Final Report - Asteroids EDA385 Group 3

Mattias Ahlstroem, ada09mah@student.lu.se

Sebastian Fabian, ada09sfa@student.lu.se

Hawoay Tao, ada09hta@student.lu.se

Zixiao Zhao, int14zz1@student.lu.se

Instructor: Flavius Gruian

October 26, 2014

Abstract

The classic arcade game 'Asteroids' is implemented on a Nexys-3 FPGA development board with custom hardware and software. This was done by creating a VGA simulator for an x86 PC so that game software and hardware could be developed in parallel. A vector based graphics controller was created for performance and style in keeping with the original game. A hardware based two channel sound synthesizer was also added. The main challenge of this project was fittng both hardware and software within the given constraints. The final result is a functioning game device with well-balanced, challenging and fun gameplay. The game features high resolution graphics, smooth 60 frames per second video, background music and sound effects.

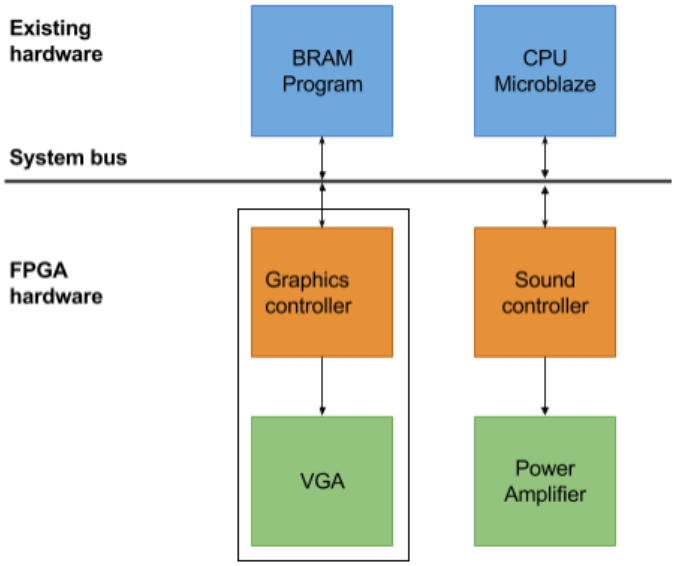
Contents

1 Introduction

The objective of this project is to implement a clone of the classic arcade game Asteroids. For those unfamiliar with the game, it's a vector based space shooter taking place on a single screen. The player controls the yaw and thrust of a small space ship and attempts to avoid and blow up asteroids by shooting at them with a projectile weapon. When an asteroid is hit, it's divided into smaller rocks and eventually disappear when shot a certain number of times.

We chose to implement this game because it gave us a great opportunity to experiment with a hardware graphics controller that is vector based, providing an interesting hardware design challenge. Because of the nature of a simple game such as this, we were also able to scale the project up or down, which we did by implementing a two channel sound synthesizer. Overall, the project was a challenge from both a hardware and a software point of view.

Figure 1: Architecture of the hardware



2 System Description

2.1 Hardware

2.1.1 FPGA utilization

Below is an overview of how much of the FPGA is being used by the various components.

As can be seen, a substantial amount of the FPGA is used merely for the AXI bus. This is because the VGA controller is clocked at 154MHz to handle the high resolution output, meaning it's running at a different clock speed than the rest of the system. To accomodate this, register slices have been added

Figure 2: FPGA utilization

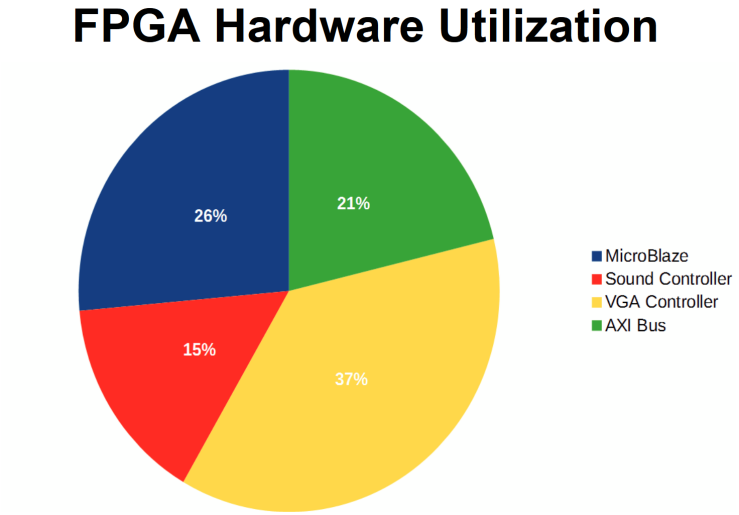


Figure 3: Slice Logic Utilization

	Used	Available	Utilization
Number of Slice Registers	5,621	18,224	30%
Number of Slice LUTs	6,511	9,112	71%
Number of occupied Slices	2,232	2,278	97%
Number of RAMB16BWERs	30	32	93%

to the AXI bus that buffer values being transfered to and from the graphics controller.

The following two sections (VGA controller and Sound synthesizer) detail hardware IPs that we implemented ourselves.

2.1.2 VGA controller

All graphics are handled by the VGA controller on the AXI bus that receives image data (from the Microblaze CPU and our program) in the form of lines. The data format is four 13-bit integers, which grouped by two gives the coordinates for each line, plus 8 bits of color information per line. No frame buffer is used; instead the vector data is kept in memory and the controller calculates which horizontal pixels to draw for each scanline.

The resolution of the VGA controller was set up to 1920x1200, which is the native resolution of the monitors available in the computer labs at LTH. Precise timing information was recovered from the EDID data of the monitor, this had to be done offline since the FPGA does not have connections for those pins. The resolution and timing parameters are easily configurable in the HDL but cannot be reprogrammed without rebuilding the hardware.

The vector unit of the VGA controller consists of a configurable number of computational units which implement the Bresenham line algorithm[1] in a

[1] http://xlinux.nist.gov/dads//HTML/bresenham.html

combinatorial circuit. This circuit is connected to a shift register in such a way that the contents of the shift register rotate through the computational unit, updating the state of the algorithm at each pass. All lines are sorted in software such that the starting point lies above the end point on the screen and the algorithm does not advance unless the current Y coordinate is less or equal to the current scanline plus one.

When a computational unit signals that a pixel is to be rasterized, which it does when the current Y coordinate is exactly equal to the current scanline plus one, that pixel is written to a block RAM resource representing the next row of pixels to be displayed on the screen. This pixel RAM is double buffered such that the VGA output is driven by one set of block RAM resources while the vector unit is drawing to another, swapping these two sets at the end of each scanline. Each block RAM unit is shared between two computational units to make efficient use of dual port memory.

During the vblank period the computational units are inactive and their shift registers daisy-chained to produce one long shift register. This shift register is in turn driven by a component called the line programmer which calculates the starting state of the line algorithm. A very simple AXI bus controller waits for two 32-bit writes to arrive, decodes the start- and end-points of a line and then triggers the line programmer. The line programmer takes a certain number of cycles to complete, during which the VGA controller does not accept any write operations. Reading from the VGA controller either returns all zeroes, indicating that the VGA controller is busy and will ignore any writes, or all ones, indicating that the vblank period is in progress and that the controller is ready for programming.

In the final configuration, our implementation has 6 computational units (BRAM resources support a maximum of 8), a shift register depth of 32 for a total of 192 lines (shift register resources cannot support 8 * 32 = 256 lines) and the line programmer takes 5 cycles to compute the starting state for a single line.

2.1.3 Sound synthesizer

A two-channel synthesizer produces the sound for our game, which is then fed into a PmodAMP1 headphone/line amplifier[2]. The sound synthesizer IP consists of three parts. Two identical snd_controller cores which generate the waveform, and a mixer IP that mixes the two channels. All three cores are connected to the AXI bus and all communication between them and the CPU is handled directly on the AXI bus. The snd_controller cores support three types of waveforms; square wave, sawtooth wave and noise. Of these only the square wave is used to create basic sound effects as well as music in our game.

The waveform generators output 12-bit PCM to the mixer component which adds two channels together adjusted by a right shift, programmable independently for each channel as a crude form of volume control. The mixer component then converts the resulting PCM data to analog signal using pulse width modulation. The PmodAMP1 module already contains a band-pass filter and PWM produces a satisfactory result without additional tweaking.

12-bit PCM was chosen because it gives us a sample rate of approximately

[2]http://www.digilentinc.com/Products/Detail.cfm?Prod=PMOD-AMP1

24kHz if the PWM module runs for 4096 cycles per sample, which in turn means that the PCM value is the exact number of cycles the output signal should stay high for each sample. Combined with the basic volume control this makes the mixer an incredibly simple component. Writing to the mixer simply updates the volume control registers, reading from it is a no-op and always returns all zeroes.

The waveform generators are programmed using 32-bit commands consisting of the period (for square waves, other waveforms use other encodings) in samples, duration, also in samples and an identifier specifying which type of waveform to use. A command remains active for the number of samples specified in the duration field, after that the waveform generator returns to idle. A 16-level FIFO sits between the AXI interface and the waveform generator, allowing the CPU to queue up commands to be executed in sequence. Reading from the waveform generator returns all zeroes if the queue is full or all ones if there is space left in the queue. Writes are accepted even when the queue is full, allowing the CPU to effectively reset the queue by writing 16 no-op commands.

Due to the fact that periods are specified as an integer number of samples, reproducing exact frequencies gets less and less accurate the higher you go. Complete meltdown occurs around the notes A6 and A#6, where there is no longer enough accuracy to even give each separate note its own distinct frequency.

2.2 Software

2.2.1 Overview

The game logic itself is written completely in software. Its responsibilities are handling player input, advancing the game frame by frame and output the resulting graphics to the VGA controller. It also feeds buffer of the sound synthesizer. This is all done using single thread, in a single main loop.

The software consists of a main loop. It starts off by reading the input of the buttons on the board. No interrupts are used for this, just a simple read of the appropriate registers. If any of them are pressed, the corresponding action is triggered, such as rotating the ship or engaging the throttle. Then, a game tick is performed. This moves all game objects the appropriate amount for one frame and runs hit detection.

The game objects are represented by structs with a variety of members. One of these is the polygon member containing a single polygon for the graphic of this game object. At the end of each game tick, this graphic is first copied, then scaled, rotated and moved as needed. Finally, at the end of the main loop, all of the polygons are sent as lines to the VGA controller. If the sound buffer is not full, sound effect and music notes are fed into the sound synthesizer. A blocking call to the VGA controller blocks the thread until it's time to update the monitor, after which software execution is resumed.

2.2.2 Memory usage

The microblaze CPU uses 75% of the available block RAM resources for a total of 48kb of addressable memory. This maximizes the amount of space available for the software while still leaving enough for the VGA controller to work reasonably

well. The final executable is slightly larger than 45kb, 25kb of which is code. The default heap allocator has been replaced with a simple pool-based allocator to allow for resetting without having to keep a clean copy of the data section in memory. The heap size is 8kb, another 8kb is data and the rest is miscellaneous statically allocated memory.

The software part of the project is fairly memory intensive. It supports 128 game objects, each with a polygon of unlimited size (although in practice this never exceeds 9 lines). Each polygon is also copied and transformed for each frame.

3 Problems Encountered

One of the first problems we ran into when trying to integrate the system was running out of memory resources. This was a result of the fact that our software was coded as intended to run on a PC. We had to figure out various workarounds and ways to improve code efficiency to bring down software size. Another thing that helped was turning on size optimization in the compiler. An example of an optimization we did was replace the standard malloc with another with much lower memory footprint. Other imports, like the math library, could be removed and replaced with our own implementation of sin() and cos() which saved a significant amount of space.

Another big issue we had throughout the project was software bugs, and especially memory corruption bugs due to the incorrect use of pointers, that would eventually cause the system to crash. These types of errors are notoriously hard to troubleshoot, and some careful code review was required before we managed to squash all bugs.

Other problems include having to name VHDL source files with certain file names to be able to import them into XPS, and other minor issues with the environment. Annoying, but they were overcome quickly.

4 Lessons Learned

Always be careful when dealing with memory in C! It can't be stressed enough that going slow and coding carefully will take a few minutes extra, but save you time in the end.

The AXI bus is not only fast, but convenient to use and easy to get started with. We don't see why you would want to use an FSL in its stead.

Another thing that surprised us was the fact that we were able to crank up the graphical effects and game objects without slowdown on the 100 MHz Microblaze core. The final version has up to 24 asteroids on screen at the same time, plus an unlimited amount of shots, running in high resolution at a buttery smooth 60 frame per second, and in reality we never noticed any problems with frame skipping or slowdown of any kind.

Shift registers are incredibly useful in hardware design, the very first iteration of the VGA controller used a naive software-like approach of addressable memory which turned out to be so resource-intensive we could not even support 16 lines on the screen at any given time.

5 Installation

The compiled version of this project can be downloaded as a file 'download.bit'. This file contains the entire software and hardware configuration. To flash this file, connect a Nexys-3 development board to a computer and download using the Digilent Adept tool. Next, connect the PmodAMP1 amplifier into the JA1 port of the board (the upper row of the connector) and connect a speaker or a pair of headphones to the appropriate output. Finally, connect a VGA cable between the board and a monitor with support for a resolution of 1920x1200 and a refresh rate of 60 Hz (timing data was recovered from a Dell U2412M, other monitors may or may not work). The game can now be reset using the middle button and played using the other buttons.

The controls are as follows:

- UP - Throttle
- DOWN - Fire
- LEFT/RIGHT - Rotate yaw

The DIP switches can be used to adjust the following parameters:

- SW0 - Display red border around screen area
- SW1 - God mode (invincible and destroy asteroids on collision)
- SW2 - 70s mode (psychedelic colors that change depending on the horizontal position of a vector)
- SW3 - Instant victory
- SW4 - Disable intro animation
- SW5 - Disable randomness (use fixed seed instead of random seed based on vblank timing/clock synchronization)
- SW6 - No effect
- SW7 - No effect

6 Credits

- VGA & Sound Controller - Mattias "Aali" Ahlström
- Gameplay & Testing - Hawoay Tao
- Game engine & Vector drawing program - Sebastian Fabian
- SFX Composition - Mattias "Aali" Ahlström, Hawoay Tao, Sebastian Fabian and Zixiao Zhao

7 References

- The PmodAMP1 amplifier http://www.digilentinc.com/Products/Detail.cfm?Prod=PMOD-AMP1
- Bresenham's line algorithm http://xlinux.nist.gov/dads//HTML/bresenham.html

8 Pictures of the game and system

Figure 4: Opening screen

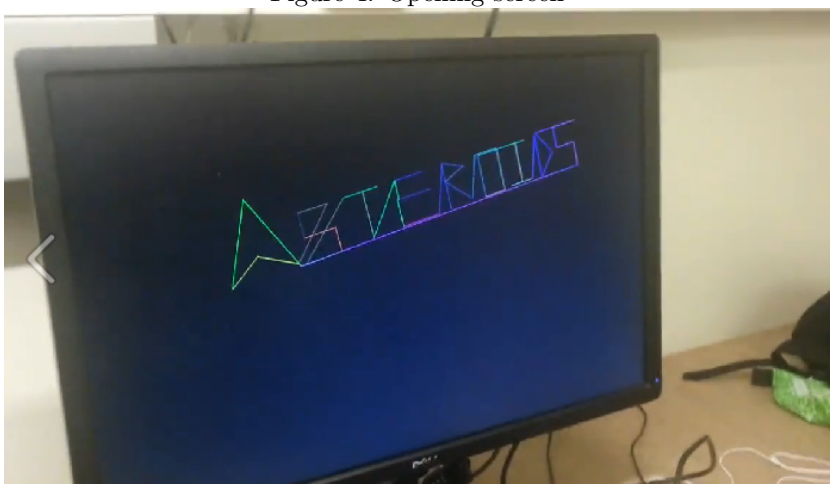


Figure 5: The game starts and shots are fired

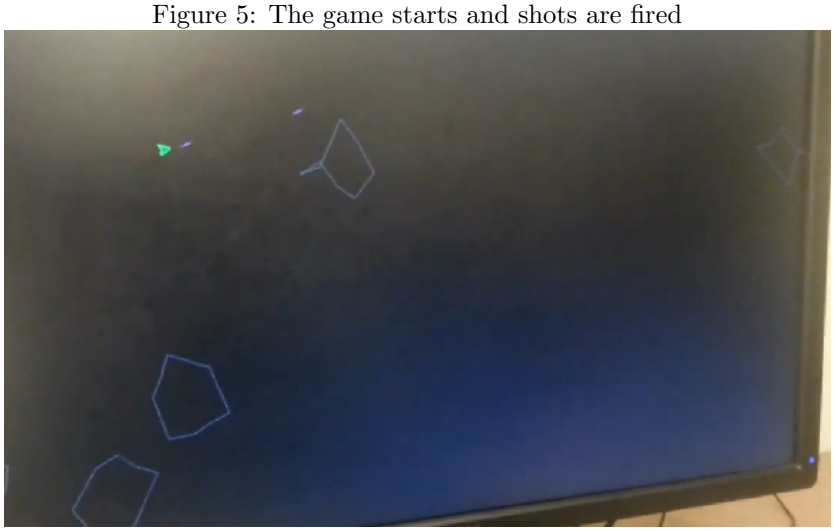


Figure 6: When astroids are hit they are divided into smaller pieces

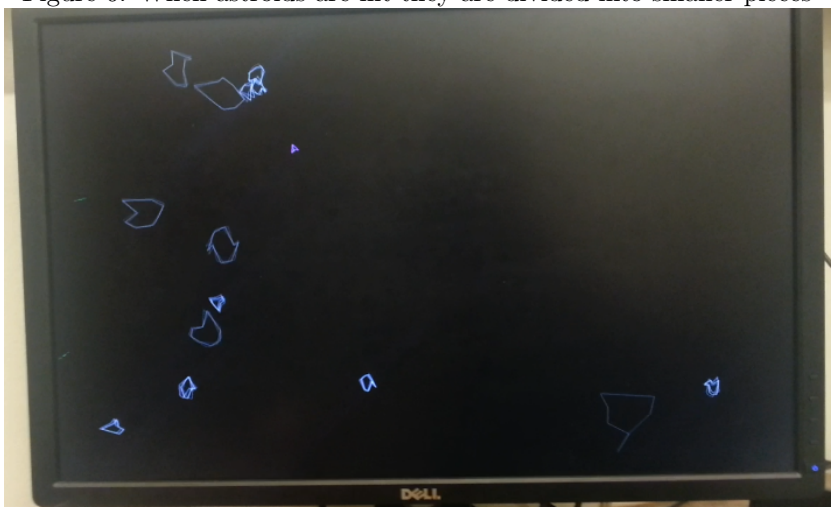


Figure 7: 70s mode activated

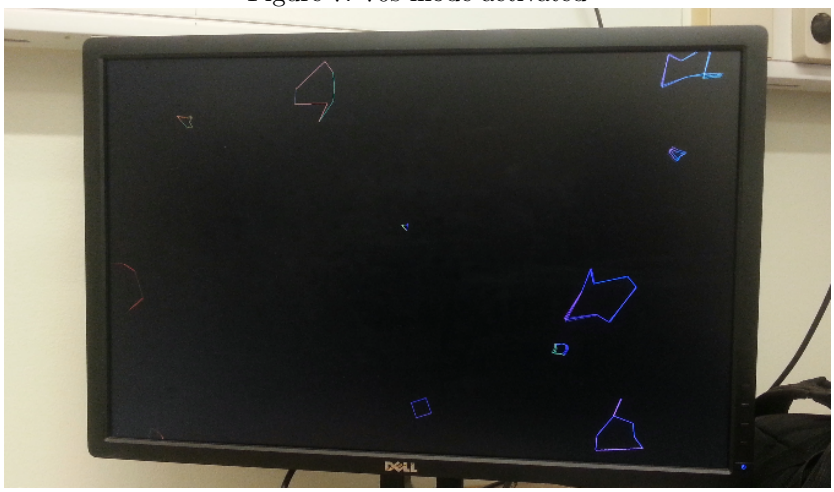


Figure 8: Memory corruption bug caused errors like this, fixed in final version

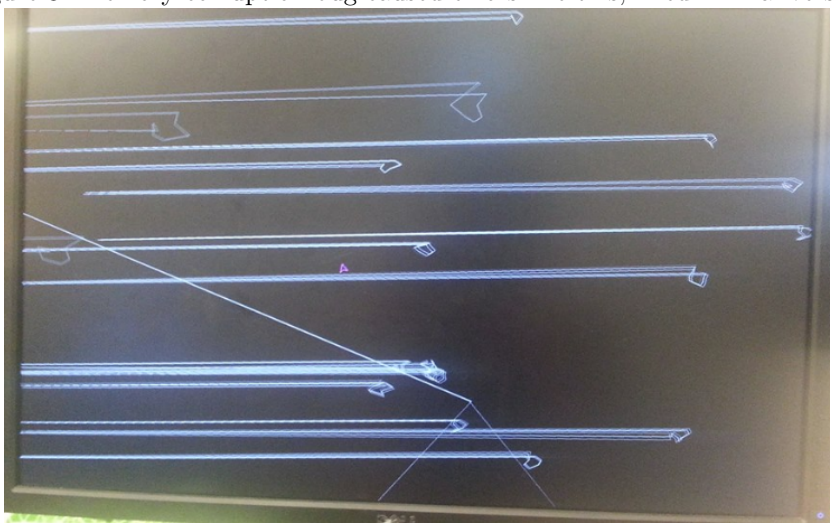


Figure 9: What the board looks like when everything is connected

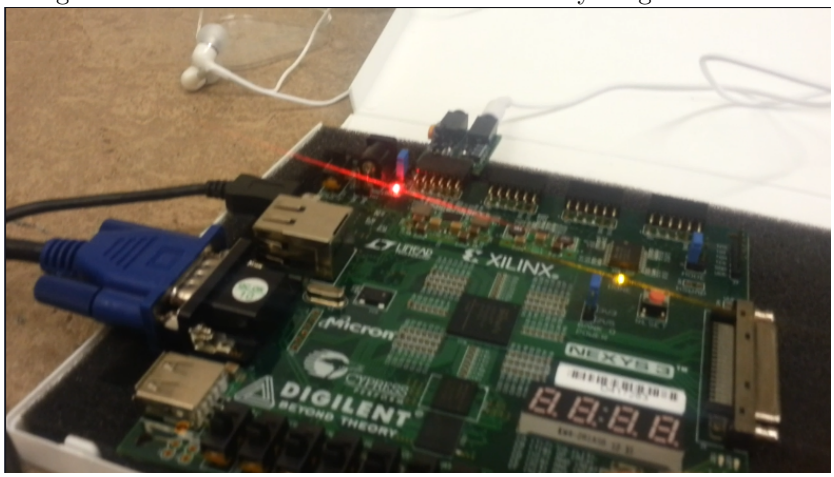


Figure 10: The credits screen after you win