Team E.T.

Fall 2015

—

Final Report

Chris Barker

Ryan Roberts

Mark Wuebbens

CMU | 18-545

Statement of Use

Our source code can be found at the following URL:

https://github.com/545/Atari7800

Feel free to contact us with questions about the project at:

chrisbarker@cmu.edu
ryanroberts@cmu.edu
mwuebben@andrew.cmu.edu

Contents

Acknowledgements

We would like to thank a few people whose work was vital to our project.

- Daniel “DanB” Boris - for the wealth of information he provides about all things Atari. Specifically:
  - The only source anywhere on the hidden control register, as well as many other posts on the atari7800wiki: https://sites.google.com/site/atari7800wiki/7800-control-register
  - His response to our question on the AtariAge forums: http://atariage.com/forums/topic/245073-hidden-control-register-for-atari-7800/
  - His site on Atari HQ, especially his commented disassemblies of the BIOS and Robotron code http://www.atarihq.com/danb/a7800.shtml
- Daniel Beer - For his RIOT and TIA implementations as part of his Atari 2600 project
  - http://people.ece.cornell.edu/land/courses/eceprojectsland/STUDENTPROJ/2006to2007/dbb26/dbb28_meng_report.pdf
- Arlet Ottens - For his 6502 implementation
  - https://github.com/Arlet/verilog-6502
- 18-545 staff - For all their support and guidance
- The AtariAge Forums and Atari Enthusiasts everywhere. Keep being awesome!

Overview

Background

The Atari 7800 ProSystem was released in 1986 by the Atari Corporation. It was intended to replace the company's unsuccessful Atari 5200 and was the first console to feature built in backwards compatibility. The 7800 will play any 2600 game and can even use 2600 controllers.

Objective

Our goal for this project was to implement the Atari 7800 game system on the Zedboard FPGA platform. Specifically, we aimed to implement a fully functioning system capable of playing any of the various game cartridges that are compatible with the 7800 including the 2600 games for which the 7800 has backwards compatibility.

This required implementing all of the various game and graphics modes available on the 7800 which are determined by the type of cartridge for each game. Game cartridges vary in the amount of included RAM and ROM, and also by the inclusion of an additional sound processing chip known as POKEY. All of these factors affect the modes which the 7800 must run in.

An additional goal for this project is to design a sleek case for our FPGA system to mimic a real video game console experience.

Results

We completed a fully functioning Atari 7800, minus two major features. Firstly, our system requires games to be run from read only memory (ROM) on the board, and is incapable of interfacing with physical game cartridges. Secondly, our system has extremely limited backwards compatibility with Atari 2600 games - the games appear recognizably on the TV and accept user input, but have major graphical glitches rendering them unplayable. Additionally, a limited set of Atari 7800 games have graphical problems. However, the majority of 7800 games run flawlessly. We used real Atari 7800 controllers for input and have working VGA video and AUX audio.

We built the following components from scratch:

- "Maria" graphics chip
- VGA Controller
    - Frame buffer for Atari 2600 mode

    - Chrominance-luminance converter
- Sound system
- Hardware interface to cartridge and controllers
- Memory map and memory clock management
- CPU halt and interrupt management
- Control Register

We used existing implementations of these components [Acknowledgements]:

- 6502 Core
- RIOT I/O Chip
- TIA Graphics and I/O
- Xilinx clock divider and RAM IP

Development Tools

FPGA

We used the Zedboard Zynq-7000 board. This board was selected because it had a considerable amount of logic cells along with enough block RAM to suit the 7800's needs.

Software

All development was done using Vivado. We used a few of the Xilinx IP blocks built in Vivado such as block RAM and clock generators.

We used Github as a version control program for the project and as a wiki for various technical documents and useful links.

Design

System

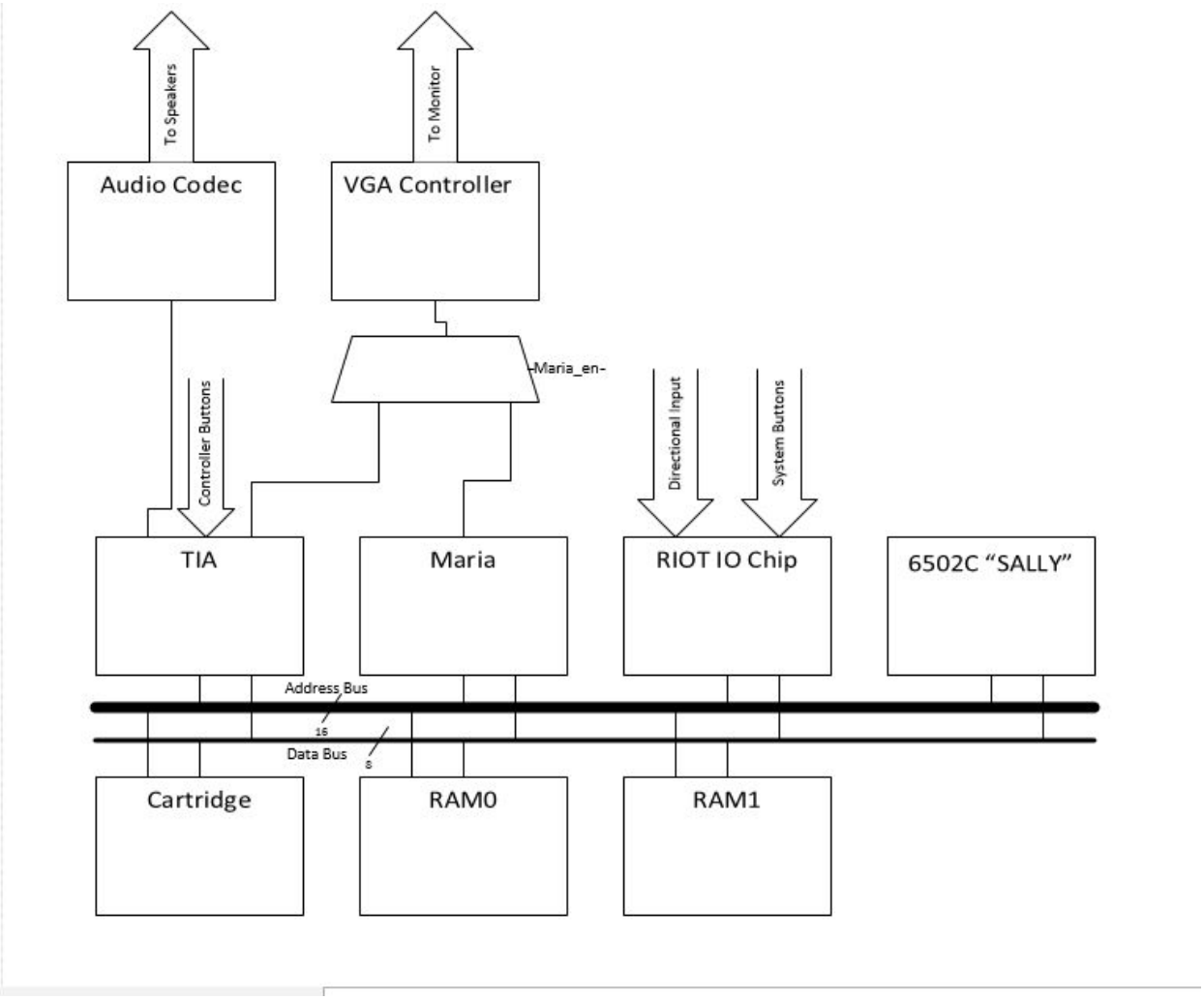
The system consists of three main components. The main components are the 6502C "Sally" core, the MARIA graphics chip, and the TIA sound and graphics chip. The 6502C is a custom MOS 8 bit microprocessor common in gaming systems of this era. The MARIA graphics chip is the 7800's main graphics chip, controlling the drawing of sprites and the playfield for all 7800 games. The TIA sound and graphics chip handles sound and button inputs for the controllers. The TIA also handles graphics when an Atari 2600 game is run.

There are also several peripheral components to the system - the RIOT IO Chip, Cartridge interface, and two 2 KB SRAM blocks. The RIOT IO Chip handles system

button presses, such as power and reset, and the directional input of the controllers. The Cartridge contains the program ROM but may also contain additional RAM or a POKEY audio chip for extra sound capability. The SRAM is low latency memory and is used to store variables and the program stack.

The system communicates and performs computations through shared Address and Data buses with most communication happening through memory mapped registers in each component.

System Block Diagram



6502C Sally

Overview

The 6502C is a custom MOS 8 bit microprocessor with 56 instructions and 13 addressing modes.

Registers

The 6502C has 6 registers:

PC - The 16 bit program counter. This stores the current instruction being executed

S - The lower 8 bits of the stack pointer. The upper 8 bits are always 1. Points to the top of the stack.

X - 8 bit index register. Used to calculate addresses in certain addressing modes.

Y - 8 bit index register. Used to calculate addresses in certain addressing modes.

A - 8 bit accumulator register. Used primarily for arithmetic and logical operations.

P - 8 bit status register. Each bit represents one flag and can be set and cleared using special instructions

Bit 0 C- The carry flag. Represents the carry out of any ALU operation.

Bit 1 Z - The zero flag. Set to one when any ALU operation results in a 0.

Bit 2 I - The interrupt flag. If this is set, IRQs are disabled.

Bit 3 D - The decimal flag. If this is set, the ALU operates in BCD mode.

Bit 4 B - The brk flag. Set whenever a software interrupt (BRK) is executed.

Bit 5 - Unused. Always set to 0.

Bit 6 V - The overflow flag. Whenever an ALU operation results in a value larger than can be represented in 8 bits, this is set

Bit 7 S - The sign flag. This is set when the ALU operation results in a negative value.

Addressing Modes

There are 13 addressing modes, though not all of them can be used on each instruction.

Accumulator - The accumulator is implied as the operand.

Immediate - The operand is preceded by # and is used directly.

Implied - The operand is implied by the instruction, so no operand is given.

Relative - The offset specified is added to the PC to compute the new address.

Absolute - A 16 bit address is specified and used.

Indirect - A 16 bit address is specified. The value stored at that address is used as the operand. The JMP instruction is the only one that uses this addressing mode.

Zero-Page - An 8 bit address is specified. The upper 8 bits are assumed to be 00 and are concatenated with the supplied 8 bits to make the address.

Absolute Indexed (X) - A 16 bit value is specified and is offset by the value in X to get the final address.

Absolute Indexed (Y) - A 16 bit value is specified and is offset by the value in Y to get the final address.

Zero-Page Indexed (X) - An 8 bit value is specified and is offset by the value in X. The 8 bit result is used as the lower byte of the address, and 00 is used as the upper byte.

Zero-Page Indexed (Y) - An 8 bit value is specified and is offset by the value in Y. The 8 bit result is used as the lower byte of the address, and 00 is used as the upper byte.

Zero-Page Indexed Indirect - An 8 bit value is supplied and is offset by the value in X. The 8 bit result is used as the lower byte of the address and 00 is used as the upper byte. The value stored in memory at this address is used as the address for the operation.

Zero-Page Indirect Indexed - An 8 bit value is supplied and is used as the lower byte of the address. 00 is used as the upper byte of the address. The value stored at this location in memory is offset by Y and is used as the address for the operation.

Interrupts

The 6502 allows for three different types of interrupts. Each of these can be triggered by asserting the signal to the core and has a separate vector in the address space from FFFA to FFFF which points to the interrupt sub-routine that is called for that interrupt.

NMI - This is a non-maskable interrupt. This means it will always be executed regardless of the value in the I register. In the Atari 7800, it is connected to the MARIA and is asserted when the MARIA finishes displaying a zone and the programmer has opted to generate an interrupt for that zone. This means the interrupt subroutine can change palettes, change the display lists, or do anything else involving that zone safely.

IRQ - This is a maskable interrupt. If the I flag is set, it will be ignored. The I flag is set automatically upon entering into the IRQ subroutine and is cleared just before exiting the subroutine. The IRQ line is hooked up to the cartridge in the Atari 7800, but no cartridge ever asserts it, so it is unused for hardware interrupts. The BRK instruction is a software interrupt which uses the same interrupt vector as IRQ

RESET - This is an interrupt triggered by pressing the reset button on the console. It will result in running the BIOS code.

RDY Signal

The RDY signal is a standard signal across all derivatives of the 6502 model. It halts the processor and holds the address and data bus constant. While the RDY signal is held constantly high by the Atari 7800, it was used in our implementation of the halt signal, so it is important to mention.

Halt Signal

The halt signal is a custom signal used in the 6502C and is identical to the RDY signal with one exception. The address and data bus are tri-stated when the halt signal is asserted, allowing another chip to drive the bus. This signal is necessary because the MARIA needs to halt the cpu regularly to perform a DMA. Our halt signal was implemented by putting the cpu in a wrapper module and asserting the RDY signal on the core while tri-stating the address and data bus.

Our Model

The 6502 model we decided to use is the NMOS 6502 verilog model by Arlet Ottens [4]. This model was seen as ideal compared to the models available on opencores for two reasons. Primarily, the core is well documented and uses very descriptive variable names. This is a huge step up from the opencores models which had sparse commenting and confounding variable names. Secondarily, it was described in Verilog, which our team is much more well versed in. This core has one major drawback. It does not use the two clock system that the 6502 was known for. The 6502 uses two clocks which are inverses of each other. External operations, such as driving buses, are done when one clock is high, and internal operations, such as latching data, are done when the other clock is high. Because of this, Ottens declared that the timing within an instruction may not match up perfectly with the original 6502; however, the core will perform exactly the same from an instruction-atomic level. This means that each instruction will take exactly the same number of cycles as the original 6502 design. Because of this, we are confident that this drawback will not have any effect on the core's functioning within the system.

Verification

We have personally verified that the core works by running the core with a test suite found online [5]. We developed a test bench and simulated the entire memory state using our custom memory module. We loaded the binary for the test suite into the memory, and it ran to completion. After modifying the core to add the halt signal, we ran the test again, halting the core for 64 cycles after 64 cycles of execution. This test

suite tests every single instruction and addressing mode as well as testing every alu instruction with every possible 8 bit operand. Because of this, we are very confident that this core will suit our needs and perform the tasks required of the SALLY.

Maria

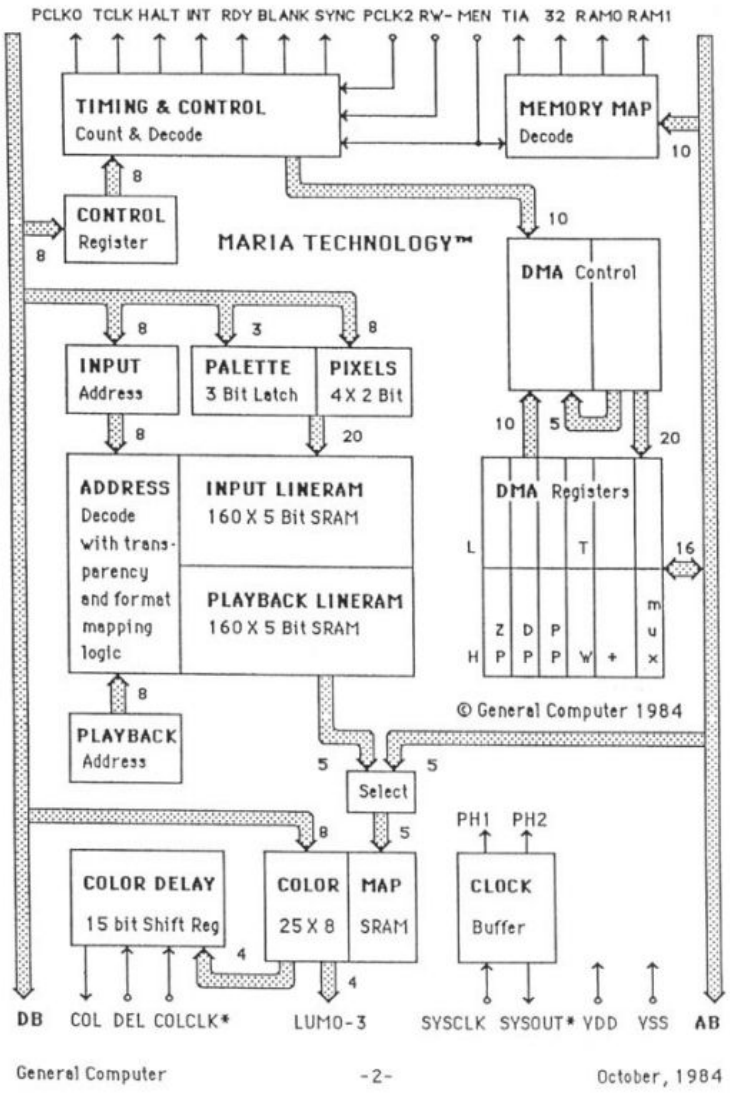
The Maria chip generates the video output for Atari 7800 games. It “makes available a hardware and software breakthrough” in graphics display [2]. It allows for more complex scenes than the Atari 2600, which uses the Tia chip and allows for only a fixed number of sprites. The Maria chip reads graphics data from an in-memory data structure on a per-scanline basis, giving much more flexible graphics options.

The original console output video in the NTSC or PAL TV formats (for North America and Europe respectively.) We are using NTSC cartridges and are therefore implementing a Maria that emulates the NTSC output, however our implementation converts this format to VGA for compatibility with modern displays.

The Maria operates at 7.154MHz. Its clock is divided by either four or six (to 1.79 MHz or 1.19 MHz) and input to the CPU, depending on which memory address is being accessed. In this section, a “clock cycle” refers to a 7.154 MHz period.

The following block diagram ([2], Fig. 1) describes the functional organization and the registers in the original Maria design. Our design divides the Maria into four modules: *Timing and Control*, *Memory Map*, *DMA Control*, and *Line RAM*. Each is described below, including its interfaces internally as well as to parts of the system external to the Maria.

Figure 1. Block Diagram



DMA Control

The Maria retrieves graphics data through Direct Memory Access (DMA). The in-memory data structure is called the Display List List (DLL), or Zone List (ZL). One item in the DLL is used per scanline and describes what needs to be written into the Line RAM during that scanline. We start with a description of the data structure format, and then move to explaining how our implementation processes it.

Display List List Format

The first item in the DLL is in memory at the address specified by the memory-mapped registers ZPH and ZPL. Each element (Zone) in the DLL is specified by three bytes:

- {DLIen, a12en, a11en, 0, offset[3:0]}
  - DLIen: Whether the *Timing and Control* module should raise a Display List Interrupt after fetching these three bytes.
  - a12en, a11en: “Holey” (sparse) graphics data. Indicates whether to ignore pixel data addresses with bits 12 or 11 asserted, respectively.
  - Offset: The number of scanlines to use this Zone for, minus one. The sum of offsets-plus-one for the entire DLL must be 242, the number of scanlines in NTSC. If the DLL describes more than 242 lines, the 243rd and later lines will be ignored.
- DPH
  - High byte of the Display List Pointer
- DPL
  - Low byte of the Display List Pointer. These bytes are concatenated to form a pointer to the Display List (DL) for this zone. The format of the DL is described next.

The ZP is reset to its memory mapped value at the beginning of every frame. Every time a zone is completed, the ZP is incremented by three and points to the next Zone for the next scanline.

Display List Format

The display list specifies an ordered list of one-dimensional Objects to be drawn on the Line RAM. Each object has a horizontal position, a color palette (3-bit index into memory mapped registers COLOR MAP), a width in bytes of pixel data, and a pointer to pixel data. Display list entries consist of either 4 or 5 byte headers. The end of the display list is indicated by a 2 byte header where the second byte is null.

Each object is written sequentially into the buffer Line RAM, which is latched into the playback Line RAM at the end of each scanline. See the *Line RAM* module for how the graphics data is interpreted.

Four byte object header:

- PPL: Low address of Pixel Pointer (address of graphics data)
- {Palette[2:0], Width[4:0]}
  - Width is two’s complement of number of bytes of pixel data
- PPH: High address of Pixel Pointer
  - Pixel data is loaded into *Line RAM*
- INPUT: Left column (0 to 159) of object in Line RAM. Each column is 2 pixels.

Five byte object header:

- PPL: Same as 4-byte mode
- {WM, 1, IND, 5’b0}

- WM: Write Mode. Determines whether each byte of graphics data occupies two 2-pixel cells of Line RAM (1) or four 2-pixel cells (0). This changes WM for all objects until another 5-byte header changes it.
- IND: Indirect Mode. If 1, the data at PP is treated as a “character map” - an array of low bytes of pointers to graphics data in ROM. The upper byte of the pointer is CHARBASE, a memory mapped register, plus OFFSET. Indirect mode is used for this object only.

- PPH: Same as 4-byte mode
- {Palette[2:0], Width[4:0]}: Same as 4-byte mode
- INPUT: Same as 4-byte mode

Null terminator:

- PPL
- 0

DMA Implementation

Our *DMA Control* module is implemented as an FSM. It is reset to a `wait` state. There are two signals it can receive from *Timing Control*: `dp_dma_start` and `zp_dma_start`. ZP DMA occurs once per frame. *DMA Control* dereferences the memory mapped `Zone Pointer` and loads the three bytes of Zone data. `DP`, `DLIen`, `A12en`, `A11en`, and `OFFSET` are loaded into registers from ZP, ZP+1, and ZP+2. *DMA Control* returns to the wait state. If `DLIen` is asserted, *DMA Control* signals *Timing Control* to raise a display list interrupt.

Upon receiving `dp_dma_start`, DMA for the DP starts. This occurs once per scanline. *DMA Control* dereferences the `DP` register and reads 2 bytes. If the second byte is nonzero in any of the lowest 5 bits, *DMA Control* is in four-byte mode. If the second byte has zero as its lowest 5 bits, but nonzero in any of the top bits, it is in five-byte header mode. If all bits are zero, the null terminator is found. This case is described last.

In 4- or 5-byte mode, the 4 or 5 bytes above are addressed. `WIDTH`, `PP`, and `IND` are read into DMA Control. DMA Control sends signals to *Line RAM* when `PALETTE`, `INPUT`, and `WM` are on the data bus instructing it to latch those values.

In direct mode, *DMA Control* places `PP` on the address bus and signals *Line RAM* to latch the values on the data bus as graphics data. `OFFSET` is added to the upper byte of `PP`. This process continues only if Holey DMA rules are not met. If the address is in cartridge space, *DMA Control* holds this address for 4 clock cycles, as the cartridge is clocked at 1.79MHz. Otherwise, it holds the address for 1 clock cycle. It repeats this process and increments `WIDTH` until `WIDTH` is zero. Then it moves to the next Object

(increments DP by 4) and repeats the DP DMA process- unless dp_dma_kill is asserted from *Timing Control*, in which case it returns to wait.

In indirect mode, PP is dereferenced to obtain CHARPTR. This takes only one clock cycle as character maps are required to be stored in fast memory. {CHARBASE+OFFSET, CHARPTR} is dereferenced to obtain pixel data, assuming holey DMA rules are not met. This can take either 1 or 4 cycles. If CHARACTER_WIDTH in the Maria's control register is set, {CHARBASE+OFFSET, CHARPTR+1} is also dereferenced to obtain pixel data. Otherwise, PP is incremented and the process repeats until PP is zero. The state machine moves to the next Object, as in direct mode.

When the null terminator Object header is found, *DMA Control* decrements the value in its OFFSET register. If OFFSET is nonzero, or if *Timing Control* signals that this is the last scanline, *DMA Control* asserts dp_dma_done to Timing Control and returns to the wait state. Otherwise, the next 3-byte Zone header needs to be fetched. New values of DLIen, A12en, A11en, OFFSET, and DP are loaded, and ZP is incremented by three. If DLIen is 1, *DMA Control* signals dp_dma_done_dli to *Timing Control* and returns to wait. Otherwise, it asserts dp_dma_done to *Timing Control* and returns to wait.

Line RAM

The *Line RAM* module is, at its core, a place for graphics data to be buffered between being read by *DMA Control* and displayed by *VGA*. It has some additional logic to implement 2 Read Modes and 3 Write Modes, controlled by the memory mapped Control Register and a bit in the five-byte Object Header respectively.

Line RAM has two RAMs: a buffer RAM and a playback RAM. Each one has 160 five-bit cells. Each cell contains the data for two consecutive pixels. When *Line RAM* receives an lram_swap signal from *Timing Control*, the contents of the buffer RAM are latched into the playback RAM, and the buffer RAM is reset to all zeroes.

Line RAM takes a read_mode input from the memory mapped control register. It loads an 8-bit input register from the data bus when it receives input_w from *DMA Control*, a 2-bit write_mode register when it receives wm_w from *DMA Control*, and 3-bit palette register when it receives palette_w from *DMA Control*.

Writing

Upon receiving pixels_w from *DMA Control*, *Line RAM* loads a byte of pixel data from the data bus. Let the value in palette be {P2, P1, P0}. Let the value on the data bus be {D7, D6, D5, D4, D3, D2, D1, D0}. If wm is 0, the following values are written to buffer Line RAM starting at input and proceeding upward:

- P2, D3, D2, D7, D6
- P2, D1, D0, D5, D4

The exception is if the index into Line RAM exceeds 159 or the pixel data is 0. Then no value is written.

Reading

The output UV data for a given column is determined by indexing 3 palette bits and 2 color bits into the memory mapped COLOR MAP. If the color bits are 0, the background color is used. Otherwise, color C of palette P is used. See the Memory Map for more detail. Then, these UV values are converted to RGB by the VGA module. See the VGA module for more detail. Here, we focus on how the palette and color bits are determined.

The VGA display has 640 columns. The original console output 320 pixels, so the VGA module sends all but the least significant bit of its column counter to *Line RAM* to request Luminance-Chrominance (UV) data for the current pixel. *Line RAM* uses all but the least significant of these bits to index into the playback Line RAM and retrieve one of 160 five bit cells. Depending on `read_mode`, the least significant bit, indicating left or right pixel within the cell, may affect the output UV data. We will call this bit `R`, for Right pixel. We refer to the bits in the playback cell as {L4, L3, L2, L1, L0}. Then the palette used for display is determined as follows:

- `RM` = 0x: {L4, L3, L2}
- `RM` = 10: {L4, 0, 0}
- `RM` = 11: {L4, L3, L2}

The color used for display is determined as follows:

- `RM` = 0x: {L1, L0}
- `RM` = 10, `R` = 1: {L0, L2}
- `RM` = 10, `R` = 0: {L1, L3}
- `RM` = 11, `R` = 1: {L0, 0}
- `RM` = 11, `R` = 0: {L1, 0}

Together, the three read modes and two write modes create six graphics modes. Note that the write mode is applied during the buffering scanline whereas the read mode is applied during the playback scanline. These graphics modes allow for different varieties of palette choices and resolution/compression tradeoffs.

Timing and Control

The *Timing and Control* module has several responsibilities, including managing timing for the Maria, generating control signals for the *DMA Control* module, and managing clocking for the whole system.

Maria Timing

The *Timing and Control* module takes an enable signal generated by the processor. If it is not asserted, it will not send commands to the *DMA Control* module. It also takes a signal from *Memory Map* indicating whether ZP has been written to yet. Before ZP is written, *Timing* will not send commands to *DMA Control*.

Both NTSC and VGA operate at 60 frames per second. VGA has 525 rows, 480 of which are visible. NTSC has 262 scanlines, 242 of which are visible. To convert from the NTSC interface to VGA, we use two VGA rows per NTSC scanline. This means a scanline will need to complete in slightly less time: 2/(60 * 525) = 1/(60 * 262.5) seconds in our system instead of 1/(60 * 262) seconds in the original system. However, since our DMA implementation takes fewer clock cycles to complete than the original system, this is not a problem. Our conversion to VGA cuts off the top line and bottom line from the NTSC format. This is acceptable because on some NTSC TVs, the top and bottom 25-26 lines were not visible anyway.

Each scanline takes 452 clock cycles. A column counter keeps track of how many cycles have elapsed since the last scanline incrementation. This is used to generate control signals as described below.

The *Timing and Control* module takes two 10-bit signals, `vga_row` and `vga_col`, as input from the VGA controller. The module uses these to determine when to assert the halt signal to the core and send commands to the *DMA Control* module. There are two types of DMA in our design, DP DMA and ZP DMA. We describe their operation in the DMA Control section, and their timing here.

ZP DMA occurs once per frame, needs to be completed two cycles before buffering of the first line of a frame begins, and takes 22 cycles to complete, so it starts 24 cycles before the end of VGA line 520 (three rows before the first scanline in the DLL) when the column counter is 420. First, the module sends a `halt` signal to the CPU and waits 9 cycles for the CPU to finish driving the address bus, then 7 cycles are used to perform the DMA, and 13 are used to return control to the CPU. The Timing and Control module asserts `dp_dma_start` to *DMA Control* after the first 9 cycles.

DP DMA occurs once per scanline. It starts when the column counter is 28, per the specification, to allow the CPU to perform computation at the start of a scanline. We assert `halt` and wait 9 cycles for the CPU to respond to the signal, then assert a `dp_dma_start` signal to *DMA Control*. This DMA is killed at 436 cycles via the `dp_dma_kill` signal, if it is not yet finished, to allow the CPU to perform computation at the end of the scanline. Otherwise, when the module receives a `dp_dma_done` signal from the DMA Control module, it deasserts `halt` and waits for the next scanline to begin. *DMA Control* also may send a `DLI` (Display List Interrupt) signal at this time, causing *Timing Control* to send a non-maskable interrupt signal to the CPU.

TIming Control asserts the non-maskable interrupt signal for six clock cycles after halt is deasserted to ensure it is seen by the CPU.

The first DP DMA happens on VGA line 521-522 (two VGA rows before the first scanline) to buffer for scanline 0, which is not displayed. On VGA line 523-524, scanline 1 is buffered. on VGA line 0-1, scanline 1 is displayed and is the first visible line. Scanline 2 is buffered simultaneously. The last visible scanline is displayed on VGA line 479-480, and the last invisible scanline (which still has a DP DMA) is "displayed" on VGA line 481-482.

At the completion of each DP DMA, *Timing and Control* asserts a signal to the Line RAM module causing it to latch the contents of the input lineram into the playback lineram and reset the input lineram. This loads the buffered data to be displayed on the next two VGA rows.

External Signals

The Maria divides the input clock by two to 3.58MHz and outputs it to the Tia. It divides the input clock by either 4 or 6 to 1.79MHz and 1.19MHz respectively, depending on whether memory is mapped to a fast device or slow device, and outputs it to the CPU.

The `ready` signal is output to the CPU. It is 1 by default. When the CPU writes to a memory mapped register `wait_sync`, `ready` goes to 0. When a scanline completes, it returns to 1. This allows the CPU to wait for the next scanline to complete.

Memory Map

The Maria, perhaps because it is the fastest clocked chip in the system, is responsible for memory mapping. It has its own internal memory mapped registers, described below, and also outputs 4 memory-select signals to other devices in the system letting them know that the current read or write is intended for them. These devices are 6532 (RIOT), Tia, RAM0, and RAM1. The external memory map is as follows, and we have implemented it per this specification ([2], Fig 2):

Figure 2. Memory Map

AB15	14	13	12	11	10	9	8	7	6	5	pav	HALT-	MEN = 1	OUTPUT
0	0	0	0	0	1	0		1					6532-	480- 4FF
														580- 5FF
0	0	0	0	0	0	1		1					6532-	280- 2FF
														380- 3FF
0	0	0	1	1								1	RAM1-	1800-1FFF
0	0	0	1	1							1	0	RAM1-	1800-1FFF
0	0	0	0	0	0			0	0	0			TIA-	0- 1F,
														100- 11F,
														200- 21F,
														300- 31F
0	0	0	0	0	0			0	0	1	*		par23	20- 3F
0	0	0	0	0	0			1				1	RAM0-	80- FF
0	0	0	0	0	0			1			1	0	RAM0-	80- FF
0	0	0	0	0	0			0	1			1	RAM0-	40- 7F
0	0	0	0	0	0			0	1		1	0	RAM0-	40- FF,
														140- 1FF,
														240- 2FF,
														340- 3FF
0	0	1	0	0								1	RAM0-	2000-27FF
0	0	1	0	0							1	0	RAM0-	2000-27FF
0	0	0	0	0	0			0	0	0			slow (TIA)	
0	0	0	0	0	1	0		1					slow (6532)	
0	0	0	0	0	0	1		1					slow (6532)	

2600 mode:

AB15	14	13	12	11	10	9	8	7	6	5	pav	HALT-	MEN = 0	OUTPUT
			0					1					6532-	80- FF
			0					0					TIA-	0- 7F
													slow	

Notes: Blank entries indicate don't-care conditions.

"par23" is the internal Maria select.

"pav" is "Processor Address Valid," an internal signal which is internal version of PCLK0 OR'ed with internal PCLK2.

"*" indicates "par23" does not have "pav" in its equation when HALT- is asserted.

"slow" indicates a device which runs at a clock of 1.19 MHz, rather than 1.79 Mhz.

The internal memory mapped registers are as follows ([2], Fig 3):

Figure 3. Register Map

Address	Register	Comments
20	P0C0	Background Color (COLOR MAP registers 0,4,8,12,16,20,24,28)
21	P0C1	Palette 0, Color 1 (COLOR MAP registers are write-only)
22	P0C2	Palette 0, Color 2
23	P0C3	Palette 0, Color 3
24	WSYNC	Wait for SYNC (Writing to this location de-asserts RDY, which is asserted again at the completion of the current scanline -- synchronizing program to video timing.)
25	P1C1	
26	P1C2	
27	P1C3	
28	STATRD	STATus ReaD [7 thru 0] = [VBlank 0 0 0 0 0 0 0] (Read only)
29	P2C1	
2A	P2C2	
2B	P2C3	
2C	ZPH	Zone Pointer, high byte.
2D	P3C1	
2E	P3C2	
2F	P3C3	
30	ZPL	Zone Pointer, low byte.
31	P4C1	
32	P4C2	
33	P4C3	
34	CHARBASE	The upper byte of character-map addresses. (Lower byte is in the map, i.e. the character code.)
35	P5C1	
36	P5C2	
37	P5C3	
38	unused	Write in 0 (Reserved for future enhancements)
39	P6C1	
3A	P6C2	
3B	P6C3	
3C	CONTROL	[7 thru 0] = [CK DM1 DM0 CWIDTH BCNTL KM RM1 RM0]
3D	P7C1	
3E	P7C2	
3F	P7C3	

General Computer -10- October, 1984

The memory mapped registers are used as follows:

- COLOR MAP: Indexed by a {Palette[2:0], Color[1:0]} pair by the *Line RAM* module. The value in register P[Palette]C[Color] is output. It represents an 8-bit chrominance/luminance pair.
- WSYNC: Wait for end of scanline. See Timing and Control: External Signals.
- STATRD: High bit indicates whether NTSC VSYNC would be active, which occurs at VGA lines 512 through 524.
- ZPH, ZPL: See DMA Control. Concatenated together, these point to the beginning of the Zone List.
- CHARBASE: Upper byte of character map address. Concatenated with items in the character map in indirect mode.
- CONTROL:

- CK: Color Kill. Not currently using. “Eliminates color artifacts for text modes.”
- DM: Display mode. Must be 0b10 for the Maria to operate.
- CWIDTH: Determines whether to read 1 or 2 bytes at a time in Indirect mode.
- BCNTL: Determines whether the border should be black or the background color. Since we are not displaying a border we don’t plan to use this.
- KM: Kangaroo mode. Determines the behavior of pairs of pixels in 320-graphics modes when one pixel is visible and the other isn’t. In kangaroo mode, the invisible pixel is transparent. Without it, it takes on the background color.
- TK: Transparency kill. Always display background color when color is 00, instead of retaining the existing color.
- RM: Line RAM Read mode. See Graphics Modes.

Memory

The Atari 7800 has two 2KB banks of SRAM referred to as Ram0 and Ram1 which are used by the Sally to create a stack and to store variables. We implement this memory by instantiating two distributed RAMs:

```
(* ram_style = “distributed” *) reg [7:0] ram0 [2047:0];
(* ram_style = “distributed” *) reg [7:0] ram1 [2047:0];
```

The memory mapping module described in the Maria section detects when addresses mapping to Ram0 or Ram1 are being accessed and outputs a chipselect signal, which sets the corresponding enable signal for the RAM modules, as well as the other memory mapped components: TIA, RIOT, Cartridge, BIOS, and Maria.

We use two separate databuses: a “write” databus and a “read” databus. This allows the core to perform a write and use data from its previous read on the same clock cycle. To allow for this, the chipselect signal is buffered. The address bus, write databus, and write enable signals are sent to the memory component selected by the current chipselect. The read databus is driven by the memory component selected by the buffered chipselect.

Complexity arises here because the buffered chipselect needs to be latched at a different clock edge depending on which memory component is being read from. The TIA and RIOT are clocked at 1.19 MHz; the cartridge and Maria memory map are clocked at 1.79 MHz; the RAM blocks are clocked at 1.79 MHz when the core is active; and the RAM blocks are clocked at 7.14MHz when the Maria is active. We use a gated clock called mem_clk to determine at what rate to latch the chipselect buffer.

TIA

Graphics

Overview:

The TIA chip is responsible for generating graphics information in 2600 mode. The Atari 7800 runs in this mode for backward compatibility with 2600 games.

In 2600 mode, the processor accesses the TIA registers in the address range 0x00-0x3c. This memory space, the RIOT memory mapped registers, the 128 bytes of memory in the RIOT, and the ROM space provided on the game cartridge are the only memory available to the processor in this mode. This mode has a special 2600 memory map that covers the same 0000-FFFF address space with a large number of shadows into each of these 4 regions.

Graphics in 2600 mode are implemented by writing values to registers in the TIA to describe the location and properties of the objects it displays. These objects are: 2 player sprites, 2 player missiles, 1 ball, the playfield, and the background. Because the TIA has no memory to store graphics data, it requires the core to set up each scanline as it happens. This makes timing between the TIA and core crucial for proper execution.

Timing:

A single frame on a TV consists of 262 horizontal scanlines of which there are (in order)

- 3 Vertical sync (VSYNC) lines
- 37 Vertical Blank (VBLANK) lines
- 192 Visible picture lines
- 30 Overscan lines

Each scanline consists of 228 TIA clock counts (3.58 MHz). The first 68 of these are horizontal blank (HBLANK), and the remaining 160 clocks are visible on the TV.

Horizontal timing is taken care of by the processor, and the HBLANK and HSYNC signals are raised in turn. The core must control vertical timing by writing to memory mapped VSYNC and VBLANK registers in the TIA. The typical method for doing this is

- Assert VBLANK at the beginning of Overscan
- Wait 30 scanlines (usually by setting the timer in the RIOT so the core can do other things)
- Assert VSYNC

- Wait 3 scanlines
- Deassert VSYNC
- Wait 37 scanlines (usually by setting the timer)
- Deassert VBLANK
- Display 192 lines of game picture

The game picture is displayed line by line by having the core write the data for that line in the TIA. This means the core must run in perfect line by line sync with the TIA. Each core clock is equal to 3 TIA clocks, so this gives the programmer 76 cycles per line (or less) to generate that line. It is customary to update the TIA every two lines to give the programmer more computation time. The 70 lines of blank give the core 5,320 cycles to update game logic, check for input, and perform other housekeeping tasks.

Since synchronization is so important, it is necessary for the programmer to be able to wait for the beginning of the next scanline before executing more code. This is accomplished by writing any value to the WSYNC register.

COLOR:

Color is output in a chrominance-luminance format just as it is in the MARIA. There are four chrominance-luminance registers which control the color of the objects (some are paired). These registers and the objects they color are

- COLUMP0 - Player 0 and Missile 0
- COLUMP1 - Player 1 and Missile 1
- COLUMPF - Playfield and Ball
- COLUMBK - Background

Each register is 7 bits with 4 bits selecting from 16 available colors and the remaining three bits controlling the brightness of the selected color.

Playfield:

The playfield is described using 20 bits split into three registers (PF0, PF1, PF2). PF0 constructs the first 4 bits of the playfield, PF1 constructs the next 8, with PF2 construction the final 8 bits, ending at the center of the screen. The playfield is drawn where a 1 is seen while the background is drawn where a 0 is seen. Writing a 0 to the CTRLPF register causes the playfield to be duplicated on the right side of the screen. Writing a 1 causes the playfield to be reflected onto the right side.

Movable Graphics:

The remaining objects (player sprites, missiles, and the ball) are movable objects. They can be positioned horizontally by writing to their reset registers. Each object will be positioned at wherever the electron beam was when its register was last reset.

Because of the write timings, this means the programmer can only position each object with a granularity of 15 clocks. This is fine tuned using horizontal motions.

Vertical positioning can be controlled by writing to each objects enable register for the scanline on which the programmer wishes it to appear.

Each object is described more specifically below:

- Missile - The missiles can be horizontally positioned by writing to their reset registers (RESM0, RESM1). They can additionally be reset to the center of their respective player sprite by writing a 1 to their reset player registers (RESMP0, RESMP1). They will remain locked to the player sprite until a 0 is written to the register. They can be enabled by writing a 1 to their enable registers (ENAM0, ENAM1). Their width can be controlled by writing to bits 4 and 5 of the number size registers (NUSIZ0, NUSIZ1) where the possible widths are 1, 2, 4, and 8 bits wide.
- Ball - The ball can be reset by writing to its reset register (RESBL). It can be enabled by writing a 1 to its enable register (ENABL). It can be stretched in width by writing to bits 4 and 5 of the Playfield Control register (CTRLPF) with the same widths as the missiles. The ball can be given 1 cycle vertical delay by writing to the vertical delay register (VDELBL). The reason for this is so the ball can still move smoothly vertically if the TIA is only updated every other line.
- Players - The player graphics can be reset by writing to their reset registers (RESP0, RESP1). They take shape by writing to their player graphics registers (GP0, GP1) which are each 8 bits wide. This allows the sprite to be 8 bits wide and as tall as desired. For each line that the sprite is on, simply write a 1 to a bit to enable the sprite for that clock and a 0 to disable the sprite for that clock. A reflection of the player sprite can be drawn by writing a 1 to the reflection registers (REFP0, REFP1). Multiple copies of the players can be drawn using the bottom 3 bits of number size registers (NUSIZ0, NUSIZ1). Missiles are generated for these copies. The player sprites can also be vertically delayed like the ball using the vertical delay registers (VDELP0, VDELP1).

Horizontal Motion:

Horizontal motion allows the programmer to move the objects relative to their current horizontal position. This allows for finer grained control of the sprites. Each object has a 4 bit register (HMP0, HMP1, HMM0, HMM1, HMBL) which uses two's compliment to load a value from 7 to -8. When the HMOVE register is written to, all registers execute their horizontal motion. This must be done directly after a WSYNC to ensure that all registers have time to execute during HBLANK time and should not

be modified for at least 24 cpu cycles after the HMOVE is written. The programmer can write to the HMCLR register to set all horizontal motion registers to 0.

Collisions:

The TIA handles collisions internally. There are 15 possible two object collisions which are stored in 15 1-bit latches and can be read in the collision registers on the D6 and D7 pins of the data bus where a 1 indicates that a collision has occurred between those objects. Conventionally they are read during VBLANK when all possible collisions have occurred, though this is not required. All collision registers can be cleared by writing to the collision reset register (CXCLR).

Inputs:

The TIA handles inputs on the buttons for the console. There are two types of inputs dumped and latched.

Dumped inputs were originally used for paddle controllers to play pong. For our system, they are connected to the left and right buttons on the 7800 controllers. These inputs are only used if the programmer has 2-button mode enabled (7800 mode only).

Latched inputs are the standard used for the fire buttons on the 2600 and 7800 (in 1-button mode). The latch can be enabled using bit 6 of the VBLANK register. If enabled, the latch register will remain high until a low value is seen on the fire button input. It will then remain low until the programmer resets the latch. If disabled, the core reads the input port directly.

Register Reference Tables:

WRITE ADDRESS SUMMARY

6 bit address	Address Name	7	6	5	4	3	2	1	0	Function
00	VSYNC							1		vertical sync set-clear
01	VBLANK	1	1					1		vertical blank set-clear
02	WSYNC		s	t	r	o	b	e		wait for leading edge of horizontal blank
03	RSYNC		s	t	r	o	b	e		reset horizontal sync counter
04	NUSIZ0			1	1	1	1	1	1	number-size player-missile 0
05	NUSIZ1			1	1	1	1	1	1	number-size player-missile 1
06	COLUP0	1	1	1	1	1	1	1		color-lum player 0
07	COLUP1	1	1	1	1	1	1	1		color-lum player 1
08	COLUPF	1	1	1	1	1	1	1		color-lum playfield
09	COLUBK	1	1	1	1	1	1	1		color-lum background
0A	CTRLPF			1	1		1	1	1	control playfield ball size & collisions
0B	REFP0					1				reflect player 0
0C	REFP1					1				reflect player 1
0D	PF0	1	1	1	1					playfield register byte 0
0E	PF1	1	1	1	1	1	1	1	1	playfield register byte 1
0F	PF2	1	1	1	1	1	1	1	1	playfield register byte 2
10	RESP0		s	t	r	o	b	e		reset player 0
11	RESP1		s	t	r	o	b	e		reset player 1
12	RESM0		s	t	r	o	b	e		reset missile 0
13	RESM1		s	t	r	o	b	e		reset missile 1
14	RESBL		s	t	r	o	b	e		reset ball
15	AUDC0					1	1	1	1	audio control 0
16	AUDC1				1	1	1	1	1	audio control 1
17	AUDF0				1	1	1	1	1	audio frequency 0
18	AUDF1					1	1	1	1	audio frequency 1
19	AUDV0					1	1	1	1	audio volume 0
1A	AUDV1					1	1	1	1	audio volume 1
1B	GRP0	1	1	1	1	1	1	1	1	graphics player 0
1C	GRP1	1	1	1	1	1	1	1	1	graphics player 1
1D	ENAM0							1		graphics (enable) missile 0
1E	ENAM1							1		graphics (enable) missile 1
1F	ENABL							1		graphics (enable) ball
20	HMP0	1	1	1	1					horizontal motion player 0
21	HMP1	1	1	1	1					horizontal motion player 1
22	HMM0	1	1	1	1					horizontal motion missile 0
23	HMM1	1	1	1	1					horizontal motion missile 1
24	HMBL	1	1	1	1					horizontal motion ball
25	VDELP0								1	vertical delay player 0
26	VDEL01								1	vertical delay player 1
27	VDELBL								1	vertical delay ball
28	RESMP0							1		reset missile 0 to player 0
29	RESMP1							1		reset missile 1 to player 1
2A	HMOVE		s	t	r	o	b	e		apply horizontal motion
2B	HMCLR		s	t	r	o	b	e		clear horizontal motion registers
2C	CXCLR		s	t	r	o	b	e		clear collision latches

READ ADDRESS SUMMARY

6 bit address	Address Name	7	6	5	4	3	2	1	0	Function	D7	D6
0	CXM0P	1	1							read collision	MO P1	M0 P0
1	CXM1P	1	1							read collision	M1 P0	M1 P1
2	CXP0FB	1	1							read collision	P0 PF	P0 BL
3	CXP1FB	1	1							read collision	P1 PF	P1 BL
4	CXM0FB	1	1							read collision	M0 PF	M0 BL
5	CXM1FB	1	1							read collision	M1 PF	M1 BL
6	CXBLPF	1								read collision	BL PF	unused
7	CXPPMM	1	1							read collision	P0 P1	M0 M1
8	INPT0	1								read pot port		
9	INPT1	1								read pot port		
A	INPT2	1								read pot port		
B	INPT3	1								read pot port		
C	INPT4	1								read input		
D	INPT5	1								read input		
										Note : I0, I2, I2, I3 can be grounded under software control. I4, I5 can be converted to latched inputs under software control		

A more detailed description of the TIA can be seen in the Stella Programmer's Guide [9]

Our implementation:

Our TIA graphics and input implementation was developed by Daniel Beer for his Atari 2600 project.

Shortcomings:

Atari 2600 mode proved to be the greatest shortcoming in our project. We tried to get VGA running in sync with the TIA as it does with the MARIA but ended up resorting to using frame buffers. We had two frame buffers that we would swap between as the TIA finished writing to them. Unfortunately, Beer's TIA implementation had some bugs in it. That combined with the fact that we were unsure if our core was 100% cycle accurate led to it being difficult to debug the TIA. We also ran short on time after spending most of the semester getting 7800 mode working, so we could not put the effort that we wanted toward debugging 2600 mode. In retrospect, we should've written our own TIA graphics, so that we could understand and debug it properly.

Sound

The Atari 7800 was originally intended to have a dedicated sound chip, but when that design fell through, the designers decided to use the sound output from the TIA. This was one of the largest sources of criticism for the console since its sound was noticeably weaker than the rest of the console.

The Tia has two 1 bit audio channels. Each is controlled are controlled by three registers

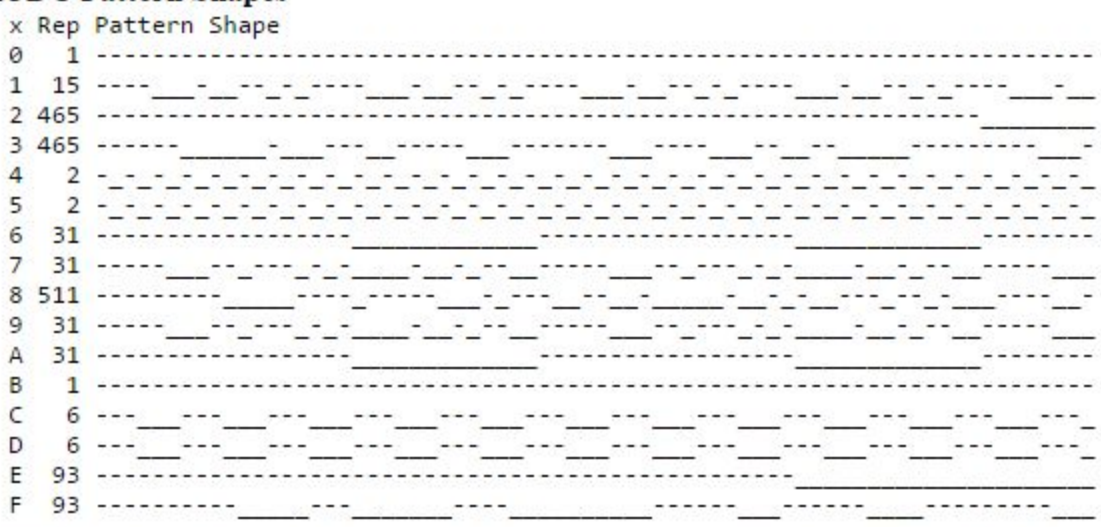
AUDVx - 4 bit register. This register controls the volume of the channel.

AUDFx - 5 bit register. This register controls the frequency of the channel. A 31139.5 Hz clock is divded by the value in this register plus 1 to get the frequency of the channel.

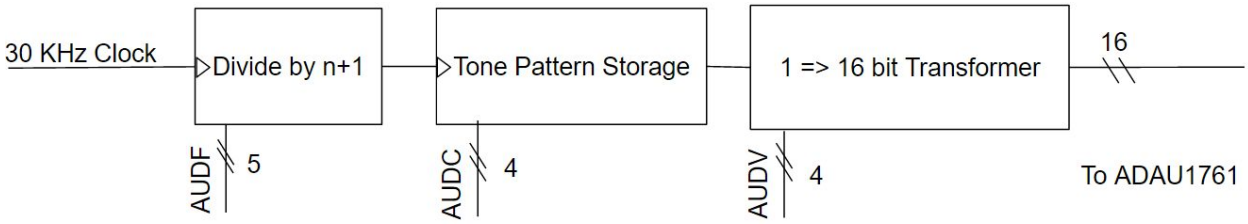
AUDCx - 4 bit register. This register controls the tone of the output. There are 11 unique tones. These tones can vary from pure flute tones to pulsing rocket sounds and explosions.

The output is created by shifting a bit out of a 9 bit shift register that is clocked by the divided clock. Depending on the tone, a new bit is shifted in. In most cases, this is based off of a 4, 5, or 9 bit polynomial counter, but in some cases it's based off a combination. This creates a pattern for each tone that varies in length from 1 bit long to 511 bits long. The creation of these patterns is beyond the scope of this report. More information on how these patterns are created can be found at [6].

AUDC Pattern Shapes



This diagram shows the beginning of each pattern. For our implementation, we generated each pattern and stored it as an array in the fabric. We then iterate through the pattern specified by the AUDC registers and output the bit.



This diagram shows our design for the sound. The original clock is divided by the value in AUDF+1. That clock is used to clock our iteration through the pattern array for the specified tone. The output bit is then scaled up to a 16 bit digital audio value. This is done in such a way that a 1 output with an AUDV value of 15 is very close to int_max, and a 0 output with an AUDV value of 15 is very close to int_min. The two channels are mixed and then sent to the audio codec (ADAU1761).

RIOT

Overview

The RIOT IO Chip or MOS 6532 was a popular IC in the late 70s and early 80s. It integrated 128 bytes of RAM, two 8 bit bidirectional ports, and a programmable interval timer. Its multiple functions meant that it could be used to replace multiple ICs. It was used in both the Atari 2600 and 7800.

RAM

The 128 bytes of RAM is laughable by today's standards, but in early computers, it was a significant amount of space. The 7800 did not use the RAM in the 6532, but it was the only RAM available in the 2600. For that reason, the chip's RAM is only used in Atari 2600 mode. While it will technically be possible to use it in 7800 mode, programming guides advised against it, so it is unlikely any program will actually use it.

Ports

This is the main feature that the RIOT is used for. Both ports are used as inputs. The directional inputs of the controllers is handled by PortA, and the system buttons are handled by PortB. These inputs are latched into a register at each clock edge to be used by the programmer.

Timer

The timer can be used to allow the programmer to time things like scanlines while executing other code. It is given the same clock as the system clock. It is used in 2600 mode, but is likely not used in 7800 mode (since it would be running on a different clock as the core and wouldn't be very useful). A value from 1 to 255 can be loaded into any one of the four interval registers, and the value will be decremented by one at each interval.

The four interval registers and their intervals are

- TIM1T - 1 clock cycle
- TIM8T - 8 clock cycles
- TIM64T - 64 clock cycles
- T1024T - 1024 clock cycles

The timer can be read by reading the INTIM register. When the timer reaches 0, it remains there for 1 cycle. It then flips to FF and begins decrementing at every clock cycle so the programmer can determine how long it has been since the timer reached 0.

Our Implementation

We used the code developed by Daniel Beer for his project of recreating the Atari 2600 on an FPGA. We have verified the code ourselves and are convinced that it will suit our needs for the RIOT IO chip.

Hardware Interface

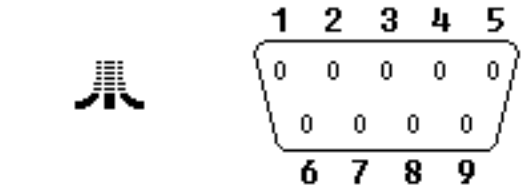
The Atari 7800 interfaces with two types of hardware peripherals: game controllers and game cartridges. In order to communicate with these peripherals in our own implementation, we built custom circuitry to connect the peripherals. This circuitry interfaces with the Zedboard through the FMC port on the board. The FMC port was chosen over other IO ports on the board (like the PMOD ports) because this single port provided 68 individual IO pins through a single interface, which was enough pins to connect both the cartridge and controllers to. We purchased an FMC breakout board from a third party seller which brought the FMC pins out onto a development matrix (similar to a perf board matrix). This allowed us to solder wires from our circuitry directly to the FMC port.

One issue we ran into with the hardware interface is that the Zedboard operated at a 2.5 voltage level and the hardware peripherals of the Atari expect to operate at 5 volts. This meant that we couldn't just connect the peripherals directly to Zedboard. Instead we had to wire connections through a set of logic level shifters which could automatically convert from 2.5 to 5 or from 5 to 2.5 volts. Although these level shifters solved the issue of different operating voltages, they resulted in a fairly dense and ugly spanse of circuitry and the protoboard this circuitry was built on added to the overall footprint of our system.

Since both of the peripherals we wanted to connect with were relatively wide (12 pins for two controllers and 32 for the cartridge) we used ribbon cable for the circuit implementation. Ribbon cable works especially well in bussing applications and was quite a boon in this project.

Controllers

Both Atari 2600 and 7800 game controllers use the standard 9 pin DSub connection shown below to connect to the system.



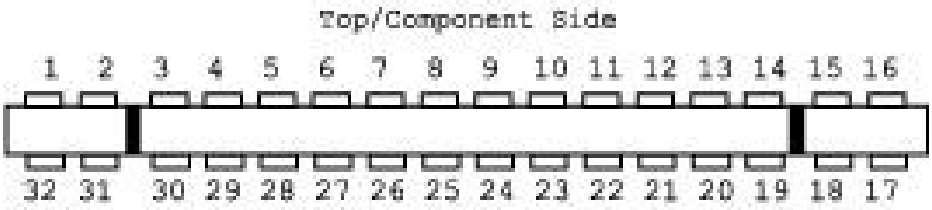
1. **Joystick Up**
2. **Joystick Down**
3. **Joystick Left**
4. **Joystick Right**
5. **B input Paddle/Touch Tablet**
6. **Input Trigger**
7. **+5V**
8. **Gnd**
9. **A input Paddle/Touch Tablet**

In our system pins 7 and 8 are attached directly the 5v and Gnd lines of the Zedboard. This solved a persistent and confounding issue we had involving inconsistent reference grounds between the Zedboard, the FMC breakout circuit, and our power supply/probes. This configuration was also optimal for demo day since it meant we didn't have to haul around an external power supply.

Pin 6 is attached to the 5v line which effectively ties the controllers to two-button mode. This configuration was the only way we could get consistent responses from the paddle buttons, so the hardware handles interpreting trigger responses based off the paddle buttons when the system is running in one-button mode. The joystick pins are driven active low and thus are attached to pull up resistors in our circuit.

Cartridge Interface

The cartridge interface consists of address and data bus pins in addition to power and ground pins. The pinout is shown below:



Atari 7800 Cartridge Pinout, looking at cart edge.

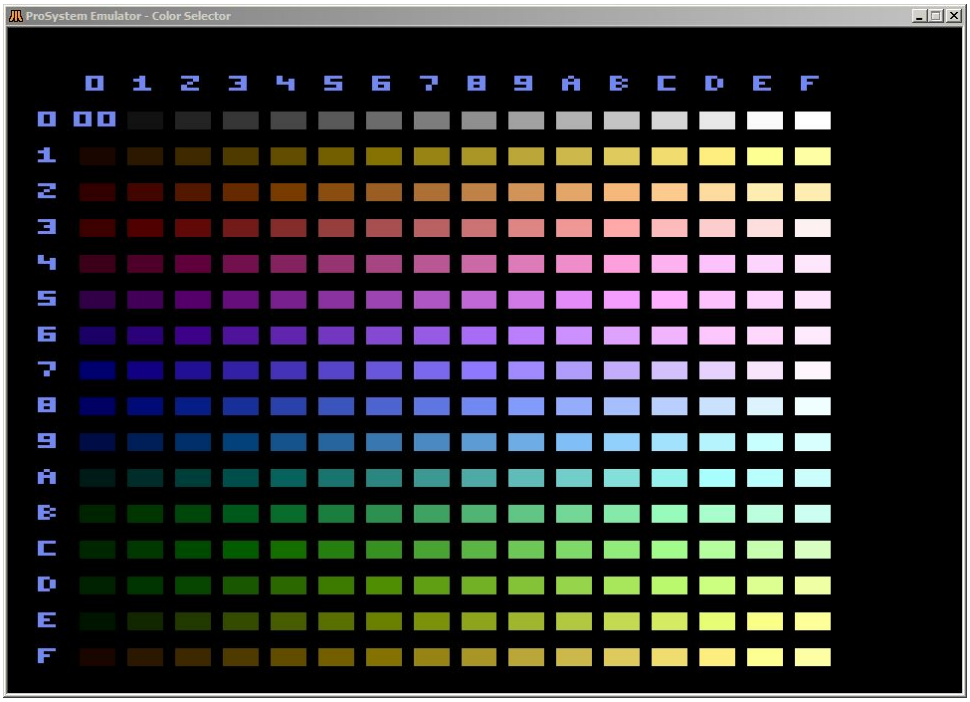
1	Read/Write from 6502, low = Write	32	Phase 2 clock from 6502
2	Halt to 6502	31	IRQ to 6502
3	D3 to/from 6502	30	Ground
4	D4 to/from 6502	29	D2 from 6502
5	D5 to/from 6502	28	D1 from 6502
6	D6 to/from 6502	27	D0 from 6502
7	D7 to/from 6502	26	A0 from 6502
8	A12 from 6502	25	A1 from 6502
9	A10 from 6502	24	A2 from 6502
10	A11 from 6502	23	A3 from 6502
11	A9 from 6502	22	A4 from 6502
12	A8 from 6502	21	A5 from 6502
13	+5 VDC	20	A6 from 6502
14	Ground	19	A7 from 6502
15	A13 from 6502	18	External Audio to system
16	A14 from 6502	17	A15 from 6502

Pins 3-14 and 19-30 are identical to the Atari 2600 and allow the 7800 to utilize existing 2600 cartridges. The remaining pins are 7800 specific pins.

Unfortunately we were never able to implement a fully working cartridge interface with our system. The final product had ROM images of the cartridges built directly into the hardware of the system. In the end, this part of the project was sunk by the real world intricacies of analog circuitry and by a late start on this portion of the project. Although we built and tested a circuit which could drive the cartridge inputs according to specification, we were never able to get the cartridges to respond as we expected. When we tested actual cartridges with repeated read signals to a single particular address the data we read back from the cartridges was garbage. Instead of responding by driving a consistent data value on the bus, our probing revealed the cartridge driving either all ones, or floating values, or values that started high and then decayed to zero with time. When we compared the waveforms we were driving cartridges with to the waveforms from the Atari, we noticed that the Atari clock wave looked very much like a saw tooth whereas our clock wave looked much more square but included a lot of ringing. We hypothesize that this might be the cause of our issues, but ran out of time by the end of the semester to verify the problem and devise an appropriate work around.

VGA

The original Atari 7800 output 8-bit chrominance/luminance (UV) values over an RCA cord to an NTSC TV. Our FPGA system has VGA output, so we convert the 8-bit UV values to 12-bit RGB. We could not find a UV-RGB conversion algorithm, probably because the perceived color output by a UV pair is television-dependent. but we did find the following image of the color palettes displayed on a CRT TV [7]:



Each row has a different chrominance value and each column has a different luminance value. We wrote a script to extract the RGB values from the cells in the image. Our UV-RGB converter simply indexes into a 256-entry ROM for each of red, green, and blue. The VGA controller itself is quite simple. It has a row counter from 0 to 524 with VSync at 490-491 and a column counter from 0 to 799 with HSync at 656-751. The row and column counters are output to the Maria for synchronization purposes.

In 2600 mode, the VGA module manages a full 160x192 pixel frame buffer. This is because the VSYNC signal generated by 2600 games does not align with the VSYNC signal frequency expected by VGA, so we could not implement row-by-row synchronization. Instead, the TIA writes to the frame buffer according to its own synchronization signals, and the VGA module reads from it at its own pace. To reduce jitter, we implemented a two-frame buffer queue, where the TIA writes to one and swaps it with the VGA read buffer when it is done writing a frame. A greater number of frame buffers would have reduced jitter further.

BIOS

The BIOS is stored in a special ROM internal to the console. It is located in the address space from 0xF000 to FFFF. Since this overlaps with cartridge space, there exists a bit in the control register to control whether data is read from the BIOS or the cartridge. The BIOS performs the following tasks (in order)

- Basic memory and core functionality test
- More complex memory test
- Copy authentication BIOS code into RAM
- Set up the MARIA DLLs and ZP to draw the boot screen (rainbow Atari logo)
- Authenticate cartridge
- Jump to start of the game in either 2600 or 7800 mode

The authentication process is a result of a number of unlicensed pornographic games that were made for the Atari 2600 console. Atari did not want these games on the 7800, so they created an authentication algorithm which iterates through the data on the cartridge generates a series of bytes that are compared with the values on the cartridge from 0xFF80 to 0xFFF9. If the data matches, the game is launched in 7800 mode. Otherwise, the game is launched in 2600 mode.

Hidden Control Register

The hidden control register is aptly named because there is extremely little documentation about it online. Most the information about it came from the Atari 7800 schematic and a blog post by Dan B.

It is essentially a latched register that is only used during the BIOS. It is written to whenever the TIA is written to and has four bits which are written to by their corresponding bits on the data bus.

- Bit 0 (Lock bit) - When this bit is 1, the control register will no longer be written to. The BIOS writes a 1 to this bit just before entering Atari 2600 mode and any Atari 7800 game is expected to write a 1 to this register immediately at the beginning of the game.
- Bit 1 (MARIA enable) - When this bit is 1, the MARIA is enabled as the video driver. It also enables the 4KB of RAM in the Atari 7800. This bit cannot be enabled at the same time Bit 3 is enabled.
- Bit 2 (BIOS enable) - When this bit is 1, the cartridge is enabled for all reads from 0x8000 to 0xFFFF. When it is 0, the BIOS is enabled for all reads from 0x8000 to 0xFFFF, though BIOS code is only located at 0xF000 to 0xFFFF.

- Bit 3 (TIA enable) - When this bit is 1, the tia is enabled as the video driver, and the console is locked in 1-button mode. This bit cannot be enabled at the same time Bit 1 is enabled.

Approach

Design Partitioning

Our project consisted of four distinct phases: Research, Design, Implementation, and Integration.

The research phase involved digging up and reviewing as many technical specs on the Atari 7800 as we could find. It also involved searching for previous solutions or projects that were similar to what we intended to produce in order to find starting ground and inspiration for our own design.

Since our aim was to recreate an existing system the design phase of this project was somewhat trivial after thoroughly researching the Atari 7800 design. We did make some key decisions in this phase however. First we affirmed that it was necessary to implement our own version of the MARIA. Additionally we made the decision to port the Atari 7800 video output to a VGA interface.

In the implementation phase we created and verified the individual pieces of the system. For some parts of the system, like the core and the TIA chip, we were able to start with an existing design and alter this design as necessary to suit our needs as well as verify their functionality. The largest effort in this phase was directed towards reverse engineering the MARIA chip from behavioral descriptions and technical documents.

The bulk of our efforts in this class were spent in the integration phase. In this phase we brought our individual pieces together into the full system and debugged the issues with our design and implementation at a system level, usually by comparing what our device was doing to what we expected it to do according to the Atari 7800.

Tools and Design Methodology

We used Vivado as our design tool for synthesis and Vivado's simulator. After designing any component, we would simulate it extensively before synthesizing it. While this worked pretty well for finding design flaws, there were still some problems that were only able to be observed on debug cores in the real system. For example, race conditions between different signals would not always be obvious in simulation, but could be debugged with a bit of effort using the debug cores.

We used the bench equipment, including the power supply, meters, and scope probe, to build and test the hardware interface. We also used it to probe an original Atari 7800 system to aid in designing the hardware interface.

Testing and Verification

Each module was tested individually using a separate test bench. First, we simulated the test bench with some input. For the Core, this was the test suite; for the MARIA, this was a hand written program that was designed to display a custom image; for the sound module, we inspected the effects of changing the tone, frequency and volume registers using the switches on the board; and so on. Afterward, we synthesized the modules to see if we could get some response on the board to indicate it was functioning correctly. After we were convinced each module was working correctly, we integrated them and ran full system simulation, using the Vivado simulator's waveform viewer to find bugs in our design. Lastly we synthesized the full system and used ILA cores to trigger on positions in the code and screen positions where we knew bugs were occurring.

Testing and verifying the peripheral circuitry involved building many quick test circuits to drive the FMC ports, a cartridge, or the controllers and then probing the responses on the scope probe. Often times we probed both our circuitry and an Atari 7800 we bought in order to compare functionality.

Status and Future Work

On demo day, we had a system with five fully working 7800 games saved as ROM images on the board, two broken 2600 games, and a controller interface. Although we were happy with our end result, further results that we would have liked to see are:

- Working cartridge interface
- Fix TIA graphics, VGA interface, and timing issues
- Further testing of the Maria and individual 7800 games to expand our working library
- Make a sleek and cool case for the system

Lessons Learned

What we wish we had known

- Figure out Vivado's intricacies early. If we had learned about synthesis attributes earlier, we would've had a much easier time debugging on ILA cores.
- ILA cores work best if you use your board's base clock as the reference clock.
- Clocking can be really hard to get right. Figure out your clocks early.
- Don't make assumptions about ambiguities in specs. Figure it out through more research and testing. If you do make assumptions, take note of it as it may be a problem later.

Vivado Tips

- When using ILA cores, if you have a mysterious port width mismatch error during Generate Bitstream, change the clock for your debug cores to the base clock for your board (100MHz for us.)
- If you can't find the net that you want to mark debug on, use the keep attribute. If that causes your design to fail, try something like this:

```
logic important_net;
(* keep = "true"*) logic important_net_kept;
 assign important_net_kept = important_net;
```

  And then mark debug on important_net_kept.

- If Vivado claims a certain net is undriven and will be tied to zero, it might just be because that net is optimized out in the synthesized design, and it will still do what you expect.
- If Vivado is giving you weird error messages related to your debug cores, it can help to go into the constraints and remove any useless lines. Vivado doesn't do a very good job of curating its constraints file when it modifies debug cores, so it often leaves extra lines in that cause errors. Unfortunately, this means you'll have to resynthesize your design.

Good Decisions

- Choosing a realistic, obtainable project that wasn't too easy. We originally wanted to do the Atari 2600, but that would have likely been too easy. The Atari 7800 challenged us throughout the semester without leaving us utterly unsuccessful in the end.
- Deciding to start with an already-implemented core design. This saved us a lot of time and frustration early on.
- Working together in the lab as a group frequently. We had two meeting times per week scheduled but often met more than that. Our important in lab activities included

    - Debugging in a pair. This was important because people who worked on different parts of the system had wider perspective about what might be causing a bug.
    - Designing in a pair. This was particularly important with the Maria since it was such a complex system.
- Testing the Maria incrementally. We first designed the VGA controller and tested some output patterns with RGB colors, then with chrominance-luminance colors as this is what the Maria outputs. Then we tested an output pattern with all the read/write graphics modes and set up a manual DMA interface to make sure the Line RAM was updating correctly in response to DMA commands. Finally, we set up a display list list in memory and verified that it displayed the correct pattern on screen. Because of this incremental testing, when it came time to integrate the system, we knew any display issues were not the fault of the display system.
- Stepping stone goals. These allowed us to set achievable goals that could actually be reached and let us know if we were behind schedule.
- Doing a lot of debugging in simulation. This helped fix a lot of easy system integration bugs.
- Designing sound early. Since sound is almost completely independent of the rest of the system, designing it early meant we didn't have to worry about it for the rest of the semester. Our sound design did not change since the day we designed it.
- Keeping each other updated on progress even when we were working on different parts. In general, having an active dialogue between the group encouraged progress and helped us overcome tough obstacles.
- Bringing larger monitors to our work station allowed higher productivity especially when using Vivado's large clunky GUI.

Bad Decisions

- Not implementing our own TIA and postponing testing 2600 games until it was too late to fix them. The TIA is not nearly as complex as the MARIA, and we could have likely designed it ourselves in a short time. This would have given us a lot more understanding of how it worked and allowed us to debug 2600 mode a lot earlier.
- We took too slow of a start at the beginning of the semester. Although we were able to produce good results by the end of the semester, we could have saved ourselves the several weeks of intense work towards the end by spreading the load more evenly across the semester.
- Debugging in simulation for too long because we couldn't figure out ILA. The problem that prevented us from using ILA was the fact that a free-running clock was not used as the reference clock. Vivado gave us weird errors that did not lead us to this problem for at least a week or two.

- We waited too long to begin implementing hardware interface and thus when we ran into a series of frustrating obstacles during the process we ran out of time to debug the cartridge interface.
- Forgetting to turn in many of the status reports, they're due every Monday folks.
- Probing indiscriminately on the zedboard. This led to some a couple accidental shorts and two dead Zedboards.

Words of Wisdom

- Start working **hard** early
- Research extensively. Don't brush over ambiguities
- Figure out Vivado early

Personal Statements

Chris Barker

What I did

I was the Maria guru for the duration of the project. During the research and design phases, Mark and I worked together to understand what sparse specifications for the Maria we could find. We came up with a set of subsystems and interfaces between these. During the design phase, it was crucial that we hashed out the details together since it was such a complex system. Then we each wrote the implementation for two of the four subsystems.

From then on, I was responsible for the Maria. I wrote the VGA controller and the chrominance-luminance converter, tested these two modules, and then moved on to test the Maria. I started by testing the Line RAM output in all of the read/write graphics modes, then moved on to test the DMA commands to the Line RAM without using memory, and then integrated it with the memory system.

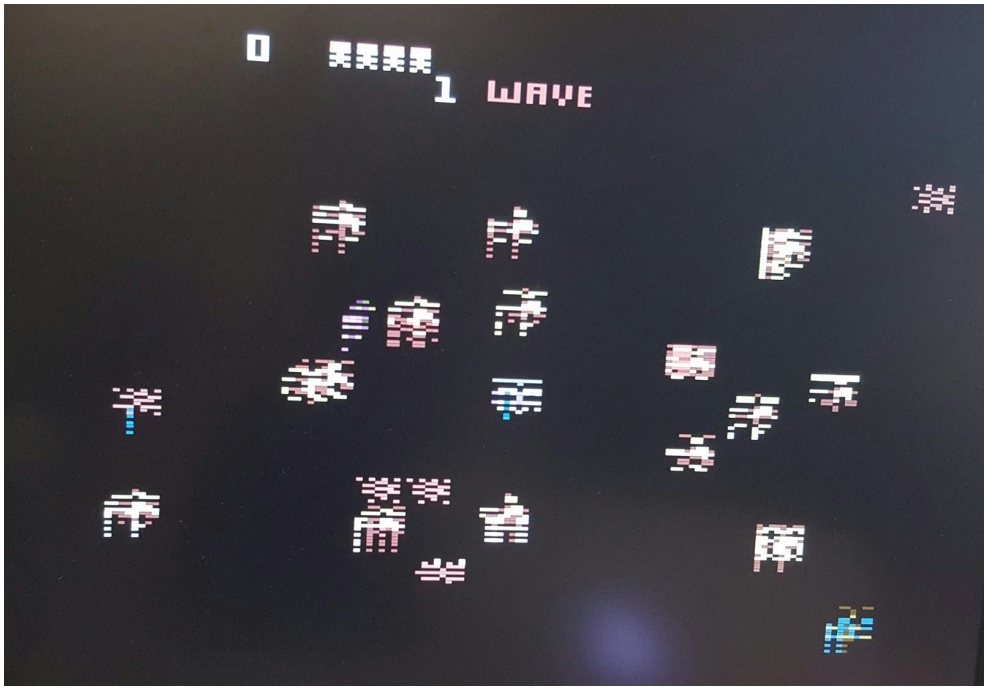
Several features were still not implemented after my first pass with Mark, including indirect mode graphics, display list interrupts, and DMA timeouts. I implemented and tested these next.

By the time I had finished implementing and testing the Maria, all the components of the system were complete and it was time for integration. I worked closely with Ryan to debug the internal system execution, starting with the BIOS code and later moving onto games. We spent a few weeks in simulation until the core was able to complete the BIOS. During this time, there were several issues with the halt signal, the display list interrupt signal, memory mapping, switching clock frequencies, and race

conditions. For most of these issues, knowledge of the Maria, Sally, and memory system were necessary to understand what the problem was, which is why it worked so well for Ryan and I to do this side by side.

Once we had finished getting the CPU to run through the BIOS in simulation, simulation runs took an entire night to run and sometimes crashed Vivado, so it was time to switch to synthesis. We encountered a new slew of bugs with BIOS here, particularly with the reset signal and timing violations that didn't occur in simulation. After we got past these, it was time to debug game code. This is where several bugs in the Maria finally got fixed, once we could see how the games expected their graphics to work. We discovered graphics data could be in slow or fast memory for both direct and indirect mode, whereas we had previously thought they could only be in slow memory for indirect and fast memory for direct. We discovered display list interrupts should be triggered *before* a zone is rendered, rather than after. We discovered that indirect mode character maps had a different format than we expected. And so on. These bugs took quite a while to find thanks to long, nondeterministic synthesis runs, trouble with ILA cores, and the time it took to parse Robotron and Ms. Pac Man code to determine the format of their display lists.

In the last couple of weeks, I worked on the controller and cartridge interfaces on the verilog side, and built the frame buffer and VGA controller for 2600 mode.



Maria expecting 7.14MHz read latency from a cartridge ROM clocked at 1.79MHz

Time Spent

During the research phase - the first 4 weeks when we had readings and labs to do - I spent about 8-10 hours per week on the course. For the next 4 weeks, the implementation phase, I spent 12-18 hours per week. For the remainder of the course, the integration and debugging phase, I spent 15-35 hours per week in the lab. There was one day in the last week when I was in the lab for 15 hours. The three of us were here until 4:30 a.m. listening to Christmas songs and trying to get 2600 mode to work. I would warn you to spread your work evenly over the semester, but that will never happen. Just try to spread it more evenly than I did.

Class Impressions

This course was a bit of a shock to me in that I had to set my own goals and deadlines. At first, I thought of the readings, labs, proposal, status updates, and the design review as the outcomes of the class that I had to meet for a grade. That may be true for the readings and labs, but the proposal, status updates, and design review are byproducts that should come nearly effortlessly if you're working sufficiently hard on the self-guided portion. Once I realized that, I was more effective in the project.

This class will be a huge pain if you don't genuinely want to implement your project. Digital design is challenging, and Vivado is rough, and I had other classes to keep up with as well, but the thought of building a working game console kept me going. If you are passionate about design, this is a great class for you. You have nearly complete freedom to choose what system you think will be cool. Money, lab equipment, and equally passionate teammates are at your disposal - all you have to do is put your skills to work.

Ryan Roberts

What I did

In the early stages of the project, I did most the research on which core to use. After looking on Open Cores, I found that most the 6502 cores were written in an extremely cryptic manner and were poorly documented. It was then that I found Arlet Ottens' 6502 implementation and told everyone else in the class about my findings. While his core did not entirely follow the spec, it was very well documented, which made it a lot easier for us to understand what the Core was doing during debug. I then validated the core using the test suite. Because of this, I was the team member with the most knowledge of the core. I worked extensively with Chris to debug the full system implementation as we tried to navigate the BIOS. In the beginning we had a lot of problems with integrating the core into the rest of the system, and this is where my knowledge shined as Chris and I were able to many system bugs in a short time. Once we had the system exiting the BIOS, I continued to debug the system with Chris. At

this point, most the bugs in the system were graphics related, so he took charge, while I helped as best I could.

I also researched and implemented the sound early on in the project. It proved somewhat difficult to find sources that understood how the sound on the TIA chip worked at a low level, and it was even harder to find sources that were well written. After a lot of digging, though, I had enough knowledge to implement it. I took advantage of the fact that we had a lot of logic elements to program the sound patterns directly into the fabric. This reduced the complexity of the sound design a good deal. Once I was confident I had implemented the sound correctly, I wrote a test bench to test it, and validated its tone authenticity with an online tool called Tone Toy [10]. I then integrated it into the full system, and, since the sound system is almost entirely independent of the rest of the console, we had no issues after that.

I was also the one who found Daniel Beer's implementations of the RIOT and TIA online and integrated them into our project. I took it upon myself to understand how the RIOT worked, though we didn't have to make any changes to the code. In the end of the project, I learned a lot about how the TIA worked and worked with Chris to attempt to debug the TIA implementation as best we could. Unfortunately, we ran short on time and had to go with what we had at the end of the project.

Time Spent

In the beginning of the class while doing research, I spent anywhere from 6-12 hours per week. As we got closer to integration, that number continued to increase, and once integration began (early November), I would spend anywhere from 20-40 hours per week in the lab. Especially in the final two weeks of the project, Chris and I spent almost every day in the lab from about 10:30 AM to 10:30 PM.

Course Impressions

Wow what a great course! This is the first time that I've ever been given such an open ended assignment as "do something cool with an FPGA." The professors and course staff were an absolute joy to work with, and my teammates and I had a ton of fun implementing this project in the lab. Even though we spent a lot of time on it, it didn't feel like a lot of work! My only complaint about the course is that I wish we were given a bit more information on how to get started with Vivado. We were just kind of told to get started when a bit of direction would have been extremely helpful. Overall, I'm extremely thrilled with the class and very proud of my group's performance.

Mark Wuebbens

What I did

In the initial stages of the project I spent a lot of time looking for and reading through technical documents for the Atari 7800. When we determined that we could start with an existing implementation of most system components I focused my efforts on researching behavioral descriptions of the Maria since we needed to implement this part from scratch. Thus while Ryan worked on finding and verifying a core implementation, Chris and I started on a first draft Maria design. The initial design required Chris and I working closely together to decode several often conflicting Maria specifications. During this stage we specified a top level interface and split the design into several key subsystems, which we then implemented separately. Although the design at this point was buggy and not fully complete, it was a crucial stepping point since our Maria implementation pinned down much of the top level interface for the system.

After this point almost of all the main system components were implemented. Ryan and Chris began debugging things like the BIOS at a system level while I worked on developing an interface for the Atari peripherals. This endeavor was what I spent the bulk of my effort on for the rest of the class. First I made the decision to interface to the peripherals through the FMC port of the Zedboard, since this port was wide enough to include all of the signals I would need. I found and purchased a breakout board adapter for the FMC port online which allowed me to solder connections directly to a perf board matrix on the adapter which were connected to IO pins on the FMC. I also discovered that the normal operation voltage of the zedboard ports was 2.5 volts whereas the Atari peripherals operated at 5 volts, so I found logic level shifter ICs that would do the work of stepping up or down the voltage levels between the FMC and peripherals.

The peripherals had many pins and wide buses to connect to so I spent over a week creating ribbon cable interconnects to serve as wiring for both the FMC and peripheral sides of the circuit. I stripped, tinned, and then crimped either male or female adapters to both sides of each wire on the interconnects to ensure reliable connections and allow quick prototyping and redesign of the circuitry. (For groups in the future planning on using ribbon cable to wire wide buses, I recommend looking into appropriate male/female ribbon cable adapters to save on time and hassle.) I also soldered wiring to female DSub adapters to connect the controllers to. These tasks were made much harder and quite frustrating by the lack of usable soldering irons in the lab.

Finally I worked on debugging this circuit connection to the system. Fixing the circuit connecting the controllers required attaching pull up resistors to the directional pins and a hack that involved tying the trigger pin to high to force the controllers into two-button mode. Unfortunately attempts to get the cartridge interface working were stymied. Although my circuit appeared to be drive the cartridge pins according the Atari specification, the cartridge never responded with data in the way we expected it to, and thus the system was not able to interface with any physical cartridges the way had hoped it would. I ended up running out of time to debug and fix this problem.

Time Spent

At the beginning of the course I probably spent on average 8-10 hours a week either doing research or meeting and working with my group on the project. As the semester ramped up and the class got into full swing I spent around an average of 15 hours a week, usually working on the peripheral interface but also meeting and talking with my group about the system as a whole.

Course Impressions

I found this one of the most uniquely challenging, but also rewarding classes at CMU. I was intrigued by the open ended nature of the projects and staggered by the sheer volume and complexity of the systems that my team and others implemented in a semester. I am left with a strong impression of how important a good team and effective teamwork and communication is in designing complex systems.

I was quite pleased with cheerful available staff for the course and with the professors who are clearly very passionate about digital design and quite willing to share their knowledge and experience in the area.

Sources

[1] 7800 Software Guide:

http://www.atari7800.org/manuals/7800_Software.pdf

[2] Maria Technology:

http://www.atari7800.org/manuals/7800_Maria_Specs.pdf

[3] GCC1702B “MARIA” CHIP

http://www.atarimuseum.com/ahs_archives/archives/pdf/videogames/7800/gcc1702b_maria_specs.pdf

[4] 6502 Model Github

https://github.com/Arlet/verilog-6502

[5] Test suite

https://github.com/Klaus2m5/6502_65C02_functional_tests

[6] TIA Sound information

http://problemkaputt.de/2k6specs.htm

[7] Atari 7800 Palettes

http://atariage.com/forums/topic/209210-complete-ntsc-pal-color-palettes/

[8] Atari 2600 Project

http://people.ece.cornell.edu/land/courses/eceprojectsland/STUDENTPROJ/2006to2007/dbb26/dbb28_meng_report.pdf

[9] Stella Programmer’s Guide

http://www.atarihq.com/danb/files/stella.pdf

[10] Tone Toy

http://www.randomterrain.com/atari-2600-memories-program-tone-toy-2008.html