Lund University
Faculty of Engineering

Embedded Systems - Advanced Course
(EDA385)

Space Invaders

Submitted By:
Axel Andersson
John Gustavsson
Victor Skarler

Submitted To:
Flavius Gruian

November 6, 2015

Contents

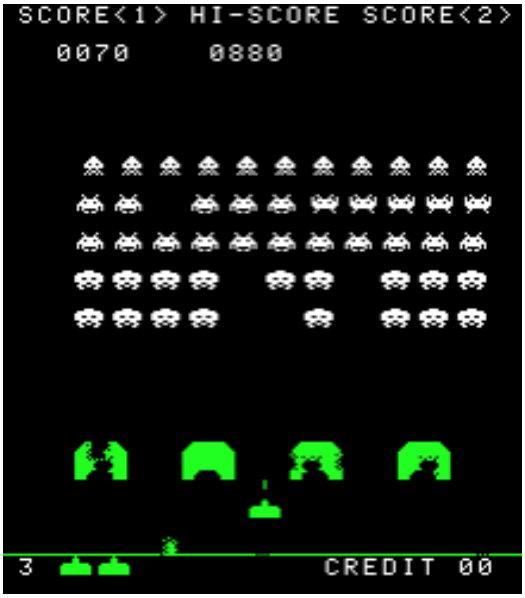

Figure 1: Screenshot of the original game. [1]

1 Introduction

The purpose of this project was to implement the arcade game Space Invaders from 1978 on a FPGA (Field-Programmable Gate Array). The idea behind the game is very simple. Figure 1 is a screenshot of the original game. The player controls the ship in the bottom of the picture. It is possible to move horizontally and the goal is to shoot all the aliens above. The aliens can also shoot bullets, which the player has to dodge or take cover from behind the green walls.

The implementation was done by designing a system containing both software and hardware parts. Figure 2 displays an overview of the system. Two custom hardware cores with AXI bus (Advanced eXtensible Interface) interfaces had to be implemented. A GPU (Graphics Processing Unit) to display the game on a VGA (Video Graphic Array) monitor and a audio core to play sounds. Two different system buses had to be used as well, AXI bus and PLB (Processor Local Bus). The reason behind this is that the PS/2 controller could only be connected to the PLB. Therefore a bridge between the buses had to be used to connect the devices on the AXI bus to

the keyboard. The software implemented on the CPU (Central Processing Unit) was used to control the game functionalities. These functionalities included moving all sprites and handling collisions between the bullets and the player or the aliens.

The differences between the implemented and the proposed systems are quite big. Originally it was assumed that the keyboard could be connected directly to the AXI bus. There were also no plans to implement audio support. This was added since the rest of the system was implemented quite fast. Text support on the GPU was also added, which means that the software developer can print text wherever is needed.

2 Implementation

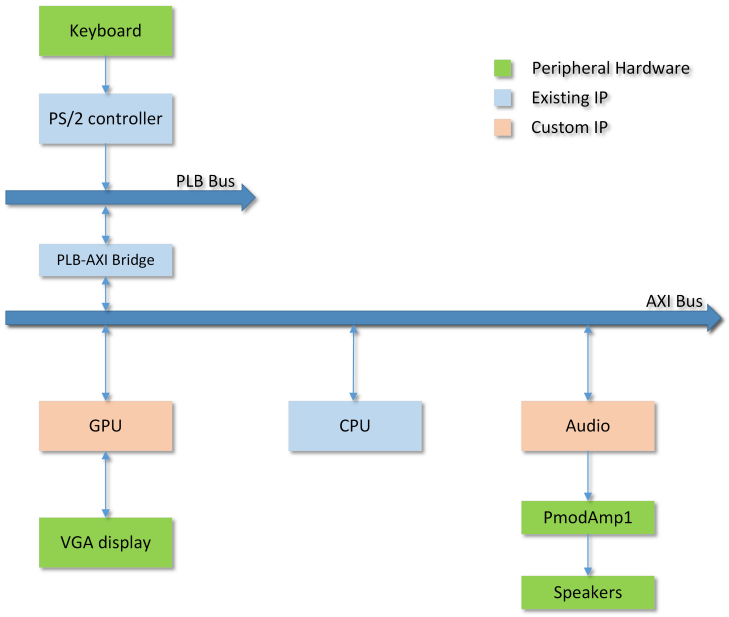

Figure 2: Block diagram.

2.1 Hardware

2.1.1 GPU

The GPU is the unit that takes game info from software through the AXI bus and generates a picture that is sent to the VGA display. The GPU itself is divided into different components as seen in Figure 3.

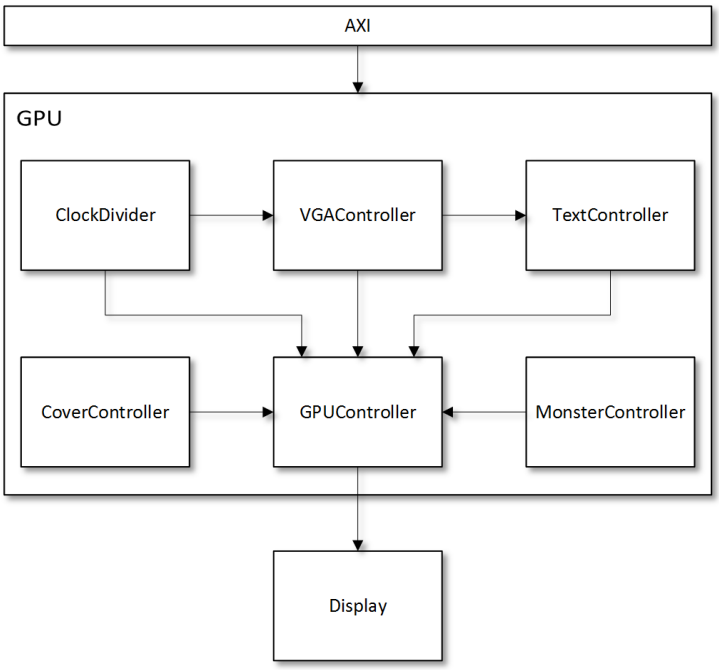

Figure 3: GPU.

The VGA controller is designed by Ulrich Zoltán from Diligent and gives out two signals that say which pixel the system are on (*hCount*, *vCount*). [2] It also produces a signal when a row is complete called *hSync* and a signal when all pixels have been painted called *vSync*. There is also a signal that tells us if the pixel, that *hCount* and *vCount* point to, are outside of the screen (*blank*). All these signals conform with the VGA standard and are driven by a 25MHz clock since a resolution of 640*480 at 60Hz was desired. The 25MHz clock are generated with a clock divider that takes a 100MHz signal and outputs the desired frequency.

The monster controller and the cover controller contain some matrices that define the covers and monsters displayed on the screen. The matrices

are written from the software through the AXI bus. The data from the AXI bus contains a enable bit that begins the data writing process, an address signal that defines what register to rewrite in the matrix and a data signal that contains the information the be written.

The text controller, Figure 4, handles all the text displayed on the monitor. It was implemented with help of a laboratory task from Brigham Young University. [3] The 640*480 display is split into 80*30 character locations, where each character is 8*16 pixels. These characters is stored in a RAM, as ASCII code, where each location describes what character to be displayed. *hCount* and *vCount* from *VGA controller* is used to decide which character location to access from the RAM. Then depending on what character ASCII the data has in that RAM location, the system accesses a place in a ROM that has the sprite saved for that specific character. This data is then sent to the GPU controller that decides if a pixel will be displayed and its color. When writing to the RAM from software the only thing sent are a ASCII code and the character location you want to write to, which makes the programming in software very easy.

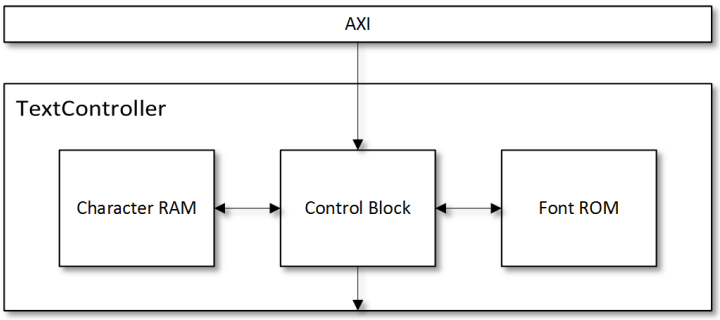

Figure 4: Text controller.

The GPU controller is the block that chooses if a pixel will be displayed or not. It takes the object coordinates from the software through the AXI bus and depending on where on the monitor the VGA controller is, it checks if a sprite exists there and its pixel of the sprite to paint. For the matrices in the monster and cover controller there are LUT (lookup tables) that check what object in the matrix the screen is on and depending on what status, it displays the appropriative sprite. The sprites are hard-coded into LUT where a logic ”1” defines that the system shall display a color and ”0” it

shall not. This minimizes the required hardware, compared to if each pixel have a different color, which require more bits.

2.1.2 Audio

The audio unit reads data from the AXI bus, written by the MicroBlaze processor, and then plays the selected sound file using a PWM(Pulse Width Modulator). [4] The block diagram of the audio unit can be seen in Figure 5.

The unit is constructed around a PWM. The inputs are determined by a controller that selects the appropriate sound file for the PWM to play depending on the input from the AXI bus. The output data is fed as a 1-bit output stream to the PModAmp1 component that smoothens the signal and amplifies it to line level output. The audio is played in mono with the left channel mirroring the right one. The sound files are stored in an eight bit level format with a sample rate of eight kilohertz, and are stored in RAMs represented by *Sound1* and *Sound2* in Figure 5.

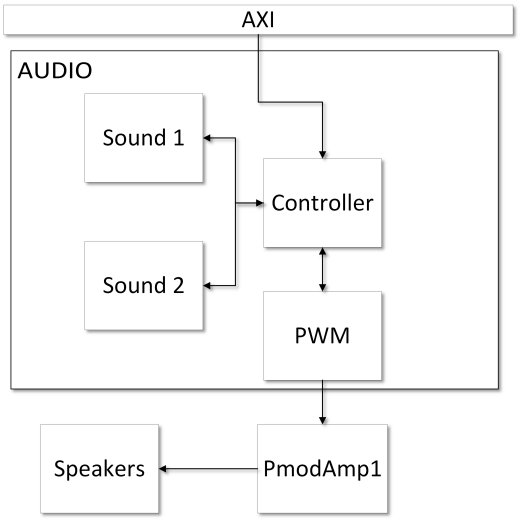

Figure 5: Block diagram of the Audio Unit.

2.1.3 PS/2

The Xilinx IP(Intellectual Property), *LogiCORE IP XPS PS2 Controller*, was used for the implementation of the PS/2 hardware controller. [5] *XPS PS2* was designed to be used with a PLB. So in order to connect the PLB

to the rest of the system a bus bridge between the PS/2 PLB and the AXI bus was implemented.

A interrupt channel was also connected between the PS/2 input and the MicroBlaze processor. This was done in order to be able to immediate address the appropriate response for each PS/2 input in the software.

2.1.4 Utilization

In Figure 6, the utilization of the different blocks in the system is visualized. The two largest blocks are the MicroBlaze and GPU core which is not surprising, considering they contain the most hardware. Most of the things in *other* can be removed since they are not paramount for the operation of the system, UART (Universal asynchronous receiver/transmitter) and debug module are not needed, but are useful during testing. If a custom made PS/2 controller were written, the area could be reduced by a large amount, since the Xilinx PS/2 is a general purpose PS/2 controller and takes up unnecessary space. The sound core does not use much logic, but instead uses a lot of memory which is not represented in this graph. The area of the AXI bus can not be significantly decreased since it is a Diligent IP that should not be modified.

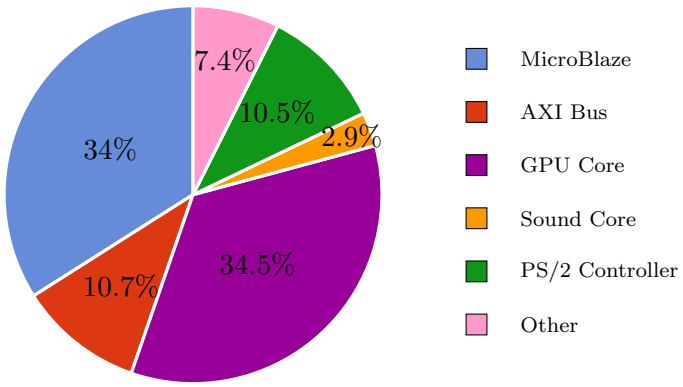

Figure 6: Slice logic utilization

In Table 1, the total utilization of our system on a Digilent Nexys 3, Xilinx Spartan-6 FPGA can be seen, where a total of 76% of the slices are

used. The system also use a total of 44% of the LUT which most can be accounted for the different sprites, matrix LUT and the MicroBlaze. The utilization of the registers is only 16%, where most of the registers are in the GPU core and MicroBlaze. The system uses 27 of 32 the block RAMs, where 16 of these are in the MicroBlaze, eight are in the sound core and 3 are in the text controller.

Table 1: FPGA utilization

	Used	Available	Utilization
Slice Registers	3,015	18,224	16%
LUT	4,274	9,112	44%
Occupied Slices	1,734	2,278	76%
RAMB16BWER	27	32	84%

2.2 Software

The software was implemented on a MicroBlaze CPU with 32 KB RAM, running at 100 MHz and programmed in C. The object of the program is to handle all necessary game logic, to make the game function as planned. A model of the game, programmed in Python, was used to evaluate the needed functionality. From the model it was derived that functions to initialize the hardware were needed as well as one main game loop. From the loop all the different functions handling the game logic itself were called upon. Functions such as moving and spawning sprites, collision detection and polling keyboard events.

The structure of the software was designed to be single threaded. The program is running with a main loop to handle the game logic functions. In addition to this it reads the input from the keyboard after receiving an interrupt. The received information from the keyboard only describes if a key has been pressed or released. Therefore extra functionality was added to the function handling the interrupt from the keyboard. The player should be able to move and shoot simultaneously, therefore an array was used to store up to five pressed keys.

Data transfers from the software to the hardware components was handled by an AXI bus. This meant that only 32-bit values could be transferred between the components. Therefore several functions was implemented to ensure that the data was sent correctly. These functions used logical SHIFT

and OR to pack several integers together in a 32-bit format. This could be done since values never needed more than 10 bits to be represented in binary. A majority of the sent data did not need more than 3-4 bits. This way data could be sent in a very efficient manner to the registers in the custom built hardware components.

2.3 Problems and Solutions

The major problem encountered was debugging, this due to the fact that it is hard to detect if the errors are occurring in the software or hardware. The solution we adopted was to have many isolated tests on the different components before integrating them into the system. By doing this there were no major bugs in the hardware at all.

We also had issues with the program not fitting in the memory when compiling the software. This was solved by cleaning up the code and making it more efficient. Also software compilation optimization was used.

3 Operation Guide

The main objective in our implementation of the Space Invaders game is to eliminate all aliens in each level. This implementation of Space Invaders has four levels of an ever increasing amount of aliens. The last level is then looped when completed so the player may compete for the high score. The player has to fire his gun in order to kill the aliens as they move ever closer to the player, simultaneously as the player has to evade their incoming shots. As the number of aliens begins to dwindle, they increase in speed and thus the difficulty. To the players assistance there are a couple of covers that can absorb the incoming shots. But as the covers are hit, they systematically get damaged and ultimately destroyed. At the start of the game the player is given two lives, which means that the player can get hit twice before losing the game. For each alien the player manages to kill, he will receive score. Sometimes, at random, a special boss alien will appear in the top of the screen and if the player manages to hit it the player will be awarded additional score.

Before the FPGA is configured with the game, a PS/2 style keyboard should be connected to the USB port and a PModAmp1 expansion module should be connected to the JA(0-3) ports on the FPGA.

After the FPGA is configured with the game, the player can control it

using the keyboard controls as seen in Table 2. The player may also utilize the hardware reset on the FPGA as seen in Table 3.

Table 2: Keyboard Controls

Action	Key
Left	A
Right	D
Shoot	Space
Reset Software	Esc
Skip Level	C

Table 3: FPGA Controls

Action	Key
Hardware Reset	Center Push Button (BTNS, Pin B8)

4 Lessons Learned

When designing a system on a FPGA, the designer has to take into consideration the limitation of resources. Especially the amount of memory available and the different kinds of memories has to be taken into account. A very obvious conclusion when looking at the utilization of the implemented system is that images and audio occupy very large amounts of memory. Therefore an external memory should be used for these kinds of data banks. Otherwise the designer has to compromise the systems functionality. For example the audio was stored in registers to begin with, but was moved to BRAMs instead, to make room for more sounds. This solution was possible because of the fact that the CPU memory was changed from 64 KB to 32 KB which made 16 BRAMs available for the rest of the design.

Software optimization is also a very important, both in the sense of coding and in compiling. Compile optimization of the program can be done to minimize the needed memory size and increase efficiency. The designer must be aware of what the compiler optimization actually does to the code. Otherwise unexpected behavior can appear, such as loops implemented to cause delays being removed for example.

5 Contributions

The design of the system was split into following areas. The report has been divided evenly where each person has written their own part, and the other chapters have been written together.

- Axel:
  - Main software
  - SW/HW integration
  - Testing
  - Final presentation
- John:
  - GPU core
  - Sound core (minor)
  - SW/HW integration
  - Testing
  - Proposal presentation
- Victor:
  - Software and hardware for PS/2
  - Sound core (major)
  - Testing
  - Final presentation

6 References

[1] Space Invaders game screenshot, Wikipedia
https://upload.wikimedia.org/wikipedia/en/2/20/SpaceInvaders-Gameplay.gif - 2015-11-04

[2] VGA Controller reference design, Ulrich Zoltán, Diligent
http://www.digilentinc.com/Data/Documents/Reference%20Designs/VGA%20RefComp.zip - 2015-11-04

[3] VGA Text Generator laboratory, Advanced Digital Logic Design, Brigham Young University
http://ece320web.groups.et.byu.net/labs/VGATextGeneration/VGA_Terminal.html - 2015-11-04

[4] PWM Audio Tutorial using a FPGA, MakerComputing
https://www.youtube.com/watch?v=4byHVqXD-UI - 2015-11-04

[5] LogiCORE IP XPS PS2 Controller, Xilinx
http://www.xilinx.com/products/intellectual-property/xps_ps2.html#overview - 2015-11-04