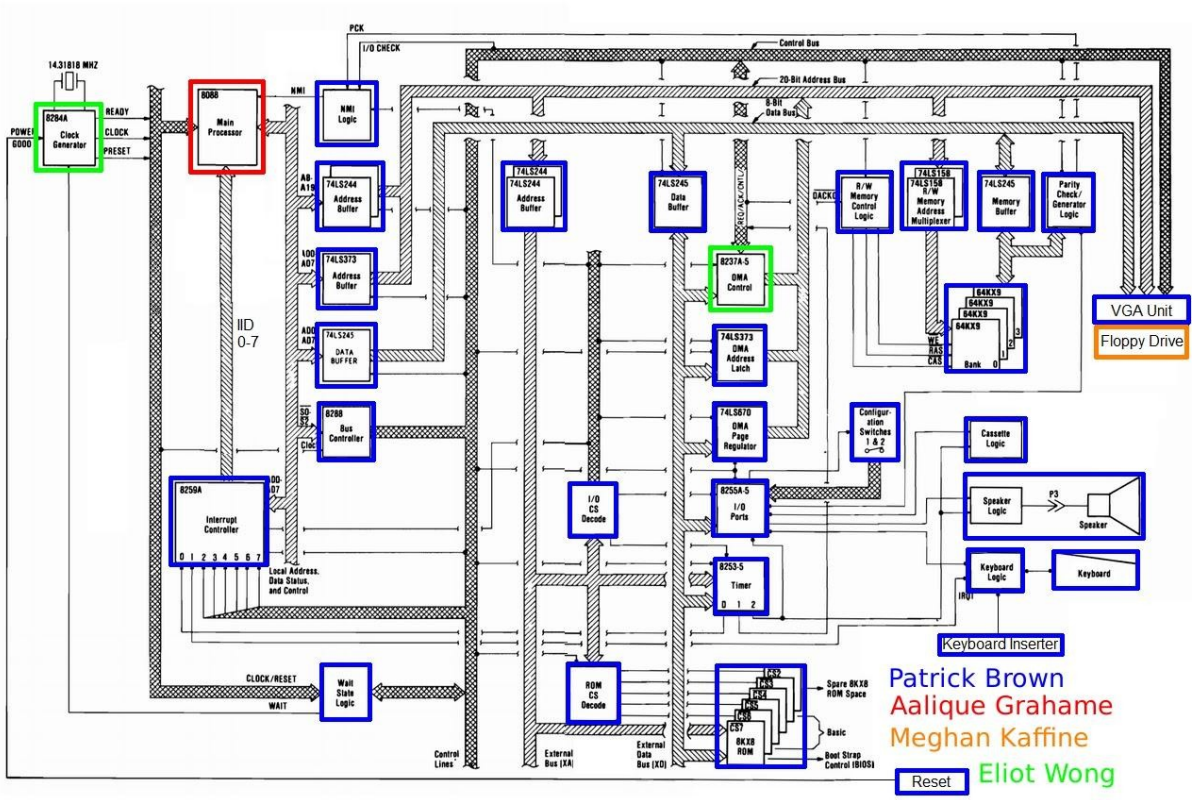
After some tweaking, we managed to get the song running on the IBM PC. The time we put into perfecting the 8253 Timer is evident, as the IBM PC is able to play the song with correct timing and pitch. This shows the advanced capabilities of our sound system. Though it would be nice to have a more complex game to show off sound, this demo program does a good job of demonstrating sound.

Detailed Hardware Description

Introduction

Our IBM PC implementation attempts to follow the original IBM PC’s design as closely as possible. As a result, we have based our design on the original circuitry of the IBM PC. However, we have made some modifications that are detailed in this report.

System Block Diagram



Hardware Overview

The following sections provide overviews of the major components of the IBM PC. These overviews provide an idea of what each component does, our design for the component, our testing strategy for the component, problems we ran into, and the result of our efforts.

CPU (Intel 8088)

The Intel 8088 was a variant of the Intel 8086, introduced on July 1, 1979. For this microprocessor, Intel decided to swap the 16-bit external of the 8086 for an 8-bit external bus. Therefore, the 8088 required two bus cycles to read or write data as opposed to the 8086's one. The features of the 8088 processor are:

- 8 bit data bus interface
- 16 bit internal architecture
- Direct addressing capability to 1 megabyte of memory
- Direct software compatibility with 8086 CPU
- 14 word or 16 bit register set with symmetrical operations
- 24 operand addressing modes
- Byte, word and block operations
- 8 bit and 16 bit signed and unsigned arithmetic in binary or decimal including multiply and divide

Although Intel moved away from the 16 external bus, they decided to keep their 16 bit internal architecture and 20 bit address bus from the 8086 unchanged.

The Intel 8088 was targeted at economical systems by allowing the use of an 8-bit data path and 8-bit support and peripheral chips. The original IBM PC was the most influential microcomputer to use the 8088. It used a clock frequency of 4.77 MHz. In order to design our 8088 processor, we decided it would be very useful to reuse a processor we found online, the Zet processor. The Zet processor was fully functional but was based on Intel's 8086 processor and not the 8088 processor which we were interested in. However, we saw this as an easy modification being that the main difference was the size of the external bus. Unfortunately, this proved to be much more difficult than we anticipated due to the implementation of the processor. The code written was very poorly documented making it difficult for us to determine what would be affected if we went in and changed the width of their data bus. This issue forced us to make the decision to avoid touching the functionality of their code and instead write our wrapper system for the Zet processor. We were able to do this because the Zet processor included a halt signal that allowed us to stop and start the processor as we pleased. Therefore, in cases such as read and write in which our processor would take more cycles, we would halt their processor to account for those extra cycles. Those allowed us to avoid any incorrectness. Unfortunately, this did cause us to lose some performance but it was a necessary trade-off to ensure functional correctness of the processor.

Basic functionality of the wrapper basically acted as a middle-man between the processor and memory or the I/O device. It waited for the processor to signal that it would like to access memory, halt the processor while it processes that request and then un-halts the processor to continue executing with that new data.

Floppy Drive

The IBM PC used a floppy drive or a cassette drive in order to boot DOS, in addition to any applications. The floppy drive used a floppy disk controller in order to communicate to the CPU. The FDC has two main I/O ports; the main register and the data register. The main register held the status of the FDC at any given point and could be accessed at any point in time. It is read only. The data register was actually the first in a stack of registers that held the data being read or written, it was the only one accessible. The FDC used Direct Memory Access to record data transfers.

The initial approach to the floppy drive was to create the floppy drive controller and use a compactflash disk and an image of DOS from the internet. At first glance, it seemed deceptively easy, but upon re-reading the IBM PC's diskette drive adapter reference, it was realized that an understanding of floppy disks would be needed, as some of the terminology being used was unfamiliar, as well as figuring out which version of the 5 ¼" floppy disk would have been used with the IBM PC. It appeared as if Team Dragonforce had done something similar, but their ultimate goal was different and their code was not helpful. With the additional research needing to be done, we had to extend the deadline to October 20th.

Not that that helped much. We ended up borrowing code from AO486 project, which is included in the citations, to get the floppy drive controller to work. The code we borrowed simulated a floppy drive, and we thought that all we needed to do was add a wrapper and make sure that it worked, that every command we expected the BIOS to give was verified.

Unfortunately, it wasn't that easy. While verifying that the commands that did not involve the evaluation step of the FDC was done relatively easily, the most important command the FDC would receive was the read command. The read command needed the evaluation step to work.

After discovering all the signals that prevented the FDC from going into the evaluation step, and making sure that they were changed, we discovered that the floppy drive code possibly expected the commands in a different order than the documentation we had provided. At this point it was too late in the semester to actually try and fix this problem in the wrapper, or to try and write our own code.

Direct Memory Access Controller (Intel 8237)

The IBM PC's DMA controller uses the Intel 8237 chip to allow data transfer from I/O to memory without using the main processor. It also allows for transfer between memory and memory. The DMA controller allows for different types of transfer include single, block, demand and cascade. There are also several different modes of transfer such as read, write and a verify transfer. The 8237 chip is capable of transferring data at a rate of up to 1.6 MByte per second. There are 4 channels for the DMA controller. The first channel, channel 0, is special and is used only for periodic DRAM refreshes. Because we had implemented the RAM ourselves we did not have to utilize the RAM refresh because it would be taken care of already. There are three additional channels that are used by I/O devices and there is a priority queue.

The documentation for the 8237 chip is limited because Intel choose not to describe the inner workings of the chip. Instead a very high level overview of the chip's function is described. As a result, it was hard to come up with an implementation from scratch. We found a basic implementation completed by a student at the Bourns College of Engineering that had some basic functionalities of the chip completed. However, this chip implementation was done in VHDL so it needed to be converted to Verilog so it could be used in the IBM PC implementation. This led to some issues because of the way VHDL and Verilog handled inout signals. Most of the debugging of the DMA was spent fixing this.

The other problem with the original implementation is that it only supported one channel which would not be sufficient for our needs. As a result, we added more registers in order to accommodate for all four channels needed. This design differs slightly from the original Intel design because the Intel uses only one set of registers for all four channels. The reason for this design change is because the implementation did not contain any type of priority selection for channels. Thus, we did not have to worry about clobbering the contents of the registers.

In line with our development goal to replicate the original IBM PC as closely as possible, the initial plan for the DMA was to write up a Verilog description based on the Intel 8237A datasheet. However, the details for this chip are sparse and it was difficult to determine the exact functionality and behavior of the DMA controller. As a result, writing a Verilog description would be tough and would be a result of many assumptions of the inner workings of the chip. We have found a couple of previous implementation of the chip, however both have issues. One of them is specifically hardcoded to work with an application and thus the timings and functionality would not be suitable for our project. We currently are using a basic implementation which implements single and block, read and write transfers. The added modifications and verification needed led to a delay in the implementation of the DMA. However, we managed to complete an implementation of the DMA controller.

Random Access Memory

The IBM PC's RAM consists of 64-256 kilobytes of memory. Given the large amount of block RAM available to us, we have chosen to instantiate 256 kilobytes to serve as our RAM bank. The RAM modules on the IBM PC consist of four RAM banks with 64 kilobytes each. Each bank consists of nine slices. Each slice holds 64 kilobits of data. Eight of those slices are the data itself, the ninth slice is for parity check. Our PC instantiates a CoreGen block to simulate the RAM.

The IBM PC's RAM utilizes two special signals for read and write, column access strobe and row access strobe. In order to read to or read from RAM, the system first sends a row access strobe and the row address to the RAM. The RAM latches the address when the strobe is received and selects the requested row. The system then sends a column access strobe and the column address to the RAM. The RAM latches the address when the strobe is received and selects the data in the row at that column. Data is then written to or read from that address.

The use of a Row Access Strobe and Column Access Strobe are unnecessary with current technology. Our original plan was to build a system that made use of the strobe signals. However, this proved far too difficult to debug. Instead, we used CoreGen to instantiate block memory, forgoing the use of strobe signals.

Read Only Memory

The IBM PC's ROM consists of 64 kilobytes of memory. This memory holds our image of the BIOS and the BASIC interpreter. BIOS stands for Basic Input/Output System. It may be accessed at any time, but it may not be written to by the processor. The ROM will be the first place the processor will look for instructions to execute. Our PC instantiates a CoreGen block to simulate the ROM from an image we found on the internet. The ROM covers addresses 0xF0000 to 0xFFFFF.

Keyboard

The IBM PC's keyboard was one of the more memorable aspects of the original IBM PC. It had 83 keys and unlike more modern keyboards, it was mechanical. On the inside, an Intel 8042 translated the keystrokes into scan codes that were sent to the IBM PC via a serial connection. The set of scan codes it utilized are now referred to as scan code set 1. Within the IBM PC, these scan codes are received by a shift register. Once a scan code is received, an IRQ1 interrupt is generated and sent to the 8259 Programmable Interrupt Controller. Once the processor has received the key press, it clears the interrupt by sending data to the 8255 Programmable Peripheral Interface at Port C. The original IBM PC keyboard, while a very nice piece of machinery, did not set an industry-wide standard for keyboards. Instead, IBM's PS/2 line of computers, which arrived in the later 1980's, made the PS/2 connector the standard way of connecting keyboards and mice for decades to come. The keyboards shipped with the PS/2 line included more keys, transmitted in scan set 2, and used a slightly different serial protocol.

The keyboards in the lab follow the newer PS/2 standard, which presented problems for our group. We needed our keyboard to work just like the original IBM PC keyboard. Our first thought was to develop a go-between module that would receive serial data from the keyboard, translate the scan codes, then send data to the motherboard that followed the IBM PC's serial protocol. We developed a Verilog module to accomplish that goal. However, it did not work as expected. We had only a vague idea of how the keyboard was actually transmitting data. Our keyboard module was built upon assumptions that were baseless. Worse than that, we realized that our original concept was monstrously inefficient. It was pointless to receive data, translate it, retransmit it, then receive it again.

The solution was to craft our own data receive module. Our keyboard data receive module takes in serial data under the PS/2 protocol, translates the data from scan code set 2 to scan code set 1, and generates an interrupt to the processor. The processor acknowledges the interrupt by sending a signal to our module that allows it to receive another scan code. This meant disregarding several chips and circuits on the motherboard that were intended to receive keyboard data from a real IBM PC keyboard.

This is an important lesson for any future 545 groups - your assumptions about how a system operates may be wrong, so have extra time in reserve to make adjustments. We thought that we understood how the keyboard sends data. We developed a Verilog module based on these assumptions that worked in simulation. However, it did not work in the real world, primarily because we did not have a sufficient understanding of how the keyboard transmits data. We wasted time developing something that ended up being worthless. However, we did have enough time to rework it for demo day. Leave time in reserve in case unexpected situations arise. Also, be sure that your assumptions about system operation are based on fact, not your own theories about system operation.

The keyboard module provided the perfect subsystem to show off on demo day, October 15th. We developed a special test bench for the occasion. Our test bench allows the user to fill up a buffer with key presses, read the key presses back from the buffer, and reset the buffer to read more scan codes. This provided not only a means of testing our keyboard module, but also

a way to demonstrate the progress we had made to that date. Work on the keyboard was conducted during early October. Patrick Brown took the lead on this part, as we had stipulated at the beginning of our project.

Keyboard Loader

In mid-November, we realized that we were going to face a big problem on demo day. Unless we completed the floppy drive, we had no way of storing programs to show off for demo. Without some way of storing programs, we would have to type in our demo program by hand. Typing a BASIC program in by hand is time consuming and can often result in syntax errors that could lead to disaster during a demo. We needed a solution to this problem and Eliot suggested a module to load in keystrokes automatically to the BASIC interpreter.

Having worked on the main keyboard modules, Patrick Brown took the lead on this part. The principle behind the keyboard loader is simple. An FSM waits for a button to be pressed on the front of the console. It waits for the keyboard to be idle, then sends a keystroke as if it was the normal keyboard receive module. Once the acknowledge signal is sent from the CPU to the keyboard, it increments an internal counter and sends the next keystroke. It keeps sending keystrokes until it reaches its limit.

However, there is an operating system-imposed limit to the number of keystrokes that can be sent at once. The operating system has an internal keyboard buffer that is only so long. If the keyboard tries inputting characters too fast, it sounds the system buzzer. When we tried to run the keyboard loader the first time, we printed only a few characters before the buzzer sounded and the system went haywire.

The solution was to add an extra counter to the keyboard loader that forced lag time between keystrokes. After receiving the keyboard acknowledge signal from the CPU, the keyboard loader starts a counter and waits until the counter reaches its maximum level before trying to send another character. This avoided overflowing the keyboard buffer. In addition, we

were able to calibrate the counter's maximum level to make the keyboard loader run as fast as possible without overflowing the buffer.

In order to have keystrokes to send to the keyboard loader, Eliot developed an online tool to translate BASIC programs into keystrokes. More information about it is located in the section "Keyboard Translator".

The keyboard loader also helped reveal the most pernicious bug in our entire system. Sometimes the keyboard loader would load characters up until a point and then stop. ChipScope investigation revealed that the system kept ending up in an invalid instruction handler and kept trying to return to an invalid part of the BIOS. Further investigation revealed that this pattern was happening after a repeat command that the processor was executing. The Zet processor handles repeat commands by having an extra-long execute state. If an interrupt occurred during this execute state, such as one from the keyboard module, the Zet processor immediately tried to service it. However, this resulted in the processor clobbering its own stack and registers. As a result, it popped the wrong return address off of the stack and ended up in an area of invalid memory. The problem was fixed by sending interrupts to the CPU only during the fetch state. The problem was solved and the system works much better now.

Work on the keyboard loader was conducted during late November. Patrick Brown took the lead on this part because of his experience with the keyboard system.

Input/Output Interface (Intel 8255)

The IBM PC's I/O interface is handled by the Intel 8255 Programmable Peripheral Interface chip. The Intel 8255 has 24 different I/O pins that can be programmed in three operating modes. Mode 0 is for basic input and output. Mode 1 is for strobed input and output. Mode 2 is for a bi-directional bus. The 24 pins are grouped into four sets: Port A, Port C Upper, Port B, and Port C Lower. Writing a control word to the 8255 changes the mode of operation of the chip. The 8255 can be programmed to accept many different combinations of operations.

However, the IBM PC BIOS sets the 8255's mode of operations at startup. It writes the control word 0b10011001 to the mode register. This control word configures the chip in mode 0, basic input and output. Port A is set as input. Port C Upper is set as input. Port B is set as output. Port C Lower is set as input. This configuration is kept for the operation of the computer and assists with communication with the keyboard, configuration DIP switches, cassette drive, and speaker. Since the BIOS configures it manually for only three jobs, we simplified the design to work exactly as the BIOS desires.

Each set of ports on the 8255 handles a different I/O device. Port B, which is the only output port, has multiple jobs. It sends data to the speaker, turns the cassette motor off and on, enables RAM and I/O clocks, and controls what data is sent to the other input ports. Signal PB2 from Port B controls which set of DIP switch settings from Block SW2 is sent to Port C Lower. If asserted, switches 1-4 are sent. If not asserted, switch 5 is sent. Signal PB7 acknowledges the

receipt of keyboard data and reads switch settings from Block SW1 if asserted. Port C Lower is an input port that receives switch settings from Block SW2. Block SW2 lists the amount of conventional RAM (times 32 kilobytes) aboard the device. Port C Upper is an input port that receives data from the cassette drive. Port A can receive either keyboard scan codes or switch settings from Block SW1. Block SW1 configures the display, floppy drive, and RAM. The SW1 settings are detailed in the table below.

Switch Number	Function
Switch 1	ON: No floppy drives connected OFF: Floppy drives connected
Switch 2	ON: 8087 not connected OFF: 8087 connected
Switches 3 and 4	ON, ON: RAM Bank 0 populated OFF, ON: RAM Bank 0 and 1 populated ON, OFF: RAM Bank 0, 1, and 2 populated OFF, OFF: RAM Bank 0, 1, 2, and 3 populated
Switches 5 and 6	OFF, OFF: MDA Monochrome OFF, ON: CGA 40 x 25 ON, OFF: CGA 80 x 25 OFF, OFF: Special expansion card
Switches 7 and 8:	ON, ON: One floppy drive OFF, ON: Two floppy drives ON, OFF: Three floppy drives OFF, OFF: Four floppy drives

Work on the 8255 was conducted during early October. Patrick Brown took the lead on this part, as we had stipulated at the beginning of our project.

Programmable Interval Timer (Intel 8253)

The IBM PC's Programmable Interval Timer consists of an Intel 8253 chip that generates three different signals. The Intel 8253 can generate six different types of signals: an interrupt on terminal count, a hardware triggerable one-shot, a rate generator, a square wave, a software triggered strobe, and a hardware triggered strobe. It can countdown in either binary or binary coded decimal. The Intel 8253 is a very complex piece of hardware that would have taken an inordinate amount of time if we had not devised ways of simplifying it.

Fortunately, the IBM PC does not use all the native abilities of the chip. It makes use of only three out of the six operating modes and does not use binary coded decimal countdown. It makes use of all three of the channels, but the BIOS configures all three of them at startup. Channel 0 is a square wave at 18.2 Hz that handles the machine's time of day functionality.

Channel 1 is a rate-generated pulse at 66 kHz that controls the memory refresh. Channel 2 is a square wave at 896 Hz that is used to produce a beep from the speaker. This knowledge allowed us to simplify the design of the 8253 and streamline the verification we needed to perform.

However, prior to studying the BIOS in depth, we had attempted to implement all of the operations ourselves. This attempt was short-lived and we endeavored to find an existing implementation in Verilog. An internet search revealed that Synopsys had included a Verilog implementation of the 8253 in the demo folder of VCS. With Professor Nace's blessing, we copied the code for the 8253 from the VCS installation on the ECE network. At first, the 8253 code seemed to be a blessing. It turned out to be a curse.

The 8253 code from VCS did not operate according to the timing specs found in Intel's datasheet for the 8253. We had developed a test bench based upon the datasheet timing specs and the VCS code failed every test. As a result, we had to rewrite large portions of VCS's code to bring it up to specifications. We ended up with code that featured workarounds and kludge operations.

Kludge code is barely maintainable, so we ended up scrapping all of the original code and rewriting it. This give us counters that were far easier to debug and operated much more reliably.

This is an important lesson for any future 545 groups - demo code is not guaranteed to run as expected. Demo code is what it sounds like - code for demonstration. Even a company as well-known as Synopsys isn't going to take the time and test the code they include as part of program demos as thoroughly as they do for code they plan to sell. It isn't guaranteed to work exactly like the hardware it is meant to emulate. Instead, use it as a point of reference for designing your own code. Don't even try to modify what they have already. We spent hours trying to modify their code until we realized that it would be impossible to make it meet our desired level of operation. In retrospect, it was far less time consuming to scrap everything that Synopsys wrote for the 8253, write our own 8253, and test it.

Work on the 8253 was conducted during late September. Patrick Brown took the lead on this part, as we had stipulated at the beginning of our project. He is the one who found out about the Synopsys code in the VCS installation files. He is also the one who rewrote the 8253 code in mid-November after all of the problems we had with the 8253.

Video Display Unit (VGA)

The original IBM PC was shipped with two types of graphics cards, the Color/Graphics Adapter (CGA) and the Monochrome Display and Printer Adapter. The Color/Graphics Adapter could operate in either color text mode or color graphics mode. The Monochrome Display and Printer Adapter could only operate in monochrome text mode.

The Zet processor has two separate graphics units, one which can handle the graphics and text operations of the CGA and another that can only handle the text operations of the CGA. The Zet processor's status page stated that only the latter worked, so we decided to modify the latter unit. Admittedly, this limits our operations to text programs. However, text mode is all that is necessary for our goals, which are to boot into IBM BASIC and PC-DOS. Our logic was that it was a better idea to use code that we knew we could rely upon rather than code that might not work.

Both the Zet graphics unit and the CGA follow roughly the same architecture. Each has onboard RAM that stores the character data. Each has onboard ROM that stores the bitmaps for each of the characters. A finite state machine takes character data from the RAM, grabs the relevant bitmap from the ROM, and prints it on the screen. The finite state machine also controls the output of red, green, blue, vertical sync, and horizontal sync signals to a VGA monitor. In addition, there is an onboard register file that stores such information as cursor location and light pen location.

The onboard RAM stores two pieces of data - character and attribute. Each piece of data is eight bits. Character data is stored at even addresses beginning at 0xB8000. Attribute data is stored at odd addresses and is paired with character data. Character data consists of the character that needs to be printed. The image for the character is found in the onboard ROM and follows IBM Code Page 437. The attribute data is also eight bits. Bit 7 is the brightness of the background, bit 6 is the red value for the background, bit 5 is the green value for the background, bit 4 is the blue value for the background, bit 3 is the intensity of the foreground, bit 2 is the red value for the foreground, bit 1 is the green value for the foreground, bit 0 is the blue

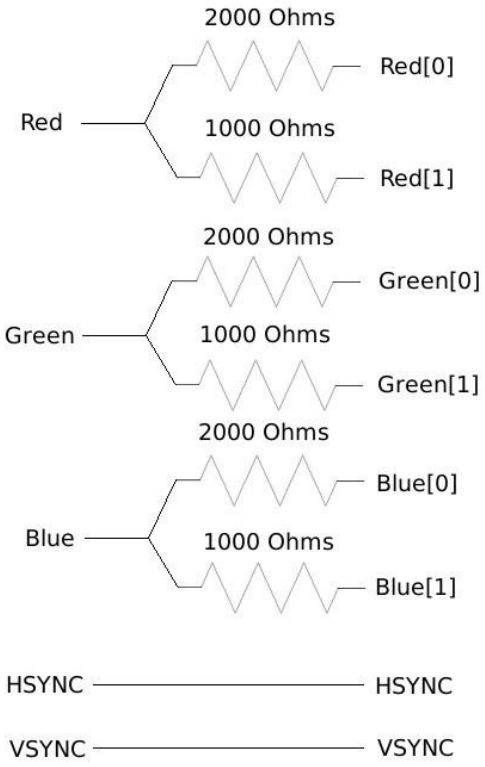
value for the foreground. Text is displayed with 80 characters per line and 25 lines in total. There are 1000 characters in all. Character data is stored in 2 kilobytes of RAM and attribute data is stored in another 2 kilobytes of RAM.

Modification of Zet's graphics unit proved to be more complex than we had expected. Zet utilized a wishbone interface to link their processor with their graphics unit. Access to the onboard RAM was restricted by the finite state machine. Finally, the Zet graphics unit was meant to be synthesized on a board that provided native VGA support. All three of these problems needed to be solved in a way that ensured operation as close to the original IBM PC as possible without damaging the existing code extensively.

The wishbone interface utilized a different set of signals than the IBM PC's 62-pin I/O interface. This required scrapping most of the code that followed these signals in favor of code that worked with the I/O interface. Rewriting this code required care, but there were enough similarities to make the process of rewriting relatively straightforward.

The finite state machine on the Zet graphics unit restricted accesses to the onboard RAM to prevent the processor from changing character values while the screen was refreshing. This prevented our processor from writing to memory whenever it desired, requiring more complex control. To simplify things, we decided to use CoreGen's true dual port memory. The processor could read and write from the A Side of the RAM while the graphics unit could read from the B Side of the RAM. By making this change, our graphics unit is a lot easier to use and program.

When we chose our FPGA board, we chose the Virtex-5 because it claimed that it would support VGA with an adapter for the DVI port. However, the documentation failed to mention that VGA and DVI were both controlled by an onboard computer chip. There was no way to natively pipe VGA signals directly through the DVI port. The only option was to use some of the GPIO ports on the board. This meant that we had to construct a crude DAC using resistors and solder together a VGA connector. We learned a lot about how to do this from the team that had constructed the 1942 game a few years ago. They had the same issue and built a DAC very similar to ours. Even though the picture wasn't perfect, we were able to transmit an image to our monitor that was readable. A diagram of our DAC is provided below.



Work on the graphics unit was conducted during mid-semester break in mid-October. Patrick Brown took the lead on this part, primarily because we were facing a time crunch and he had just completed the Intel 8259 Programmable Interrupt Timer.

Piezoelectric Speaker

The audio for the IBM was provided by a relatively simple, 2.25” piezoelectric speaker. The speaker is driven from two sources. The first source is the 8255 Programmable Peripheral Interface. By sending data to bits 0 and 1 of Register B on the 8255, the user can modulate the signal and send it a pulse train of data. The second source is Channel 2 of the 8253 Programmable Interval Timer. By adjusting the timer’s maximum count, the user can produce a waveform at different frequencies. The speaker can be driven in three different ways, all of which may occur simultaneously. The first is to create a pulse train from the 8255. The second is to create a waveform via the timer clock channel. The third is to modulate by activating or deactivating the 8253 timer via the 8255 Programmable Peripheral Interface.

We originally thought that we would have to acquire a piezoelectric speaker to use with this project. As it turns out, there is a piezoelectric speaker available on the Virtex-5 FPGA. At first we were concerned that the speaker would have insufficient power to deliver audible sound. However, it proved to be loud enough for our purposes.

We did not have to implement the speaker as its own subsystem because its operation was governed by the 8253 and 8255. However, we discovered early on that sound did not work as we expected. Sometimes beeps did not occur, other times beeps could not be disabled, requiring a reboot of the system. These problems led us to make revisions to the 8255 and scrap most of the code that we had modified for the 8253. The necessity of getting sound working motivated us to scrap and rebuild the 8253 completely. However, the sound works now and can play a nice arpeggio for the demonstration.

Since the speaker system is driven by the 8255 and 8253, responsibility for the system fell to Patrick Brown, who wrote both the 8255 and 8253. Since the 8253 needed to be scrapped and rebuilt, completion of the speaker system did not occur until mid-November.

Programmable Interrupt Controller (Intel 8259)

The IBM PC has two types of interrupts, maskable and non-maskable. Non-maskable interrupts, or NMI, are inputted directly into the processor through the NMI pin. Maskable interrupts pass through the Intel 8259 Programmable Interrupt Controller before being sent to the processor via the IRQ pin. The purpose of the Programmable Interrupt Controller (PIC) is to control how the processor receives interrupts. This involves setting priority of interrupts, masking interrupts, and determining which interrupt to send.

There are three important registers on the Intel 8259 that determine what data is delivered to the processor: the Interrupt Request Register (IRR), the Interrupt Mask Register (IMR), and the In-Service Register (ISR). The IRR is a series to latches that are set by data coming in on the interrupt lines, IRQ0 - IRQ7. They are cleared by the ISR when their respective interrupt is being serviced. They are activated regardless of the mask on the IMR. The IMR is a bit mask that determines which interrupts are blocked and which ones are not blocked. If a bit on the IMR is asserted high, this means that the respective interrupt is blocked. The ISR holds the interrupt currently being serviced by the PIC. It is cleared when the interrupt service routine is done.

The interrupt service routine consists of six steps. First, one or more of the interrupt request lines (IRQ0-IRQ7) are raised high, which sets the corresponding IRR bits. Second, the 8259 evaluates the requests and sends an INT signal to the CPU if the request needs servicing. Third, the CPU acknowledges the INT by sending back an INTA pulse. Fourth, the 8259 receives the INTA pulse. It then sets the highest priority ISR bit and clears the corresponding IRR bit. The 8259 does not drive the data bus during this step. Fifth, the CPU will initiate a second INTA pulse. The 8259 makes an 8-bit pointer available on the data bus that points to the interrupt vector table in the BIOS. Sixth, the 8259 waits for an End Of Interrupt (EOI) command from the processor before clearing the relevant ISR bit.

Like many of the other chips in the Intel family, the 8259 has many operating modes that the IBM PC does not utilize. For example, the 8259 can be configured in automatic end of interrupt mode, in which it does not have to wait for an EOI command to clear the relevant ISR

bit. Once again, examining the BIOS code proved to be the key to simplifying the 8259. The processor starts the 8259 by issuing a series of command words that govern its modes of operation and the 8-bit pointer to the interrupt vector table. The 8-bit pointer to the interrupt vector table is 0x08. As a result, our 8259 bypasses the BIOS's configuration processes and configures itself for operation.

This is an important lesson for any future 545 groups - the BIOS is your friend. We were lucky in finding documentation from IBM for their BIOS. It was not just an assembly dump - they had comments that explained how each chip is configured. This saved us a huge amount of time by allowing us to know what we had to implement and what we could safely ignore. There are many chips that are used in early computers that can handle a multitude of different configurations. It was not uncommon for companies to make a chip that was versatile enough to be used in different computers. Look at the BIOS, datasheets, and documentation before you start slinging Verilog to implement these components.

Work on the 8259 was conducted during late October. Patrick Brown took the lead on this part, as we had stipulated at the beginning of our project.

Clock Generator (Intel 8284)

The IBM PC uses an Intel 8284A chip as the clock generator for the main processor. The 8284A is a single chip clock generator which has a divide-by-three counter. The chip has a special input called CSYNC which allows the 8284A to be synchronized with another external clock, for example, other 8284A clocks throughout the system. The Intel 8284A also has the job of generating READY and REST signals for the main processor.

The clock generator chip implementation ended up being a pretty simple combination of several clock dividers. Because we are implementing this on a FPGA we can use the existing clock and simply divide the time to get the correct period and duty cycles. In total, the IBM PC required 4 clocks each with different periods and duty cycles. The FPGA clock has a duty cycle of 50% and a 10 ns period which translate to a 100 MHz clock. The first new clock needed was the CLK which was used as the clock input for the 8088 processor as well as its local bus. It required a 33% duty cycle and had a 210 ns period which translates to a 4.77 MHz clock. The next clock required was the peripheral clock (PCLK), this was used as input to the timer chip (8253) as well as the IO ports. It requires a duty cycle of 50% and has a 420 ns period.

The next clock was the oscillator output (OSC) which is used as the clock input for the IO connectors. This clock had a duty cycle of 50% and a 70ns period which is a 14.32 MHz clock. However, because the original FPGA clock had a period of 10 ns, we could not achieve a 50% duty cycle for this clock. Instead we ended up with a 57.1% duty cycle still with a 70 ns period, we predict that this will not affect the behavior of the system. Finally, the last clock is used for input to the VGA which is called (VCLK), this has a duty cycle of 50% and a period of 40ns. This was not originally part of the 8284a but was added out of necessity for the VGA implementation.

Bus Controller (Intel 8288)

The IBM PC utilizes an Intel 8288 bus controller to manage the address and data buses. The bus controller receives three status signals, s0_n, s1_n, and s2_n, directly from the 8088 that indicate what sort of signals it should broadcast to control the bus. These bus control signals include interrupt acknowledge (inta_n, when s_n = 000), read I/O (iorc_n, when s_n = 001), write I/O (aiowc_n, when s_n = 010), code access (mrdc_n, when s_n = 100), read memory (mrdc_n, when s_n = 101), write memory (amwc_n, when s_n = 110), and halt/passive (no signals asserted, when s_n = 011 or 111). In addition, there are special latch controls that help the processor send and receive data. These latch control signals include data transmit/receive (dtr), which determines the direction which data is sent, address latch enable (ale), which latches the address the processor has sent, and data enable (den), which latches the data the processor has sent.

Whether these signals are sent is determined by two control signals, aen_n and cen. The 8288 sets its signals to high impedance if cen is not asserted. The reason why this is the case is that the 8237 DMA controller can assert some of these signals as well. The logic that governs DMA accesses is responsible for asserting cen. The other control signal, aen_n, is address enable. If asserted low, it allows the 8288 to broadcast its bus control signals. If asserted high, all the 8288 bus control signals remain asserted high. As is the case with cen, aen_n is determined by the DMA logic in module 2 of the motherboard.

An important feature of the 8288 is the strapping option that configures it for use with a multi-master system bus and separate I/O bus. Placing it into I/O bus mode requires asserting pin iob high. However, the IBM PC asserts pin iob low, permanently placing the 8288 into system bus mode. Since we only use the 8288 in system bus mode, we did not implement I/O bus mode.

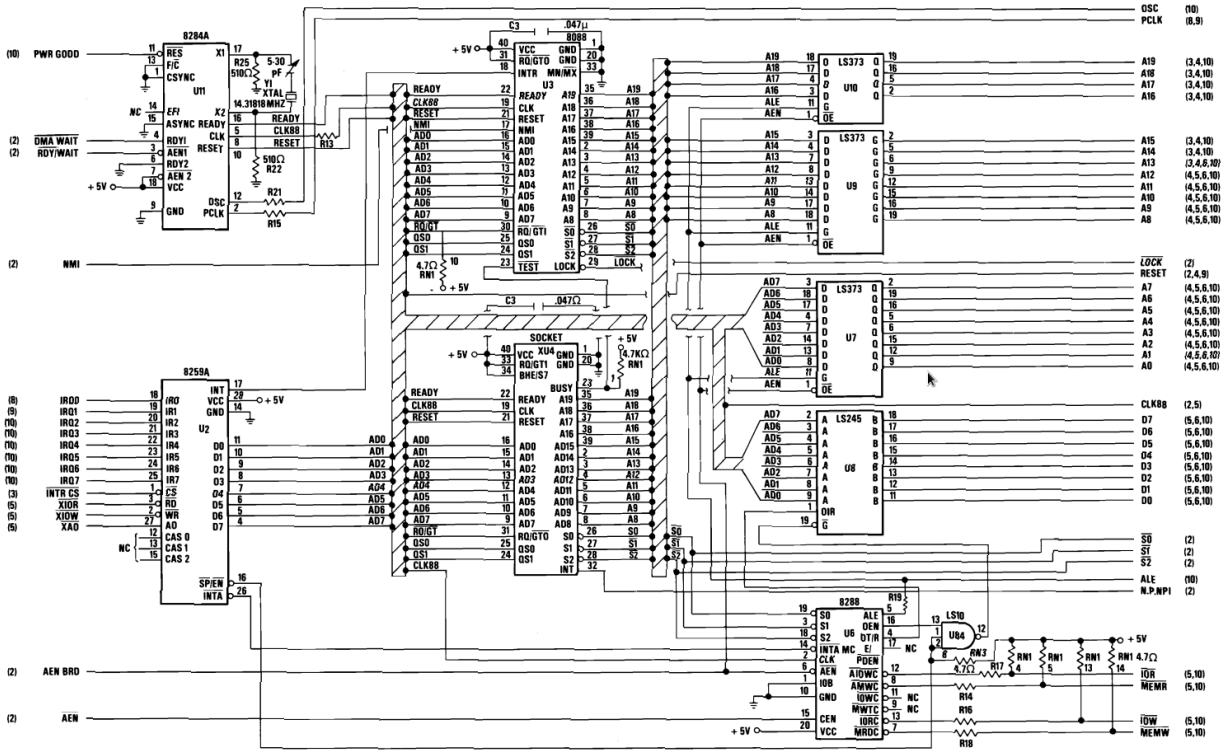
Likewise, there were several signals that we did not implement because the IBM PC leaves them unconnected. Signals mwtc_n and iowc_n are alternate signals used for memory and I/O reads. They are sent one clock pulse later than amwc_n and aiowc_n. They are unused and are unimplemented in our 8288. Likewise, signal mce is for master cascade enable and peripheral data enable. It is intended for use with cascaded Intel 8259 Programmable Interrupt Controllers. However, the IBM PC utilized only one 8259, so this signal was unnecessary in our implementation of the 8288.

Work on the 8288 was conducted during late October. Aalique Grahame was originally the lead on this part. However, verification of the 8088 was not complete by the required deadline. As a result, responsibility for this part was given to Patrick Brown.

Motherboard

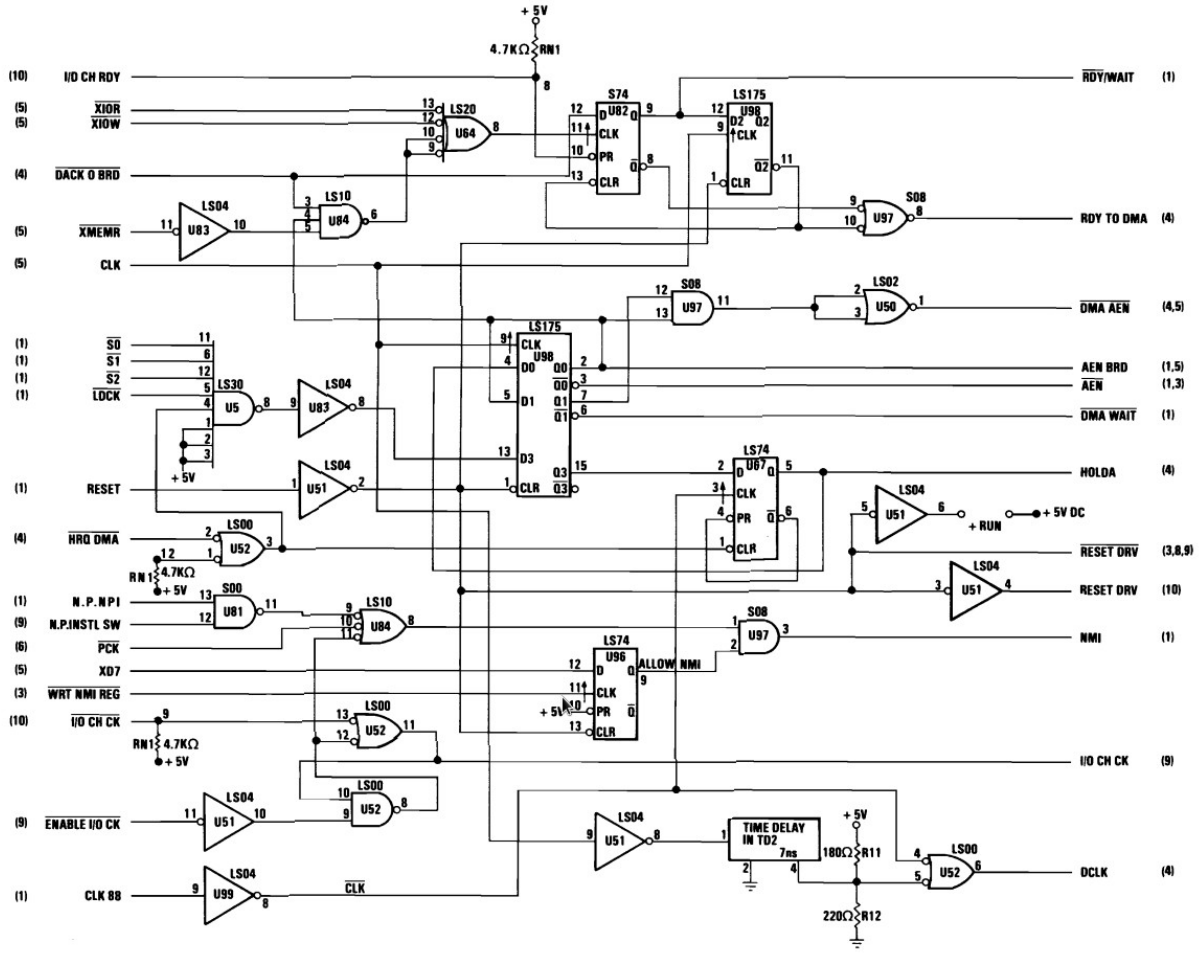
The IBM PC's motherboard integrates all the various components described above into a complete computer system. The IBM PC documentation provides a complete description of the motherboard, including all of the chips and logic required to make the IBM PC run. However, the IBM PC's motherboard is a massively complex piece of computer hardware. The diagram for the system takes ten pages, all of which needed to be converted into Verilog. In order to simplify the process of design and debugging, each page of the motherboard diagram was converted into separate Verilog modules. Each page contains different sub-systems critical to the functionality of the system as a whole. Some of these pages required the implementation of smaller chips in the 74LSxxx family. Patrick Brown completed these chips in mid-September not long after we decided on our project. These smaller chips are latches, parity checkers, and bus arbiters. For the sake of simplicity, descriptions of these small chips will not be covered in this report. In the next few sections, we will describe the ten sections of the motherboard and what chips go into them.

Motherboard Section 1



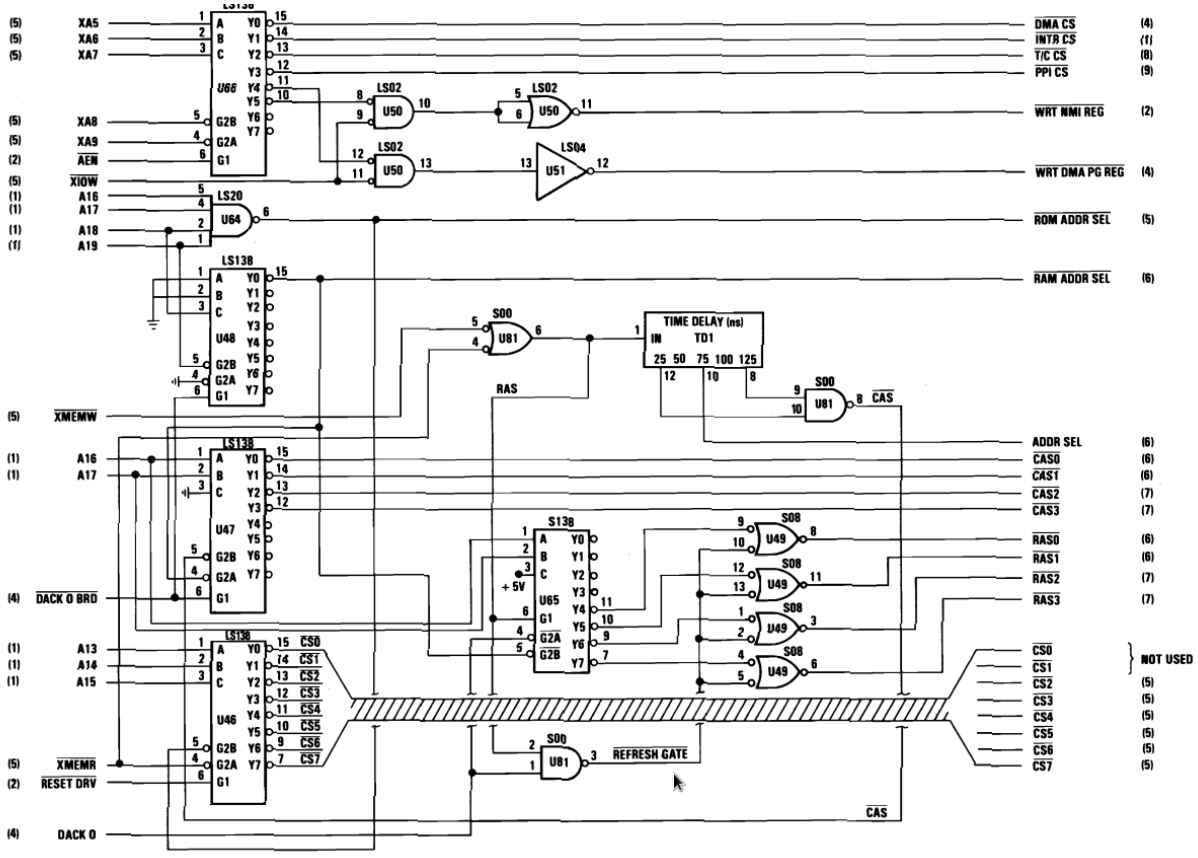
The first section of the motherboard includes the most important part of the system, the Intel 8088 processor. It also includes several important chips that assist the processor, including the 8284 timer, the 8259 programmable interrupt controller, and the 8288 bus controller. The 8284 timer generates the desired clock signals for the processor. The 8259 programmable interrupt controller alerts the processor if an interrupt occurred. The 8288 bus controller determines which system buses are activated. In addition, it orders several LS373 bus driver latches to output the address and a LS245 bus driver to receive or send data.

Motherboard Section 2



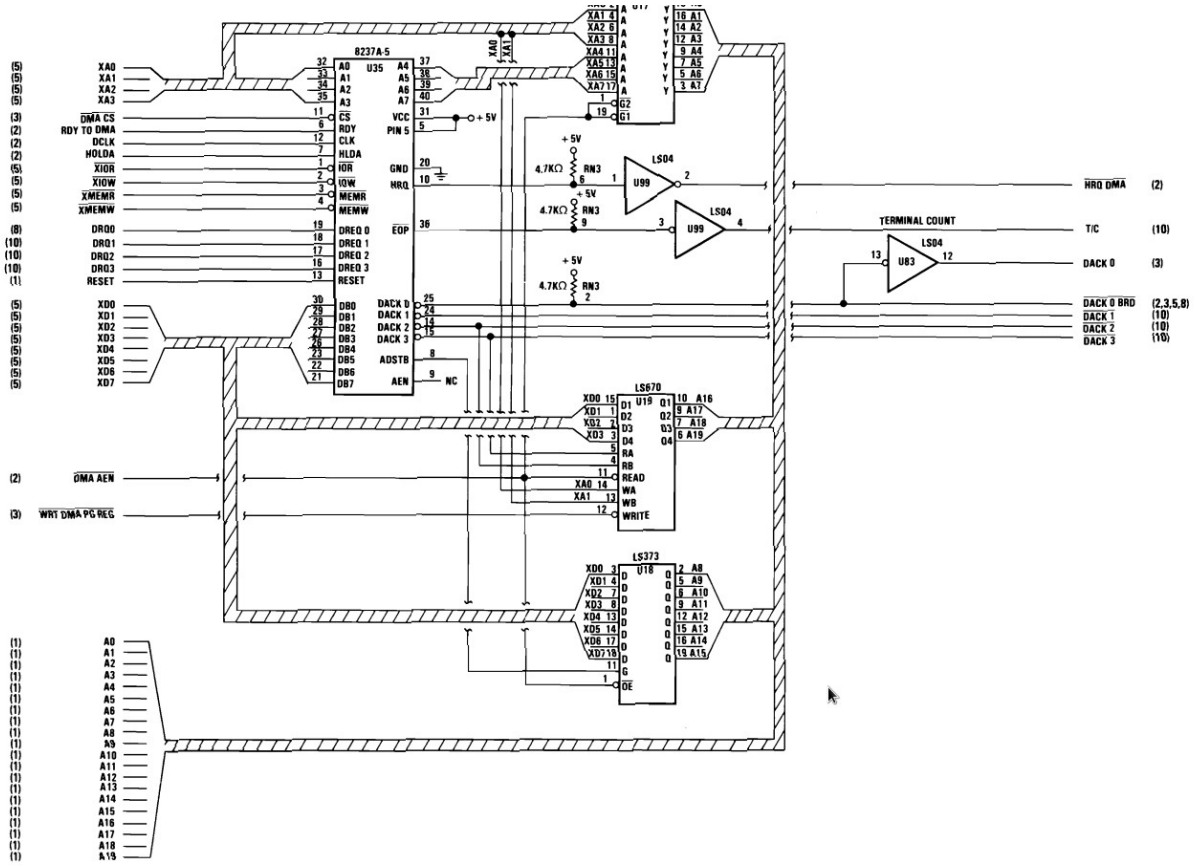
The second section of the motherboard is a complex web of combinational and sequential logic to determine several system signals. These signals handle such jobs as DMA transfers, bus arbitration, NMI, device reset, and I/O check. For this section, we decided to copy the combinational and sequential logic directly from the IBM PC documentation.

Motherboard Section 3



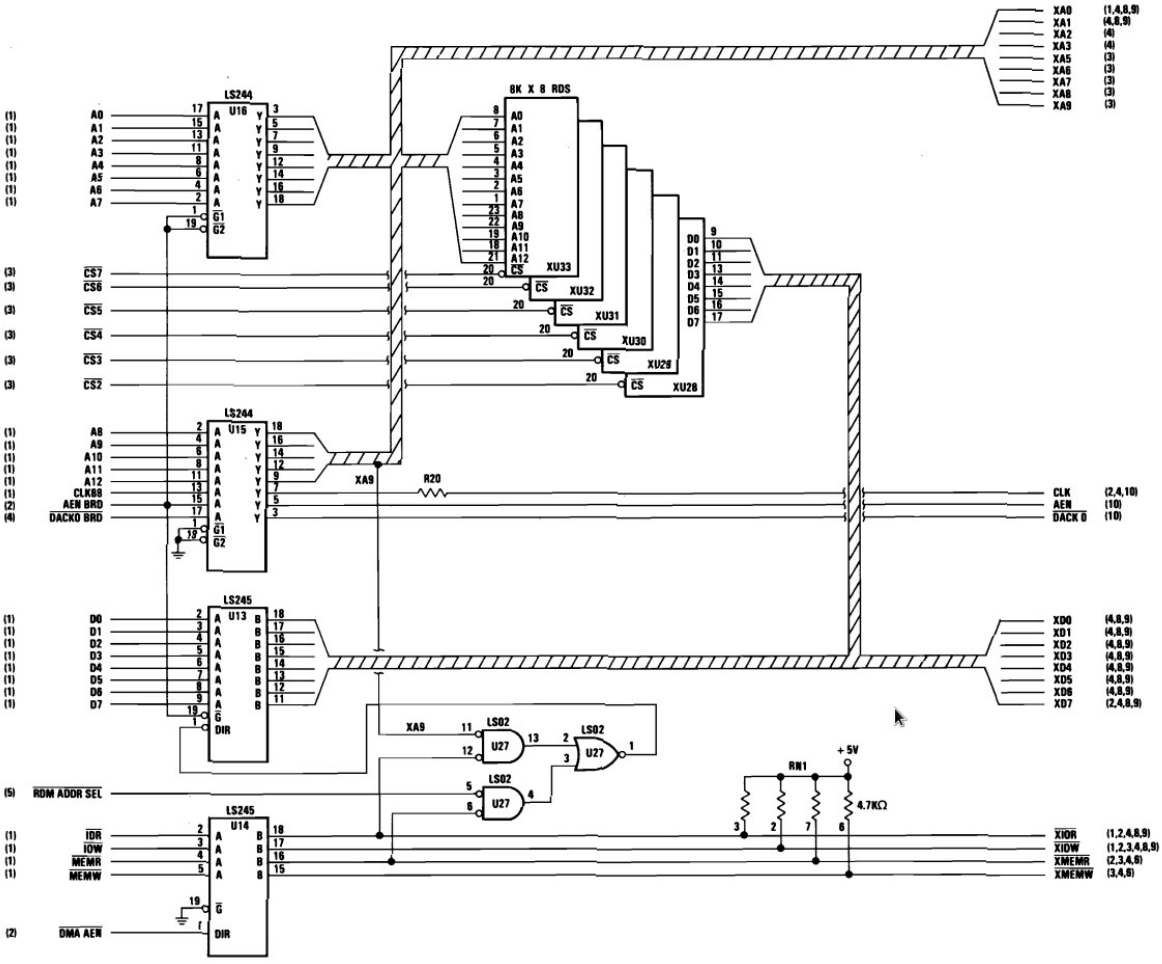
The third section of the motherboard governs access to the RAM and ROM banks. A series of LS138 demultiplexers translate the ROM or RAM request into access requests for the specific bank that the user requires. ROM requests are fairly straightforward. The address is decoded, the particular bank is selected, and the data at the address is sent to the data line. In the original implementation, RAM requests require the use of row and column access strobes. A special time delay module was constructed to assist with this process by delaying the column access strobe until 100 nanoseconds after the row access strobe. This ensures that data is latched at the correct moment. However, we decided to modify the RAM implementation, which meant removing much of this circuitry.

Motherboard Section 4



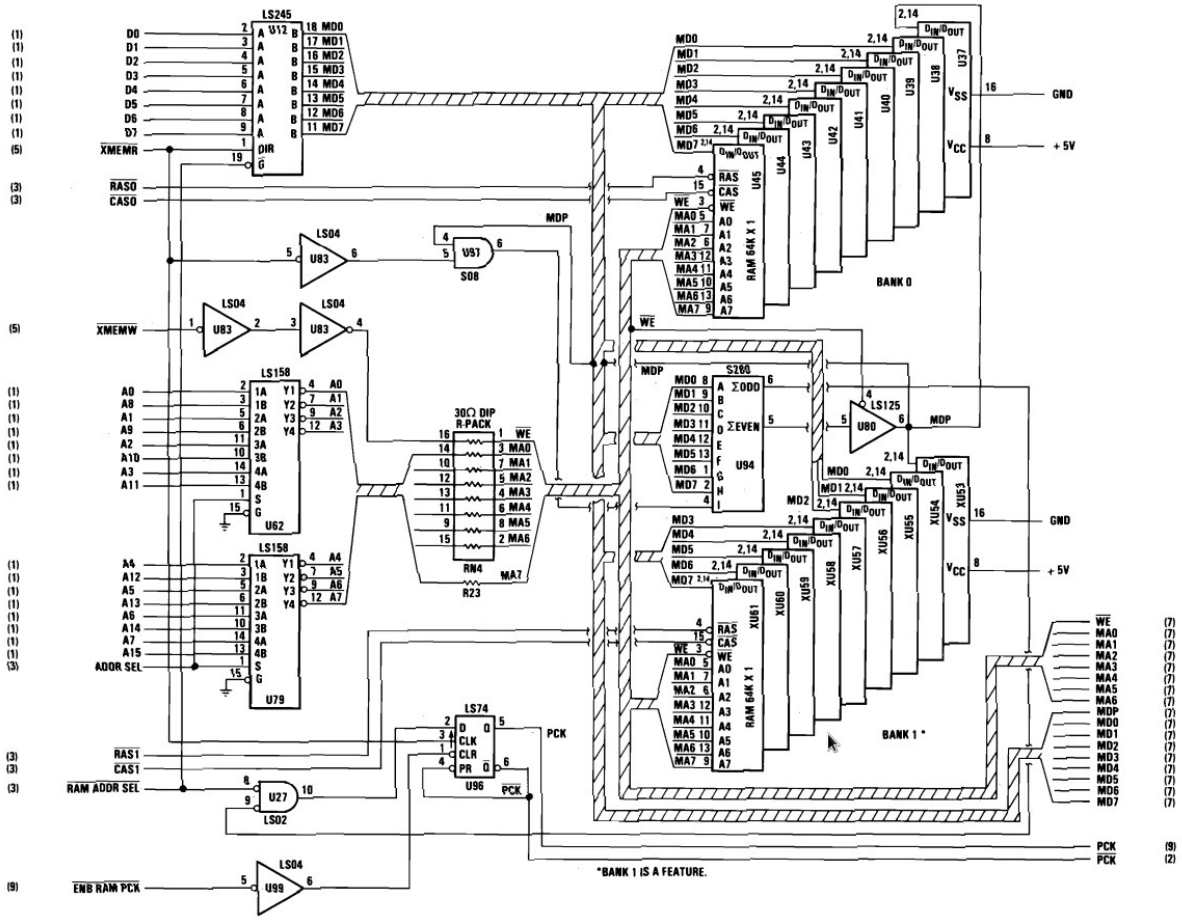
The fourth section of the motherboard consists of the 8237 DMA controller and its associated support chips. The purpose of this module is to perform DMA reads and writes. It includes latches to grab data from I/O devices and send it to memory. Most of the DMA operation takes place within this module.

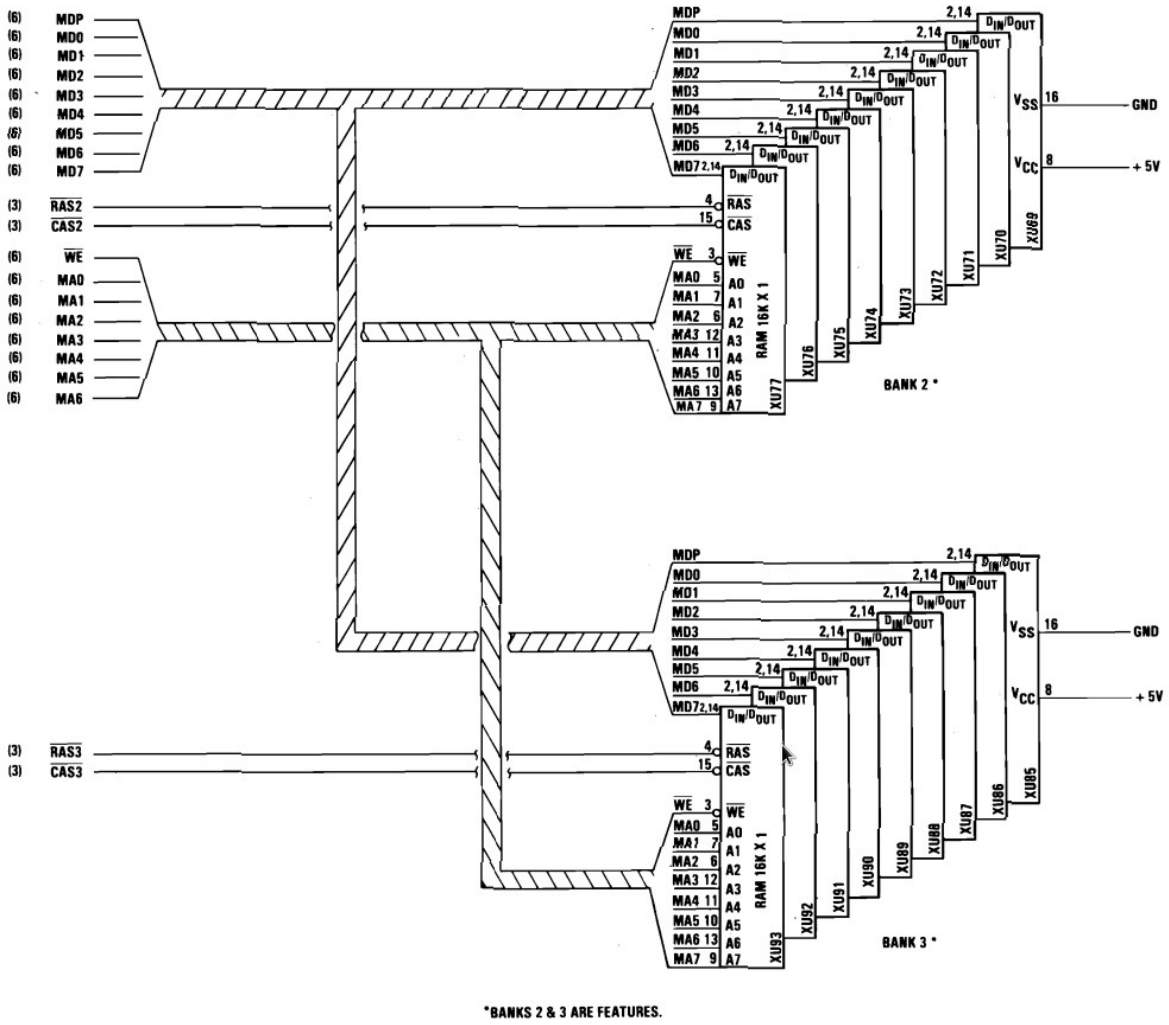
Motherboard Section 5



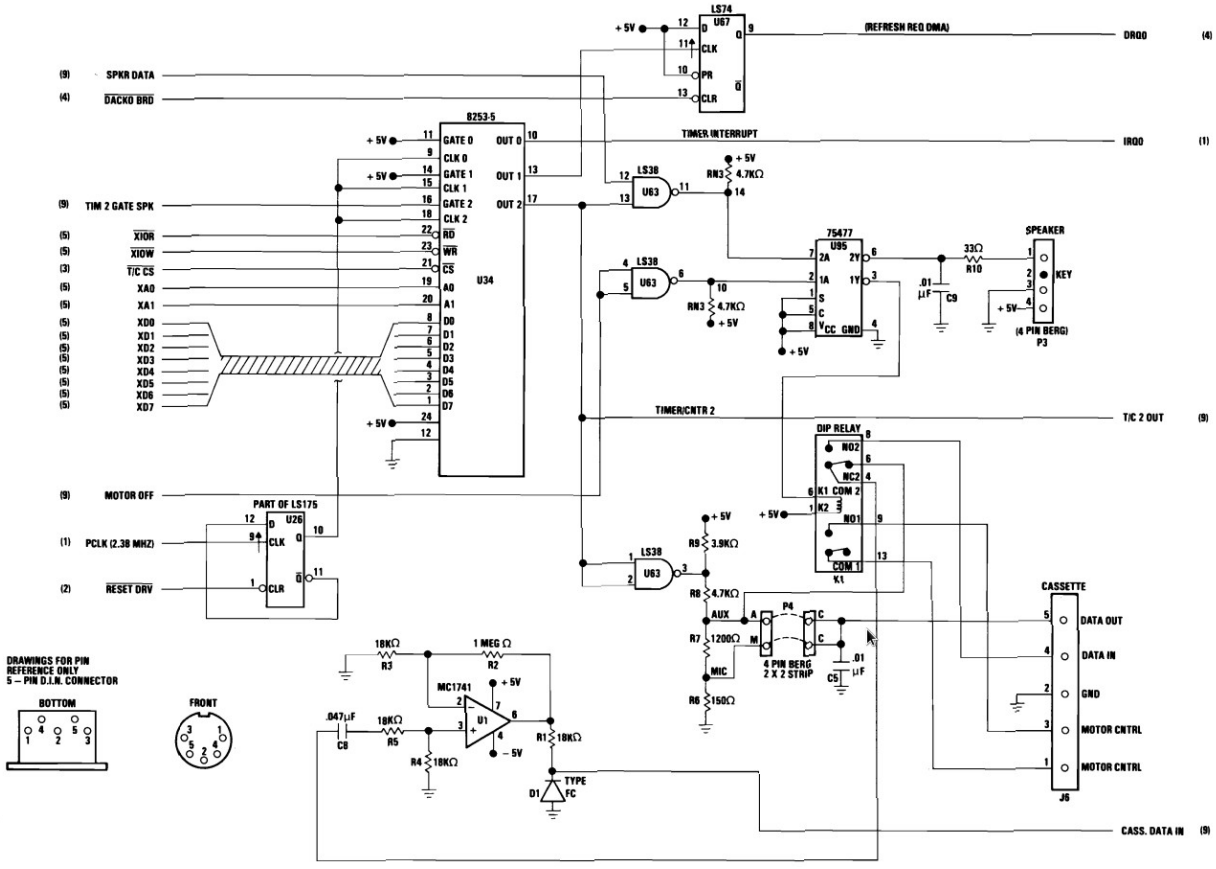
The fifth section of the motherboard contains the ROM. The ROM consists of eight banks, each of which is eight kilobytes, adding up to a total ROM size of 64 kilobytes. The ROM covers addresses 0xF0000 to 0xFFFFF. The first 16 kilobytes are reserved because the original IBM PC had only six banks of memory. The BIOS images we obtained were 64 kilobytes, hence we have instantiated eight banks of memory. These images leave the first sixteen kilobytes free.

Motherboard Section 6 and 7


LS245
U112
D0
D1
D2
D3
D4
D5
D6
D7
XMEMR
DIR
G
MD0
MD1
MD2
MD3
MD4
MD5
MD6
MD7
RAS0
CAS0
LS04
U83
MDP
U57
S08
XMEMW
WE
LS158
U62
U79
A0
A8
A1
A9
A2
A10
A3
A11
A4
A12
A5
A13
A6
A14
A7
A15
ADDR SEL
30Ω DIP
R-PACK
RN4
R23
MA0
MA1
MA2
MA3
MA4
MA5
MA6
MA7
LS74
U96
PCK
RAS1
CAS1
RAM ADDR SEL
U27
LS02
ENB RAM PCK
U99
RAM 64K X 1
U45
U44
U43
U42
U41
U40
U39
U38
U37
BANK 0
S286
U94
ΣODD
ΣEVEN
LS125
U80
U53
U54
U55
U56
U57
U58
U59
U60
U61
BANK 1 *
VSS
VCC
GND
+5V
*BANK 1 IS A FEATURE.

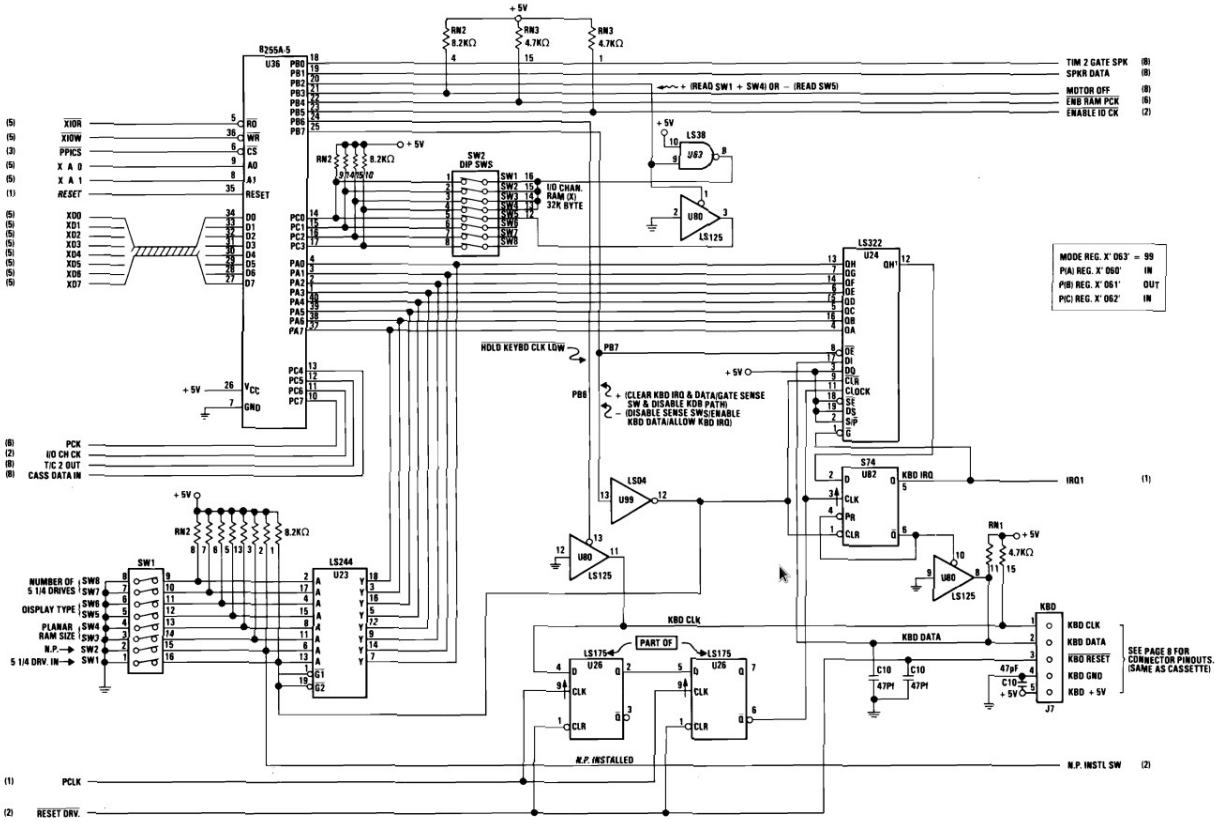


The sixth and seventh sections of the motherboard contain the RAM. The IBM PC could have anywhere from 64 to 256 kilobytes of RAM. We have chosen to instantiate the maximum of 256 kilobytes. The RAM is divided into four banks of 64 kilobytes each. Since there is plenty of RAM aboard the Virtex-5, we decided to instantiate all 256 kilobytes. In the original IBM PC, each RAM bank would consist of nine slices of RAM, each of which would hold 64 kilobits. The extra ninth slice would hold the parity of the other eight banks. This extra check was necessary at a time when RAM was less reliable than it is now. As a result, we decided to forego the parity storage. In addition, we decided to eliminate the row access strobe and column access strobe, which greatly simplified the implementation of RAM. The parity bit is fixed to zero, indicating a successful RAM read or write. The parity bit is made available to the processor via the 8255 programmable peripheral interface.



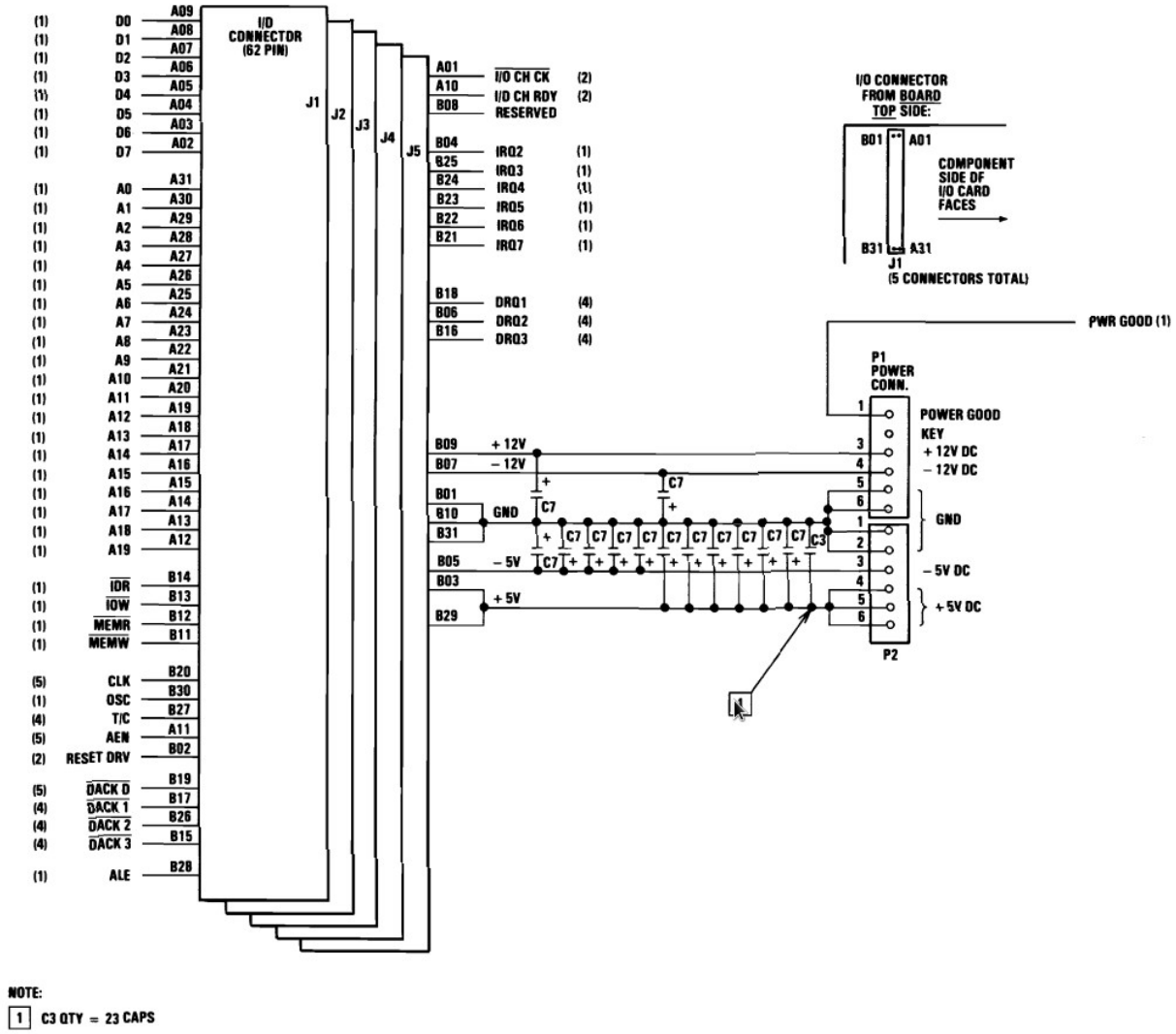
The eighth section contains the 8253 programmable interval timer, the speaker control, and the cassette drive control. The 8253 has three channels and controls three important devices. Channel 0 controls the timing for DMA refresh. Channel 1 controls the timing for the time-of-day functionality. Channel 2 controls the timing for the speaker. Channel 2 broadcasts a pulse train that, if activated by speaker data, makes noise on the speaker. Normally, the eighth section would have several relays and an opamp to control the cassette drive. However, we did not implement the cassette drive. Hence, it is not included in this section.

Motherboard Section 9



The ninth section holds most of the peripheral I/O. At its core is the 8255 programmable peripheral interface, which has been described in detail in an earlier section. It includes simulated DIP switches that govern system functionality. It also includes the keyboard controller, which receives data from the keyboard and interrupts the processor to make this data available. This section also generates several signals relevant to the operation of the cassette drive, which we did not choose to implement.

Motherboard Section 10



NOTE:
[1] C3 QTY = 23 CAPS

The tenth section holds the link between the main system and the I/O devices. In particular, it includes the VGA controller and the floppy drive. These devices would usually be connected to the system via cards that plug into a 62-pin I/O connector. We implement this functionality by declaring these modules inside the tenth section and linking them to our main system bus.

Interface Definitions

Control Bus

The control bus is one of the three major buses on the IBM PC. It includes all the signals that govern the operation of the PC. Such signals include: Address Enable, Address Latch Enable, System Clock, DMA Acknowledge 0-3, DMA Request 1-3, I/O Channel Check, I/O

Channel Ready, I/O Read, I/O Write, Interrupt Request 2-7, Memory Read, Memory Write, Oscillator, Reset Drive, and Terminal Count.

Address Bus

The address bus is one of the three major buses on the IBM PC. It is 20 bits wide and it holds the address of the data being accessed. It can allow for the access of up to one megabyte of memory. A0 is the least significant bit and A19 is the most significant bit. Either the processor or the DMA controller generates these bits. These signals are active high.

Data Bus

The data bus is one of the three major buses on the IBM PC. These lines provide data bus bits 0 to 7 for the microprocessor, memory, and I/O devices. D0 is the least significant bit and D7 is the most significant bit. These signals are active high.

Detailed Software Description

Software/Hardware Partition

The software on the IBM PC is stored in two hardware areas, the Read Only Memory (ROM) and the Floppy Drive. The ROM stores the IBM PC's BIOS and the Floppy Drive stores the IBM PC's Operating System. The software stored in each area is executed by the Intel 8088 processor. We have obtained the binary file of the IBM PC's BIOS.

ROM and BIOS

The ROM on the IBM PC consists of 64 kilobytes of memory. It contains two important pieces of software, the IBM PC's BIOS and a copy of IBM PC BASIC. The IBM PC starts by reading from its BIOS and determining if its floppy or cassette drive contains an operating system. If they do not, the BIOS starts loading IBM PC BASIC. This BASIC interpreter is very similar to BASIC found on other early computers, such as the Commodore 64.

We have made use of ISE's CoreGen to obtain block RAM to hold the binary file for our BIOS. We started out by testing it under simulation and our image can be read from block RAM. Once we had the whole system together, we utilized the BIOS to see if our system boots correctly.

Our first goal on start up was to get the BIOS to run without halting the system. If an error is discovered on startup, the BIOS halts the system. This proved a good way of debugging our design. Once we passed all the BIOS tests, we were able to move on to our second goal, which was to get the BIOS to boot us into IBM PC BASIC, which is included on the binary file

we have obtained. The below table details the 38 tasks the BIOS performs as part of the initialization and self-test routine. Some of these tasks were bypassed or failed. However, because their correctness did not interfere with the system operation as a whole, we could safely ignore them.

Step	Action	Comment	Reference
1	8088 PROCESSOR TEST - PASSED!	If the test fails, halt the CPU.	Page 5-34
2	DISABLE NON MASKABLE INTERRUPTS - NMI BYPASSED!	Disable NMIs from reaching the CPU.	Page 5-35 (line 370)
3	DMA CHAN 1 PAGE REGISTER - PASSED!	Zero the page register for DMA channel 1.	Page 5-35 (line 371)
4	DISABLE VIDEO - PASSED!		Page 5-35 (lines 372-376)
5	8255 OPERATION - PASSED!	8255 PPI chip. Set port B lines to outputs.	Page 5-35 (lines 377-378)
6	8255 OUTPUTS - PASSED!	Set the 8255's port B lines to various states.	Page 5-35 (lines 379-380)
7	ROS CHECKSUM TEST I - PASSED!	Verify that the checksum of the BIOS ROM, U33, is 00. If verification fails, halt the CPU. (U33 = 8 KB block at FE000)	Page 5-35 (lines 381-380)
8	TEST TIMER 1 - BYPASSED!	Timer #1 on 8253 interval timer chip. Used in RAM refresh process. If the test fails, halt the CPU.	Page 5-35 (lines 402-430)
9	INITIALIZE TIMER 1 - PASSED!	Initialise timer #1 with divisor of 18 - results in one pulse per approx. 15 µs.	Page 5-35 (lines 435-436)
10	8237 DMA TEST - PASSED!	Test 8237 DMA controller chip. If the test fails, halt the CPU.	Page 5-36 (lines 439-462)
11	START RAM REFRESH - PASSED!	Initialise and start DMA for RAM refresh (RAM refresh	Page 5-36 (lines 464-486)

		done via dummy DMA transfers).	
12	EXPANSION I/O BOX - ENABLE - PASSED!	Write 1 to port 213h.	Page 5-36 (lines 498-500)
13	BASE 16 KB RAM TEST - PASSED!	If a cold boot, test the first 16 KB of RAM. If the test fails, halt the CPU.	Page 5-36 (lines 502-508)
14	ZERO MOTHERBOARD RAM UP TO 64 KB - PASSED!	Size determined by examination of switches 3/4 on switch block SW1.	Page 5-36 (lines 509-520)
15	ZERO RAM PAST 64 KB - PASSED!	Conventional memory only. Upper limit determined by examination of switches 1 to 5 on switch block SW2.	Page 5-37 (lines 525-551)
16	8259 INITIALISATION - PASSED!	Initialise the 8259 interrupt controller chip.	Page 5-37
17	SET UP STACK SEG AND SP - PASSED!	Set up the stack segment and stack pointer.	Page 5-37
18	8259 TEST - PASSED!	If the test fails, beep 1 long then 1 short, then halt the CPU.	Page 5-38
19	TEST/SET TIMER 0 - PASSED!	Timer #0 on 8253 interval timer chip. If the test fails, beep 1 long then 1 short, then halt the CPU.	Page 5-38
20	INIT/START VIDEO CONTROLLER - PASSED!	Examine video switches on motherboard (SW1:5 and SW1:6) to see which type of video card is selected.	Pages 5-39 to 5-41
21	EXPANSION I/O BOX - TEST - PASSED!	If an extender card for the IBM 5161 Expansion Unit is	Page 5-41

		fitted in the 5150, then test communications with the 5161.	
22	CALCULATE TOTAL RAM - PASSED!	Calculate the total RAM.	Page 5-42 (lines 934-951)
23	ADDITIONAL RAM TEST - PASSED!	If a cold boot, test the RAM past 16 KB.	Page 5-42 (lines 952-998)
24	KEYBOARD TEST - PASSED!	If the test fails, display a "301" error.	Page 5-43
25	INTERRUPT VECTORS - PASSED!	Set up the interrupt vector table.	Page 5-43 (lines 1034-1049)
26	CASSETTE PORT - INTERNAL WRAP/LOOPBACK TEST - NOT PASSED! Current system works fine without it		Page 5-44
27	EXPANSION ROM SEARCH - PASSED!	Look for BIOS expansion ROMs in address block C8000 - F5FFF	Page 5-44 See note 1 for more info.
28	ROS CHECKSUM TEST II - PASSED!		Page 5-44
29	DISKETTE ATTACHMENT TEST - PASSED!	Only performed if switch 1 on switch block SW1 is in the OFF position.	Page 5-45
30	SET UP KEYBOARD BUFFER - PASSED!		Page 5-45 (lines 1188-1194)
31	8259 - ENABLE TIMER/KYB INT - PASSED!	Enable interrupts from 8253 timer [chan. 0] and keyboard.	Page 5-45 (lines 1195-1197)
32	DETERMINE LPT (PARALLEL) PORTS - PASSED!	Check for printer ports at the following I/O addresses, in that order: 3BC, 378, 278	Page 5-45 (lines 1198-1216)
33	DETERMINE COM (SERIAL) PORTS - PASSED!	Check for RS-232 serial ports at the following I/O addresses, in that	Page 5-45 (lines 1217-1241)

		order: 3F8, 2F8	
34	GAME CARD - PASSED!	Is a game card present?	Page 5-46 (lines 1242-1246)
35	SET LPT/COM TIMEOUTS - PASSED!		Page 5-46 (lines 1249-1259)
36	ENABLE NON MASKABLE INTERRUPTS - PASSED!		Page 5-46 (lines 1263-1264)
37	BEEP 1 SHORT TONE - PASSED!		Page 5-46 (lines 1267-1268)
38	DO BOOTSTRAP - PASSED!	Attempt to boot from floppy drive 0 (if switch 1 on SW1 is OFF), else try hard drive (if present), else run cassette BASIC.	Page 5-46 (line 1271)

We located BIOS images from several locations, but the most relevant images were located at http://www.vintagecomputer.net/ibm/5150/BIOS_dumps/. Of particular interest is the fact that there were three different versions of the BIOS. The first BIOS revision was April 24th, 1981 and is only for 16K/64K motherboards. We did not find an image of this BIOS. We did not want an image of this BIOS because we were implementing a 64K/256K motherboard. The second BIOS revision was from October 19th, 1981 and is only for 16K/64K motherboards. We found two images of this BIOS at the vintage computer website, 0159618.BIN and 0192562.BIN. There was no significant difference between these two images. Again, we did not use either of these images because they were for 16K/64K motherboards. The third BIOS revision was from October 27th, 1982 and is for 16K/64K motherboards and 64K/256K motherboards. We found two images of this BIOS at the vintage computer website, 0239462.BIN and 10048725150.BIN. The former features a copy of IBM PC BASIC 1.00 and the latter features a copy of IBM PC BASIC 1.10. We were initially unsure if BASIC 1.10 would work, so we used the former image with BASIC 1.00. Once we gained more confidence in our design, we added the code for BASIC 1.10 to the previous ROM image.

Floppy and Operating System

The BIOS on the IBM PC tests to see if any peripherals are connected that contain an operating system, such as a cassette drive or diskette drive. For our system, we have chosen to emulate a diskette drive rather than a cassette drive. Our original plan was to use CoreGen to instantiate block RAM to hold our floppy disk image. We found a copy of DOS on the internet to run on our PC. However, we did not finish the floppy drive. After getting DOS to run, we planned

to get a floppy disk image of a game and run it on the IBM PC. Again, the lack of a floppy drive prevented us from doing so.

BASIC and Games

Despite lacking a floppy drive, we were not deterred from developing a demo for our project that was fun and informative. Without a floppy drive, the IBM PC automatically loads its BASIC interpreter. One can write a program in this interpreter and execute it by hitting “F2”. However, typing a program in from scratch can be a long and error-prone process. To make demo easier, we developed a keyboard loader and keyboard translator to convert a BASIC program into keystrokes and load them directly into the BASIC interpreter. The keyboard loader is described under “Keyboard Loader” and the keyboard translator is described under “Keyboard Translator”.

By using the keyboard loader, we were able to load in a demo program that features two games, Hi-Lo and Game of Life, as well as two sound demos, Arpeggio and Imperial March. These demo programs are described in detail under “Game Description”. The games were obtained from a book of BASIC games from 1978. The wrapper program, Arpeggio, and Imperial March were written by us.

Keyboard Translator

In order for our keyboard loader to work correctly, we needed a way to translate text into scan set 1 keystrokes. Eliot, who had experience with scripting and web design, developed an online tool to convert BASIC programs into keystrokes. The translator can be found here (http://www.contrib.andrew.cmu.edu/~ecwong/). It provided us with a way to translate our prototype BASIC programs into keystrokes that could be placed into a CoreGen core as part of the keyboard loader. The code did require some debugging, but it was up and running within a few hours.

How We Built It

Our Approach

Our initial approach was based on making a prototypical IBM PC. In other words, we decided to build the system based on the the original motherboard blueprints. The module definitions and subsystem divisions were based off of the signals going into and out of the components on the original IBM PC. For example, the module for the Intel 8253 features most of the same signals as the real Intel 8253. The reason for adopting this approach was to ensure that the system works correctly by making the various parts match those on the IBM PC.

From a theoretical standpoint, this philosophy should work well. In practice, it was grossly unrealistic to make every component work exactly like it does in the real world. In some cases, we needed to make simplifications to the components to complete the project in time. For example, the Intel 8259 Programmable Interrupt Controller has multiple modes of operation, but the IBM PC only uses it in EOI mode. In addition, the operation of some chips needed to be modified to make them run predictably. For example, the Intel 8259 operates without a clock, latching everything asynchronously. However, making an asynchronous system work predictably is difficult, and debugging it is even harder. We ended up implementing the 8259 with an FSM that runs based on the CPU clock.

However, our prototypical philosophy carried us to our final design, with a few caveats. The final design has many of the same signals and interface definitions as the original IBM PC. However, many of the signals have been changed and parts simplified in order to suit our purposes.

Design Partitioning

Our design partitioning philosophy closely followed our initial approach to the design. Since we decided to implement the components as closely to the original as possible, we partitioned the project similarly. Everyone was given a number of chips to work on. Aalique Grahame would handle the Intel 8088 CPU and Intel 8288 Bus Controller. Meghan Kaffine would handle the Floppy Drive and Video Controller. Eliot Wong would handle the Intel 8237 DMA Controller and Intel 8284 Timer. Patrick Brown would handle the Intel 8255 I/O Controller, the Intel 8259 Programmable Interrupt Controller, the RAM, the ROM, and the various smaller chips. Once all the chips were complete, the group would implement the motherboard glue logic to tie the system together.

As in most projects, there often arises a situation in which responsibility for certain parts need to be reassigned. Verifying the 8088 took a bit longer than expected, which meant that Patrick Brown took responsibility for the 8288 Bus Controller. The floppy disk controller was several orders of magnitude more complex than we had anticipated. Thus, responsibility for the video display module was shifted to Patrick Brown.

Tools and Design Methodology

For an FPGA, we selected the Virtex-5 LX-110T board from Xilinx. There were two reasons behind our choice. First, it provided plenty of logic space for our project. We projected that we would need a lot of logic space for a project as immense as ours. Second, it appeared to provide VGA capabilities via the DVI port. We discovered later that it was not as simple as sending VGA signals from our modules to certain pins on the DVI connector. The DVI connector has its own processor which could not be bypassed. Instead we built a VGA connector of our own, but we really should have done more research before choosing our board.

For a toolchain, we made use of Xilinx’s ISE Design Suite. We would have preferred using the more modern Vivado, but Vivado does not work with this board. We had a number of

gripes with ISE. ISE takes a lot of time for synthesis, as much as ten minutes for our project. ISE's CoreGen tool generates extra files that interfere with synthesis. ISE is prone to dumping its own core. This happens more frequently with simulation than it does with synthesis. However, there is nothing more irritating that waiting ten minutes for a synthesis job then finding out that ISE has fallen flat on its face. Finally, ChipScope has a bad habit of editing out signals that you really need.

On the other hand, ISE was a relatively simple tool to learn. ChipScope proved to be a powerful debugging tool. That is, when it doesn't edit out signals. Even though synthesis was slow, it forced us to give careful thought to our changes before trying to re-synthesize. Having talked with other groups, neither ISE nor Vivado were especially fun tools to work with. Vivado was faster, but there was a steeper learning curve.

Our design methodology was relatively straightforward. We divided up the project into the various components on the motherboard. Each member of the group was responsible for a few components. We set deadlines for completion of each component. Each member was expected to have written and verified the component before the deadline. All the parts, minus the floppy disk controller, were completed by the end of October. Once all the parts were complete, we integrated them together. Once the parts were integrated, we began the debugging of the project. The debugging phase took the entire month of November. After debugging was complete, we developed our demonstration program and refined our final design.

Testing and Verification Methodology

Our testing and verification methodology closely paralleled our design methodology. We assigned each group member responsibility for a component. Each component is written and verified before a set due date. In theory, if all of the components work as expected, then the system should work as expected.

Reality is very different from theory. We knew early on that we were making assumptions about the operation of components that might be incorrect. In certain cases, we would have to rewrite components entirely, such as the Intel 8253. In other cases, we needed to make serious modifications to some components, such as the Intel 8259. However, this begs the question of how we find out if our assumptions were wrong.

Our salvation was the IBM PC's BIOS. The BIOS runs a gauntlet of 38 tests before letting the user into the BASIC interpreter. Within our IBM PC Hardware Manual was a well-commented description of all 38 of the tests. These tests formed the core of our testing and verification strategy after integration.

We quickly developed a post-integration testing and verification strategy based on the BIOS. For the first few tests, the system halts if the test fails. To figure out what went wrong, we needed to find where the test was failing. The Hardware Manual listed the address of each

instruction in the BIOS. We then use ChipScope to determine which test we have failed. We did this by setting ChipScope to record data when we pass a test's address. We then reset the machine. If ChipScope gets no data, then we need to backtrack farther. If ChipScope gets data, we look to see if we halt at the end of the test. If so, we pull any relevant signals and check the BIOS to see what we are doing wrong.

We gradually improved our strategy as we moved through the BIOS tests, fixing our components along the way. It became almost second nature around mid-November. By the time we passed all the tests, we had fixed most of the bugs in the system. However, using BASIC revealed a few more that we needed to solve.

The first was the beep function. If we called beep, the system would keep beeping and hang completely. We could not stop it after it started beeping. Everyone in lab hated us until we fixed the bug. It turns out the Intel 8253 was the cause of the problem. This prompted us to rewrite it from scratch. As it turns out, the computer keeps track of time using the 8253. The beep function is programmed to run for one-fourth of a second. If the 8253 does not work, the beep function will run forever!

The second bug was the execute state interrupt bug. This was the most evil, pernicious bug any of us had ever seen. When using our keyboard loader or typing on the keyboard, the system would lock up randomly, requiring us to do a hard reboot. The cause was the execute state of the Zet processor. There are a few instructions that cause the Zet processor to have an extra-long execute phase. If the Zet processor is interrupted during the execute phase, it will try to service the interrupt. Trying to service the interrupt causes the Zet to clobber its own stack, destroying the return vector. As a result, the Zet returns to an invalid location, loads an invalid operation, and traps itself in the invalid instruction internal exception handler. The solution was to only allow interrupts to occur during the fetch state.

Status and Future Work

At the end of the semester, we accomplished all but one of our goals. We built a machine that runs well, passes all of the BIOS tests, boots into IBM PC BASIC, and runs games from BASIC. However, the floppy disk controller did not come to fruition in the time we had available. If we had one more month and we all worked on it together, we think we could do it.

The floppy disk controller is the perfect thing for a future project. In fact, there are several parts of the IBM PC that we encourage future groups to work on. If anyone would like to base their project on ours, then by all means do!

Here is a list of things that can be done by future groups:

1. Make a floppy drive controller and boot into DOS.
2. Expand the VGA unit's capability to include 640 x 200 pixel graphics instead of our 80 x 25 character text.

3. Develop an implementation of the IBM PC's cassette drive.
4. Get a mouse working with Windows 1.0.
5. Look at the IBM 5160 documentation and make a hard drive controller.
6. Develop the Intel 8087 Math Coprocessor.
7. Get a joystick and the IBM PC Game Card working.

Original Schedule Plan

Task	**Due Date**
BIOS, RAM, and ROM	9/30
Keyboard and I/O Interface (Intel 8255)	9/30
Programmable Interval Timer (Intel 8253)	9/30
Direct Memory Access (Intel 8237)	10/9
CPU (Intel 8088)	10/10
Floppy Drive	10/10
Timer (Intel 8284)	10/15
Video (VGA) and Speaker (Piezoelectric Speaker)	10/22
Programmable Interrupt Controller (Intel 8259)	10/22
Bus Controller (Intel 8288)	10/22
Basic System Functionality	10/31
Debugging	11/15
Design Improvements	11/23
Demonstration Ready	11/25

Revised Schedule Plan

Task	**Due Date**
BIOS, RAM, and ROM	9/30
Keyboard and I/O Interface (Intel 8255)	9/30
Programmable Interval Timer (Intel 8253)	9/30

Direct Memory Access (Intel 8237)	10/14
CPU (Intel 8088)	10/17
Floppy Drive	11/07
Timer (Intel 8284)	10/27
Video (VGA) and Speaker (Piezoelectric Speaker)	10/22
Programmable Interrupt Controller (Intel 8259)	10/22
Bus Controller (Intel 8288)	10/27
Basic System Functionality	10/31
Debugging	11/15
Design Improvements	11/23
Demonstration Ready	11/25

What We Learned

What We Wish We'd Know at the Beginning of the Project

The number one thing that we wish that we had known at the beginning was how complex the floppy disk drive system was. If we had known that it would become the most difficult part of the project, we would have placed more people on the system than Meghan. We thought that the floppy disk drive system would take only two weeks. However, Meghan found herself unable to finish the system despite having spent several months on it.

The big mistake we made early on was not looking at project reports from previous years, especially those from projects similar to ours. In Fall 2012, two teams worked on implementations of the Apple II. Both teams tried to implement a floppy disk drive system. Both teams failed at that objective. If we had looked at reports from prior years as part of our research program, we would have realized that the floppy drive system could only work with a lot of hard work and team effort.

It would have been prudent to assign two people to the more complex components such as the floppy disk and main processor. This is something that we ended up doing in the later part of the semester, but looking back it would have been better to do this in the beginning.

Particularly Good Decisions We Made

The best decision we made was to make a plan early on. We developed a schedule at the beginning of the semester, laying out what parts we needed to complete and when we needed to complete them. We targeted October 31st for system integration, which gave us all of November to debug the system. Having a plan benefitted us in two ways. First, it told us when we had serious problems. If a part was running behind schedule, we needed to work together to solve the problem or work harder to complete the part. Second, it provided a bit of stress relief for us throughout the semester. Knowing that sticking to the plan meant completing the project on time provided peace of mind during the project's rough patches.

Another good decision was canceling the floppy disk controller before the last, most critical week. Instead of stressing about whether or not we'd be able to get it to work and integrated into the system without breaking everything, we just took what we had working and made it better. That allowed our demo to be more interesting. It also reduced the team's stress considerably, allowing us all to get some sleep before the public demo.

Particularly Bad Decisions We Made

Our worst decision was not appraising how long each part would take accurately. When we made our schedule, we assigned at two week completion period to nearly every part. What we should have done was research the part more before making assumptions about completion time. Modifying and verifying the processor was a very complex task that took longer than expected. The Intel 8253 was a far simpler part than we thought, but because we thought that it was very complex, we used the horrendous code from VCS. Both the 8255 and 8288 were completed in less than a day each, yet we allocated two weeks for each. Finally, the floppy disk controller was the cherry on top of our misappraisal sundae. We thought that it would take two weeks. In the end we did not finish it. If we had to do it all over again, at least two people would have been assigned to it for the entire semester.

Words of Wisdom for Future Generations

1. Look at project reports from previous years to get an idea of the practicality of your project proposal. If we had looked at the reports for the Apple II groups, we would have taken the floppy disk controller more seriously.
2. Take the time early on to research each part of your project and come up with a good estimate for how long it will take. If we had made better estimates for how long each part would take, we would have been able to balance work better between us.
3. Make a plan after you come up with good estimates for how long each component will take. Having a plan helps focus your energy, which prevents the end of the semester from becoming a panic.
4. If your project is going to include a floppy drive, have **at least** two people work on it for the semester. If you don't have two people to assign to this task full time, you aren't going to complete a floppy drive by the end of the semester. And the one person working on it will go insane and start having nightmares about verifying FDC code.

5. Don’t trust code that you didn’t make yourself. We were burned by the floppy disk controller and Intel 8253 code. It’s like the old saying goes, “if you want something done right, you have to do it yourself”.
6. Voraciously research your project before starting. The IBM 5150 Hardware Guide was our bible for the semester. The BIOS was full of helpful comments. There were system schematics and helpful insights. This project would have been impossible without the Hardware Guide.
7. Your assumptions may often be wrong, so have time in reserve to make changes. We made many assumptions about how the keyboard operated early on that turned out to be incorrect. However, we budgeted enough time to correct our early mistakes.
8. Get the right FPGA for the job. We chose the Virtex-5, but we planned to use the VGA module from Zet. The Virtex-5 does not have native support for VGA, it only has DVI that a VGA dongle can be attached to. We spent a couple of days building our own VGA connector and connecting it to our FPGA.

Individual Pages

Patrick Brown

Throughout this project I had multiple jobs. My primary job was as the team’s unofficial project manager. In this role I oversaw the project’s scheduling, balanced work between myself and my teammates, made major design decisions, and kept the project mostly on schedule. My secondary job was as a Verilog programmer. I oversaw the design and verification of the Intel 8253 Programmable Interval Timer, the Intel 8288 Bus Controller, the Intel 8259 Programmable Interrupt Controller, the Intel 8255 I/O Controller, the RAM, the ROM, the speaker system, the keyboard, the keyboard loader, various simple LS-series chips, the Video Display Unit, and the motherboard glue logic. In addition, I carried out much of the debugging work for the whole system. My tertiary job was as a researcher, obtaining documentation, BIOS images, and BASIC code. Much of the documentation I harvested early on provided the backbone of our reference materials.

One thing that I did not do was log how many hours a week I spent working on this class. I know that I spent considerably more time towards the end of the semester doing debugging than I did early in the semester writing the components. Other than this class, I had a relatively light load of classes, which meant that I spent a considerable amount of time on this project. I would estimate that at most I spent 40 hours a week working on this project. A few of the people in the other groups commented that they saw me there all the time, so maybe it was more!

18-545 was the best course I had at CMU. It was the first time that I had the opportunity to start a project and run with it. I got caught up in the inertia of the project very quickly and found myself enjoying every moment of it. I think the reason I became so engrossed in it was because two factors intersected perfectly. First, the project made use of all the skills I had developed throughout my CMU career. I found myself drawing on knowledge that I had obtained in 18-340, 18-341, 18-213, 18-349, and 18-220. Second, the project was challenging but not overwhelming. I never doubted that we could have something good to show off on demo day.

However, having something good to show off required a lot of sweat equity, which kept me motivated through a very long semester.

The best part of 545 was the ability to choose a project for the semester and work at it for the whole semester. Every day was a new challenge and I found it enjoyable. In addition, I picked up skills in debugging that will no doubt be useful in the future. It was beneficial to learn the tools that engineers in the field use, such as ISE. I also learned quite a lot about how to manage projects and work in groups. Overall, 545 was an exciting and enjoyable course that I would highly recommend to anyone in the ECE department interested in hardware design. In addition, I have no major complaints or concerns about the course. We had plenty of tools at our disposal in terms of hardware and software to get the job done. Even though we didn't place any orders for equipment or parts, it was nice to know that we had the option to purchase whatever we needed through the ECE department.

Aalique Grahame

My biggest responsibility in the project was the design of the main processor. As mentioned earlier, our goal was complete implementation of Intel's 8088 processor which was very similar to the 8086 processor. We were somewhat fortunate in finding a complete verilog implementation of the 8086 processor by another team online. I say somewhat because although this meant we didn't have to implement the processor from scratch which could have taken sometime, it also meant that we had to understand their code. If we didn't understand their code completely we were risking not having completely correct functionality once modifying the code. Actually, we may not have even known where to start when trying to modify their code because we simply would not know what did what.

Unfortunately, like most of the code that one finds online, this code was very poorly documented. Just about the only comment in the code was their big licensing statement at the very beginning. There also wasn't any good online documents that went along with the code to describe exactly how the system was functioning. Thankfully the modifications that would have had to be made to this code was very little, the main one being changing the size of the data bus from sixteen to eight bits. My initial approach was to simply seek out their data bus, change its width and then make little needed tweaks to their code to make sure that it still ran and gave correct results. I expected this to take no more than a few hours. However, a few hours went by and I was still pulling hairs trying to understand how their code was working, specifically how they were reading and writing data and how that data was being used. After a few more hours I realized that this just wasn't going to work. So I took a step back and after some discussion with my group, I came up with a new approach.

This new approach was to avoid touching any of their code as much as we could and write our own system that simply wrapped their system and used it for processing. This solution was very promising once we realized that we had the ability to stop and start the processing of their processor. Also, I was able to find some pretty good timing diagrams on the memory reads and writes for Intel's 8088 processor which I followed when implementing our wrapper. This took me just a few hours to design but I then ran into another problem. Although we had the

possibility to start/stop, it wasn't completely clear when I needed to do it. Fortunately, I just do happened to be reading one of the blogs on the Zet processor and came across a presentation that one of the members had given which included their FSM and datapath. This gave me an idea of when I needed to halt and it also revealed that I needed to add an extra state to their FSM, a fetch state. I believe their reads were combinational so they didn't need one. All in all, the processor took me about 6 hours to write and days to debug.

Once I was done with the processor, we jumped right into integration since we had about all of the different components completed. The rest of my time was spent helping with integration and with the long hours that we spent debugging everything.

I have to say that I really did enjoy this class. I like the idea of choosing a project of our own and taking the entire semester to implement. I also like that this course allows us to exercise much of the skills that we had acquired in previous ECE classes. My only complaint was about the tools we were using, specifically ISE. Every time we wanted to resynthesize our project, it took us about ten to fifteen minutes which was a huge waste of time. Besides that I think this is an awesome course and I which there were more courses that were strictly project based.

Meghan Kaffine

Originally, I was supposed to have had more than one job this semester. However, the first component I was assigned to turned out to be harder than expected, so I ended up working on the floppy disk controller for the entire semester, and still don't have a working FDC to show for it.

I had been waiting to take this class since I took 18-240 my sophomore year, and went to see the projects in Fall 2012. I knew this was the capstone I wanted to take, and planned my course schedule so that I would have all the requirements done. I also tried to make sure that I had a relatively light semester so that I could dedicate a lot of time to this project.

Unfortunately, in the beginning of the semester I got sick, and was a little behind schedule. I definitely was not able to dedicate all the time I had intended to when I was trying to make up work from other classes. However, as the semester moved along into October, I was able to spend more time working on the FDC. This lead to me discovering, a bit later than would have been liked, that the FDC was more complicated than expected. I ran into more problems like this, where code we borrowed was complicated and I had to spend a lot of time re-reading all the documentation and code we had so that I could come up with a verification program.

During November, I spent a considerable amount of time in lab with my eyes glued to the computer and all the documentation I had, trying to work on the verification of the FDC wrapper. ISE's waveform reader was nice, and made finding out why things weren't working easier, although still didn't solve all the problems. I spent most of the last three weeks of November in lab, trying to get the read command to verify.

Besides the fact that I spent three months of my life working on something that didn't actually end up in our final project, I enjoyed the class. I learned a lot about relying on other people's code (don't do it), and documentation (there was a straight up contradiction in the diskette documentation on what the memory addresses were), and the reasons why commenting code is important. It was also my first time really working on a group project that didn't have an answer. While at times annoying because I would like to have the answer to how to make the FDC work, it was also fun because I got to try and figure everything out without being able to have something to fall back on. There was no 'golden FDC' to compare it to, and I had to come up with the verification program on my own, figuring out what values should be expected, and when they weren't being received, try and find out if it was because I had calculated the values wrong, or there was something wrong with the code. It was also strange having other people relying on me to get my part done (even if it didn't happen).

Our project was interesting, and even though we couldn't do really complicated games, I found that our demo was still fun. The Game of Life can get really competitive. And I totally beat Patrick at it.

Eliot Wong

I admit that in the beginning of the semester, I felt pretty overwhelmed with the scope of our project and all the parts that we would have to implement for the IBM PC. This was combined with the fact that we had to learn how to use ISE and ChipScope and my skills with Verilog were average at best. It was very hard to picture being able to complete the entire project and being able to get it to a demo-able state. However, I'm very glad that as a team we were able to work together and complete the project.

My tasks for the project included designing and implementing the Intel 8237 direct memory access (DMA) chip, the Intel 8284 clock generator, the keyboard translator, a large portion of BASIC coding for the demo program, maintaining the team VCS, and initial software setup. The DMA controller was the most difficult task for me because type of documentation provided for the chip. Unlike other chips, Intel kept the internal design of the chip to themselves and only revealed a description of the behavior.

Once we knew that floppy disk was not going to be a feasible option for demo day, I began to brainstorm ideas for a more interesting demo. At the current time, our only plan was to type in a small BASIC program. I came up with the idea to use simulated keyboard to allow us to program much larger programs. As a result, I ended up working on a keyboard translator which would take in any text (or in our case BASIC program) and translate it to the keyboard scan codes.

Given that I was taking three ECE classes as well as being a TA for a 400 level CS course, my time that I could devote to this class was not as much as I would have liked. I estimate that I spent a minimum of 8-10 hours per week on the class. As the semester progressed the number of hours definitely increased.

Overall, I enjoyed the time I spent working on the project for 18-545. However, I am not sure that it was the right Capstone class for me. I am primarily interested in software and given that there is no software capstone, I ended up picking 545 because the projects seemed interesting. However, the class ended up being extremely hardware design focused which was not really my expertise or interest. I definitely enjoyed the class much more once we had gotten the system to run the BASIC interpreter and I was able to start creating the demo program. Given more time, I wish we could have gotten floppy disk and advanced VGA graphics to be implemented. This would have allowed me to increase the complexity of the demo program and it would have catered more towards my interest.

Appendix

Code Sources

Intel 8237: http://www.cs.ucr.edu/~dalton/i8237a/
Intel 8088: https://github.com/marmolejo/zet/tree/master/cores/zet
Video Display Unit: https://github.com/marmolejo/zet/tree/master/cores/vdu/rtl
Floppy Disk Drive: https://github.com/alfikpl/ao486/tree/master/rtl/soc/floppy
BIOS Images: http://www.vintagecomputer.net/ibm/5150/BIOS_dumps/
BASIC Games: http://www.vintage-basic.net/games.html
IBM PC Info Source: http://www.minuszerodegrees.net/index.htm
IBM PC BIOS Images: http://www.vintagecomputer.net/ibm/5150/BIOS_dumps/
Intel 8253: Located in the Synopsys VCS Installation Directory on AFS at /afs/ece/support/synopsys/synopsys.release/vcs-mx-2005.06-SP1/doc/examples/i8253/

Statement of Permission

If you have questions about the project, you can email us at:
Patrick Brown: newyinzer66@gmail.com
Aalique Grahame: aalique.grahame@gmail.com
Meghan Kaffine (only about floppy disk controller): mkaffine@andrew.cmu.edu
Eliot Wong: eliotw@cmu.edu
The source code can be found at: http://github.com/eliotw/IBM_PC

Errata

Project Proposal Video: https://www.youtube.com/watch?v=22WM9m0oEi8

Keyboard Translator: http://www.contrib.andrew.cmu.edu/~ecwong/