1942 Final Review

12/9/2012

18-545

Team Arcade

Tyler Huberty, Greg Nazario, Isaac Simha

Table of Contents

Introduction

Our group built the arcade game 1942 on an FPGA. 1942 is a vertically scrolling Capcom arcade game from late 1984 set in the Pacific Theater of World War II. The player's goal is to shoot down enemy planes and avoid enemy fire. Two players could play in an alternating cooperative mode. It was a popular game followed by several sequels.

Platform

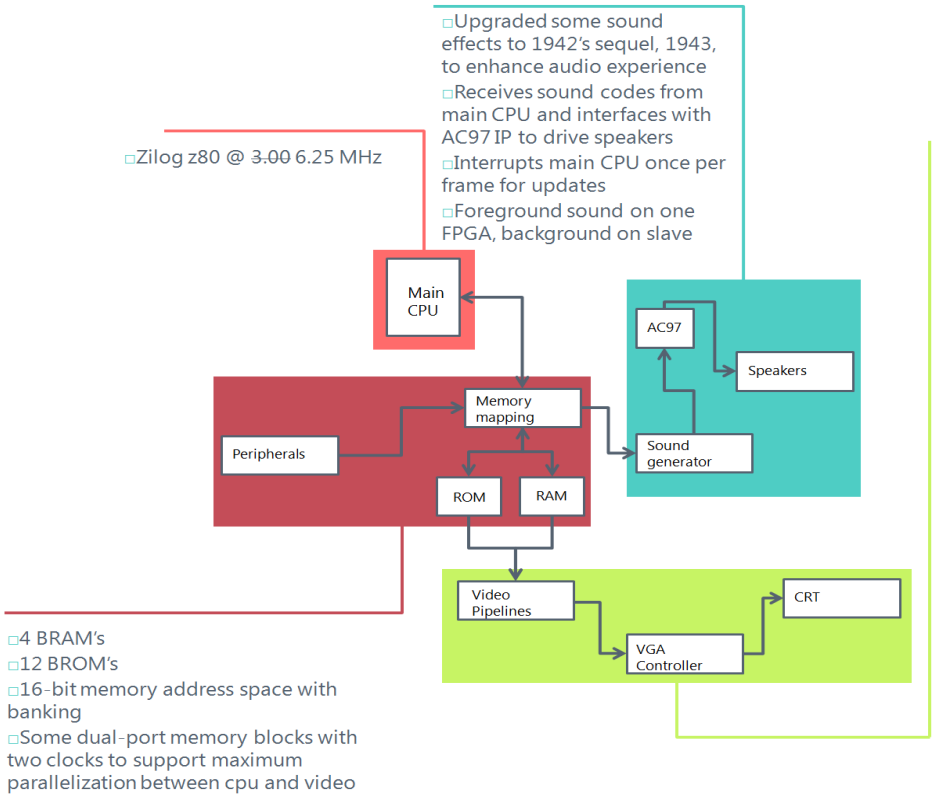
We built 1942 on a Xilinx Virtex-5 LX110T FPGA. We did not use a PowerPC core.

System Overview

The 1942 arcade system had the following hardware specifications:

Main CPU	Zilog z80 @ 4.0MHz
Audio CPU	Zilog z80 @ 3.0MHz
Sound Chip	2x AY-8910 @ 1.5MHz
Display	224 x 256 pixels, 256 colors, raster CRT

Our implementation of the 1942 platform consisted of the Z80 for the main CPU, a display system, a memory mapping system, and a sound controller linked to the AC97. We made the decision not to use the second Z80 and two AY8910 sound chips, which allowed us to record our own upgraded sounds.
The display system was powered by our VGA controller (see “VGA controller” section) and our sprite and tilemap hardware (see “sprite and tilemap hardware” section). The behavior of these components was not documented at the cycle level and as such we sought only to replicate the functionality. A block diagram of the system follows.



Main CPU

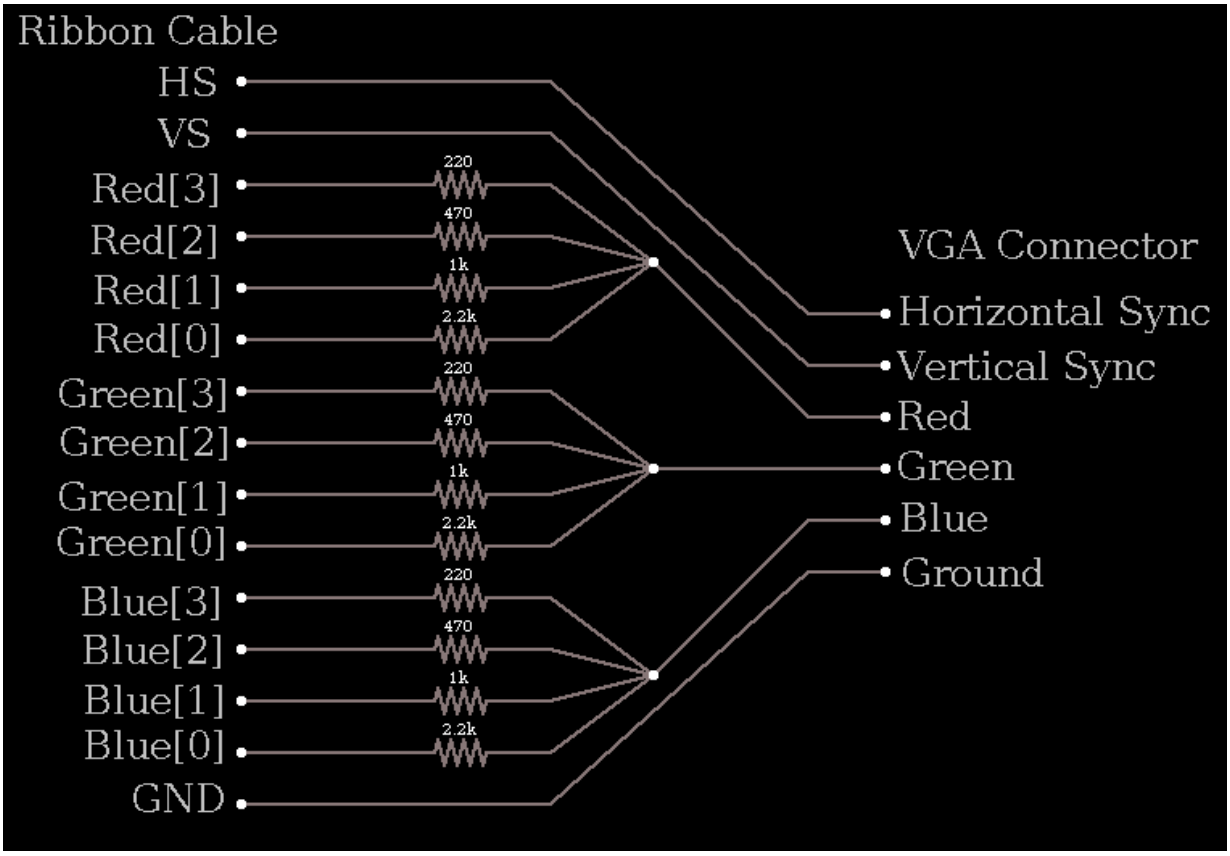
The Zilog z80 is an 8-bit microprocessor. Our project made use of the OpenCore TV80, a Verilog port of the z80. Our testing included verifying the working and timing accuracy of this model.

VGA Controller

For our design, we decided to skip the DVI controller entirely and make our own VGA controller. This was motivated by: our schedule not allowing for learning curve of Chrontel chip and DVI protocol; and the fact 1942 was an analog system itself. Building our own VGA controller involved making sure that we were sending the vertical and horizontal syncs at the right speed in order to make a 640x480 resolution at 60Hz. Therefore, we would be closer to the original hardware of the arcade machine, which actually uses scanline interrupts to pace many of the in-game events.

The VGA controller had a couple of counters to keep track of the sync signals and what row and column are currently being used. The outside interface required the

hardware using the controller to specify a color based on the current row and column location. The colors was given in 4-bit arrays which then were converted into analog colors by using the resistor diagram in the below picture.



To test the colors, we created a simple test screen which would output various colors to test how well the system would work. It tested both row and column changes, but it did have a few color bleeding issues which seem to be due to certain changes of colors. A picture of the given test screen can be found below.


DELL

Peripherals

Our joystick system was borrowed from the Multi-Williams team. Our improvements include cleaning up the wiring and documenting the pins. We had no need for most of the buttons and the 2-player co-op capability. It used a ribbon cable to connect to the 8-way joystick, 2-play buttons, 2-start buttons, and insert coin button. We created a custom graphic (see appendix) to use as an overlay on the controls to both enhance the visual experience and ease the small learning curve.

Sound Controller

Our sound controller intercepted the “sound codes” which the main CPU writes to the address 0xC800 and used them to decide what sound to play. We used our own recorded sounds from 1942 and 1943 (1942’s sequel), which we had put onto BROMs using the ISE Block Memory Generator. Our sound controller adopted Team Dragonforce’s code to interface with the AC97 and was modified to use our sound BROMs instead of sounds from flash. It selected which sound BROM to send the data from to the AC97.

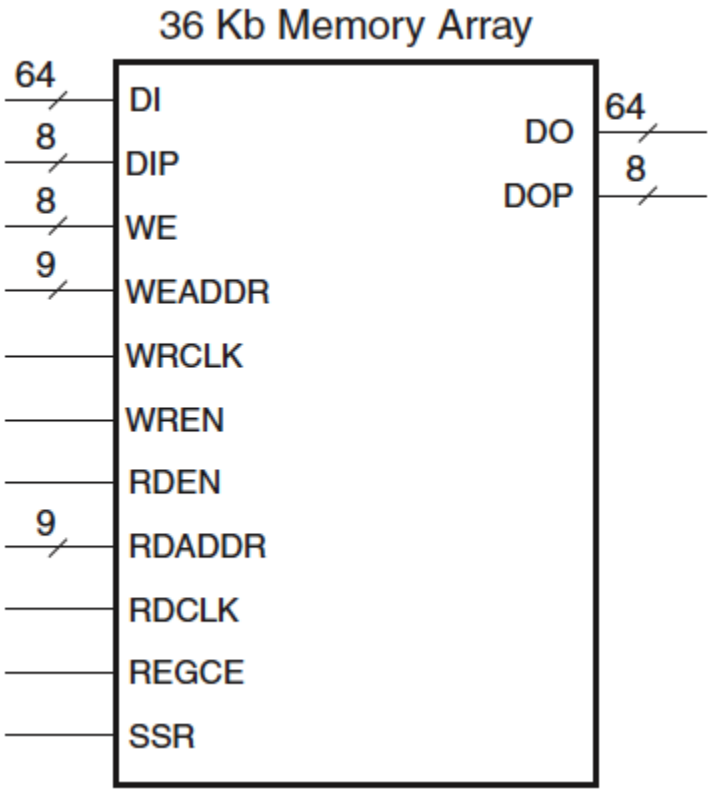
In order to fit all of our recorded sounds, we used two FPGAs, one of which had the foreground sounds, and one with the background sounds. A simple one-bit signal interfaced the two boards. This necessity for two boards came from the fact we ran out of block memory on one board alone. The background music played simultaneously with the foreground sounds, and its starting and stopping was triggered by two sound codes. This allows for a more complete audio experience.

Sound	**Sound code from Main CPU**
Fire	0x04
Flip	0x06
Explosion	0x02
Take-off	0x0D
Retry	0x12
Insert coin/power-up	0x07
Background music	Start (0x11), Stop (0x10)

BROM’s and BRAM’s

We chose to use the FPGA’s block memory to hold 1942’s RAM and ROM data. The ROM data was programmed onto the ROMs when the board was programmed, so that no setup needed to be done (loading flash data into BROM’s, etc.) at the beginning, and we were able to avoid using Flash memory.

Our system used 11 BROM’s, in addition to 7 BROM’s with our recorded sounds. Each BROM was a single-port ROM generated from the ISE Block Memory Generator. They were divided up so that each BROM does not need to be read more than once at a time, so that they could use single-port.

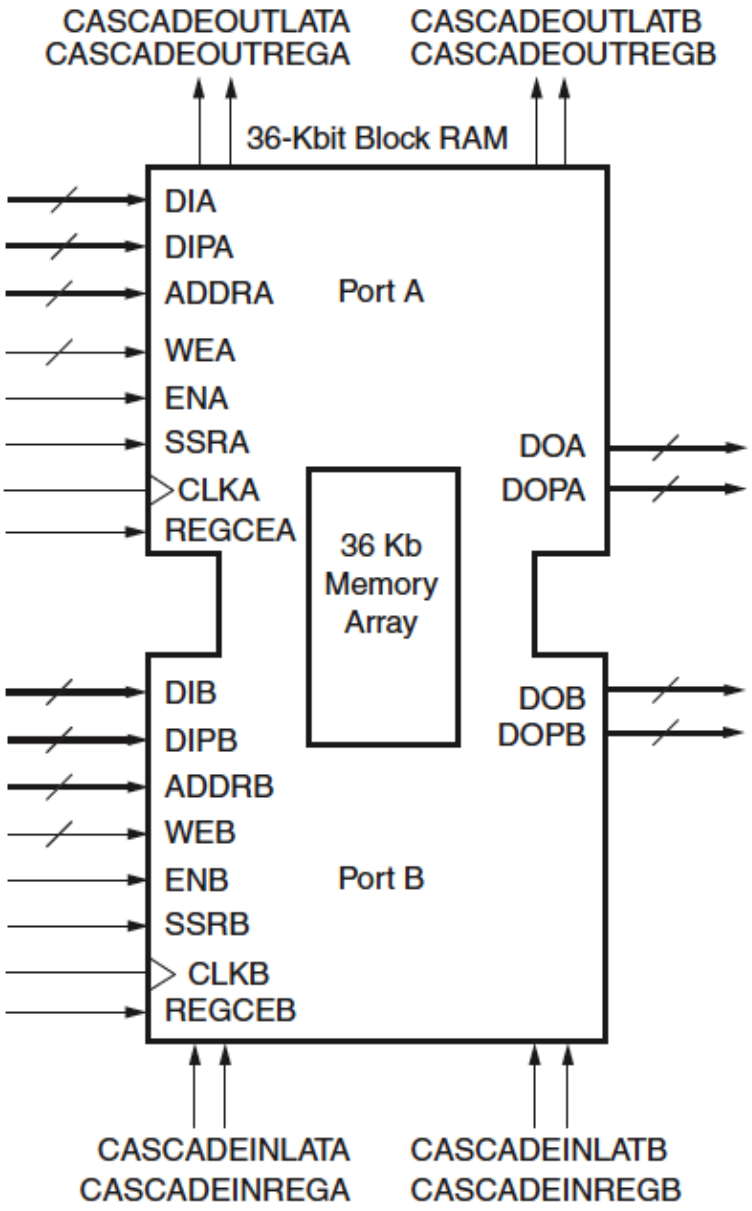


BROM’s

Name	Purpose	Word Size (bits)	Depth (# addressable locations)	Memory Range (hex)	Resources Used
maincpu	1942 main cpu game ROM	8	32768	00000 - 07FFF	8 x 36K BRAMs
maincpu bank1	1942 main cpu game ROM	8	16384	10000 - 13FFF	4 x 36K BRAMs
maincpu bank2	1942 main cpu game ROM	8	8192	14000 - 15FFF	2 x 36K BRAMs
maincpu bank3	1942 main cpu game ROM	8	16384	18000 - 1BFFF	4 x 36K BRAMs
gfx1	character tile object data	8	8192	00000 - 01FFF	2 x 36K BRAMs
gfx2_1	background tile object data 1/3	8	16384	00000 - 03FFF	4 x 36K BRAMs
gfx2_2	background tile object data 2/3	8	16384	04000 - 07FFF	4 x 36K BRAMs
gfx2_3	background tile object data 3/3	8	16384	08000 - 0BFFF	4 x 36K BRAMs
gfx3_1	sprite object data 1/2	8	16384	00000 - 07FFF	4 x 36K BRAMs
gfx3_2	sprite object data	8	32768	08000 - 0FFFF	8 x 36K BRAMs

	2/2				
palette	Computed from red, green, blue, tile, and sprite ROM’s and used instead	16	1536	00000 - 00BFF	1 x 36K BRAM

There were 4 BRAM’s in our system. The BRAM for maincpu used Single-Port BRAM, while the BRAM’s for fgvideo, bgvideo, and sprites used True Dual-Port. The layout of a true dual-port BRAM on the Virtex 5 is shown below. The design reason for the dual-ports was to eliminate the requirement for arbitration logic when both the main cpu and video hardware may have been making read/write requests simultaneously. Likely, the original 1942 system did not use dual-port RAM’s, but this simplifies our design, while leveraging the power provided by the Virtex-5.


CASCADEOUTLATA
CASCADEOUTREGA
CASCADEOUTLATB
CASCADEOUTREGB
36-Kbit Block RAM
DIA
DIPA
ADDRA
Port A
WEA
ENA
SSRA
CLKA
REGCEA
DOA
DOPA
36 Kb
Memory
Array
DIB
DIPB
ADDRB
WEB
ENB
Port B
SSRB
CLKB
REGCEB
DOB
DOPB
CASCADEINLATA
CASCADEINREGA
CASCADEINLATB
CASCADEINREGB

BRAM's

Name	Purpose	Ports	Word Size (bits)	Depth (# addressable locations)	Memory Range (hex)	Resources Used
maincpu	main cpu RAM	Single-Port	8	4096	E000 - EFFF	1 x 36K BRAM
fgvideo	foreground tilemap RAM	True Dual-Port	8	2048	D000 - D7FF	1 x 18K BRAM
bgvideo	background tilemap RAM	True Dual-Port	8	1024	D800 - DBFF	1 x 18K BRAM
sprite	sprite RAM	True Dual-Port	8	128	CC00 - CC7F	1 x 18K BRAM

Memory Map

Our memory map module mapped addresses from the CPU to the appropriate BRAM/BROM. It took an address from the CPU, a bank_switch signal, and outputted the appropriate data. See the BROM and BRAM tables above for the memory ranges of the CPU BRAM's and BROM's. The memory mapping module was also responsible for memory mapped IO to the peripherals. The interface to the cpu was like any other memory interface, but the memory module also interfaces with a number of external pins and memories through their respective interfaces in a read-only fashion (i.e. peripherals and ROM's). It also sent the sound codes from the main CPU to the Sound Controller, which played sounds using the AC97.

Sprite and Tilemap Hardware

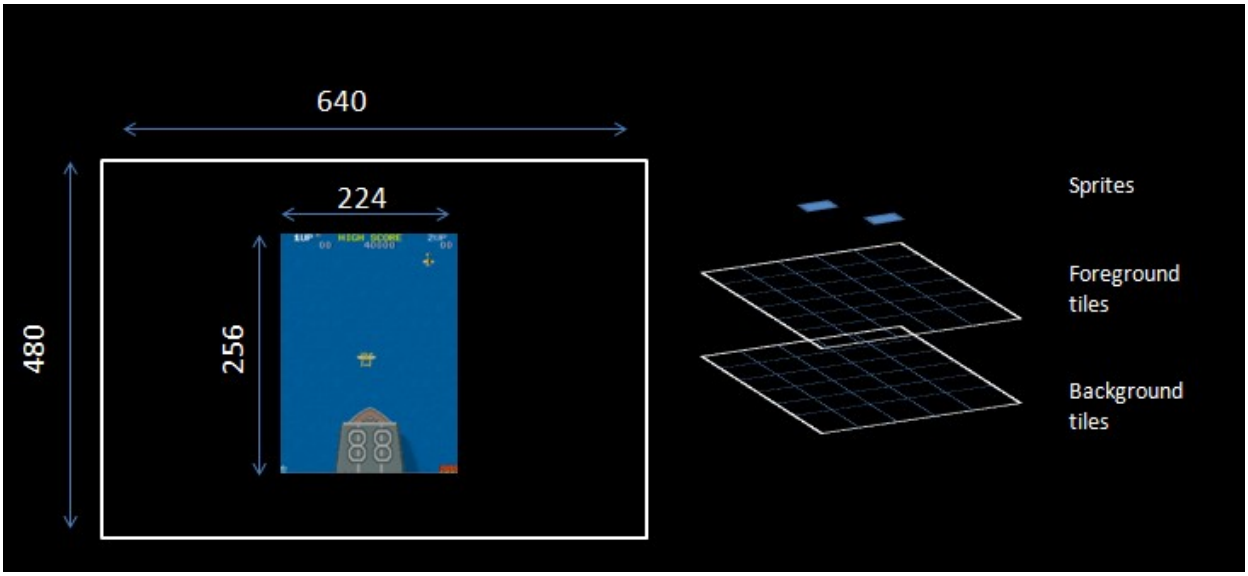
The pixel generation hardware consisted of sprites, tilemaps, color lookup tables, and color palettes. These together were used to reduce the memory bandwidth requirements in the arcade hardware, while providing decent graphics. Because of this memory constraint, 1942 did not have a framebuffer. Pixel colors were generated in near real-time with the CRT pixel gun position and fed directly to the VGA controller. MAME did not faithfully replicate this hardware, but provided the same end result. As such, our implementation of this hardware was in part our own

design and like MAME achieves the same result. Our design drew a compromise between the extra speed and bandwidth we could achieve on the FPGA and attempted to be faithful to the original hardware's implementation.

Screen

The screen is 224x256 pixels. It was made up of a 32x32 grid of 8x8 foreground tiles overlaid on a 16x32 grid of 16x16 background tiles. Therefore, the background tile grid was not always completely visible and allowed for the seamless vertical scrolling effect. Sprites were overlaid on top of these tilemaps, not restricted to a grid granularity. The two tilemaps--the foreground or character tilemap and the background tilemap were RAM's that the main CPU writes to. There was also a sprite RAM for holding the sprites on the screen.

There were 256 scanlines, each ended with an HSYNC. The timing for these scanlines were important for both the VGA controller and providing the timing interrupts to the main cpu to control game speed. The main cpu was interrupted on VSYNC (end of sending all scanlines for a given frame).



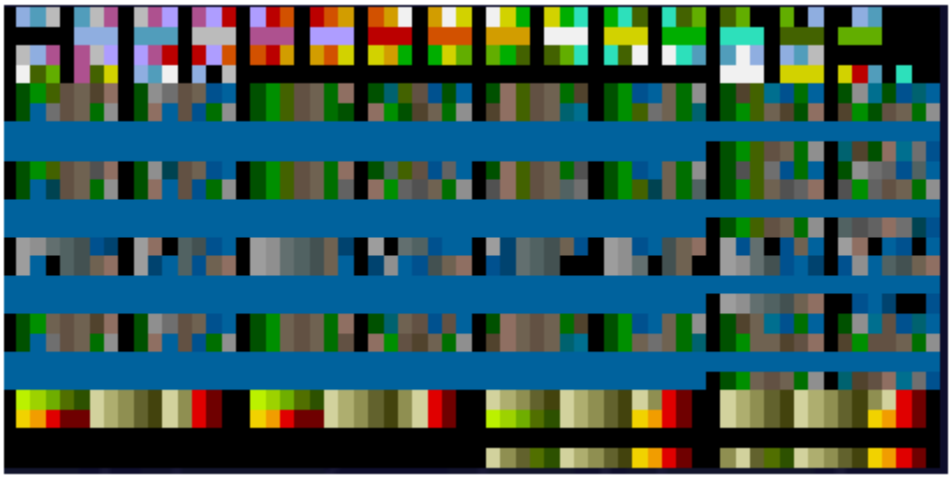
As seen in this diagram, the resolution of the game was such that it did not fill the lowest resolution available on our monitor. Doubling the resolution of the game couldn't easily be done because of the skewed aspect ratios.

Color

The VGA controller accepted three 4-bit colors (rgb red, green, and blue values). A palette provided the appropriate 12-bit value to the VGA controller, with the desired color being the index into this palette table. This reduced the number of bits required for specifying a color for objects, but restricted the available colors that could be displayed on the CRT. The palette is 1536 entries effectively. It was built using color lookup tables and color ROM's. It was built as follows.

1. Characters (foreground tiles) had 16 available unique colors. There was a color lookup table of 256 entries of 4-bits each that indexes into the color ROM's (entries 128-143). The resulting 256 entries of 12-bits were stored in the palette.
2. Background tiles had 64 unique colors available. There was a color lookup table of 256 entries of 4-bits each plus a 2-bit palette selector register that indexed into the color ROM's (entries 0-64). The resulting 1024 entries of 12-bits were stored in the palette.
3. Sprites had 16 unique available colors. There was a color lookup table of 256 entries of 4-bits each that indexed into the color ROM's (entries 64-79). The resulting 256 entries of 12-bits each were stored in the palette.

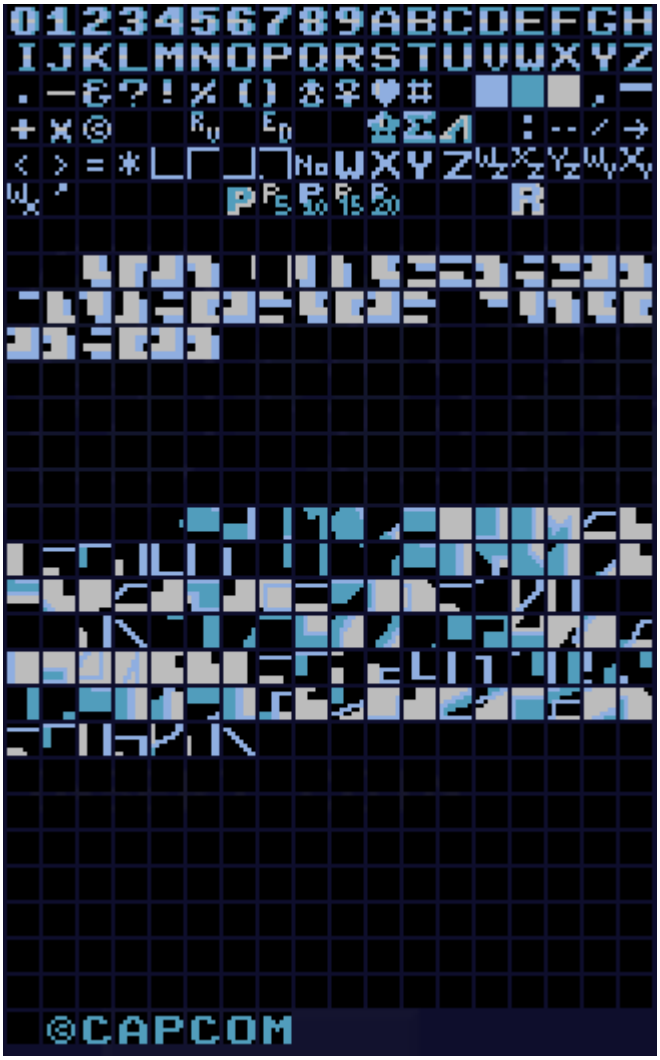
The complete palette can be seen below (generated using MAME). Notice the distinct regions of the palette (the largest being the background palette).



In the real 1942 system, the concept of a palette was really an abstraction built on the aforementioned color ROM's and color lookup tables. Our design decision was to implement this indirection once offline and generate the complete palette as I described and stick it in a large BROM. The Virtex-5 was more than capable of this and it simplifies our design (i.e. removed requirement for the color ROM's and color lookup table ROM's).

Character Tilemap

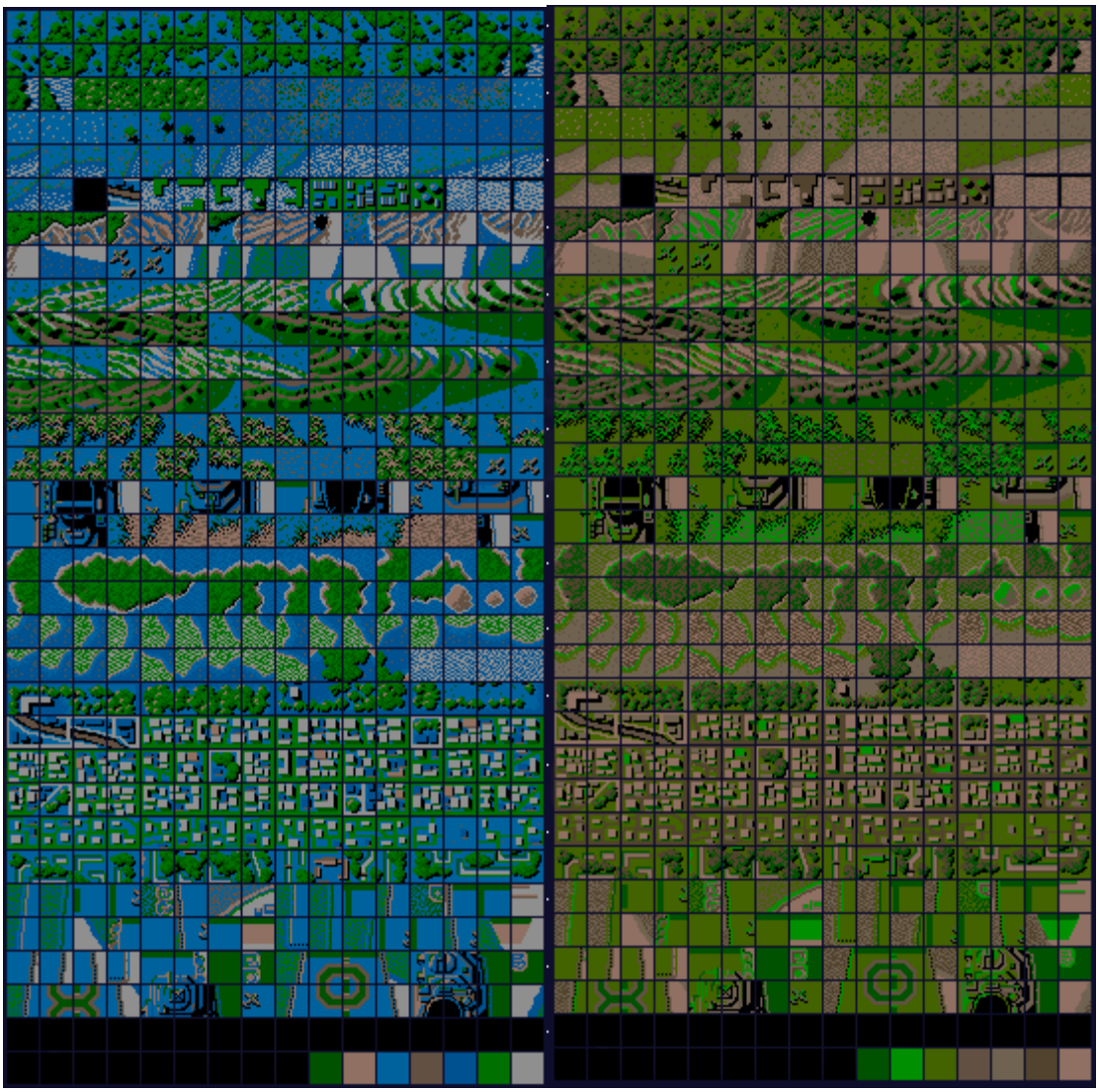
The character tilemap contained 32x32 2-byte tiles. One byte of each tile was the index into the foreground tile ROM, which contained the color offsets for each of the 64 pixels of the 8x8 tile. Foreground tiles were 8x8. The other byte in the tilemap contained the colorbase into the palette. Therefore, the colors used by the pixels of the tile could be controlled by the colorbase provided in this tilemap RAM.



The palette index was computed as the tile colorbase + tile pixel color offset. Pictured are the foreground tiles from 1942.

Background Tilemap

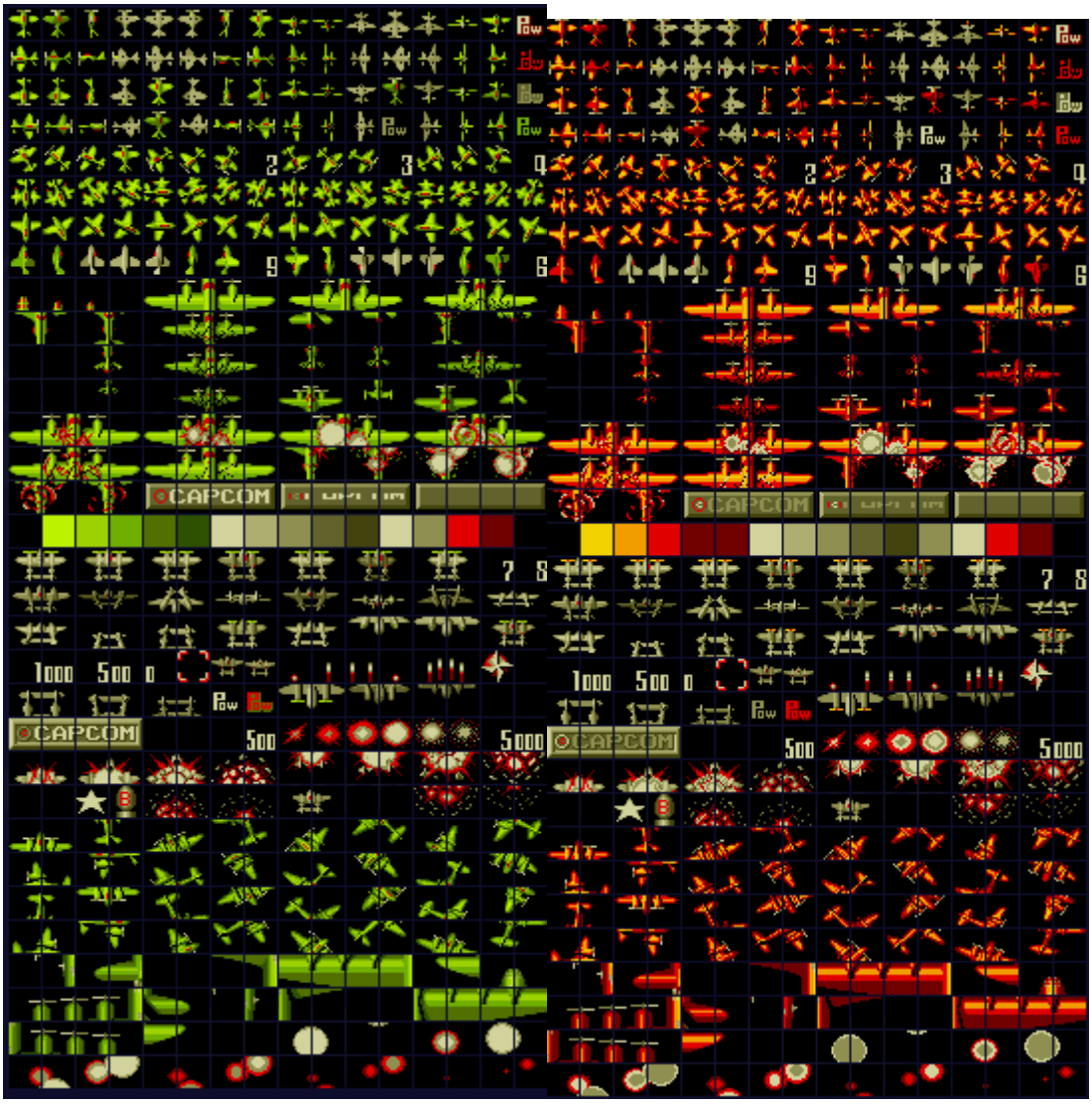
The background tilemap contained 16x32 2-byte tiles. One byte of each tile was the index into the background tile ROM, which contained the color offsets of each of the 256 pixels of the 16x16 tile. Background tiles are 16x16. The other byte in the tilemap contained the colorbase into the palette. Therefore, the colors used by the pixels of the tile could be controlled by the colorbase provided in this tilemap RAM. Also encoded in these 2 bytes was an x and y flip bit to allow for more variations of the tile with minimal storage overhead.



Pictured here are two complete renderings of the 1942 background tilemap. The pixel color offsets read from ROM obviously do not change, but the color can still be changed in RAM by altering the colorbase value. This is how the game at runtime can repurpose a tile from representing land (on the right) to water (on the left).

Sprites

Sprites are a little different since they require more freedom in positioning and change more frequently in position. A 128-byte sprite RAM contains 32 4-byte entries representing sprites currently visible on the screen. The four bytes contain the x coordinate, y coordinate, index into sprite ROM, and colorbase for the sprite. The sprite ROM provides color offset values for the 256 pixels that make up a particular 16x16 pixel sprite.



The complete sprites stored in ROM are shown here, rendered with two different colorbases, which is how the game achieves different color aircraft.

Pixel Rendering

Pixel rendering is accomplished with a set of relatively complex pipelines that read the required data, perform any necessary transformations, buffer the data momentarily, and finally decide on and provide a color to the VGA controller. There are four main pipelines working simultaneously.

Timing

The pixel rendering pipelines operate in the video clock domain. This clock is the same as the pixel clock of the CRT at this resolution so that the final pixel color generation pipeline can operate in lockstep with the VGA controller. This frequency is 25.175MHz.

Foreground pipeline

This pipeline produces 8 pixel colors every 8 clock cycles. As such, it begins 8 cycles before the first pixel needs rendering to the screen. The steps in this pipeline:

1. fetch foreground tile from RAM (2 sequential reads)
2. load tile from ROM (2 sequential reads)
3. set foreground linebuffer

Background pipeline

This pipeline produces 8 pixel colors every 8 clock cycles. As such, it begins 8 cycles before the pixel needs rendering to the screen. The steps in this pipeline:

1. fetch background tile from RAM (2 sequential reads)
2. load tile from ROM (3 simultaneous reads)
3. transform tile
4. set background linebuffer

In order to facilitate one-pixel granularity background tile scrolling, a buffer was made for the entirety of the current scanline. Therefore, it was able to then offset the start in this scanline and choose which pixel it would start at. This allowed for scrolling with ease as the background pipeline would be tricked several scanlines ahead to be able to line the background up correctly.

Sprite pipeline

This pipeline produces 224 pixel colors every scanline. Because this takes a considerable number of cycles, it operates one scanline ahead of the current pixel

rendering to screen. The steps in this pipeline are performed 32 times in building the linebuffer (once for each sprite in the sprite RAM). The steps in this pipeline:

1. fetch sprite from RAM (4 sequential reads)
2. evaluate y-coordinate
3. load sprite from ROM (4 sequential reads from 2 ROM's)
4. transform sprite
5. set sprite linebuffer

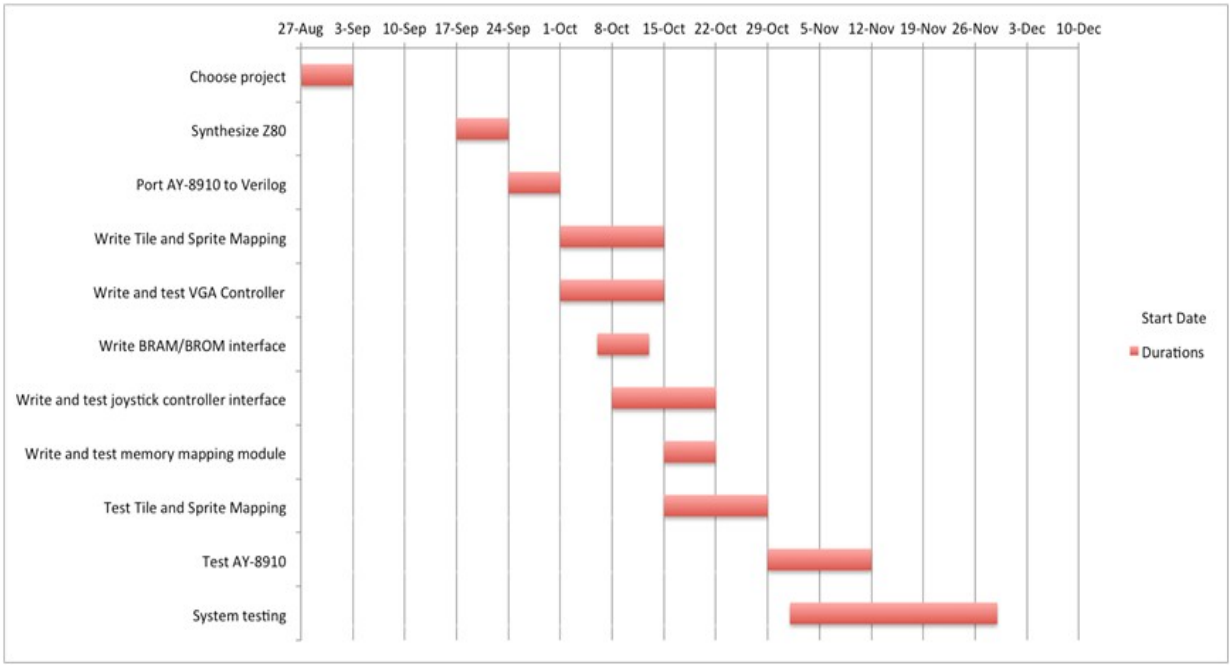
Priority logic evaluates at any given pixel coordinate from the VGA controller which sprite from the sprite linebuffer is relevant. To simplify this priority logic, only 8 sprites are allowed per scanline and the rest are dropped. There are therefore really 8 sprite linebuffers, each with an x register that holds the starting column at which it should be rendered. Sprites can overlap because they have a transparency color.

Pixel rendering pipeline

This pipeline produces 1 pixel color per clock. The output of this pipeline is a 12-bit color for VGA controller. The stages in this pipeline:

1. read/choose pixel color from (sprite || foreground || background) linebuffer
2. read palette color
3. output 12-bit color to VGA controller

Schedule



Our project, as of design review time, was on target with this Gantt chart. We kept an aggressive schedule in order to keep ourselves ahead of the late November, early December rush of assignments. We found that keeping an aggressive schedule worked out well. As of Thanksgiving break, we had a project that had a fully working game without sound, with a few minor graphical glitches, and without one-pixel granularity scrolling. From there we had enough time to focus on the details such as, scrolling, sounds, and most importantly arcade control artwork and CRT bezel artwork.

Methodology

Our team used git for version control. We maintained a Team Arcade wiki (dokuwiki) where we document 1942 and the workings of our implementation. This was also the home to our schedule and list of tasks.

Results

Our project was a success. It received positive feedback from students, faculty, and an Apple representative. The game was fully working with a great audio and visual experience. There are no known bugs. We believe our goal of implementing 1942 on an FPGA was met and although some of our initial visions changed, we believe the end result exceeded our initial expectations. The finished product was a great blend of sensory experience and technical merit.

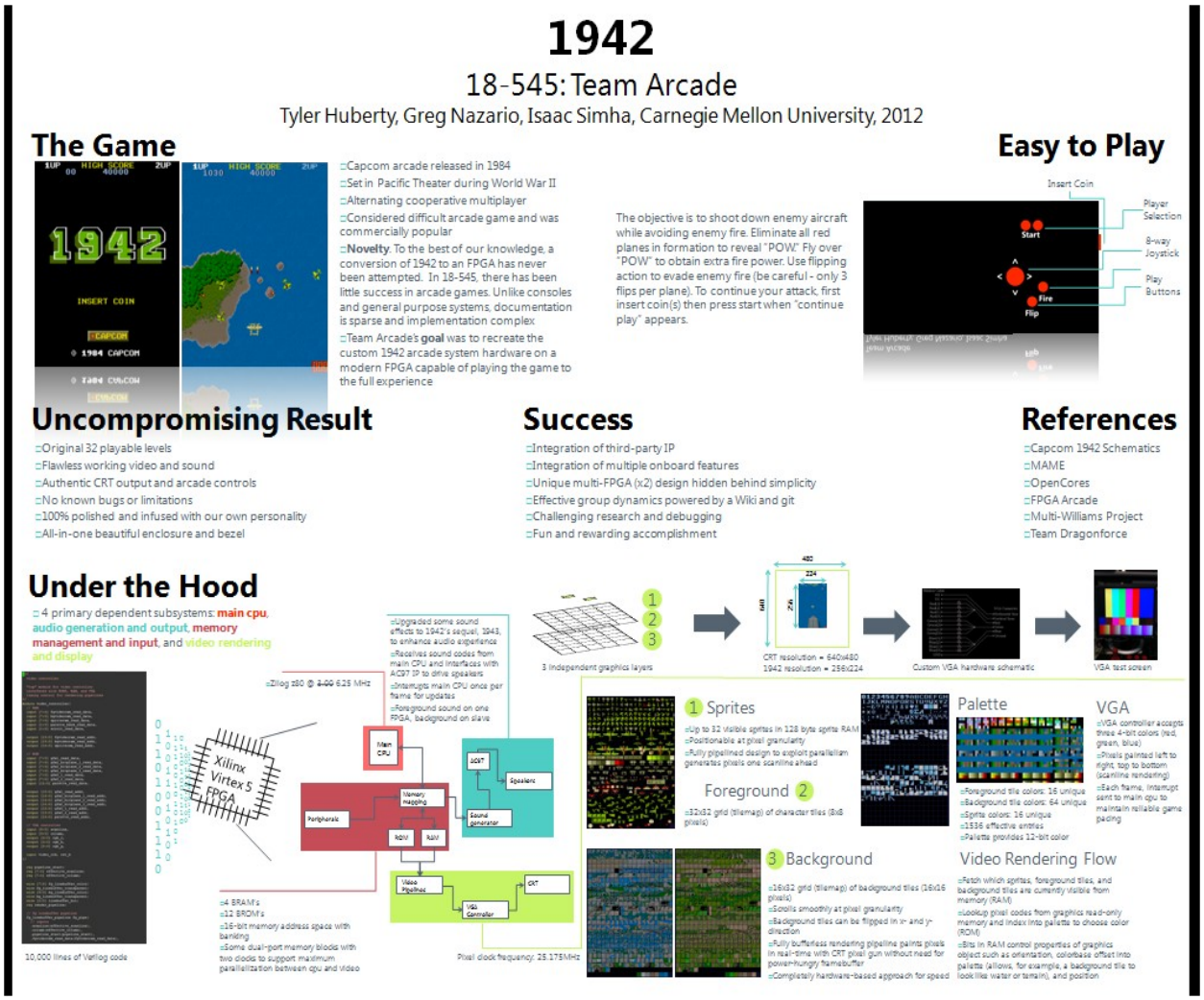
Sources

1. Multi-Williams Project Report (www.ece545.com)
2. MAME (www.mamedev.org)
3. Virtex-5 User Guide (http://www.xilinx.com/support/documentation/user_guides/ug190.pdf)
4. Wikipedia (http://commons.wikimedia.org/wiki/File:AY-3-8910_pinout.JPG)
5. OpenCores (http://opencores.org/project,tv80)
6. FPGA Arcade (http://www.fpgaarcade.com/scramble.htm)
7. 1942 Schematics

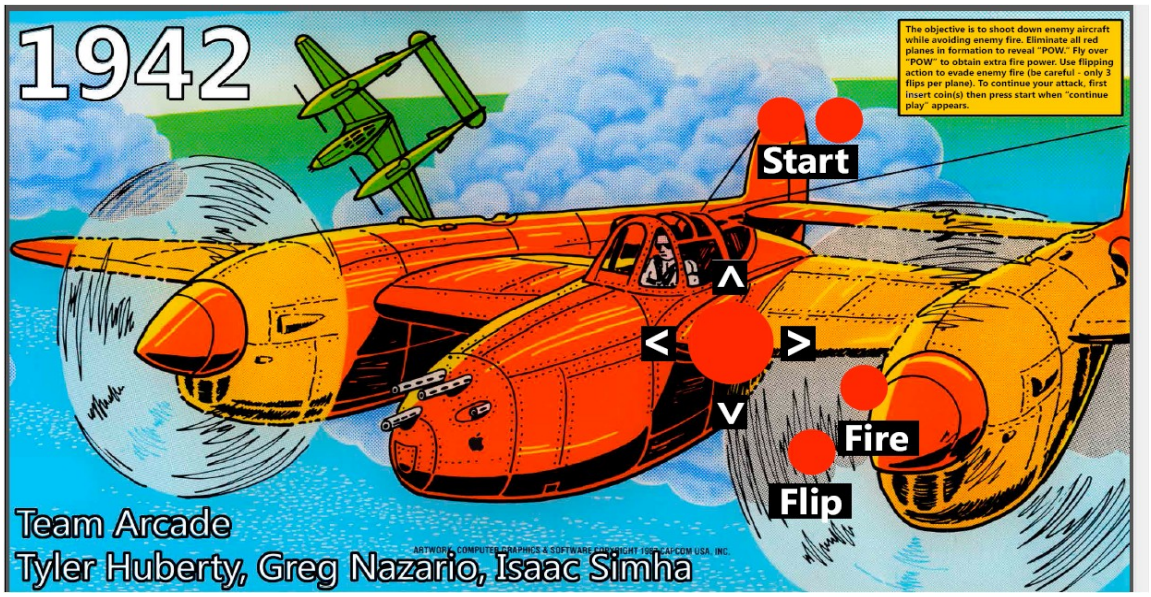
8. 1942 Game (CAPCOM)
9. 1943 Graphics and Sounds
10. Team Dragonforce (AC'97)

Appendix

Poster



Arcade controls overlay graphic



CRT bezel graphic

Personal Statements

Tyler Huberty

My initial efforts were focused on finding an arcade game of the appropriate challenge and scope for the course. I spent much time on the frontend to make sure we chose an exciting game and studied the MAME source code thoroughly to understand and assist our project as the main source of documentation. I was capable of editing and compiling MAME after much work and used this ability on several occasions for debugging (since testing changes in MAME was quicker than testing our hardware changes, I could “break” MAME in different ways to rapidly check if that introduced bug might be what we were seeing in hardware--an interesting type of debugging). I wrote the initial implementation of the foreground,

background, and sprite pipelines and subsequent rendering. I assisted with the debugging of these units as well as the debugging of the audio and memory mapping units. The marketing materials, poster, and graphics for our presentations and final demo were largely my work. I also hosted the wiki on my server for our internal documentation.

My average time spent on class including lectures, reading the course textbooks, and working the labs was 12-16hrs/wk. One key to our group's success was undoubtedly that the early weeks especially were closer to the higher extreme of this range. The extensive and smart research throughout and hard work on the right units at the right time to keep the project moving were other keys to our success. Finally, I had strong teammates with effective dynamics and collaboration.

This course was a fun, challenging, and rewarding experience. The most exciting accomplishment was using technical skills developed throughout CMU to create a polished, beautiful product that entertained many during demo. Having industry sponsorship from Apple was a plus as well.

In picking a project, my advice would be to read all the previous project reports to learn from the success and be critical of the causes of failure. While it's important to choose a project with appropriate scope in order to have a polished product by demo time, problems previous groups encountered (such as the CRT timings of the Multi-Williams group) may be obstacles your group can overcome with early warning and mindfulness to the issue.

Gregory Nazario

I was the one who was the Linux guru and figured out first how to get all the tools working on the machine and how to use the toolchain. It seemed to have a lot of issues with drivers, simulation, and synthesis problems with 32-bit and 64-bit versions of the software. Many times, we had to switch between one or the other due to the fact that it would say place and route successful, and then place and route failed immediately afterwards. Also, the machine would not recognize the programming wire without using 64-bit drivers.

I spent time building the external devices for the system. In the first few weeks, I had made a VGA breakout board and a VGA test screen in order to show that it was possible to output to the screen without using the DVI controller. Also, the controls which we borrowed from the Multi-williams had to be rewired and adapted to the new system and reconfigured to adapt to the new system. The acquisition of a CRT monitor was also something I went out of my way to do. It allowed for a more authentic gameplay experience as well as the ability to move the graphics around the screen to the appropriate place.

I also, spent a lot of time working on debugging and integration of systems. I don't know what I could estimate my approximate time, but I spent about 8-14 hours a week outside of class, where I'd just sit down and try to work on debugging or adding whatever feature of the week. Usually, 4 hours was with my team on Saturday afternoons. I spent more time working on the project at the beginning of the semester in order to meet our goal of having the game work mostly well before thanksgiving break.

I started working on some graphical bugs and lining up the system that Tyler had made with the VGA output, and eventually worked on interrupts in the CPU which allowed the game to finally progress, as it is interrupt driven. The Multi-williams team before said that it was very hard to imitate these video interrupts from the pixel gun, but we found no problems with it. From there, I fixed some problems with sprites, background tiles, and foreground tiles with some simulation and trial and error. The final graphical bug was the scrolling to make sure that it scrolled at 1 pixel granularity.

Finally, I worked on having sound working as both background and foreground sounds. I had previously had bad experience of using the flash memory from the lab. However, it seemed that it was not the flash memory that was inconsistent, but instead it was the AC'97 interface. I spent time getting the two boards to work in tandem to create the sounds for the game.

Overall, I thought the class was a great experience, and my team worked great together on the project. It seemed that there needed to be a clearer way to program things onto the board, as sounds for some reason would not work correct when programmed from the onboard flash PROM, but everything else would work. Also, either finding tools that always work on the 64-bit version correctly or just installing the 32-bit version of linux on the machines to avoid the 64-bit - 32-bit problems would be a good choice.

Isaac Simha

When we started this project, we wanted to try something not attempted before that would be challenging yet feasible to do over a semester. Once we decided on 1942, I Initially worked on the memory mapping and CPU. We decided early that we would take the Z80 from OpenCores, and my task began to test it and look for any significant issues in the implementation, of which we found few.

For the memory mapping, the MAME documentation for 1942 provided a lot of hints. We decided after doing lab 2(the sound lab), that flash was too inconsistent, and I found that using the ISE Block Memory Generator to create block memory provided the best alternative. Then it was mostly a matter of figuring out what BRAMs were needed, and which ones needed to be dual-port vs. single-port (Video controller might need to read while CPU writes, etc.) I created about 20 BRAMs/BROMs initially (we had originally planned to write the AY-8910 for sound, which also needed BRAM), and I soon realized that simply waiting for them to synthesize could be a significant portion of time needed for this project (once we had a lot of BROMs/BRAMs in our system, synthesizing could take up to 20 minutes, which made testing using trial and error inconvenient to say the least).

One of the less documented things I spent time on was how the video interrupts were supposed to work, and the timing issues involved. The interrupt comes from an instruction we send to the Z80 that tells the Z80 to jump to the video interrupt handler. Although this seemed simple, figuring out when to assert and de-assert the signal lines proved to be a bit challenging since following the diagrams in the user guide did not work, and simulating the Z80 until it got to an interrupt was never a fast process. Interrupt issues were the last major hurdle in making the game run, and after this point, I focused more on sound.

Once we decided to not use the AY8910 and instead use the AC97, I started modifying Team Dragonforce's code to use BRAM instead of flash, and recording sounds that would be triggered by the various sound codes generated from the CPU. Using the AC97 ended up allowing us to avoid incorporating a system bus and other timing issues, and it even allowed us to make the game sound better; I believe this was one of our best design decisions.

Time I spent on this class varied from week to week, usually about 10 hours, but because of our relatively early success with the project, I managed to avoid a significant increase in hours put into the final weeks that many other teams had. I feel that I have learned a lot from completing this project, especially about research and going through documentation, and putting an entire system together. I had lots of fun with the class, especially once I started seeing results from our hard work, and I am very happy with Team Arcade's final product.

Acknowledgements

Team Arcade would like to thank Professor Nace, the ECE department, and Apple for their support and generosity throughout the project. We would also like to voice our appreciation to those that gave us advice or attended our public demo.