Department of Computer Science
Embedded System Design - Advanced Course (EDA385)

Project Report
FPGA Invaders

Per Edgren (dt06pe3)
Marthin Nielsen (dt05mn7)

October 19, 2009

Abstract

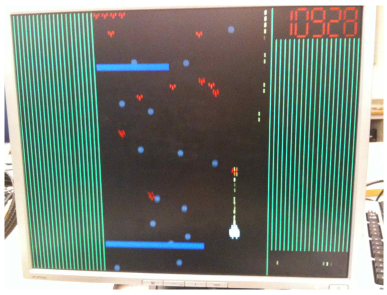
This report is part of a project where a two-dimensional space shooter game is designed on a Digilent Nexys2 development board (FPGA) using the Xilinx EDK. The system is designed using a hardware description language (VHDL) and a software programming language (C). The report covers everything from input-output (keyboard and VGA monitor) to the inner architecture of the system. Results and possible improvements are described both in terms of implementation and project work. Basic knowledge of C and VHDL is assumed throughout the report.

Contents

1 Introduction

Two main goals when choosing a subject for this project were that it had to be challenging yet attainable. Due to a lack of experience in hardware design and FPGAs, it was very hard to estimate workload and feasibility. A space-shooter game in a scrolling world seemed both suitable and appealing.

1.1 Legend



This is FPGA Invaders.

The goal of this game is to survive as long as possible in a lethal environment surrounded by asteroids, enemy ships and other obstacles. Everything that is hurled at the player can be destroyed with the ship's laser turret.

1.2 Deviation from Proposal

The first idea was to use framebuffers, i.e. to keep a copy of both the foreground and background in memory. After some research it was clear that this model would not fit on the board. An alternative to the framebuffers was to store the different objects such as the player, asteroids and enemy ships as bitmaps. The background would then be generated on the fly using a random algorithm implemented in hardware.

Instead of rendering bitmaps, the object representation was reduced to a set of rectangles. The background effect was omitted in favor of better game experience (improved controls, smarter enemies etc.)

1.3 General Design

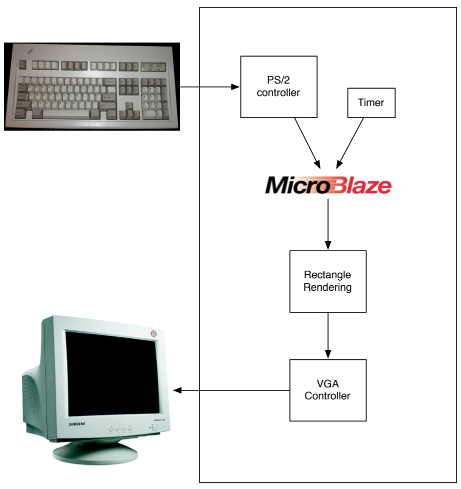


Figure 1: System Overview

A rough figure of the system design is shown to the left. A program written in C is run on the MicroBlaze softcore processor. This program acts on input from a keyboard and sends commands to the VGA controller that in turn renders the desired output. More detailed descriptions of the hardware and software are found in their corresponding sections.

There are two major communication interfaces between hardware and software. Firstly, there is the input (PS/2) that is read from software.

Secondly, the graphical output is controlled in software and rendered in hardware. Rectangles are encoded into 32 bit words as two 16 bit coordinates. In software, the internal object representation is packed to this format. It is then unpacked and displayed by the VGA controller.

1.4 Software / Hardware Partitioning

There are some tasks that naturally need to be implemented in hardware, such as I/O that are routed to physical connectors on the board. In an analogous manner there are tasks that are meant to be implemented in software. These can often be identified by the sequential logic they perform. The final partitioning of the system is listed below.

Hardware

- PS/2 input
- Rectangle rendering
- VGA Output

Software

- Game logic
- Collision detection
- Object representation (rectangle combinations)

1.5 Environment and Tools

The target board in this project is the Digilent Nexys 2 with a Spartan3E-1200 FPGA. The development environment for software and hardware is Xilinx Platform Studio, Xilinx ISE for simulation and Diligent ExPort for downloading the gate configuration to the board.

2 Architecture

This section describes the system architecture and the interaction between the different IP cores. Several Xilinx standard IP cores are used, the most prominent of them being the MicroBlaze softcore processor. Softcore means that the processor is synthesized along with the rest of the IP cores, resulting in a very portable design. Additional cores include bus controllers, a timer and an RS232 interface. A lightweight PS/2 controller has been reused from the course homepage, slightly modified for use with the PLB. A custom VGA controller was designed. A block diagram with the IP cores and the bus communication is shown below.

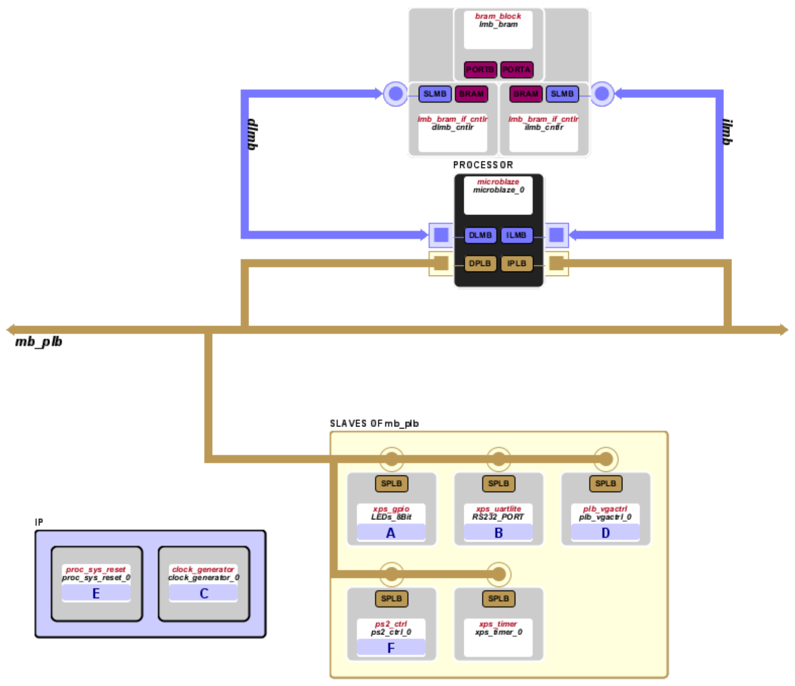


Figure 2: Block diagram of IP cores

The slice utilization of the entire project is 88%. In contrast to older projects in the course, the limiting factor was the number of interconnections. This problem is discussed extensively in section Problems and Solutions.

Design summary, entire project:

Number of 4 input LUTs	12,219 out of 17,344	70%
Number of occupied Slices	7,704 out of 8,672	88%

The communication interface between software and the VGA controller is based on a rectangle representation. As the figure indicates a 32 bit word is divided into four parts, X_{start}, Y_{start} and X_{end}, Y_{end}.

The screen is divided into two different parts. Between pixel columns 0-160 and 460-640, hardware controls and renders the graphics while columns 160-460 are controlled by software but rendered in hardware.

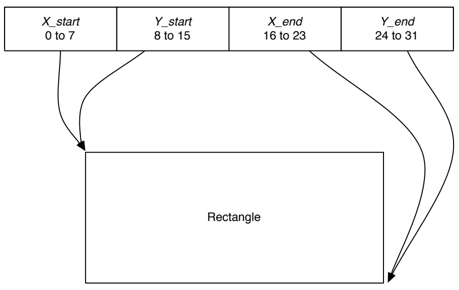


Figure 3: Bit representation

3 Hardware

3.1 VGA Controller

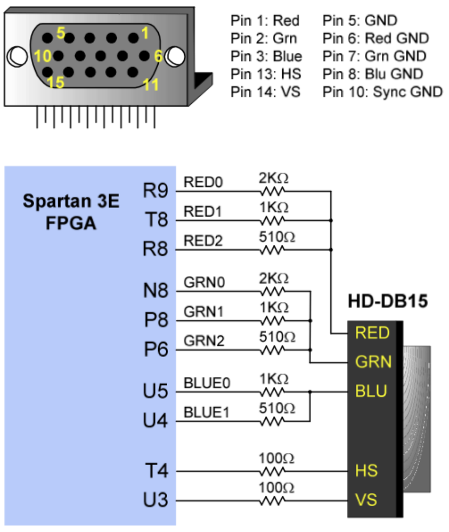


Figure 4: VGA connector

The VGA controller is divided into two logical parts. The top-level component is responsible for providing a representation of the current pixel that is drawn on the monitor. It contains the external ports (R, G, B, HSYNC and VSYNC) that are physically connected to the monitor. The schematic of the VGA connector is shown to the left (Nexys2 Reference Manual). The top-level component instantiates the `User_logic` entity that controls the pixel output. It also sends the next X and Y pair (denoted $next\_x$ and $next\_y$) to be displayed on the monitor.

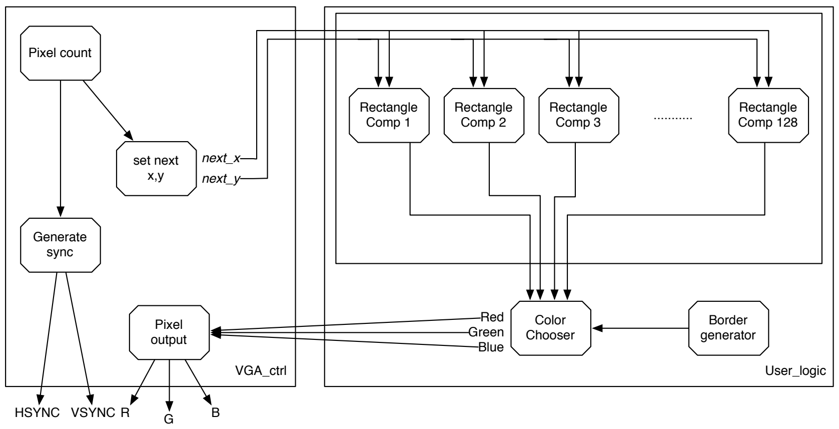


Figure 5: VGA Controller schematic

The basic idea of the graphical rendering entity is to compute each rectangle in parallel. This way the parallel nature of hardware is exploited and a short critical path is guaranteed. In order to accomplish this, a separate process is generated for each rectangle. Part of the VHDL code used to generate the processes is shown below. `User_logic` contains a BRAM implementation that is accessible from the MicroBlaze application over the PLB. Each process retrieves two coordinates (the corners of a rectangle) from the BRAM and checks if $(next\_x, next\_y)$ is within the boundaries of the rectangle. The result is sent to a multiplexer-like process that sends the desired RGB values back to the top-level component.

```
for obj_index in 0 to RECTANGLE_VECTOR_LEN - 1 generate
begin
  FG_SELECTOR : process (Bus2IP_Clk) is
  begin
    -- render logic
  end process;
end generate;
```

The device utilization of the VGA Controller is notably high. This is an effect of the numerous processes that are generated. This highlights the trade-off between area and speed that is part of every design process.

Design summary, VGA Controller:

Number of 4 input LUTs	10,287 out of 17,344	59%
Number of occupied Slices	6,609 out of 8,672	76%

3.2 PS/2 Controller

In order to get a smooth game experience and a quick response on the user-input an interrupt based PS/2 controller was to be implemented. However, due to an unknown error, MicroBlaze was unable to handle the incoming interrupt. After several attempts with and without external interrupt controllers and after discussing the issue with the course tutors, a polling based solution was chosen. After some tuning (e.g. embedding PS/2 polling into the collision detection algorithm) the controls were acceptable.

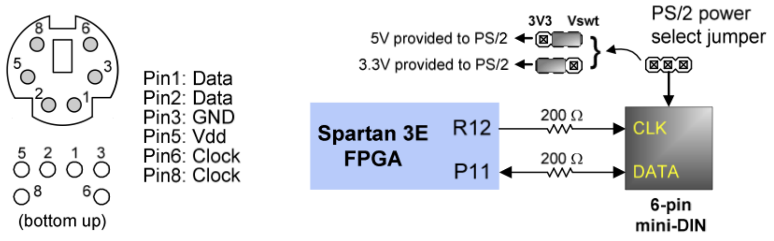


Figure 6: PS/2 connector (Nexys2 Reference Manual)

Different PS/2 implementations were tested when trying to fix the interrupt issue, including the Xilinx PS/2 IP core. As the interrupt issue remained, the most lightweight PS/2 controller was chosen. In contrast to the Xilinx core, this controller is designed to treat the CLK and DATA ports as pure inputs. Tri-state ports are not required in order to receive data from a keyboard.

Design summary, PS/2 Controller:

Number of 4 input LUTs	106 out of 17,344	<1%
Number of occupied Slices	83 out of 8,672	<1%

4 Software

The software running on the MicroBlaze is a simple sequential game engine. It is responsible for interpreting user input, the interaction between objects and commanding the VGA controller. The flowchart below describes the different steps involved when executing the game, from start to game over. As PS/2 interrupts are not used, keyboard polling is performed between the different steps in order to reduce the risk of missing a key event.

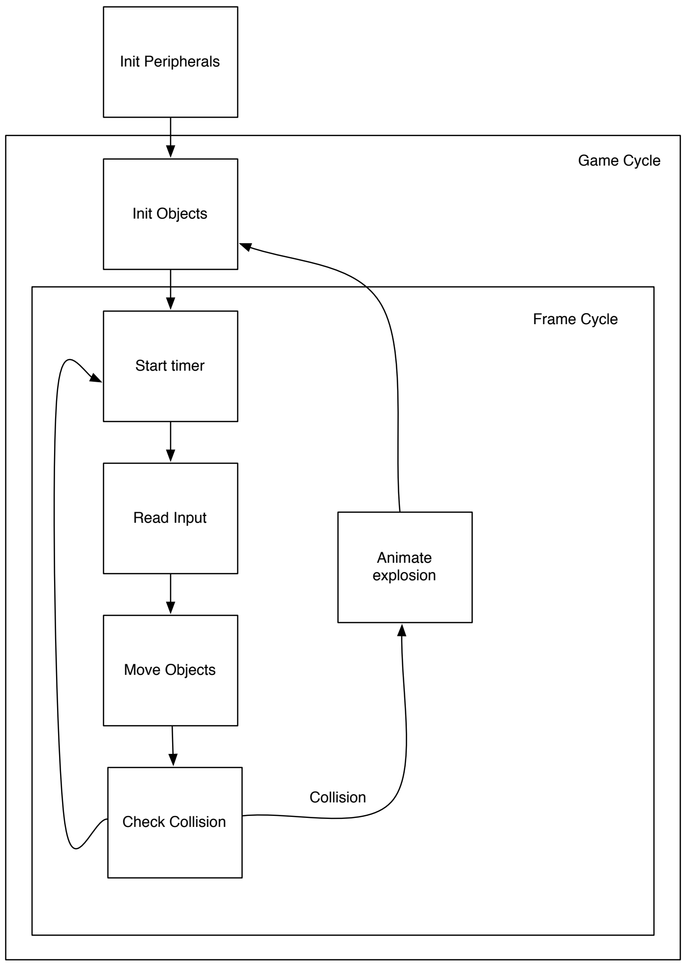


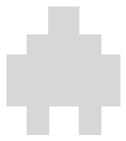
Figure 7: Software flowchart

4.1 Collision detection

The collision detection algorithm performs simple boundary checks between each of the objects. This is done once in every frame and is not an efficient implementation. This is not a problem as CPU time is abundant - MicroBlaze is in idle wait 95% of the time.

4.2 Painting objects

Visual objects are created by combining different kinds of rectangles. A sample object can be seen to the right. In order to paint this object, the following function calls are performed.



```
void draw_player(void)
{
  put_rectangle(4 + p.x0, 0 + p.y0, 6 + p.x0, 13 + p.y0, 0);
  put_rectangle(2 + p.x0, 3 + p.y0, 4 + p.x0, 15 + p.y0, 1);
  put_rectangle(6 + p.x0, 3 + p.y0, 8 + p.x0, 15 + p.y0, 2);
  put_rectangle(0 + p.x0, 7 + p.y0, 10 + p.x0, 13 + p.y0, 3);
}
```

In `put_rectangle` the rectangle data is packed into 32 bit words and written to the VGA controller. The implementation is shown below.

```
put_rectangle(short x1, short y1, short x2, short y2, short id)
{
  int *addr = (int*)(XPAR_PLB_VGACTRL_0_MEM0_BASEADDR + id*4);

  int data = 0;

  data = x1;
  data <<= 8;
  data += y1;
  data <<= 8;
  data += x2;
  data <<= 8;
  data += y2;

  *addr = data;
}
```

5 Game Description

A "screenshot" of the game can be seen below. The asteroids are randomly positioned, and so is the initial position of the enemy ships (a.k.a. "kamikazes"). The kamikazes are ships that have a random speed and start to move sideways, also at a random speed, after reaching a distance of 200 pixels from the player. Both asteroids and kamikazes can be destroyed by the player with a single shot. In addition to the kamikazes and asteroids there are bars posing as obstacles at random sizes between one third and two thirds of the game field width. These can be destroyed if needed with a fusillade of 70 shots. At half their health they start flashing, indicating that they have been damaged.

Control	*Action*
←	Move player left
→	Move player right
↑	Move player up
↓	Move player down
Space	Fire

The table above shows the commands that the user can send and their corresponding keys.

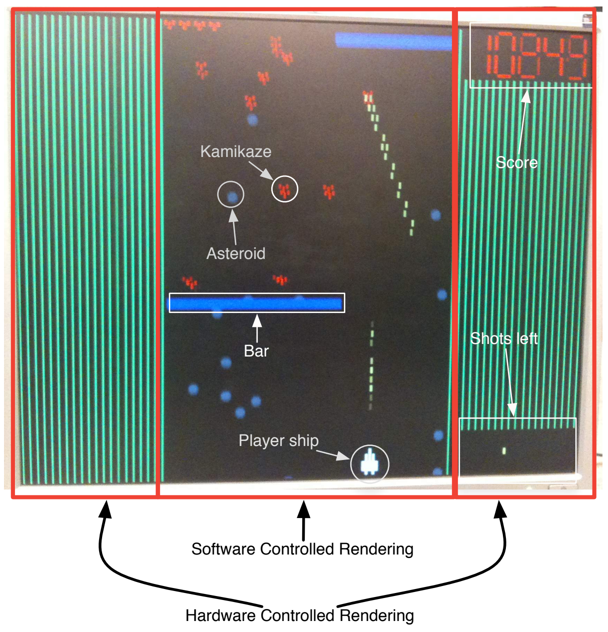


Figure 8: A sample screenshot

In the bottom right corner the ammunition is displayed. There is a total of 14 shots available that are returned to the player once they either hit a target or move out of the screen. In the upper right corner there is a software rendered seven-segment display of five digits, showing the player's current score. The score is incremented constantly as long as the player stays alive. The player will also get rewarded for destroying the different objects: 100 points for asteroids and bars, 200 for kamikazes.

In the beginning of the game there are no enemies and once every 1,000 points, a kamikaze is released from the depot in the upper left corner. At 14,000 points all kamikazes are detached. This results in an increasing difficulty level as the game progresses. When a ship or an asteroid is destroyed by the player it is re-spawned at the top of the screen.

6 Problems and Solutions

6.1 Adapting the Sample VGA Controller to PLB

A sample implementation of a VGA controller with a 1 bit color depth framebuffer is given on the course homepage. The BRAM is connected to the OPB, while the rest of the IP cores are connected to the PLB. Because of limited experience it was not realistic to implement a PLB slave by hand.

The solution was to create a new IP core using the "Create or Import Peripheral" wizard in Xilinx Platform Studio. This stub contained a BRAM implementation with a PLB interface. The VGA related logic was ported from the sample implementation after which the real customization began.

6.2 Choosing a Rendering Method

As framebuffers are expensive in terms of gate utilization, storing color information would not have been possible without greatly reducing the resolution. Pure framebuffer implementations are also inefficient in terms of speed as the rendering is performed in software.

In the end rectangles were chosen as the rendering method. This representation is very memory efficient as a large area can be covered by a single rectangle, described only by a coordinate pair. The implementation of a rectangle renderer is very straightforward and the result is a very flexible platform (objects are painted by combining rectangles in software).

6.3 PS/2 Interrupts

A time consuming problem that remains unsolved is the PS/2 interrupts. The source of the error has not been identified and a polling based solution has been implemented instead.

6.4 Routing Issues

In the end of the project the hardware synthesis took longer and longer to complete, often surpassing 30 minutes. While slice utilization remained at a controlled level, the number of interconnections rose aggressively. In some cases this resulted in designs that could not be routed at all.

The numerous interconnections are most likely caused by the memory access method in the rectangle processes. Instead of hot-wiring the comparators to the BRAM signal, appropriate read processes should have been used. As the problem arised during the last week of the project, there was not enough time to change the implementation. A blur effect where the rectangles would look smoother was abandoned because of the routing problem.

7 Conclusion

This project has provided a great challenge as well as a lot of hands on experience in hardware design, something that was missing among the project members. It was a great success with a satisfying result. The game is very fun to play thanks to the well balanced difficulty levels and the interesting tactical possibilities. We enjoyed creating ”FPGA Invaders” and are very proud of what we have accomplished, especially considering that our group only consists of two members.

If we had the chance to start over, the main focus would have been to implement the BRAM access according to standards. This way we would not have been limited by routing issues and could have spent more time on improving the game even more.

7.1 Possible Improvements

In addition to the problems mentioned in section Problems and Solutions, there are two features in particular that would have added to the gaming experience. The proposal suggested a background containing stars that was to be rendered independently from the rest of the game. This, in combination with the blur effect that was attempted would have contributed much to the graphical finish.

Another way of improving the project is to extend the game logic with more features - the software is very extensible.

7.2 Lessons Learned

When writing HDL code it is common practice to simulate it in a controlled environment prior to using it in a real design. This saves time as the lengthy synthesis process can be avoided when solving a particular problem. The simulation environment (ISE) also provides adequate tools needed for analyzing signals making debugging easier. In future projects, simulation will be used more often in order to increase productivity.

8 Contributions

The project proposal, game content and the final report were created by Per and Marthin.

The system architecture was designed by Per. The VGA controller was constructed by Per and Marthin, with influences from a sample implementation given on the course homepage. The software solution was written by Per, with MicroBlaze as the intended target platform from the start.

9 User Manual

Hardware requirements:

- Digilent Nexys2 FPGA prototyping board
- PS/2 Keyboard for user input
- VGA monitor for output

Installation:

- Download the project files
- Open the project in Xilinx Platform Studio
- Update Bitsteam (synthesis and compilation)
- Start Digilent ExPort
- Plug the USB, PS/2 and VGA contacts and power on the board
- Select Auto-Detect USB, click Initialize Chain and select check the ROM box
- Click browse and locate *download.bit* under *implementation* in the project folder
- Click Program Chain and enjoy the game!

10 References

Gruian, F: *Tutorials and documents* (sample VHDL implementations), (http://cs.lth.se/english/course/eda385_design_of_embedded_systems_advanced_course/tutorials_and_documents/).

Digilent: *Digilent Nexys2 Board Reference Manual*, (http://fileadmin.cs.lth.se/cs/Education/EDA385/HT09/docs/Nexys2_rm.pdf).