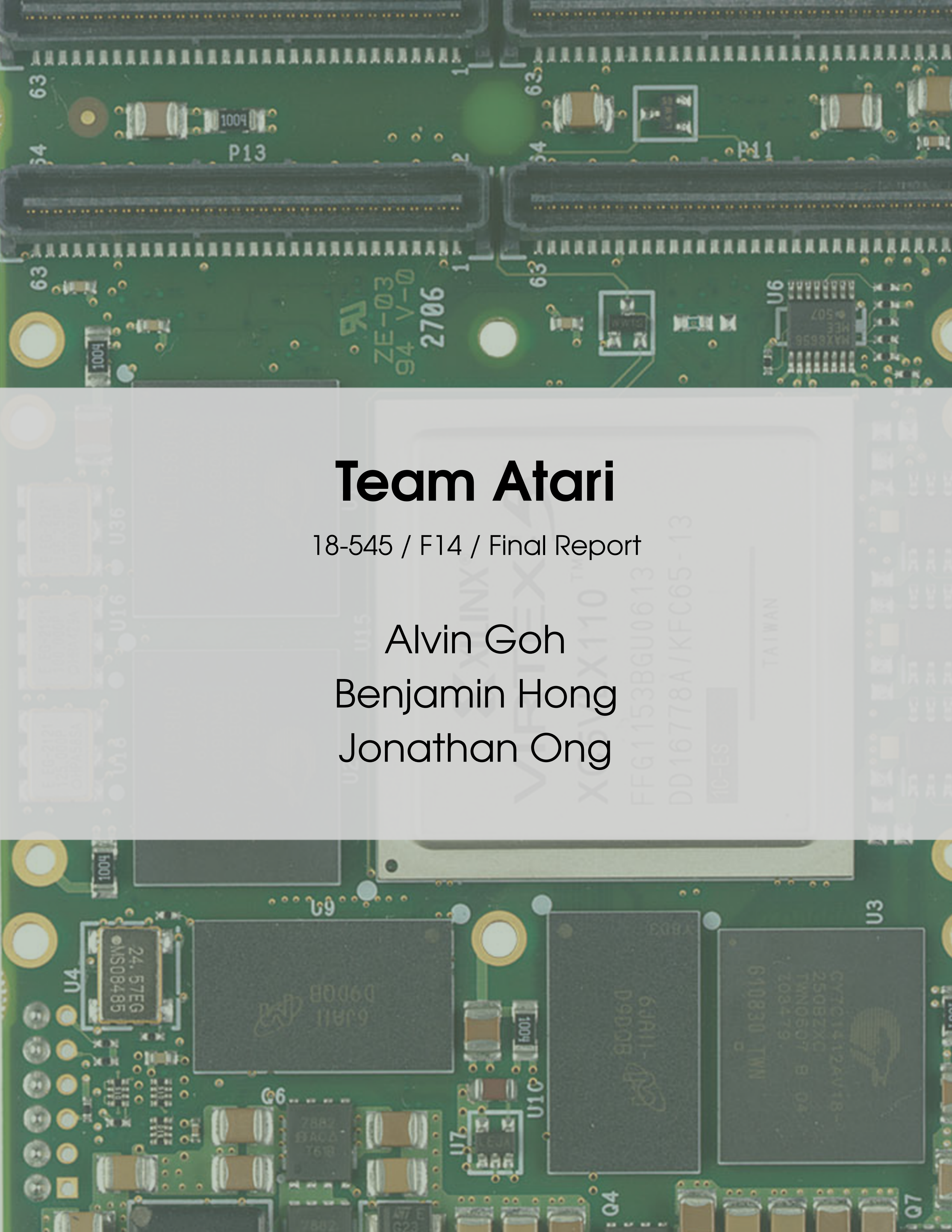


Team Atari

18-545 / F14 / Final Report

Alvin Goh
Benjamin Hong
Jonathan Ong

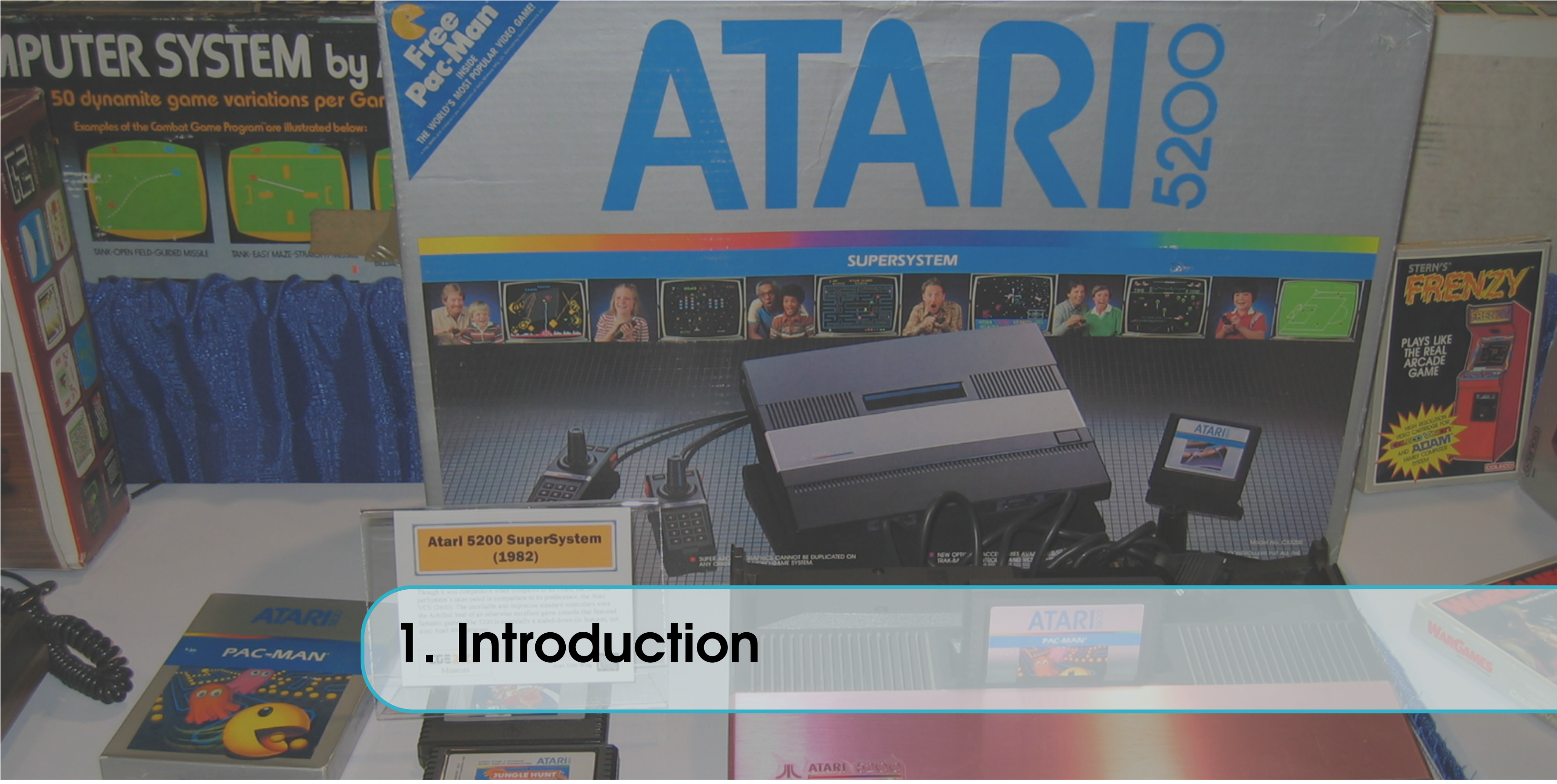
Statement of Use and Contact Information

Should you have any questions about this project, we can be contacted via email at:
Alvin Goh: chuehsig@andrew.cmu.edu
Benjamin Hong: bhong@andrew.cmu.edu
Jonathan Ong: jonathao@andrew.cmu.edu

The code repository can be found at:
https://github.com/chuehsien/TeamAtari-F14

December 8, 2014

Contents


ATARI 5200
SUPERSYSTEM
Free Pac-Man
Atari 5200 SuperSystem (1982)
PAC-MAN
FRENZY
PLAYS LIKE THE REAL ARCADE GAME

1. Introduction

1.1 Overview

1.1.1 Background

The Atari 5200 SuperSystem was a video game console released by Atari in 1982. It was based off Atari's existing line of 400/800 computers, and was the successor to the Atari 2600 console. It featured an analog joystick and numeric keypad, and could run a total of 69 officially released games.

1.1.2 Objective

The goal of Team Atari was to build a fully functional Atari 5200 video game console on an FPGA. We planned for our implementation to behave in a completely identical manner to the original console, with controller and cartridge inputs, as well as video and sound output.

We completed our objective of creating a working hardware emulation of the Atari 5200 on a Xilinx Virtex-5 LX110T FPGA. We also acquired original game cartridges and controllers, and managed to interface these physical components with the FPGA to create a realistic gaming experience.

We acquired 3 physical cartridges for testing, which contained the classic games of Defender, Mario Bros, and Pacman. These games allowed us to test different aspects of our implementation, such as two-player functionality, multi-directional joystick sensitivity, map and character display modes, and various types of sound effects.

1.1.3 Hardware Description

The Atari 5200 comprises 4 main hardware components – CPU 6502C, ANTIC, GTIA and POKEY. The CPU 6502C is a custom MOS technology 8-bit microprocessor with a clock speed of 1.79 MHz. The Alpha-Numeric Television Interface Controller (ANTIC) and Graphics Television Interface Adaptor (GTIA) chips handle the playfield graphics, as well as sprite and color overlay, to produce the television video output. The Potentiometer Keyboard Integrated Circuit (POKEY) is a digital input/output chip, responsible for input/output control, audio generation and interrupt handling.

We interfaced the original game cartridges and controllers with the FPGA. We also sent the graphical display output to a DVI/VGA monitor. Sound output was sent to external speakers.

1.1.4 Software Description

Figure 1.1 shows the structure of our source code on GitHub.

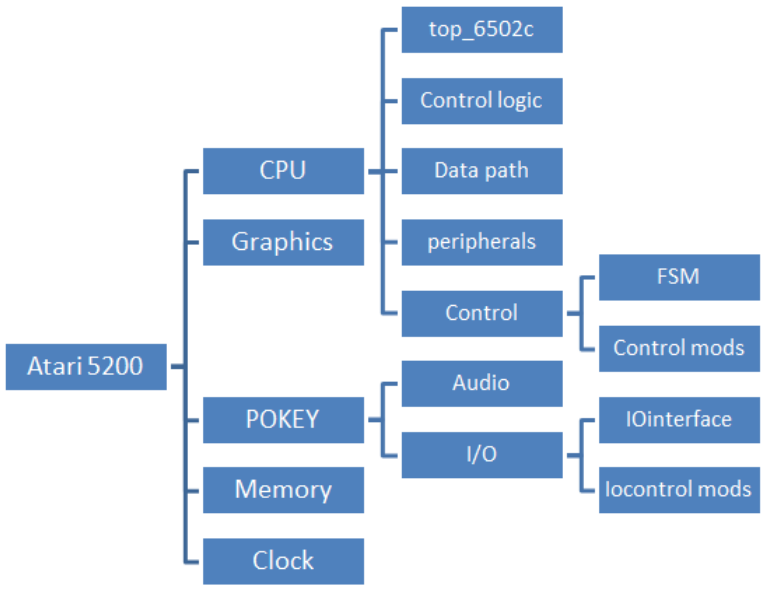


Figure 1.1: Source Code Structure

1.2 Development Tools

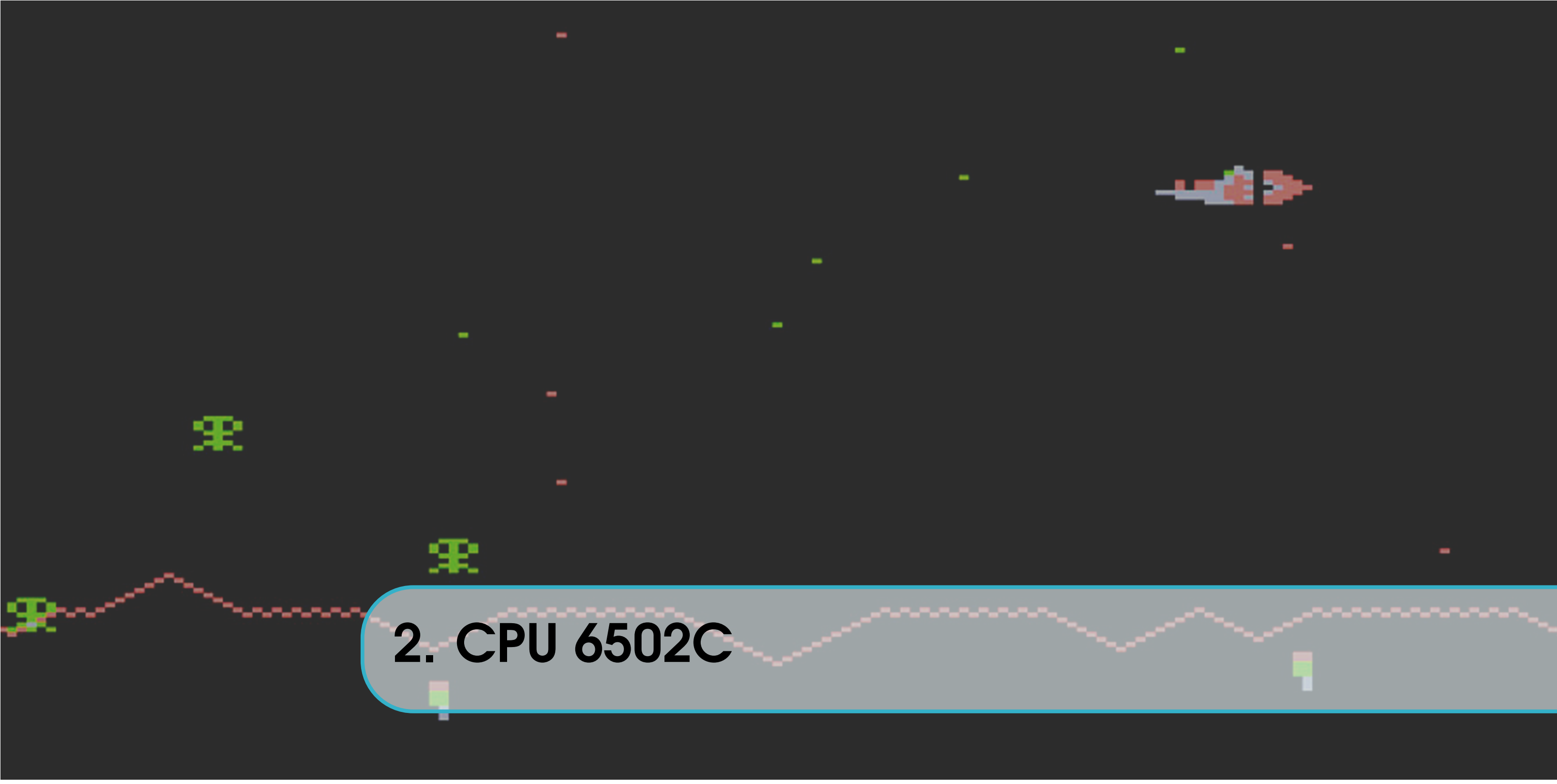
1.2.1 FPGA Board

Our project used a Xilinx Virtex-5 LX110T FPGA. We chose this option over the other available FPGAs, as we felt that using ISE might prove more reliable than Vivaldo. There was also considerably more information regarding debugging common errors on ISE and the Virtex-5 boards.

1.2.2 Software

We used ISE 14.2 to interface our Verilog code with the FPGA board. Other useful Xilinx tools that we utilized include the CoreGen program to generate common modules, such as clock dividers, block memory, and so on. We also initially used ModelSim to test our code in simulation, and later on heavily utilized ChipScope to view the bit values on board while attempting to debug our code on the actual hardware.

We used GitHub for our code repository, as it provided version control for our project, and we were familiar with its usage.



2. CPU 6502C

2.1 Specifications

2.1.1 Overview

The 6502C is an 8-bit microprocessor designed by MOS technology. Many variants of the 6502 were used in popular video game consoles, such as the Atari, Apple II and Nintendo Entertainment System. The ‘C’ in 6502C refers to the customized version of the 6502 that was used in the Atari 5200 console. This ‘C’ version introduced an additional ‘HALT’ pin, which allows other chips in the console to halt the execution of the CPU at any time. More information will be provided about this pin.

General characteristics:

- 8-bit microprocessor
- 16-bit address bus
- 8-bit data bus
- 56 instructions
- 13 addressing modes

Figure 2.1 shows a block diagram which contains detailed information about the 6502C microprocessor. The data path components are outlined in red, and the control path components are outlined in blue. These two components will be explained further in detail.

2.1.2 Data Path

The 6502C uses a 2-phase non-overlapping clocking scheme, with clocks phi1 and phi2. Interactions with external modules (e.g. memory) occur on phi1 clock ticks. Internal operations, such as loading of registers and latching of signals, occur on phi2 ticks.

The data path consists of registers and an Arithmetic Logic Unit (ALU) connected by 4 different bus lines – the Data Bus (DB), Special Bus (SB), Address Line Low (ADL), and Address Line High

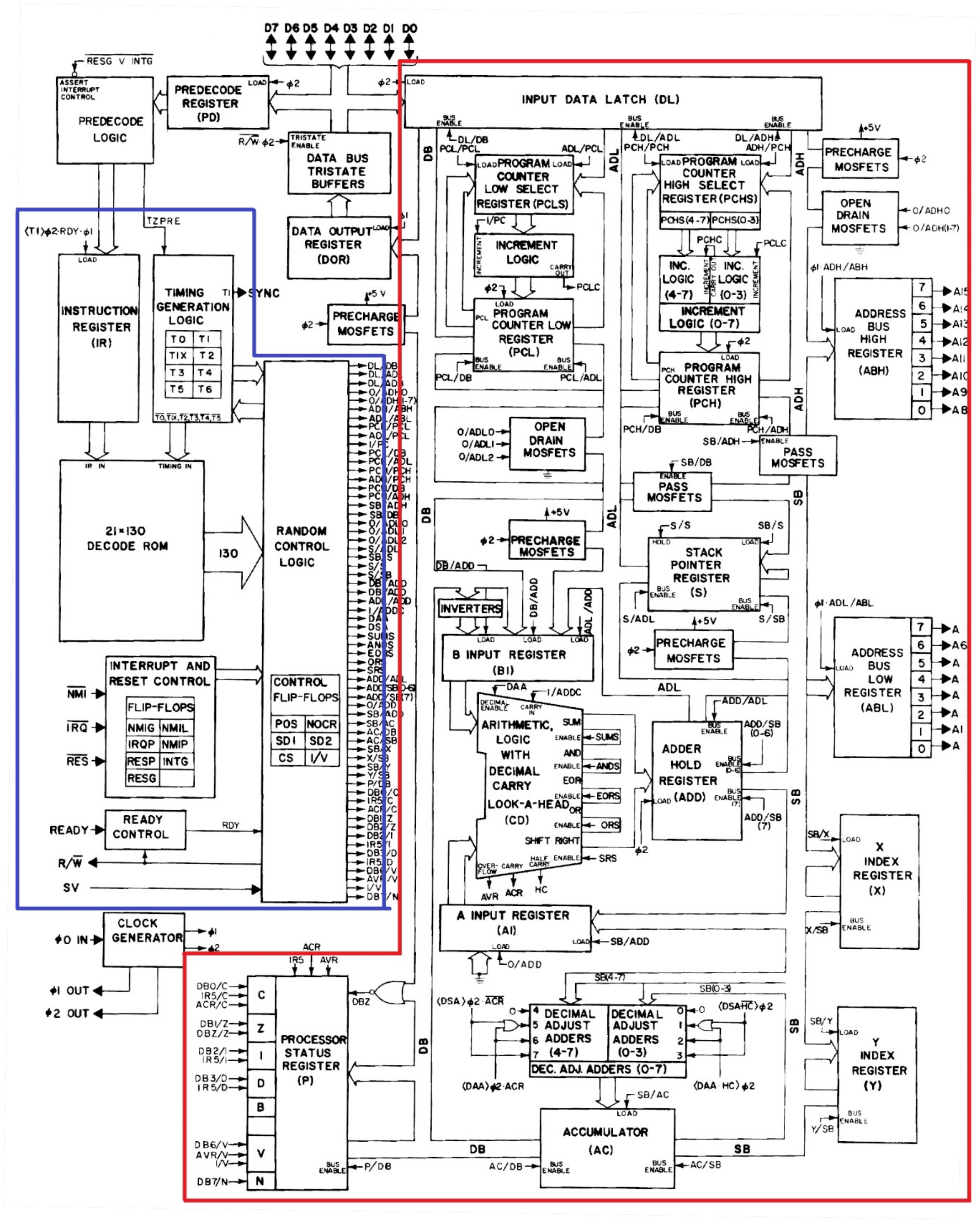


Figure 2.1: CPU 6502C Block Diagram

(ADH). These buses are connected to pull up MOSFETS, which pull up the data lines on every phi2. Open drain MOSFETS are also connected, and can be activated via control signals to force bus lines to 0. Bidirectional pass MOSFETS are also activated via control signals to bridge the SB and DB, as well as the SB and ADH when needed.

2.1.3 Control Path

The 6502C does not use a FSM in a fetch-decode-execute cycle. Each instruction consists of 2 or more micro cycles, which go from T2 -> T3 -> … -> T6 -> T0 -> T1, and the current state of execution (termed 'T' state) is maintained by a 'timing generation logic' which encodes the T states with one-cold encoding, and uses a shift register to move from state to state.

The original 6502C employs a primitive form of pipelining by fetching the new opcode in the T1 state of each instruction. The control signals to the data path are determined combinationally in the following sequence, as shown in Figure 2.2.

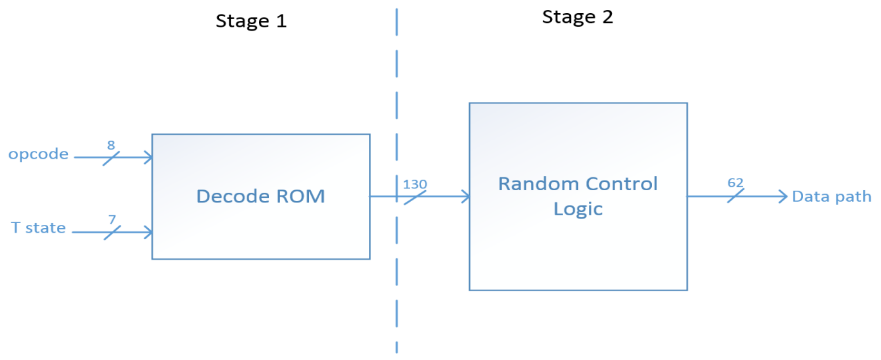


Figure 2.2: Opcode Sequence

Stages:

- Stage 1: The opcode and the 'T' state are inputs to a combination block, the decode ROM, which is a 130 x 21 bits Programmable Logic Array (PLA). All lines of the PLA compare the instruction and the current state, and if they match, the corresponding line(s) fires. The output is 130 bits width, which goes into the Random Control Logic.
- Stage 2: In the Random Control Logic block, a series of combinational circuits determine the exact datapath control signals to be fired. These 62 signals are then connected to the datapath components.

Since control signals are determined combinationally, there is no decode state involved in the 6502, resulting in a simple fetch-execute cycle.

2.1.4 HALT Signal

The HALT signal is used by other off-chip components to stall the CPU. In the Atari 5200, this served the purpose of allowing for Direct Memory Access (DMA) by the ANTIC chip. The CPU writes display instructions into RAM, which must be subsequently retrieved by the ANTIC chip. To prevent conflicts on the bus lines, the HALT line is asserted low by the ANTIC chip before DMA takes place. When the HALT line is held low, internal execution still occurs, but read/write operations are stalled.

2.1.5 RDY Signal

The RDY signal is often confused with the HALT signal. The RDY signal is available also on 6502 chips without the ‘C’ suffix, and is used to stall the processor, except on write cycles. When the RDY line is held low, the processor will finish up any write cycles, and stall at the next cycle. The main difference from the HALT pin is that when the RDY line is held low, the bus lines are not tri-stated, and the current address is maintained on the bus. The purpose of the pin was to allow the CPU to interface with slower PROMs, thus the address must remain on the bus. This pin also allows other chips/developers to single cycle the CPU if needed, and observe the external bus lines. In the Atari 5200, the RDY pin is always held high.

2.2 Implementation

2.2.1 Data Path

The data path is not complex, so we were able to implement all of the components following the schematics found online. The only difference between our implementation and schematic are the decimal adjuster. The schematics were rather unclear of the exact connections in and out of the adjuster, so we wrote our own decimal adjuster to attain the required behavior.

2.2.2 Control Path

The decode ROM is quite well documented on the internet, and we were able to figure out how it works, and which lines fire for which instructions. However, the random control logic is a black box, and no one has reverse-engineered the internals of the component. All we knew was that the 130 inputs will affect the 62 control signals which come out of the random control logic. Since there was no point getting 130 outputs from the decode ROM and not knowing how they affect the control signals, we felt there was no point implementing the decode ROM.

In the end, the design decision made was to combine the decode ROM and the Random Control Logic block, into a bigger combinational circuit. From each opcode and T-state combination, we checked the outputs of an online 6502 Javascript simulator, and set up a look-up table to derive the needed control signals. A few miscellaneous inputs such as status register flags and interrupt status were also needed by this combinational logic. For example, if decimal mode is set, the decimal adjuster is activated to modify results before being fed into the accumulator.

The BRK instruction is used for the initialization, and handling jumps to interrupts vectors. That is, when NMI_L is asserted low, the opcode 0x00(BRK) is inserted into the instruction register. The BRK instruction is then carried out, with the NMI jump vector loaded instead of the BRK jump vector. This is the same for IRQ_L and RES_L as well, with IRQ and RES jump vectors loaded instead.

We decided to use a FSM to store and update the T-states. Figure 2.3 illustrates the FSM and the combinational circuit working together. Thus, on every clock tick, the T-state is updated, and the updated T-state, combined with the opcode and other signals go into the combinational logic to form the datapath control signals. The next T-state is also determined, and ready to be ticked in at the next edge.

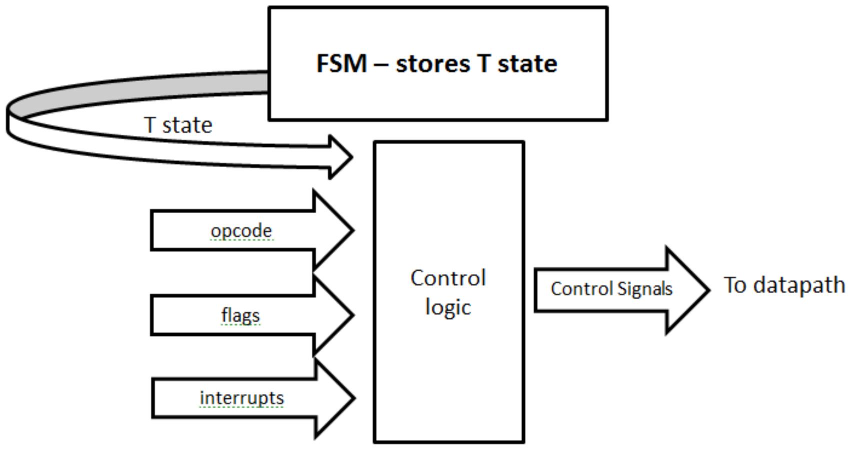


Figure 2.3: FSM Implementation

2.2.3 HALT Signal

Our implementation also handles the use of the HALT signal. The HALT signal is latched on the phi1 tick, and all CPU activity stops. When the HALT signal is de-asserted again, it is updated on the phi1 tick, and CPU activity restarts from the phi2 tick.

2.2.4 RDY Signal

The RDY signal as described above is not implemented, because schematics show that the RDY signal is always held high in the Atari 5200. For our implementation, the RDY signal is used as an output to signal that the external buses are tri-stated and that the graphic chip (ANTIC) can take over the data and address bus lines.

2.3 Challenges

2.3.1 Page Crossing Detection

Some of the 6502C instructions result in page crosses, which means that the Hi byte of the address line needs to be incremented (e.g. fetching an absolute address from 0x0099 would require reading from 0x0099 and 0x0100). These page crosses result in additional cycles, and can only be determined from the carry out of the ALU in the previous micro cycles. These flags need to be sent to the control logic at the right cycle as well, in order to get the T state updated correctly. The schematics do not show these intricacies of the execution cycles, so we coded in workarounds to carry out these instructions properly. In particular, the ACR and AVR of the ALU is latched into the ALU hold register on phi2, then latched into another register on phi1. This allows the lifetime of the ACR and AVR to be extended and used to determine the control signals in the next cycle.

2.3.2 Hazards During Clock Ticks

There were stability issues caused by hazards. For example, during the phi1 tick, data on the internal ADH is ticked into the ABH (which links to the external address bus). However, during the tick, data on the ADH changes momentarily while combinational logic determines the new control signals. These hazards in the control signals destabilized the ADH, causing wrong values to tick into ABH because of hold-time violations. To solve this, we had to use latches clocked independently to stabilize register inputs during clock ticks.

2.3.3 Interrupt Timing

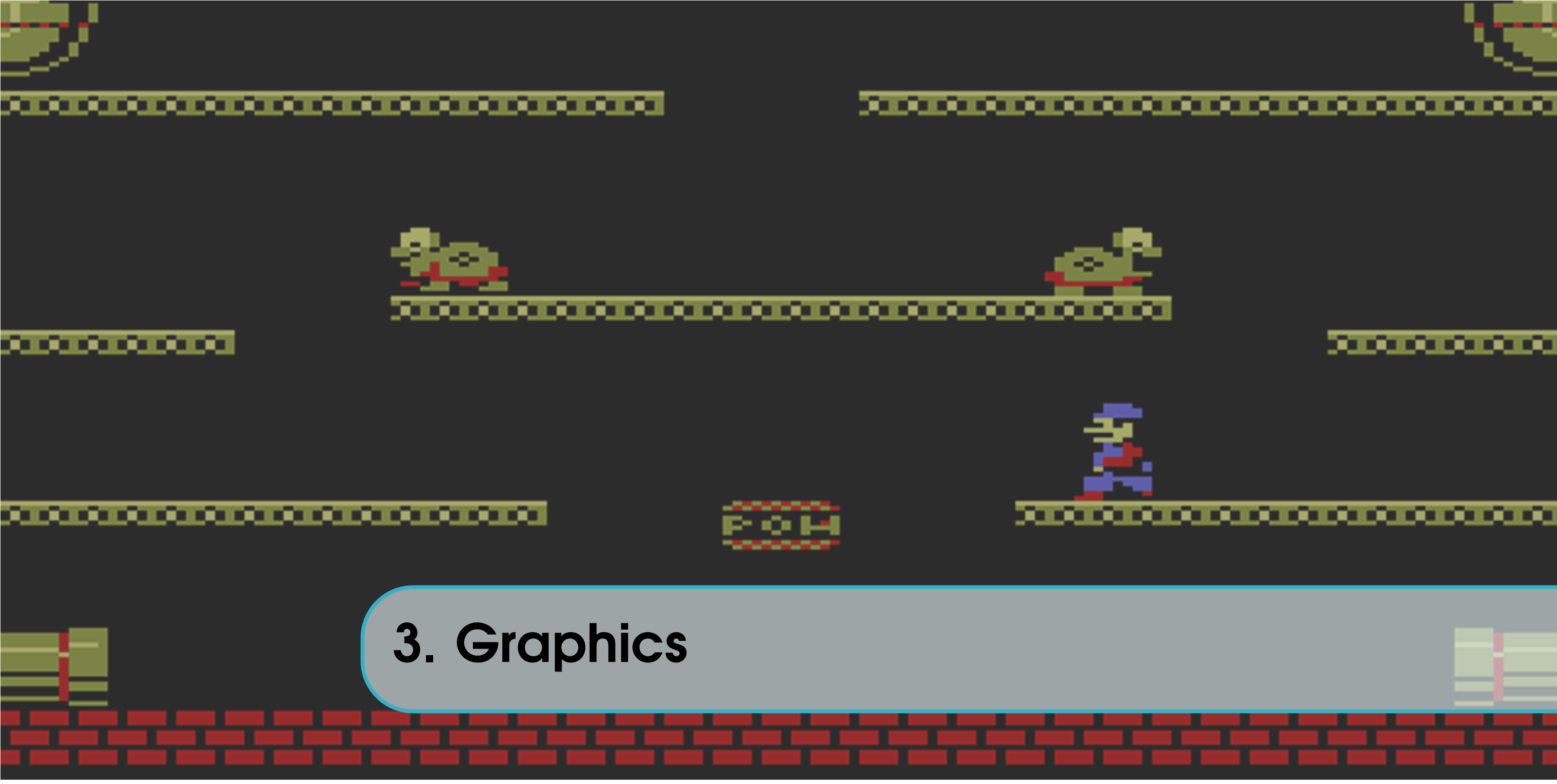
The processor also uses both maskable and non-maskable interrupts. Timings had to be controlled very carefully, as the 6502C seems to have rather weird behavior when it comes to triggered interrupts concurrently, or one after another. For example, the interrupt must arrive before the T0 state of the next instruction, in order for that instruction to be “replaced” with the BRK instruction (which brings control flow to the interrupt vector). If it arrives any later, the next instruction would get executed before the BRK instruction runs.

2.3.4 Pass/Transmission MOSFETS

Another challenge is that there is no bi-directional pass MOSFETS in the FPGA. To replicate the behavior, the number of drivers on the buses SB, ADH, and DB are stored in variables adhDrivers, sbDrivers and dbDrivers. These are inputs to ‘transbuf’ modules, which determine direction of data flow depending on which side has more drivers. In simulation, we were able to use the tranif1 primitive to model this behavior, but in synthesis, we had to use a behavioral model.

2.4 Verification

The CPU was verified using an asm test suite found online. The CPU was designed to be cycle accurate, as the T-state logic simply deduced the next T-state according to the opcode and other inputs. The test suite is located at https://github.com/Klaus2m5/6502_65C02_functional_tests. We obtained a hex dump of the binary, and loaded it into a blockram for the CPU to read from. The test covers all possible input combinations, and numerous edge cases were uncovered in this way. We are glad that we did not need to write our own verification tool, which meant one less thing to code/debug.



3. Graphics

3.1 ANTIC

3.1.1 Overview

The Alpha-Numeric Television Interface Controller (ANTIC) is responsible for the generation of 2D graphics to be displayed on a screen. It can halt the CPU to read display data from memory, via Direct Memory Access (DMA). The retrieved data is known as a display list, which comprises a set of instructions that produce the playfield graphics. Display list instructions will be elaborated upon in the next section.

ANTIC contains 15 memory-mapped read/write registers, which control the playfield display parameters, player and missile base addresses, character set base address, DMA access control, vertical and horizontal fine scrolling, light pen input, and interrupts. Some registers also have shadow copies in RAM. The CPU writes to these shadow registers, and the values are copied over during the horizontal or vertical screen blanks.

Besides the standard 16-bit address bus, 8-bit data bus, and their corresponding registers to read from memory, ANTIC also has RDY and HALT outputs to perform its DMA operations. HALT is used to suspend the CPU while ANTIC reads from memory, while RDY is used to halt the CPU for horizontal blank syncing (WSYNC).

ANTIC has a 3-bit data output (AN[2:0]) which allows it to interface with GTIA. Playfield mode information is transmitted via this output to GTIA, and each playfield type has a corresponding register with color & luminance values, allowing GTIA to display the pixel of the correct color on screen.

ANTIC operates on two different clock speeds. A 1.79 MHz clock allows ANTIC to synchronize its DMA accesses with the RAM and CPU, while a 3.58 MHz clock, which is also known as the color clock, allows it to communicate with GTIA to transmit playfield graphics data.

ANTIC Instruction	Mode Type	Bytes per Mode Line (narrow/normal/wide)	TV Scan Lines per Mode Line	Color
2	Character	32/40/48	8	1.5
3	Character	32/40/48	10	1.5
4	Character	32/40/48	8	5 (multi-color)
5	Character	32/40/48	16	5 (multi-color)
6	Character	16/20/24	8	5 (single-color)
7	Character	16/20/24	16	5 (single-color)
8	Map	8/10/12	8	4
9	Map	8/10/12	4	2
A	Map	16/20/24	4	4
B	Map	16/20/24	2	2
C	Map	16/20/24	1	2
D	Map	32/40/48	2	4
E	Map	32/40/48	1	4
F	Map	32/40/48	1	1.5

Table 3.1: ANTIC mode specifications

3.1.2 Display Lists

Display lists are instruction sets in memory designed to direct ANTIC to generate the required graphical output. There are 3 main instruction types – blank instructions, jump instructions, and mode instructions.

- Blank instructions specify a number of scan lines, from 1-8, for which a blank line will be displayed in background color on screen.
- Jump instructions are used to change the address of ANTIC's display list pointer. This usually occurs when the display list address crosses over a 1K boundary in memory, or when the current display list sequence is completed.
- Mode instructions are used to set the ANTIC graphics mode. There are 14 different graphical display modes, comprising 6 character modes (modes 2-7) and 8 bitmap modes (modes 8-F). Detailed mode specifications are listed in Table 3.1.

The are also additional bits tagged on to the mode types for different purposes. Bit 7 indicates whether a Display List Interrupt (DLI) should occur at the end of the current mode line. Bit 6 informs ANTIC that the Memory Scan Counter should be reloaded, and the next 2 bytes of data contain the low and high bytes of the address it should be loaded with. The Memory Scan Counter is an address pointer to the location in memory where the pixel data or character pointers can be found, for map and character modes respectively.

3.1.3 Character Mode

There are a total of 6 different character modes, each with variable character display heights, widths, and number of colors. In character mode, the pixel data in memory contains pointers to a character set located at the page specified in the register CHBASE. There is a default character set location at 0xF800, which contains the Atari version of ASCII characters (ATASCII). Alternatively, games can also choose to provide custom character sets for display. Some games, such as Pacman, use a custom character set to create its game display, since many parts of the display can be generated via reusable 8 by 8 pixel blocks.

3.1.4 Map Mode

There are a total of 8 different map modes, each with a variable screen display heights, widths, and number of colors. Map mode is the bitmap display mode for ANTIC, and directly translates the pixel data into the appropriate playfields to be sent to GTIA. Some of the modes allow for a higher resolution or many colors, and hence require a much larger number of bytes per mode line in memory. Map mode is the usual display mode for most games, since it allows for a large degree of flexibility in drawing the game screen. ANTIC allows for modes to be mixed freely, giving the game designers the ability to display both text and graphics together on the same screen with minimal trouble.

3.1.5 Implementation

Figure 3.1 shows a block diagram which outlines the main components of our implementation of ANTIC.

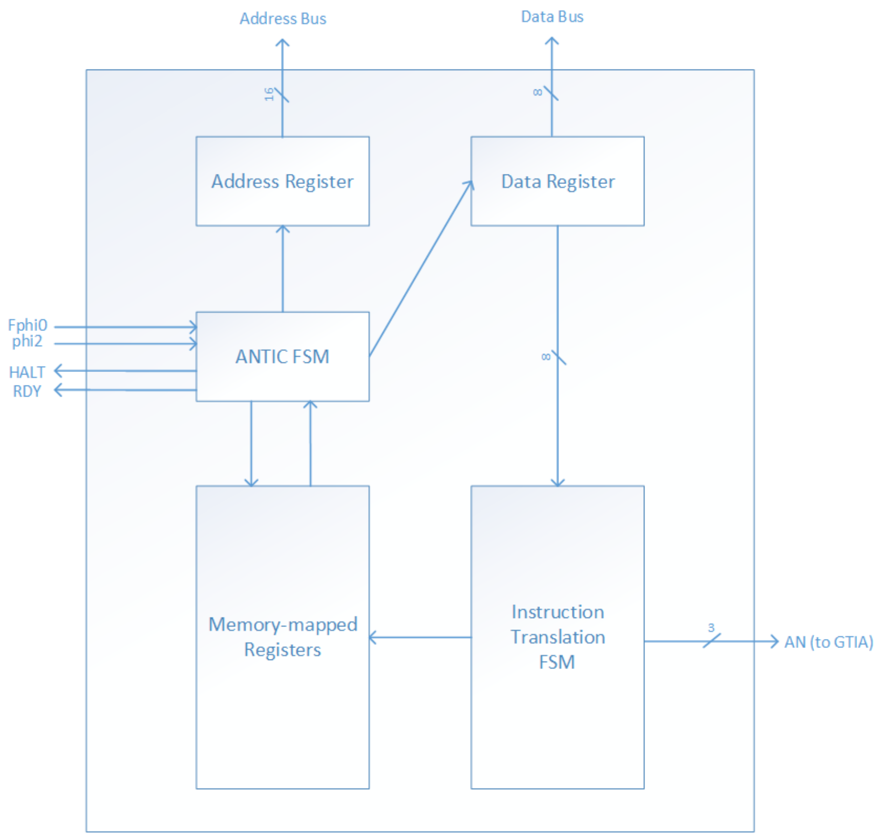


Figure 3.1: ANTIC Block Diagram

Due to a lack availability of detailed documentation on the exact hardware design of the internal components of ANTIC, we had to make an educated guess about the modules within ANTIC. We have produced behavioral components which take the known inputs to ANTIC and produce the desired outputs, as outlined in the block diagram in the next page.

The ANTIC FSM consists of a series of states, which initialize when ANTIC is enabled via the DMACTL register. Load states allow ANTIC to perform DMA operations to retrieve new data from memory, while the idle state is maintained when current display instructions are still being executed, since some instructions take multiple color clock cycles to process.

The instruction translation FSM serves the purpose of converting display list instructions into a series of playfield graphics mode instructions to transmit to GTIA. This FSM was designed behaviorally, and uses a case statement to set the parameters for the desired display mode, as well as transmit the necessary signals to the CPU, memory module, and GTIA, to produce the desired graphics as specified in the display list.

The memory-mapped registers, as well as the address and data registers, are straight-forward components and hence were directly instantiated.

3.2 GTIA

3.2.1 Overview

The Graphics Television Interface Adaptor (GTIA) is responsible for adding color to the play-field graphics produced by ANTIC, as well as adding overlay objects for player and missile graphics, also known as sprites.

GTIA contains 54 memory-mapped read/write registers, which control player/missile graphics, playfield colors, joystick triggers, and console keys. Some registers also have shadow copies in RAM. GTIA operates at the same clock speed of 3.58 MHz.

3.2.2 Colors

GTIA is capable of display 128 different colors. Colors are selected via 8-bit values which are stored in color registers. The upper 4 bits determine the color value, while the lower 4 bits determine the luminance value. These values index into a color table, as shown in Figure 3.2 below, to select te final output pixel color. In our implementation, these colors were converted into RGB values and placed in a display buffer for the DVI module to read from.

3.2.3 Sprites

GTIA adds sprite overlays to the playfield pixel data sent by ANTIC. There are 2 types of sprites, players and missiles. Player sprites are 8 pixels wide, while missile sprites are 2 pixels wide. Sprite pixel data is located in memory, at the page specified by the value in the PMBASE register. Sprites can technically span the entire vertical space of the screen, but they are usually used to display smaller player graphics.

There is also a fifth player mode which allows the four 2 pixel wide missiles to combine to form a fifth, 8 pixel wide player sprite. In fifth player mode, all the missile display the color of

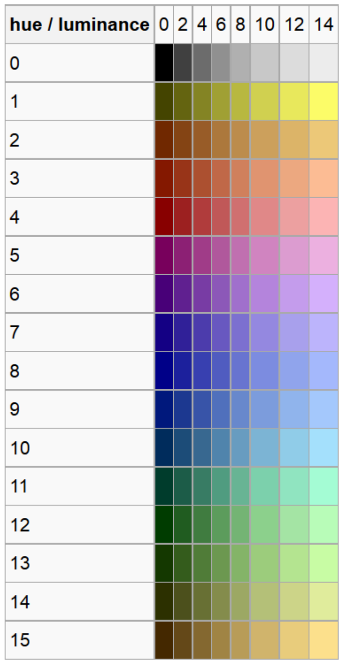

hue / luminance
0
2
4
6
8
10
12
14
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Figure 3.2: Color Table

playfield 3.

3.2.4 Collisions

GTIA has 60 bits which provide automatic collision detection for interaction between player sprites, missile sprites, and playfield pixels. For each pixel, a collision check is performed. When no collision is registered the background color is displayed for that pixel. If a collision is detected, a priority check of the various elements involved is performed, and the color of the object with the highest priority will be displayed for that pixel. There are 4 different priority modes, which can be adjusted via the PRIOR register. These allow the game to select the priority between the various player/missile sprites and the playfield. Collision detection is all performed in real-time, and allows GTIA, along with ANTIC, to offload a significant amount of calculation from the CPU.

3.2.5 Implementation

Figure 3.3 shows a block diagram which outlines the main components of our implementation of the GTIA chip.

The original GTIA chip was designed to produce an output for a Cathode Ray Tube (CRT) type of display, in NTSC format. We decided that in order to simplify our design, we would do a direct conversion of the output of GTIA to a directly displayable format, which in this case would be DVI. This would be more efficient than first producing the original CRT format output signal, and then converting it to a digital version, since the basis for displaying color is significantly different on the two types of display platforms.

The collision detection FSM is coded behaviorally to check priorities on each element that is

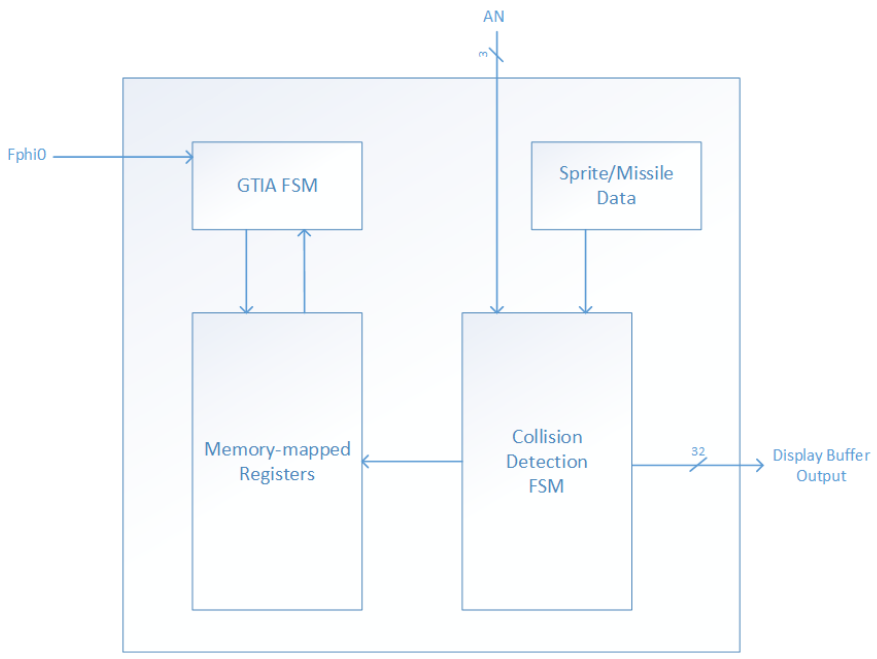


Figure 3.3: GTIA Block Diagram

selected to be displayed at that pixel. It resolves the collision by displaying the pixel color of the object with the highest priority, or the background color if no collision exists.

3.3 DVI

3.3.1 Overview

Digital Visual Interface (DVI) was selected as our mode of graphical output, since the Virtex-5 LX110T FPGA board only has a DVI output port. We used the on-board Chrontel CH7301C chip to read display information from an instantiated block memory display buffer, and then output it to the DVI port. We used an external DVI to VGA converter to translate this data to output on a small VGA monitor.

We referenced the DVI display code written by Team Dragonforce, and made a number of significant modifications to adapt it to our needs. The main components retained were the modules used to take in RGB data and perform Dual Data Rate transmission to the DVI monitor.

3.3.2 Implementation

The display buffer was instantiated using the CoreGen block memory generator. We needed 24 bits of data to display each pixel, with 8 bits for each of the three RGB values. Alignment requirements meant that each pixel would take up 32 bits. The maximum display size of the console output was 320 by 192 pixels, and multiplying by the 32 bits per pixel, this equated to a 1,966,080 bit display buffer, or 245,760 byte buffer.

Our 640 by 480 DVI output needed to be running at 25 MHz, in contrast to the 3.58 MHz clock speed of GTIA. As such, the display buffer allows the DVI display module to constantly refresh the screen based on the current state, while GTIA updates pixels in the buffer at a much slower rate. This solves the problem of GTIA not producing output at a fast enough speed for the refresh rates of modern display monitors.

Figure 3.4 shows an example of a final display output on a DVI input monitor. We were able to successfully recreate the display, despite needing to bridge the gap between the old CRT displays and the modern LCD screens.

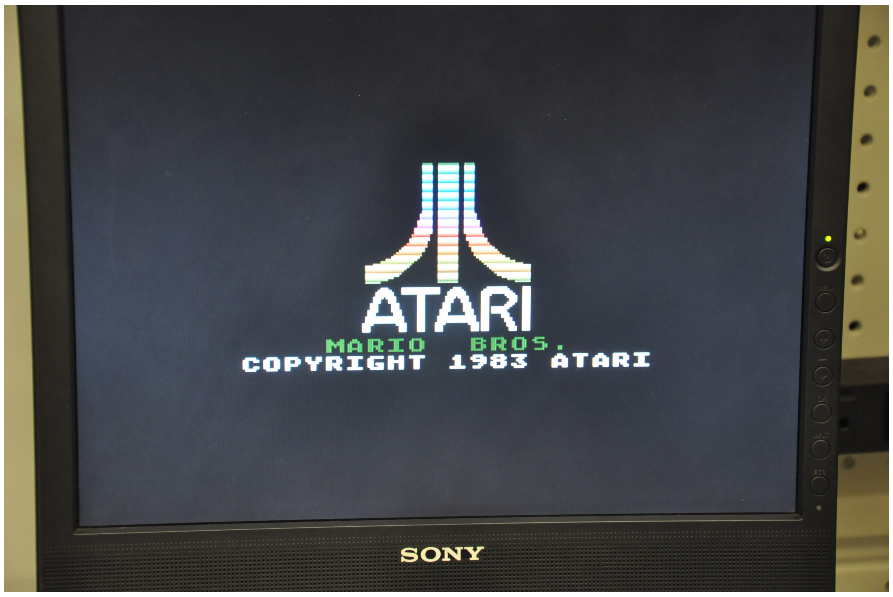
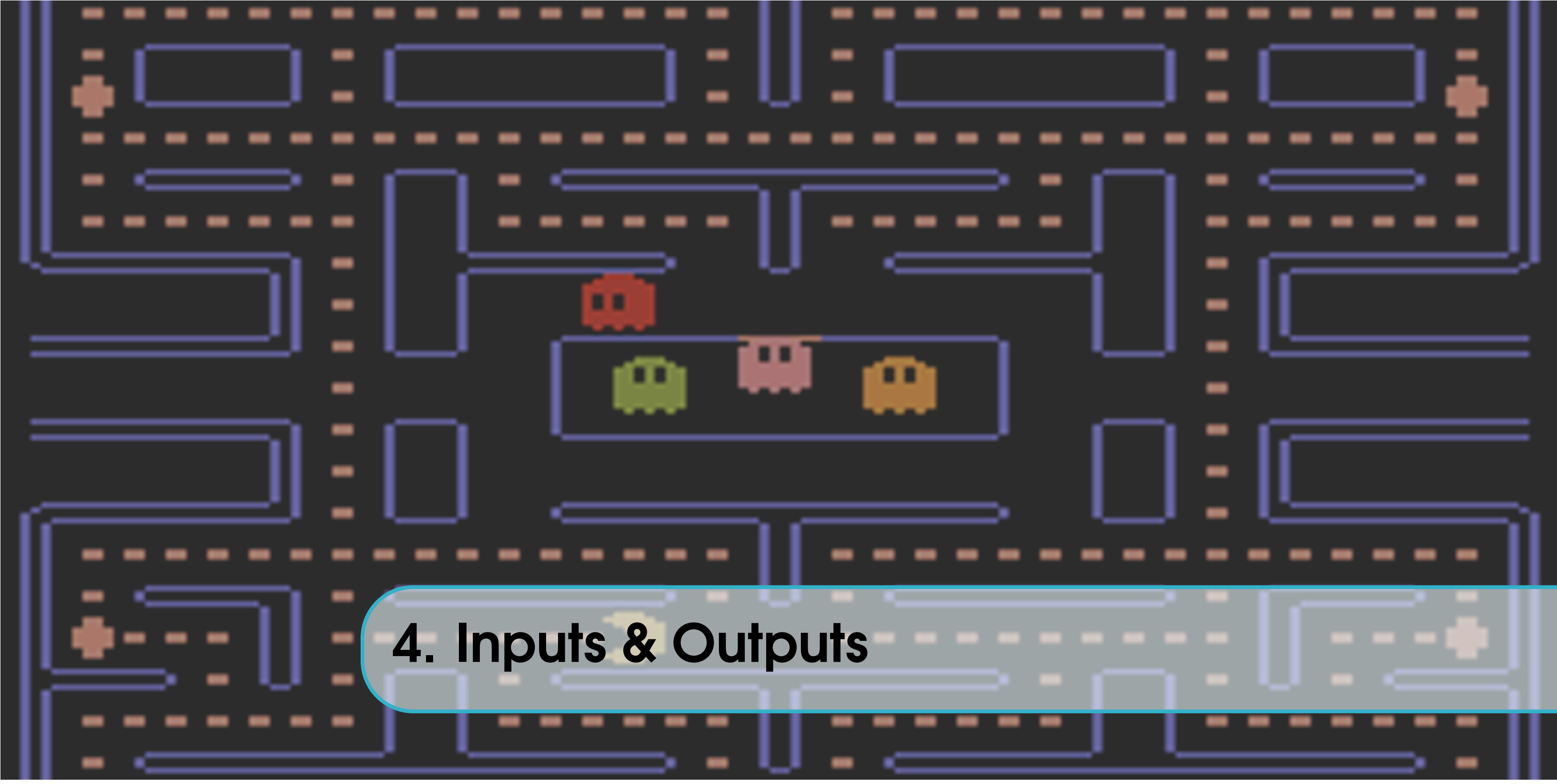


Figure 3.4: DVI Display



4. Inputs & Outputs

4.1 POKEY

4.1.1 Overview

The Potentiometer Keyboard Integrated Circuit (POKEY) is responsible for I/O control, sound generation, and maskable interrupts; it handles I/O control by sampling potentiometers and scanning switch matrices in the manner of an analog digital converter (ADC). These potentiometers and switch matrices are connected to the controller, and are changed by a user manipulating the buttons and joystick on the controller itself.

POKEY contains a series of registers that can be written to and read in order to get inputs and outputs, with different registers corresponding to different functions outlined below. For sound generation, POKEY uses a frequency divider on the input clock signal to generate tones of different frequencies, noise content and volume, based on associated values set in registers. There are a total of 4 separate audio channels, each with their own individual frequency, noise and volume select registers, while there is also a separate set of registers handling interrupt statuses and other components.

The implemented design mostly follows POKEY design documents found online, with omissions of components that are not specifically used by the Atari 5200.

4.1.2 I/O Control

I/O control for the Atari 5200 primarily involves inputs from 64-key keyboard, and potentiometers from the joystick.

- Keyboard Scan:

  There are 6 key scan lines, which hold a value between 00 and 3F, and allow the decoding of 64 individual keys. There is also a 6 bit binary counter, a 6 bit compare latch, and an 8 bit key-code latch, with a control state machine to handle key de-bouncing. A 6 bit binary counter keeps track of which key is being checked (incremented once per line); if the KR1_L line goes low, the value of the binary counter is transferred to the compare latch, and is then tested for de-bounce.

If KR1_L goes low before the next time the binary counter matches the value in the compare latch, it indicates that there are two keys depressed and both key presses are thus ignored. If the binary counter value matches the value in the compare latch and KR1_L is high, the key is bouncing and is also ignored. If KR1_L is still low, the key is valid and the value is then transferred to the key-code latch for reading by the CPU. An IRQ is also sent, indicating that the key is ready. Based on the value of KR1_L on 2 consecutive checking cycles, the key is either read as still being pressed, or released, in which case a new key can be looked for.

The KR2_L is a separate sense line that is dedicated to decoding the SHIFT, BREAK and CONTROL keys. They do not get de-bounced and are decoded only when the key scan lines are at certain values (once per cycle).

- Potentiometers (Joystick controller):

  There are 8 potentiometer input lines, each with its own dump transistor and an 8 bit latch; each potentiometer is one paddle controller, and they are all connected to the 8 line potentiometer port. The system also contains a binary counter that counts up to 228, which starts counting when the potentiometer scan sequence is started, while the dump transistors are released as well. The binary counter counts once per line, and the potentiometer lines start charging at this point. When each line reaches logic 1, the current counter value is latched into the corresponding latch to be read by the CPU. After the counter reaches a value of 228, the dump transistors are turned on to pull the potentiometer lines back to ground.

4.1.3 Audio Control

Overall audio control is handled by AUDCTL, with individual bits selecting clock frequency, random noise source, and channel manipulators. There are 4 semi-independent audio channels, each with adjustable frequency, noise and volume controls, with each channel having associated AUDFX and AUDCX registers to adjust the relevant fields. Frequency for each audio channel is controlled by a divided-by-N frequency divider register, while the noise component is controlled by bits 5, 6, 7 of the AUDCX register, selecting input from a polynomial counter. Volume is controlled by bits 0 to 3 of the AUDCX register, with 4’b0000 setting the audio output to off, 4’b1000 setting volume to half-strength, and 4’b1111 setting the volume to maximum. In addition, the “Volume Control only” mode can be invoked by setting bit 4 of AUDCX; this mode disconnects the dividers, noise counters and filter circuits from the channel output, and leaves the channel output current solely controlled by bits 0 to 3 of AUDCX.

4.1.4 IRQ Interrupts

There are 8 different interrupts available, namely:

- BREAK KEY (depression of the break key)
- OTHER KEY (depression of any other key)

- SERIAL INPUT READY
- SERIAL OUTPUT NEEDED
- TRANSMISSION FINISHED
- TIMER #4 (audio divider #4 has counted down to zero)
- TIMER #2 (audio divider #2 has counted down to zero)
- TIMER #1 (audio divider #1 has counted down to zero)

These interrupts are enabled/disabled by software, and there is a register present to read interrupt statuses as well.

4.2 Implementation

4.2.1 Overview

Since POKEY is responsible for 3 separate functionalities (namely I/O control, sound, and interrupts), each of these functionalities were implemented independent of each other, but all accessing the same set of memory-mapped registers to store and read values from.

4.2.2 I/O Control

- Keyboard Scan:

  With a total of 15 keys on the Atari 5200 controller keypad (including START, PAUSE and RESET), we only needed to use 4 of the 6 available key scan lines in order to refer to a specific key when a KR1_L signal is received. This scanning process is accomplished by cycling a zero value into pins 5-8 one at a time (through the use of a multiplexer), and checking the value of KR1_L (which is the value being output on pins 1-4, checked one at a time using a demultiplexer). The select values being used for the mux and demux are the key scan lines mentioned earlier, as shown in Figure 4.1.

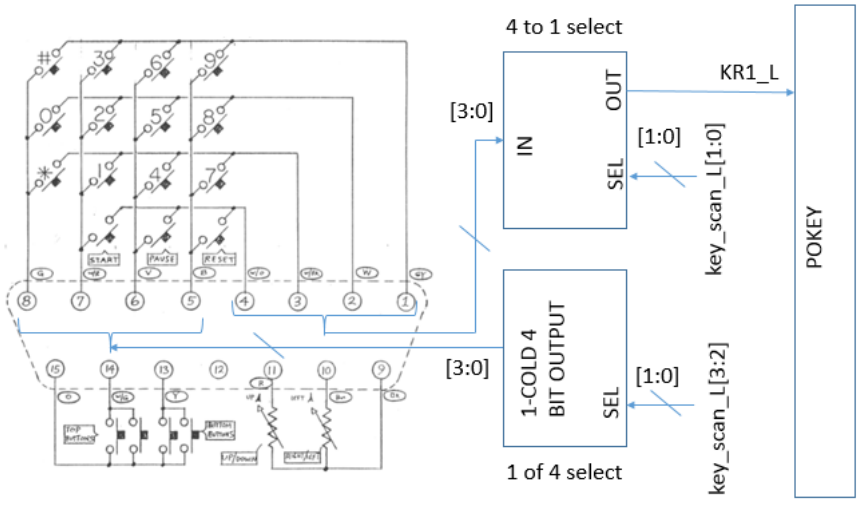


Figure 4.1: Implementation Diagram of I/O Control interface with POKEY

- Potentiometers (Joystick controller):

  In order to use the potentiometers to affect the digital value being read into the associated register in POKEY, we had to set up an external resistor-capacitor (RC) circuit, which would take a different amount of time to charge depending on the value of the potentiometer. This in turn would affect the number of lines that would have passed by the time the capacitor was charged up, and the potentiometer could be used in this manner to determine the position of the joystick. Getting accurate and consistent reads required careful customization of each RC circuit, in order to coordinate the speed of the capacitor charging (which was primarily an analog operation) with the digital counter and capturing the right digital value in the associated register.

4.2.3 Audio Control

Audio control was implemented for a single audio channel by applying a frequency divider to the main clock, based on the value located in AUF. After that stage, distortion is applied using random number generators, with the extent of distortion determined by the value in AUDC, before finally being fed to the output for the speaker. This is repeated for all 4 audio channels, and allows us to generate the unique sounds that are associated to the Atari 5200.

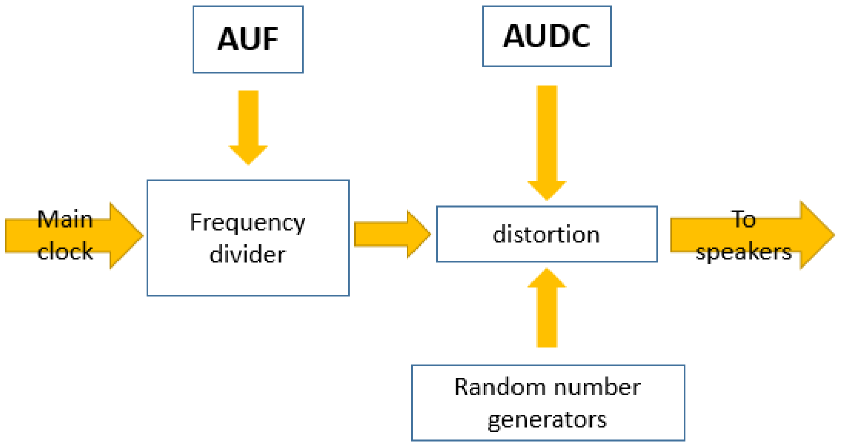


Figure 4.2: Implementation Diagram of Audio Control

4.2.4 IRQ Interrupts

IRQ interrupts were implemented by setting a bit in the IRQST register whenever an interrupt event occurred, as well as pulling the IRQ_L line to low, such that the CPU would recognize that an interrupt event occurred, and could check the value in IRQST to determine which interrupt had occurred and handle it accordingly.

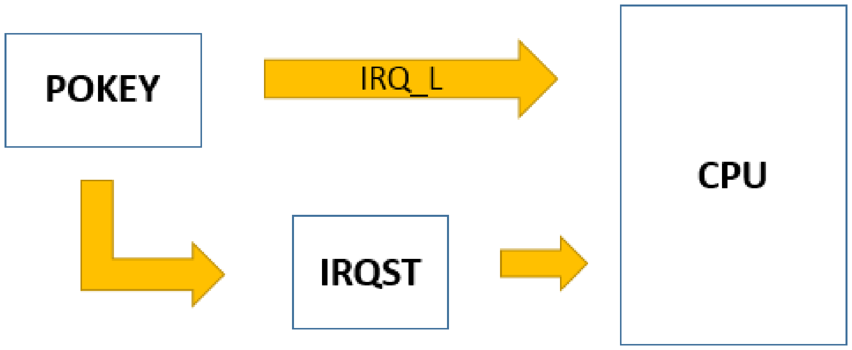


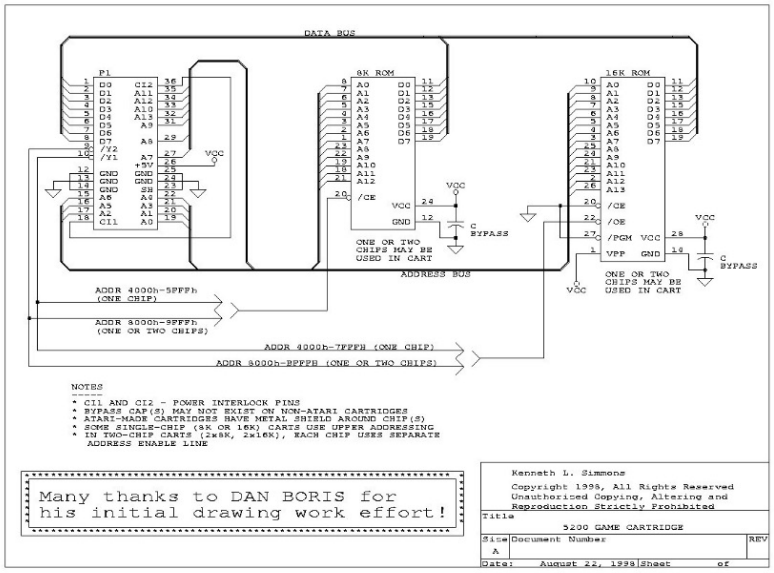
Figure 4.3: Implementation Diagram of IRQ Interrupt Handling

4.3 Additional Hardware

4.3.1 Cartridges

We acquired 3 original Atari 5200 game cartridges: Defender, Mario Bros, and Pacman. Each of the cartridges contained 32kb worth of ROM data, and interfaced with the Atari 5200 system using a cartridge connector. We purchased a 32-pin cartridge connector to allow the original cartridges to interface directly with our FPGA pinouts.

Figure 4.4 shows the pinout of an Atari 5200 cartridge.


DATA BUS
P1
8K ROM
16K ROM
VCC
BYPASS
ONE OR TWO CHIPS MAY BE USED IN CART
ADDRESS BUS
ADDR 4000h-5FFFh (ONE CHIP)
ADDR 8000h-9FFFh (ONE OR TWO CHIPS)
ADDR 4000h-7FFFH (ONE CHIP)
ADDR 8000h-BFFFH (ONE OR TWO CHIPS)

NOTES
-----
* CI1 AND CI2 - POWER INTERLOCK PINS
* BYPASS CAP(S) MAY NOT EXIST ON NON-ATARI CARTRIDGES
* ATARI-MADE CARTRIDGES HAVE METAL SHIELD AROUND CHIP(S)
* SOME SINGLE-CHIP (8K OR 16K) CARTS USE UPPER ADDRESSING
* IN TWO-CHIP CARTS (2x8K, 2x16K), EACH CHIP USES SEPARATE ADDRESS ENABLE LINE

Many thanks to DAN BORIS for his initial drawing work effort!

Kenneth L. Simmons
Copyright 1998, All Rights Reserved
Unauthorized Copying, Altering and Reproduction Strictly Prohibited

Title: 5200 GAME CARTRIDGE
Size: A | Document Number | REV
Date: August 22, 1998 | Sheet of

Figure 4.4: Cartridge Schematic

4.3.2 Controllers

The Atari 5200 controller contains both a keyboard as well as a joystick, and connects to the Atari 5200 system via a connector, as shown in Figure 4.5. We acquired two Atari 5200 controllers, and used breakout circuits to connect the pins of the controller connector to the appropriate input ports on our FPGA, based on the schematic shown in Figure 4.6.

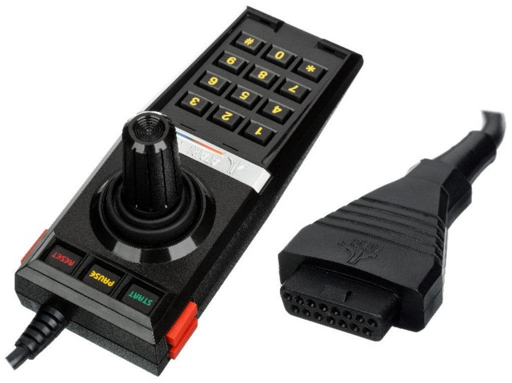


Figure 4.5: Atari 5200 Controller

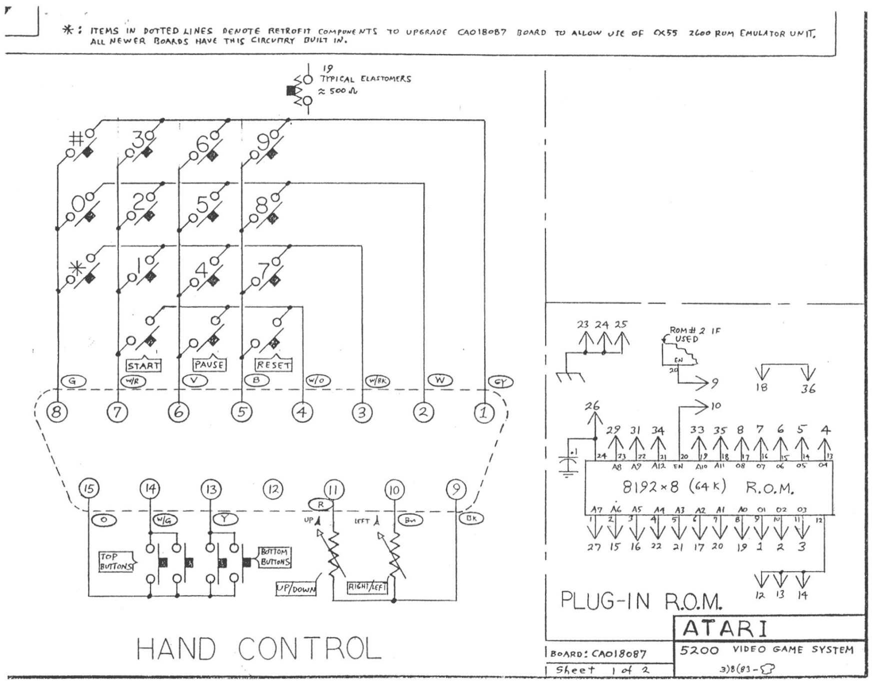
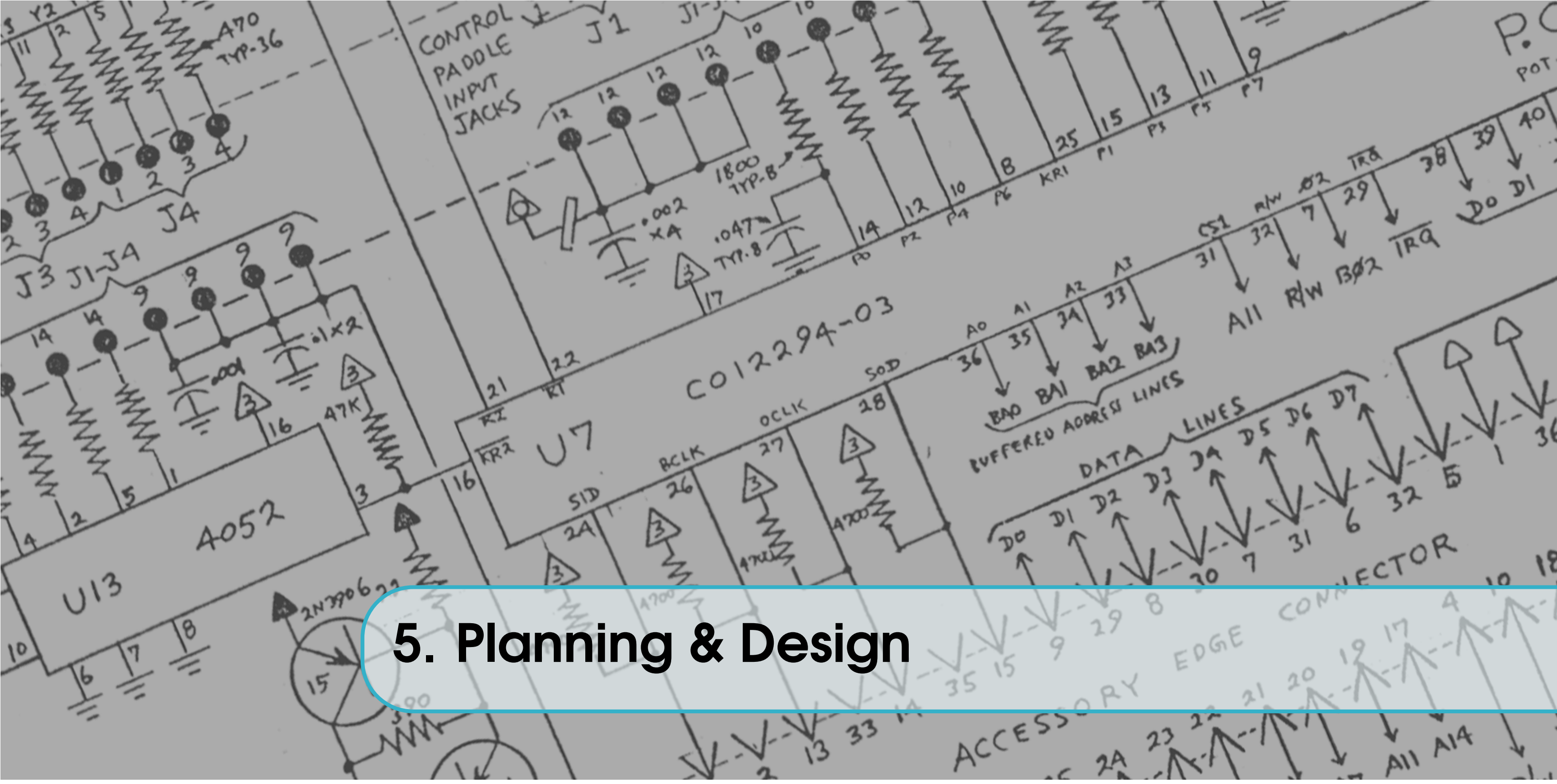

* : ITEMS IN DOTTED LINES DENOTE RETROFIT COMPONENTS TO UPGRADE CA018087 BOARD TO ALLOW USE OF CX55 2600 ROM EMULATOR UNIT. ALL NEWER BOARDS HAVE THIS CIRCUITRY BUILT IN.
19 TYPICAL ELASTOMERS ≈ 500 Ω
START
PAUSE
RESET
TOP BUTTONS
BOTTOM BUTTONS
UP/DOWN
RIGHT/LEFT
HAND CONTROL
ROM # 2 IF USED
8192×8 (64 K) R.O.M.
PLUG-IN R.O.M.
ATARI
5200 VIDEO GAME SYSTEM
BOARD: CA018087
Sheet 1 of 2

Figure 4.6: Controller Schematic



5. Planning & Design

5.1 Design Approach

5.1.1 Design Partitioning

The project was split into design, implementation and integration phases.

The design phase consisted of researching on the main parts of the system. We first found out the schematics of the Atari 5200 on atariage.com. It showed the layout of the chips and the connections between the 6502C, ANTIC, GTIA, and POKEY. However, the individual chips were blackboxes. There were only behavioral descriptions of the parts, so we intended to reverse engineer the components based on the intended behavior. At the end of the design phase, we had rough designs of the chips and could begin coding. The design phase took up the first 3-4 weeks of the semester.

In the implementation phase, we partitioned the system into 3 main parts: the CPU, the graphic chips, and the I/O chip, each member responsible for implementing one part. This phase took up the majority of the semester, and we spent 2 months on this phase.

The integration phase was intended to begin a month before thanksgiving. We anticipated that the integration would uncover a lot of bugs, especially the interaction between the CPU and the graphics, because of DMA and interrupts used by the game.

5.1.2 Tools & Design Methodology

We used ISE as a design tool for synthesis, together with ModelSim for simulation. We took care to simulate designs extensively and verified everything in ModelSim. Only then we would try to synthesize it. This narrows down the type of bugs we get to timings issues rather than implementation mistakes.

5.1.3 Testing & Verification Methodology

The CPU was tested using an ASM test suite found online. It verifies every single valid opcode with all possible operands. The test suite uncovered numerous edge cases which would have given us problems later on if not found.

The graphics were primarily testing by taking memory dumps for each game display screen, using an Atari 5200 emulator. These memory dumps were then placed into block ROM, and the graphics modules would attempt to read from these memory blocks and display the output to the LCD monitor. Bugs would then be discovered either directly on the monitor, or through ChipScope.

The I/O could only be verified after synthesis. The sound chips were tested with the whole range of frequencies and distortion settings. The I/O controllers were also tested extensively using the 2 purchased controllers, by customizing and tweaking each controller output with analogue components to ensure that the analog values being sent back to the FPGA were within appropriate thresholds that could be converted into clear digital values.

5.1.4 Schedule & Time Management

We were mindful of the fact that delays and other unforeseen circumstances were commonplace in projects, and therefore allocated buffer time for each phase of the project. We also added a week's worth of buffer time at the end of the entire project, which was designed to be used to deal with unexpected bugs and other issues.

In terms of project management decisions, we started off deciding that we should collectively work on researching the CPU 6502C, since we would need to have at least a rudimentary understanding of how the arguably most important part of the Atari 5200 worked before we could split up to work on the other components such as ANTIC, GTIA and POKEY. This decision was also made since none of us had taken an in depth computer architecture course, and we felt that our combined efforts would make the learning process a little smoother than if we had parcelled it out to just one person.

After spending a good amount of time working on a hardware implementation of the CPU 6502C, we decided we could split up our efforts to start working on the other bigger components while we continued to test the CPU implementation.

We maintained close contact with each other, and kept a good awareness of each individual's work. This helped us ensure that we were all on the right track, and smoothened the process of integration later on in the project.

6. Lessons Learned

6.1 What We Wished We Had Known

- There is never enough time. Start early!
- Pouring time at a problem may not always be sufficient. Pour time at the right places.
- Don't spend too much effort if it's not worth it, design decisions must be made swiftly given the time constraints.
- Don't look back after making decisions.
- CoreGen has a ton of useful modules which can be instantiated. Lots of time can be saved if you don't have to write and debug small modules yourself.
- ChipScope is insanely useful for debugging. Also, you can view pre-trigger data by adjusting the 'position' value.

6.2 Good & Bad Decisions

6.2.1 Good Decisions

- Partitioning the project earlier on because implementing is best done individually. Group meetings are for making decisions, not coding together.
- Deciding to write all the main components of our project from scratch early on. This gave us a steeper initial learning curve, but ended up better during the integration and debugging phases, as we were very in tune with what our code did, and could more easily understand and solve problems when we found them.
- Deciding to use the Virtex-5 boards and ISE was probably a good decision since it was a more stable release (no sudden updates) and there was information online about common errors.
- Intensive testing for freshly written modules, both in simulation and synthesis, ensured that the integration process later on ran much smoother.
- CPU: Using a lookup table instead of trying to reverse engineer the random control logic block.
- Graphics: Writing the code behaviorally based on the known outcome, rather than trying to

follow non-official hardware diagrams.

- Graphics: Skipping the original CRT-based video output, and instead hacking out a bridging module to directly display to an LCD screen via DVI.

6.2.2 Bad Decisions

- We accidentally carried over a 65kb linear memory module from simulation to synthesis, rather than instantiate one using CoreGen. This led to horribly long synthesis times (about 30mins). We only realised the cause after a couple of days.
- We spent quite some time working on the same portion of the project at the start. This may or may not have been a bad decision, but perhaps we could have done a proper schedule planning and division of labor a little earlier on. Having clearer division lines helped the workflow within the team.

6.3 Words of Wisdom

This is an awesome course if you enjoy the process of creating something cool in hardware from scratch. Make sure you pick a project about something you like as a group! Hopefully you also actually enjoy digital design, otherwise be prepared for a rough semester...

6.4 Individual Pages

6.4.1 Alvin

What you did and approximate the time spent

I was responsible for implementing the CPU. I spent an average of 15-20 hours a week for the first 2 months to research, code, and debug the CPU. The research phase consisted of reading documentation, and gaining an understanding of the timing designs and micro cycles used by the CPU. The coding phase was relatively quick, as I had simple schematics of the bus lines, the registers and latches used in the CPU. The debugging phase took the longest time. I used a test suite found on the Internet. Initial testing was done on Modelsim, to eliminate most bugs. In many cases, the bugs arise due to the schematics being incomplete, and my own modifications had to be added to replicate undocumented parts of the schematics. When I moved to debugging using synthesis on the FPGA, more issues cropped up, mostly regarding timing hazards, violation of rise/fall constraints. These issues were eventually resolved by latching input signals before clock ticks.

After the CPU was completed, I helped my teammate with the controllers, as he was having some trouble with the analog RC circuits used to detect joystick positions. I also helped to code the sound modules, which were quite straightforward.

The remaining month of the project was spent doing integration. This was a slow and painful process because of the need for the CPU and the graphics component to interact. During this phase, we faced many problems with the timing of interrupts and DMA behavior, but these were resolved over time. Me and Jonathan (responsible for graphics) spent around 20hours a week working on this integration.

During this month, I also spent 10 or so hours building the circuits for the project. This included the sound circuits, the cartridge interface, and the controller interface.

After the graphics and CPU were working together perfectly, the I/O was done (sound and controller), and we went through a week of final debugging and product packaging. I spent another 10 or so hours refining the joystick circuit, because the RC circuit was having trouble producing steady readings of joystick position. We finished a week before the deadline, and I am glad we stuck strictly to our schedule and built in some buffer time in the schedule.

Class impressions/improvements complaints

Overall, I feel that this is a great course, especially for anyone who enjoys designing and implementing your own project. It was cool to be given the opportunity and resources to make something useful, instead of following instructions in a lab handout. As a capstone class, you are expected to already have the required knowledge to implement a system on a FPGA, so there is minimal guidance from the course staff. Expect to be challenged by the project, yet you will emerge as a more competent engineer and hopefully with a sense of accomplishment. The course trains you to be resourceful, to understand specifications, and seek help on forums. However, if you do not enjoy digital design or engineering in general, you should not take this class, because you will find the hours in lab torturous. A successful project requires the commitment of every team member, so enter this course with the mindset of challenging yourself and emerging victorious!

The project also forced a team to work under stress, which reveals the true nature of your teammates and their work etiquettes. Through this project, I have learnt a lot about my teammates, and there is a shared sense of accomplishment of conquering this project.

6.4.2 Benjamin

What you did and approximate the time spent

During the initial phases of the project, we were focused on gathering and understanding as many resources as possible that would help us in the process of planning and executing the tasks required in building the Atari 5200. I started off by searching for and working my way through the documentation available online, including the schematics and other descriptions written by hobbyists about the various components. This took the bulk of the first few weeks, averaging 10-15 hours a week, especially as the team engaged in frequent meetings to determine how we would lay out the course of the project, and discussed the initial decisions that we needed to make to ensure the project's smooth progress in the later weeks, including timeline and complexity decisions.

We decided that we should all have some familiarity with the CPU component to begin with, since we felt that it would probably be one of the most challenging components to put together, especially given that our team had no prior experience with building CPUs. After we were satisfied that the team didn't need our total combined effort on the CPU, I moved on to working on understanding and implementing POKEY. The next few weeks was spent on looking for schematics and descriptions of the different functions that POKEY needed to handle, and consulting with the team regarding potential synchronization issues that were likely to arise through the interaction between POKEY and the rest of the system. In addition, after we made the decision to purchase original controllers from external sources to interface with our project implementation, I started testing each component of the controller to ensure that it was in proper working condition, as well as making sure that values from the FPGA could be fed to the controller and read back correctly. This also involved the hardware description implementation of the key-scan and potentiometer-scan process, and took an average of 15-20 hours a week to create and test. There was significant delay encountered during this stage due to the analog nature of the original controllers and the requirement to adjust analog components to ensure that clear analog signals that were well within acceptable thresholds could be sent to the FPGA to be captured as digital data.

After this portion was complete, we began to move into the integration and testing phase, to put our individual components together and get them to work correctly. This took the bulk of our time towards the later portion of the semester, especially with the CPU and graphics processing operations which needed to be well coordinated since they used the same memory map. Since the POKEY component was largely complete by this time, I was able to provide support to my team with other miscellaneous tasks such as putting the components physically together and mounting them on PCBs, as well as preparing for our final presentation and demonstrations, in addition to helping with the integration process, which took an average of 13-15 hours a week.

Class impressions/improvements complaints

This class was designed as a capstone class in the ECE department's curriculum, and was clearly different from any other class we had taken before. Besides only having one main assignment to work on for the entire semester, we were also given a great deal of autonomy in deciding how we would implement and work on our project. Resources were easily available to us upon request, and that was extremely useful in making the course focused on solving the critical issues that provided the greatest learning experience and exposure, instead of spending time worrying about how we would obtain materials for our project to move forward. In addition, being able to observe other teams at work and hearing about the problems that they were encountering and how these problems were overcome was extremely useful in allowing us not to make similar mistakes, and also gave us a great opportunity to collaborate and support one another in offering suggestions and giving feedback in order to improve our individual projects as a whole.

Overall, this course is an extremely enriching learning experience, especially in terms of honing project management skills over and beyond the technical skill required to write hardware description code, and is a useful and critical part of any ECE major's education.

6.4.3 Jonathan

What you did and approximate the time spent

I was responsible for all things graphics-related for this project. I spent around 10 hours a week in the initial research stage. Upon switching to working solely on the graphics, this went up to about 20-30 hours a week, and remained roughly the same until the end of the integration phase.

I started the project researching about CPU along with the rest of the team, as well as working on some of the smaller combinational logic modules used in the CPU. After some time, I planned out Gantt chart to give us a rough scheduling for the project. Realizing the need to begin work on the other components, I switched to working on graphics, and researched it for about a week.

After getting a sufficient basic understanding, I began coding a simple behavioral model in Verilog. I first wrote the FSM to initialize and read data from RAM via DMA, then wrote the FSM to translate the data for simple instructions such as blank lines, jump instructions, and a single map mode. I then moved to an initial testing stage to ensure that I was on the right track. I realised that since the data received by GTIA was originally then meant for a CRT display, I could potentially modify the output directly to display via DVI, rather than generate the data for CRT and then convert it to a more modern format.

At this point, I decided that it would be better to have a working chain all the way from the initial reading of the display list in ANTIC, to the color processing in GTIA, to the storing of RGB data in a display buffer RAM and then an output display to an LCD screen. I thus started work on the display module. Rather than start from scratch, I decided to capitalize on the working DVI modules from a previous 18-545 group, Team Dragonforce. I still needed to modify a significant number of areas of their source code to fit my needs of displaying RGB from a fixed block RAM display buffer. It took a week or two to get a stable display of color test bars on screen.

I next moved on to creating a basic GTIA module to receive the playfield data bitstream from ANTIC and translate the bits into the correct RGB values, by running the values in the color registers through a color lookup table. The pixel data then had to be placed into the appropriate positions in the display buffer, with some additional space required for the front and back horizontal and vertical porches, as well as the sync areas of the display screen. This portion took perhaps another week or two.

I then went back to finishing the various display modes in ANTIC. This proved harder than I initially thought, as there were 16 different modes, each of which had its own quirks in terms of display size, resolution, and color. I tested mainly using memory dumps of screen data and display lists from a working emulator. This was an effective way of ensuring a correct final output, since the online text descriptions of each mode did not always explain the specifics clearly enough. This stage took about 2-3 weeks, and when I could successfully display entire memory dumps of screens on the LCD, it was then time to integrate the graphics with the CPU.

The integration phase revealed a number of errors, both with the CPU and the graphics modules. I worked with Alvin to resolve these errors, as well as ensure that the timing coordination of the clock cycles for common elements such as DMA and interrupts were correct. It took a 1-2 weeks to get the CPU to successfully run the BIOS, and then display the classic Atari loading screen. We also had to trace and fix problems with how some synchronization signals between the graphics and CPU worked. At this point, our synthesis time had become quite significant (about 10mins), and small bugs and typos wasted a much larger amount of time. I spent the final weeks of the project fixing graphical bugs for the different display modes, and getting the sprite display and collision mechanisms working.

Class impressions/improvements complaints

Ultimately, this was a good class and I took away many learning lessons, not only in Verilog coding, but also lessons in team dynamics, time management, and project planning. It is important to know the strengths and weaknesses of your team members, as different parts of a project require different sets of skills. Also, make sure that you like the course content, and pick a project that you'll enjoy doing. Otherwise the class might not be enjoyable, and time spent in the lab will feel wasted. It felt great to have a working project at the end of the semester.