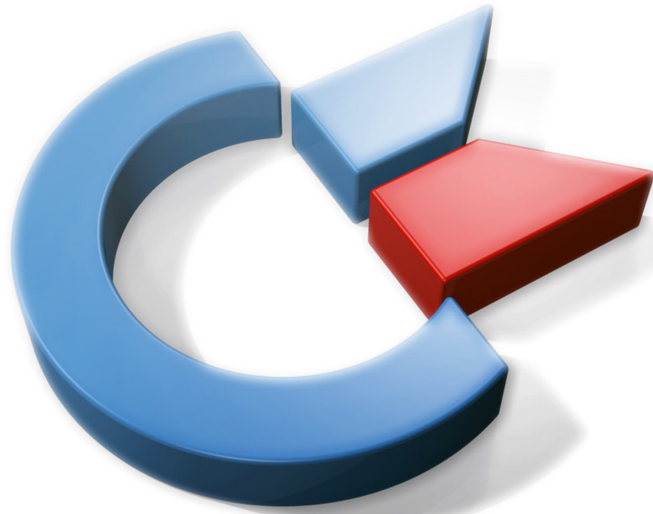


Commodore 64

<u>Team Angry Nerds</u>

Abeer Agarwal

Anirudh Reddy

Joe Berman

Sahil Jolly

Use or modification of this code for educational, research or personal use is allowed. Any attempt to make any financial gain off the use or modification of this code is strictly prohibited, without the expressed written consent of all of the aforementioned group members of Team Angry Nerds.

Table of Contents

WHAT WE BUILT 4
GAME DESCRIPTION 5
HARDWARE DESCRIPTION 6
SYSTEM DIAGRAM 6
MOS 6510 PROCESSOR 6
MOS 6567 VIDEO CHIP (VIC II) 9
MOS 6581 (SID 6581) 10
MEMORY/BUS SUBSYSTEM 12
INPUT DEVICES 13
SOFTWARE DESCRIPTION 13
HOW WE BUILT IT 14
MOS 6510 CPU 14
KEYBOARD DRIVER/ FRAME BUFFER 15
BUS 16
BLOCK RAM/ SOUND 18
TESTING AND VERIFICATION 20
CPU 20
BUS 20
GENERAL SYSTEM 21
WORDS OF WISDOM 21
INDIVIDUAL PAGES 23
ABEER AGARWAL 23
ANIRUDH REDDY 24
JOE BERMAN 25
SAHIL JOLLY 26
CITATIONS 27

What we built

Our group had set out to develop a fully functional Commodore 64 on a Virtex 5 LX110T FPGA. Commodore International developed the Commodore 64 in 1982 as a general-purpose computer on which you can play game ROMs. During its entire lifetime, it sold between 12 and 17 million units, making it one of the most popular computers of all time. The overall system architecture is fairly simple when compared to modern CPUs, which makes it a good fit for the scope of this class. We currently have a working model on which we can demo a variety of games and run the basic interpreter.

The system has three main chips within it. The processor is based on a MOS Technology 6510 microprocessor, the video chip is a MOS Technology 6567 chip, better known as the VIC II, and the sound chip is a MOS Technology 6581 chip, better known as the SID 6581. The details of each of these chips will be fleshed out below.

These three chips provide the brains behind the whole machine; however, the most crucial components of this system are the address bus and memory subsystem that allow for efficient communication between the chips. The bus proves to be an effective arbiter between the various chips and the memory subsystem. The memory itself is divided into a 64KB RAM, hence the name of the system, and a 20KB ROM, where the kernel and other data is stored that is used to run the machine.

On top of these main chips, we had to also take into consideration how we interacted with the board, in order to get the various inputs and outputs in and out of the system. We have developed several I/O board interfaces to allow for this. Each of the output interfaces will be discussed along with the chip that utilizes it. We currently use the keyboard for I/O by developing the relevant drivers. We also emulate the joystick using the numpad on the keyboard.

This project was primarily an exercise in hardware design, so we got our kernel and game ROM's from the Internet, although we gained knowledge in how game ROM's work and how they interact with the system.

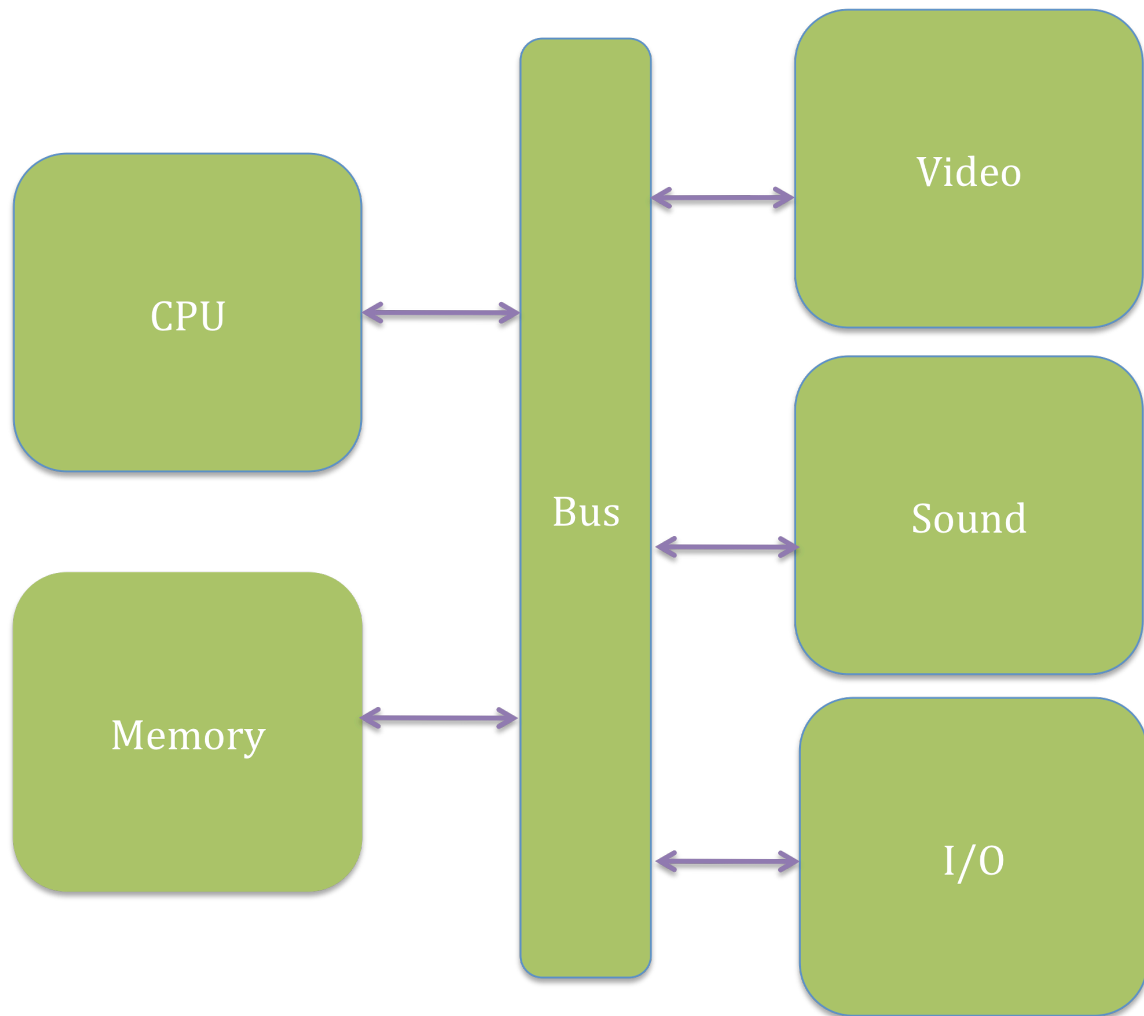
Game Description

Currently we have the following games working:

- Galaxian (8-bit): It is a popular old-school arcade shooter developed by Namco in 1979. The game involves the user controlling a ship at the bottom of the screen and shooting the descending enemy ships. This game fully works with correctly synced sound and no screen jitter or input lag. The scores and user lives update correctly and the explosions display correctly.
- Mario's Brewery (8-bit): This is an unofficial Mario game developed by Jeremy Thorne in 1983. This game is very similar to donkey kong involving having your character climb from the bottom to the top using ladders and jumping over barrels.
- Centipede (8-bit): This is a vertically-oriented shooting game developed by Atari in 1980. This game fully works with a slight delay in sound output.
- Pac-man: This game only partially works, the walls are not drawn correctly and the bottom half of the maze is not visible.
- Frogger: This game randomly works, most of the time the frog fails to get displayed.
- Basic Interpreter: Can execute BASIC commands and simple programs.

Detailed hardware description

System Diagram:



MOS 6510 Processor:

The Commodore 64 uses the MOS Technology 6510 microprocessor. The 6510 is a descendant of the much more ubiquitous MOS Technology 6502. The 6510 has an additional 8-bit general purpose I/O port that the 6502 does not. The 6510 does this by allowing the address location 1 to act as an 8-bit parallel input/ouput port. In the 6502, this location is usually used to bank switch between the Kernel/ROM and the usable RAM.

Apart form the above, the core architecture of the two processors is essentially the same. The 6510 is an 8-bit processor with a 16-bit address bus and 8-bits of data input and output. The processor uses a simple two stage pipeline, so that the next instruction is fetched and starts executing only if the current instruction that is executing does not store anything in memory.

Additionally, the processor has a two-phase clock that supplies two synchronizations per cycles, however our design currently only uses one. We do, however, use this one clock to emulate the behavior of two, so as to maintain correct timing across the bus. We are in the process of testing this to determine whether the overall number of cycles taken for a given instruction remains the same. The processor has a special AEC (address enable control line) that tristates the address bus, so that the address bus is valid only when this line is asserted.

The 6510 has the following registers:

- Accumulator register (8-bit): Used for arithmetic and logic operations.
- Index registers X and Y (8-bit): General-purpose registers used for loads, addressing modes etc.
- Program Counter (16-bit)
- Status register (8-bit): Which contains the status flags
- Stack pointer (8-bit): Mainly used for the JSR (Jump to Subroutine) and RTS (Return from Subroutine) instructions and for interrupt handling.

Additionally, the reserved memory locations used by the processor are:

- 0002 - 00FF (zero page): reserved for zero-page pointers. These pointers are used by zero-page addressing modes.
- 0100 - 01FF: 256-byte stack
- FFFA - FFFB: Pointer to NMI interrupt-processing routine.
- FFFC - FFFD: Pointer to a program handling Reset signal.
- FFFE - FFFF: Pointer to IRQ interrupt-processing routine.

The processor uses a variety of addressing modes (the indirect instructions are usually used for loops and array processing, the index registers are literally used as indices):

- Implied: the data value/data address is implicitly associated with the instruction
- Accumulator: uses the value stored in the accumulator as an effective address
- Immediate: the 8-bit data is provided as the second byte in the instruction
- Absolute: The 16 bit address is provided as the second and third instruction
- Zero Page: The second byte of the instruction points to a location in page zero (0002 – 00FF).
- Indexed zero page
- Indexed absolute
- Relative: Used for branching instructions. Branch instructions use an 8-bit offset relative to the instruction after the branch. Therefore, branches are limited to 128 bytes backward and 127 bytes forward from the current instruction.
- Indexed indirect
- Indirect indexed
- Absolute indirect

The processor also supports two types of interrupts, the IRQ (maskable) and NMI (non-maskable interrupts) which are handled as follows:

- NMI: When this occurs the program counter and processor status are stored in the stack, further interrupts are disabled and the processor jumps to the location of the handler at address 0xFFFA and returns using the RTI instruction. This interrupt cannot be disabled.
- IRQ: When the interrupt occurs the PC and processor status registers are stored in the stack, further interrupts are disabled and the processor jumps to memory location 0xFFFE, returns using the RTI instruction. This interrupt can be enabled or disabled using the CLI/SEI instructions.

The 6510 instruction set consists of around 56 legal instructions, including data moving, arithmetic, logic, control transfer and some assorted other instructions.

MOS Technology 6567 Video Chip (VIC II)

The MOS Technology 6567 video chip, better known as the VIC II, provided the video output for the Commodore 64. In the Commodore 64, the VIC II chip was also responsible for generating the system clock, as well as refreshing the DRAM, which is functionality that we will be bypassing in our design, as the LX110T already handles this on our behalf.

The VIC II chip has three character display modes and two text display modes. It also supported both the PAL and NTSC standards for video output, but we will be sticking with the American standard, NTSC. The VIC II chip is capable of displaying 16 colors, though some programmers have been able to hack the chip in order to create more visible colors. Additionally, the chip is able to display up to a resolution of 320 x 200 pixels, although that number may be reduced depending upon the video mode. The chip also utilizes

sprites to display certain patterns and has the ability to handle up to eight of these at any one time.

The chip utilizes 47 registers to generate the correct video signals. The processor can set these registers by writing to certain memory addresses that point to the registers. Those register include:

- X and Y coordinate Registers for each sprite
- Color Registers
- Background Color Registers
- Control and Interrupt Registers

The VIC II alternates memory accesses with the 6510 every clock cycle. The VIC II has control of the bus on the first half of the cycle and the 6510 on the second half. We plan to emulate this alternating behavior in our bus design. The VIC II also has a dedicated .5KB RAM used for generating colors. Every clock cycle the VIC II can access both the system RAM, as well as the color RAM. When the chip needs more than one cycle to load data in from the memory, such as when trying to load a sprite, the VIC II can take control of the bus for several consecutive cycles.

MOS Technology 6581 (SID 6581)

The sound card used in the Commodore 64 is the MOS Technology 6581, or as its more commonly known, the SID 6581. The SID 6581 was one of the most popular aspects of the original Commodore 64 and it is still popular today with many avid video game music fans and electronic artists. For many who played on the Commodore 64, the aspect that brings the most nostalgia is the auditory experience.

The 6581 consists of three synthesizer “voices” which can be used independently or in conjunction with each other. Each voice consists of a Waveform Generator, an Envelope Generator and an Amplitude Modulator.

The Waveform Generator controls the pitch of the voice over a wide range. It produces four waveforms at the selected frequency, with the unique harmonic content of each waveform providing simple control of timbre. The volume is controlled by the Amplitude Modulator, under the direction of the Envelope Generator. When triggered, the Envelope Generator creates an amplitude envelope with programmable rates of increasing and decreasing volume. In addition to the three voices, a programmable filter is provided for generating complex, dynamic tone colors via subtractive synthesis.

There are 29 eight-bit registers in SID that control the generation of sound. These registers include:

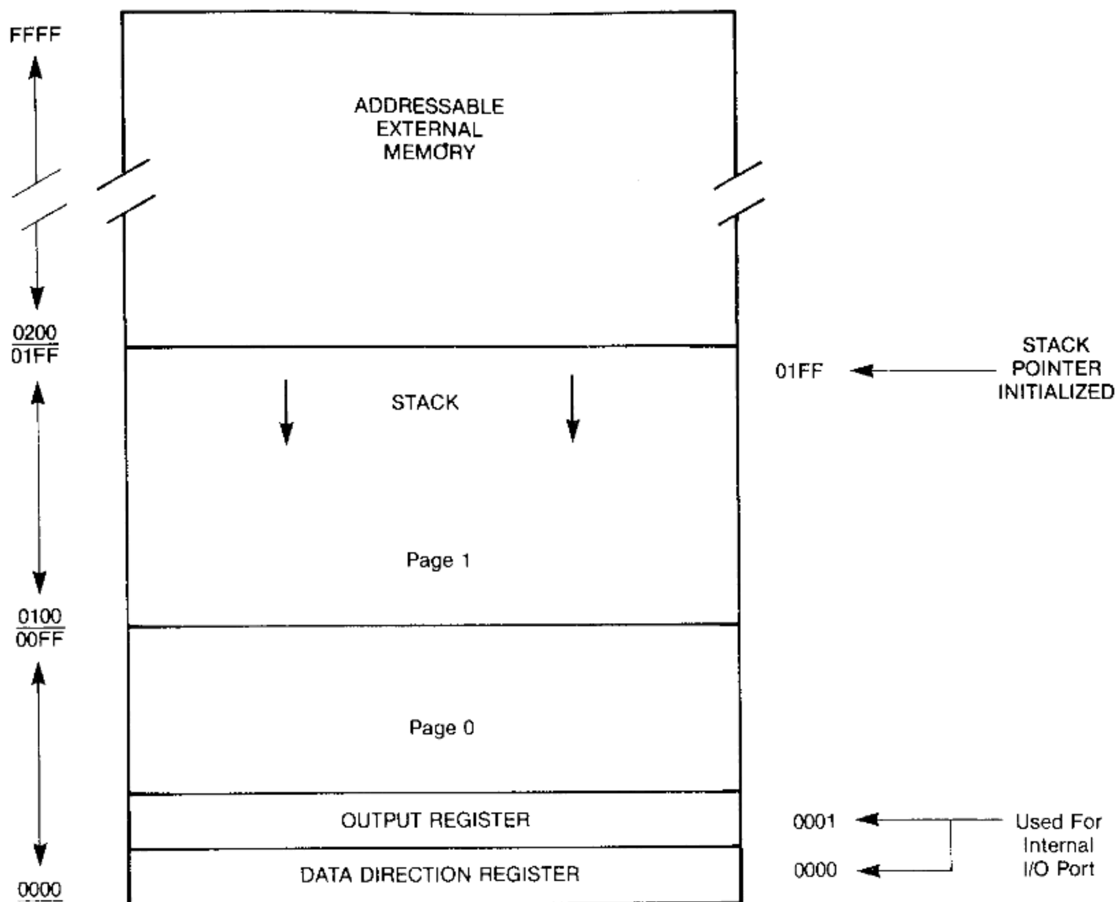
- Registers to control each of the 3 voices
- Filter Control Registers
- Overall Control Registers, including volume

These registers can be written and read in a manner similar to the registers in the VIC II, through memory mapped accesses. The SID 6581 allows the microprocessor to read the changing output of the third Oscillator and third Envelope Generator. These outputs can be used as a source of modulation information for creating various effects, and also as a random number generator for games. Two Analog- Digital converters are provided for interfacing the SID with potentiometers. These can be used for "paddles" in a game environment or as front panel controls in a music synthesizer, however, we won't be implementing this feature. The SID 6581 can process external audio signals, allowing multiple SID chips to be daisy-chained.

V. The Memory/Bus Subsystem

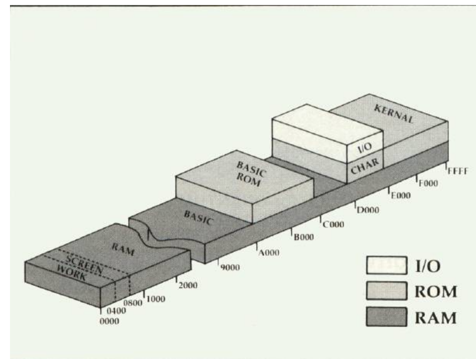
The memory bus interface connects the 6510 processor, VIC II graphics chip, 6581 sound chip, 64kB RAM and 20kB ROM together to efficiently and effectively communicate data. The interface has tri-stated address data lines, in order to allow multiple chips to control the bus. On a normal cycle, the video chip and processor alternate access to the memory bus, however, as discussed above, there are some cases when the VIC II takes control of the bus for several cycles.

The 64KB RAM is mapped as below from the processor's point of view. The processor can write and read from the registers in both the sound and video cards by accessing the memory values from $D000 to $E000, which are multiplexed with the RAM.



6510 MEMORY MAP

The 20 KB ROM is broken up into 3 main pieces, 9 kB for BASIC 2.0, 7 kB for the kernel, and two 2 kB character sets. It is also interleaved above the 64KB address space, in a manner similar to the video card and sound registers, as shown at right.



The Virtex-5 boards have 36kB blocks of memory, including parity bits, which can be configured as RAM or ROM. We, therefore, intend to use 2 of the blocks for the 64kB RAM and 1 block for ROM. The bus interface has separate address, write enable and data in pins from each chip and it outputs a single set of address, write enable and data in signals to query the memory on the board. The data that the bus receives is subsequently broadcast to all the chips simultaneously.

VI. Input Devices

The original Commodore 64 had support for a keyboard, as well as a mouse and joysticks. These input devices were controlled by a separate chip, which handled the timing and contention of the devices.

Software description

Ran the BASIC operating system that allowed BASIC commands to interact with the kernal.

How we built it and our approach

CPU 6510

The 6510 processor that we wrote was a simply lookup table based processor written in verilog. The 6510 ISA was simple enough that encoding each instruction separately in a 256-entry lookup table was the best option. Therefore, for each opcode read in we had a case statement that determined exactly which instruction was being read in.

At the same time we had to carefully follow the specifications that showed exactly how many cycles each instruction took. This was accomplished by putting in place a state machine that, depending on which instruction we were currently executing and which stage of the execution we were in, determined the next cycle to jump to. For example, on reset we had to read in the bytes at addresses 0xfffd and 0xfffe and jump to that address. The specification described that this should be done using a jump absolute instruction that took exactly three cycles including the opcode fetch cycle. Therefore, when reset went high the processor would generate a pseudo jump absolute instruction and the state machine would proceed onto the next cycle after the opcode fetch cycle. On this second cycle we would read in the byte at 0xfffe and move onto the final cycle of the jump absolute instruction. In this final cycle we would read in the last byte at 0xfffd, concatenate the two bytes that were read in and jump to that address on the next clock cycle. The state machine would then start again in the opcode fetch cycle.

We followed this general approach for the entire ISA because of the relatively small number of instructions. However, the problem with this approach was that generally each instruction had a variety of addressing modes like the jump instruction, therefore the opcode encodings jumped to around 256 towards the end. However, most implementations like the reference one we had, already had the hard coded opcode and state

encodings/transitions, so it was easier to use the existing working ones. This approach made it much easier in the debugging stage when trying to figure out which intermediate stages were going wrong.

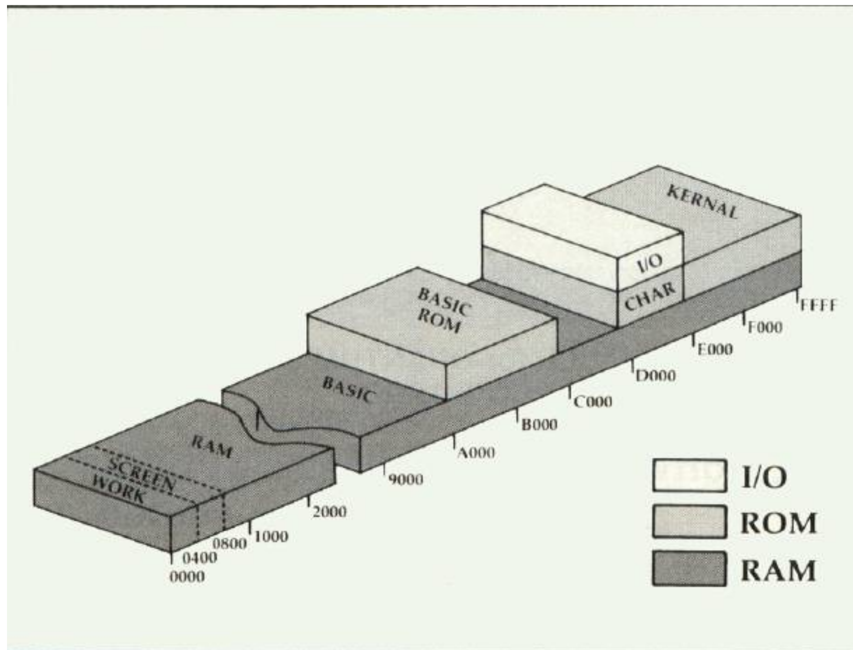
Keyboard driver (joystick emulation), frame buffer:

In order to get the keyboard working we had to first interact with the ps2 interface on the board. The ps2 interface uses a set of 2 pins, one as a derived clock and one as the data lines to transmit the 1 byte scan code and 1 bit of parity error checking. We also needed to filter out jitter that came across the lines. We wait for the derived clock to start cycling and then latch the incoming data for 9 cycles, at which point we check the parity. If it's good, we set a ready pin and pass the scan code out. The scan code we pass out, however, is not the same as what the Commodore 64 expects. We, therefore, have to pass it through what amounts to a huge case statement to change it to a legacy scan code. It is in this conversion that we emulate the joystick functionality, as we just change the press of a number pad key to the equivalent button press on a joystick. The new legacy scan code is then passed onto the CIA chips, which will signal an interrupt to the 6510 chip, which will then read the scan code.

In order to get the video output of the VIC II chip into a format we could use for the framebuffer, which talks to the DVI interface, several things had to be done. First off, the VIC chip outputs a new pixel every 2 cycles, but the framebuffer writes to the DVI interface every cycle. This means we needed to latch the output of the VIC, which includes the color, vertical sync and horizontal sync outputs to be used by the framebuffer. However, the color data that is outputted by the VIC is only a 4-bit color index, as the VIC only supported 16 colors. We, therefore, had to take the color index and produce the corresponding RGB values before it could be passed to the

framebuffer. The framebuffer itself is based off of the design of Team Dragonforce. We passed the data into the framebuffer from the aforementioned queue, which has the appropriate sync signals, as well as the transformed color signals. The framebuffer then transmits them through to the DVI interface using the appropriate control signals.

Bus



The C64 used 16 bit adresses. However, the total visible memory space was greater than 65536 bytes. So, the C64 used bankswitching, which we emulated in the bus. Depending on how certain bits are set, and which chip is using the bus, reading from the same address can mean several things. Ready from and writing to various chips is also memory mapped.

The CPU has three bankswitch bits coming out of it: LoRAM, High RAM and Char Enable. This first two decide which region of RAM to read from, while the third bit enables the character ROM. These are controlled by the CPU writing to a series of memory addresses, functionality implemented within the CPU. Various combinations of the Game and Exrom lines were

used for different sized game cartrdiges; originally, these were driven were resistiors built into the cartridges, whcih drove them high or low when they were plugged in.

10000	LHGE 1111 default	LHGE 101X	LHGE 1000	LHGE 011X 00X0	LHGE 001X	LHGE 1110	LHGE 0100	LHGE 1100	LHGE XX01 Ultimax
F000 E000	Kernal	RAM	RAM	Kernal	RAM	Kernal	Kernal	Kernal	ROMH(*
D000	IO/C	IO/C	IO/RAM	IO/C	RAM	IO/C	IO/C	IO/C	I/O
C000	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM	-
B000 A000	BASIC	RAM	RAM	RAM	RAM	BASIC	ROMH	ROMH	-
9000 8000	RAM	RAM	RAM	RAM	RAM	ROML	RAM	ROML	ROML(*
7000 6000 5000 4000	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM	-
3000 2000 1000	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM	-
0000	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM	RAM

We replaced this functionality by wiring these to a switch on the board, which allowed us to switch between the kernel and games on the fly.

Using these bankswitch bits, we generated various chip-select signals. Then, we used a simple demultiplexer with these chip-selects to decide which location we were reading from or writing to.

The bus is shared between the CPU the VIC. These are the only two chips that can initiate a read or write. The original protocol involved a complex two-clock system. We chose to not implement this. Instead, each of our clock cycles is divided into 32 sub-cycles, and either the CPU or the VIC has control of the bus during these sub-cycles: the VIC from cycles 13-16 and the CPU from cycle 17-32. They have to initiate a memory request every cycle. In case there is no need for a request, a read from a dummy address is performed.

Block RAM and Sound

After looking into options for RAM, we figured using auto generated block rams on the board would be the most convenient to use. So we generated the RAM using ISE coregen as per the spec for the address line width and data width, and tested it on the board to ensure correctness. We created a dual ported block memory module which was integrated into the bus and was used for the 64kB of RAM and 5kB color RAM needed by the VIC.

For the ROMs initially we wanted to use the on-board NOR flash, but after spending a couple of weeks trying to program the NOR flash using ISE and failing to get the data on as wanted, we gave up. Instead we used VHDL synthesizable arrays to put the ROMs directly on to the board. This worked really well, since the board had enough resources to hold the 20kB ROM as well as the game ROMs. We wrote an address translation module since the address queried by the CPU from the ROMs was bankswitched, and each individual component (kernel, basic and character) of the ROM had its own address space (0-$2000).

For the game ROMs, we found cartridge images of games online, which we had to scrub to remove the data stored for emulators, and get to the pure cartridge data. We then converted it to hex, and broke the hex file into

individual bytes using a Java script we wrote. The output of the java script was in a format that could be used directly by VHDL arrays which we put in a module to be connected to the main bus. We then wrote a small address translation module for the game roms.

For the sound, since we were running short on time, and since the SID 6581 was a popular sound chip, we found a VHDL model online. However the module had some Altera-specific components. One of the Altera specific components was a multiplier, which we were able to generate using coregen for our Xilinx Virtex 5 board. The other module was an analog to digital converter, which we did not need since those were being used by the paddle controllers in the original C64, however since we were not supporting paddle controllers, we figured we can do away with the component. we then integrated the sound chips with the rest of system. The model we found was outputting an 18-bit digital audio signal as well as a single bit analog PWM signal. We tried to get the digital signal to go through the AC97 and to the audio jack, however after toying with that for about a week we gave up, and the night before our public demo, we read online that we only needed to pass the analog output through an RC filter. So we hurriedly put together a 3.3V circuit, using 3.3kOhm resistor and a 4.7 nF capacitor, we drove the audio out from the SID module through an I/O pin on the board and were able to get sound out of the speakers.

Testing and Verification methodology

CPU

We wrote a test bench that stored the opcodes in memory and passed the memory array to the processor. I used a 6502 assembler to generate the object code and stored those hex codes in memory. We tested each opcode and the different addressing modes associated with that opcode and examined the results to see whether the execution took the correct number of cycles. I used these tests for both the reference implementations and our verilog implementation.

Bus

We spent a significant amount of time debugging the bus. Getting the timing to work was quite complicated. We could not test much of the bus in simulation, since the C64 took millions of cycles to boot up, and synthesis was slow for this. Also, we were using block RAMs and other components on the board including the keyboard, and could not easily be simulated. So, we used Chipscope along with its triggering mechanisms: this was a real lifesaver, definetely the most useful tool we had.

The initial verification for the bus was done by testing whether the processor could read instructions from memory, and do a simple read and write. We then tested whether it could write to various registers in the VIC and CIA chips: we added these registers as debug outputs to the top module, and tested their values using Chipscope. Once we had all the basic functionality working, we tried to get individual components of the system to start functioning fully, starting with the display, then the keyboard and finally the Game ROMs. In this process, we discovered bugs in both those modules as well as the bus. All of this was done using Chipscope.

General system

We put a lot of work into developing a strong testing infrastructure. The first part of this was to build a simulator script that streamlined the compilation and running of our mixed Verilog and VHDL code. When it came to actually testing out our system we did it in an incremental way. We worked to get the most basic components working first and then would slowly add more features as we found out that the previous version worked. This allowed us to really identify the problems that we were getting. However, at a certain point, we really had to make a leap of faith and synthesize our design on the board. At this point, Chipscope became an invaluable tool. Even with Chipscope, we still spent time in simulation trying to fix timing issues and trying to recreate problems in order to identify solutions. Once we got to the point of actually running kernel and game code on the system, the time spent in simulation to get to problems became extremely long, so we had to go almost exclusively to synthesis for the last few weeks.

Words of wisdom for future generations

Don't be afraid to ask for help from the other groups. They can be your best resource, as you will probably face many of the same problems, at least early on. Additionally make sure you stay on schedule and be realistic about what you can in a certain amount of time. When you start missing your own deadlines the pressure will start to build and that's just no fun.

Also, ChipScope is a big lifesaver so you should really put in the time to understand how it works. Getting the timing right is a very important factor so you have to be very careful when dealing with different clock domains.

We also learnt a lot of VHDL in the process of testing the CPU and other components. Most of the existing 6502 implementations were almost

completely written in VHDL, therefore understanding this language was very important. Also, rather than trying to reinvent the wheel by designing our own processor from scratch, using our own encodings and state transitions, we looked at existing implementations for ideas and used those while implementing it. This can be attributed to the ample research we did before we actually went ahead and got our hands dirty.

Individual Pages

Abeer Agarwal

Initially, I worked on getting the tool-chair set up. This consisted of setting up the Rodin TCL compiler for mixed System Verilog/ VHDL synthesis. After that, I looked into various options that we had for the processor, including converting a 6502 model into a 6510 one. I then set up a testing harness for the 6510 model we had, and wrote some initial tests for it using an assembler that I found, using a simple memory module for simulation. I then looked into getting a SID model that we could use a reference. We ended up using this model as is at the end of semester since we ran out of time.

Then, I started working on the bus, which ended up consuming a lot of time. We began by writing a simple interface between the VIC, memory and the 6510 and testing it. Once we had it working in simulation, I was part of an effort to use system ACE for flash memory on the bus. However, we could not get this to work. I then worked on integrating the generated RAM modules with the bus and putting it on the board. I then engaged in significant amounts of debugging to get the processor executing opcodes from the kernel. Once we had this, I worked on getting the conversion from the format output by the VIC to VGA working. This was quite challenging, since we had to deal with multiple clock domains. Once this was done, we had the kernel working and could run BASIC programs on it. Once this was done, we assumed that getting the Game ROMs to work would be trivial, since we were able to execute opcodes off it. However, we spent the next 3 weeks debugging the bus, and found several subtle bugs in it using Chipscope. This was extremely difficult, as it was hard to trace down the souce of the problems.

We finally got Games working on the Monday before lab demo day. We then worked on getting sound working: this went fairly quickly once we went the analogue route.

Anirudh Reddy:

Initially I helped out with generally researching our project and finding various references that would be useful for us in the future. At this point in time I was spending anywhere between 6-8 hours per week on this class. I initially did things like figure out how to boot from the flash card and put our bit file on it, instructions that we later used for our game cartridges.

Once we got started I dealt mainly with the CPU portion of the system. I wrote extensive tests for the reference implementation. However, before even starting I had to figure out how the CPU booted up, what it did on research and understand its timing. Once done, I wrote the CPU that was based off of the reference one. Again, I had to do extensive testing to correct timing issues and other bugs. At this point in time I was spending significantly more hours on the class.

The class was pretty challenging on the whole, apart from 18-487 I never had a chance to work on just one big project the entire semester. I prefer this to working on smaller inconsequential projects through the semester, as is the case with most classes.

Professor Bill Nace and TA Lincoln Roop were very helpful throughout the semester and helped us keep on track. (They also understood when we submitted status reports a day late ☺).

Joe Berman:

I did a number of things on the system. I was primarily responsible for developing the interfaces with the board. I worked on the framebuffer, the keyboard modules, as well as developed an AC97 module that we had planned to use, but didn't end up needing for sound.

I was also responsible for integrating all of the components together and therefore, did a lot of the debugging of the bus and the control signals. A lot of my time in the later half of the semester was spent using Chipscope to debug problems in the system.

For the first few weeks of the semester, I probably spent 6-8 hours a week on the class. By mid-semester break I was probably up to 12 or so a week. The last few weeks have seen me in the lab upwards of 20 hours per week. I don't know what that comes to in total for the semester, but it certainly was one of the most, if not the most, time consuming classes I've had at CMU.

I really enjoyed this class. I've always been interested in Hardware since I got to CMU and this course provides a really good culmination of all of the teaching I've received here. I think the video games aspect of the class definitely makes it more interesting. Additionally, I really like the freedom that Professor Nace provided us to do what we wanted to with the FPGA boards. I definitely learned a lot about team dynamics and the larger sense of project management. I feel that this course definitely provided a more real world perspective on what designing hardware in the real world holds for me.

Sahil Jolly

For the C64 I mainly worked on the Memory and Sound components of the module. The first couple of weeks were spent doing a lot of research, getting familiarized with the system, tool chain and reading a lot of documentation to understand how the various parts of the computer worked and talked to each other.

I also spent time initially figuring out mixed language synthesis and simulation which turned out to be a fairly easy task, which I found out only after hours of research and trying several simulation tools. There were missing C++ files on ISE's ISIM which I wasn't aware of. After also trying ModelSim, we ended up using VCS for mixed language simulation.

Overall I enjoyed the class, definitely got to learn a lot about the inner workings of a computer system. I feel we had pretty good team dynamics and people were aware of their responsibilities. Taking this class with people you already know well helps. Also, I feel the course staff guided us towards a realistic project to begin with and I'm glad that we had an almost fully functional system running by the end.

Citations

http://www.springerlink.com/content/3875725575657610/

http://www.zimmers.net/anonftp/pub/cbm/documents/chipdata/64doc

http://www.commodore.ca/manuals/funet/cbm/schematics/computers/c64/manual-html/large/Page_03.gif

http://www.6502.org/documents/datasheets/mos/mos_6510_mpu.pdf

http://www.syntiac.com/index.html

http://www.masswerk.at/6502/6502_instruction_set.html

http://codebase64.org/doku.php?id=base:6502_6510_coding

http://www.zimmers.net/cbmpics/cbm/c64/vic-ii.txt

http://www.oxyron.de/html/registers_vic2.html

http://www.antimon.org/dl/c64/code/missing.txt

http://www.waitingforfriday.com/index.php/Commodore_SID_6581_Datasheet

http://ist.uwaterloo.ca/~schepers/MJK/c64__.html

http://www.skinfaxi.com/cli/C64/hardarch.htm

http://www.unusedino.de/ec64/technical/aay/c64/krnromma.htm