

# **FPGA BASED VGA DRIVER AND ARCADE GAME**

ARMANDAS JARUSAUSKAS  
THIRD YEAR INDIVIDUAL PROJECT  
2009/2010

SCHOOL OF ENGINEERING & DESIGN  
UNIVERSITY OF SUSSEX



This work is licensed under the Creative Commons Attribution 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/3.0/>

## **Abstract**

Field-Programmable Gate-Array (FPGA) technology is gaining popularity among Application-Specific Integrated Circuit (ASIC) designers. Ease of development and maintenance makes FPGAs an attractive solution to many speed and efficiency-critical applications. The purpose of this project is to explore the world of FPGAs by implementing an arcade game on top of a VGA driver. The project was implemented on Xilinx Spartan-3E development board using VHDL hardware description language.

The project was started by learning VHDL as well as familiarising with the Spartan-3E development board and Xilinx ISE WebPACK design software. A number of simple applications were developed in order to comfortably proceed on to investigation of working principles of a VGA driver. It turned out that the synchronisation logic was rather trivial and that the custom implementation would not add much value to the project.

VHDL implementation of the classic Pong game was the first major task of the project. Pong, as it was named, is a two-player game, where each player tries to hit a ball towards the opponent. A number of basic features, such as acceleration of the ball, sound output and textual information display, have been implemented. As this game was developed in the early stage of the project, complexity was consciously avoided.

The objectives the second game, on the other hand, were focused on technical features and other improvements. Still monochrome images were exchanged for colourful animations, sound output was made more sophisticated by adding a tune player, Nintendo game pads replaced on-board switches as a user input interface and a number of other modules were developed. The theme was chosen to resemble the Space Invaders game, but all graphic elements were designed from scratch.

To make the project more complete and self contained, the two games were combined into a single application, with a menu screen to allow user to choose which game to play.

Lastly, an adaptor board, for connecting NES game pads to the development board, was designed and built. It facilitates the communication between the controllers and the FPGA. The board also hosts a piezoelectric buzzer for sound output and a reset switch.

The project was complete and fully functional before the deadline. While there is still a lot of room for improvement, all set objectives, and more, were met.

**Statement of originality**

I hereby declare that this report is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or which have been accepted for the award of any degree or diploma at any educational institution, except where due acknowledgement is made in the report itself.

## **Acknowledgements**

First supervisor: Dr Ahmet Aydin

Second supervisor: Dr Christopher Harland

I would like to thank the following people for all the help and support:

- Ahmet Aydin for his time and support during the project and for letting me borrow the FPGA development board.
- Sam Beardsmore-Rust for general support and help with project proposal.
- Philip Watson for 100-pin connector design files and other help.
- Martin Nock for help with PCB manufacture.
- David Smith, from the mechanical workshop, for his excellent work on a plastic case for my adaptor board.

# Table of Contents

<b>1 INTRODUCTION</b>	<b>9</b>
1.1 About this report.....	9
1.2 Project aims and plan.....	9
<b>2 BACKGROUND</b>	<b>9</b>
2.1 Field-Programmable Gate Arrays.....	9
Xilinx versus Altera.....	9
2.2 Hardware description languages.....	10
VHDL.....	10
Digital systems theory.....	10
Other HDLs.....	10
2.3 Development board.....	10
2.4 VGA.....	11
Hardware.....	11
Timing.....	11
2.5 Nintendo controllers.....	12
2.6 Principles of game implementation.....	12
Logic blocks.....	12
Objects.....	12
<b>3 IMPLEMENTATION</b>	<b>13</b>
3.1 VGA.....	13
Code.....	13
Results.....	13
3.2 NES controller interface.....	14
Code.....	14
Results.....	14
3.3 Graphics.....	14
Storing images.....	15
Getting the pixels.....	15
Moving images.....	16
Colour image storage.....	16
3.4 Text generation.....	17
Font.....	17
Grid.....	17
3.5 Binary to BCD conversion.....	17
Theory.....	17

VHDL implementation.....	18
<b>3.6 Sound.....</b>	<b>18</b>
Frequency generator.....	18
Sound ROM.....	18
Player.....	18
<b>4 PONG.....</b>	<b>19</b>
<b>4.1 Structure.....</b>	<b>19</b>
<b>4.2 Graphics.....</b>	<b>19</b>
Main logic.....	19
Paddle logic.....	20
Ball logic.....	20
Scores.....	20
<b>4.3 Sounds &amp; User input.....</b>	<b>21</b>
Sounds.....	21
User input.....	21
<b>5 SPACE SHOOTER.....</b>	<b>21</b>
<b>5.1 Gameplay.....</b>	<b>21</b>
<b>5.2 Graphics.....</b>	<b>22</b>
Main logic.....	22
Aliens.....	22
Spaceship and missile.....	22
Explosion logic.....	22
Score and level display.....	23
<b>6 COMPLETING THE PROJECT.....</b>	<b>23</b>
<b>6.1 Game selection menu.....</b>	<b>23</b>
<b>6.2 Combining the applications.....</b>	<b>23</b>
<b>7 ADAPTOR BOARD.....</b>	<b>23</b>
<b>7.1 Reverse engineering the controllers.....</b>	<b>23</b>
<b>7.2 TTL / CMOS level conversion.....</b>	<b>24</b>
FPGA to Shift register.....	24
Shift register to FPGA.....	24
<b>7.3 Board design.....</b>	<b>24</b>
Schematic capture.....	24
PCB layout design.....	24
Manufacture.....	24
<b>8 DISCUSSION.....</b>	<b>25</b>

<b>8.1 Issues.....</b>	<b>25</b>
Pong ball acceleration.....	25
Synchronisation problems.....	25
Adaptor board.....	26
Binary-to-BCD conversion.....	26
<b>8.2 Further work.....</b>	<b>26</b>
<b>9 CONCLUSION</b>	<b>26</b>
<b>REFERENCES</b>	<b>27</b>
<b>BIBLIOGRAPHY</b>	<b>27</b>
<b>APPENDIX A</b>	<b>28</b>
Initial technical proposal.....	28
<b>APPENDIX B</b>	<b>29</b>
Project parts list.....	29
<b>APPENDIX C</b>	<b>30</b>
Project plan.....	30
<b>APPENDIX D</b>	<b>31</b>
NES Controller: equivalent schematic diagram.....	31
<b>APPENDIX E</b>	<b>32</b>
Python script for generating VHDL ROM from images.....	32
<b>APPENDIX F</b>	<b>33</b>
Adaptor board schematic diagram.....	33
<b>APPENDIX G</b>	<b>34</b>
Adaptor board PCB: top and bottom layer masks.....	34
<b>APPENDIX H</b>	<b>35</b>
Photographs.....	35




## 1 INTRODUCTION

### 1.1 About this report

This report covers the work done during the course of the project. An introduction to core concepts relating to the project is given in Chapter 2 section. Chapter 3 describes how the main building blocks, used to create a game, were developed, while Chapters 4, 5 and 6 go into game-specific details. Design of an adaptor board, connecting game pads to the development board, is described in Chapter 7. Report is finished by discussing some of the issues experienced during the project, and Conclusion.

Throughout the report, you will see notes like this:

---

 Project - filename.vhd

---

It is a pointer to a relative source file. It may be helpful, if one needs to see the implementation details.

The report is accompanied by a printed VHDL code listing and a Compact Disc. The CD contains all the relevant material in digital format, including this report, VHDL code, scanned version of engineering logbook and more.

### 1.2 Project aims and plan

The aim of this project was to explore the capabilities of modern programmable logic devices while getting hands-on experience of FPGA development.

The above definition is rather abstract and can be hard to assess. For this purpose, a number of objectives were set. The initial technical proposal can be found in Appendix A. Below, is the amended list of objectives.

- Learn about FPGAs, development tools and VHDL, start programming.
- Produce a simple application, that uses a VGA driver to display graphics on a computer monitor.
- Create a simple game.
- Create a more advanced game.
- Design some hardware.

A plan, available in Appendix B, with objectives

and milestones was created to give a rough estimate of duration of each part of the project.

## 2 BACKGROUND

### 2.1 Field-Programmable Gate Arrays

FPGAs are modern programmable logic devices that can be configured to perform any logic operation.

An FPGA typically contains a matrix of programmable elements, also known as, Configurable Logic Blocks (CLBs). CLBs contain Look-Up Tables (LUTs), that can be used as logic or storage elements. The configuration data is stored in the memory.

Spartan-3E series FPGAs, used in this project contain the following structures, as described in [1]:

- **Configurable Logic Block** – logic and basic storage elements are implemented using the Look-Up Tables (LUTs).
- **Input/Output Block (IOB)** – control the data flow between the I/O pins and internal logic of the device. LVTTTL and LVCMOS logic standards are supported among others.
- **Block RAM** – memory used for data storage. Organized as 18kb dual-port blocks.
- **Digital clock manager block** – provides management for clock signals.

### Xilinx versus Altera

There are two major manufacturers in the programmable logic business – Xilinx and Altera. Xilinx is a current market share leader, with Altera being a close second.

The decision which one to choose should not be based on the architectural differences of the two technologies. There are many more important aspects, such as device availability, overall price, development tools, existence of Intellectual Property (IP) cores and educational material and so forth.

Xilinx Spartan-3E development boards were readily available at the university and, therefore, Xilinx technology was used for this project's implementation.

# FPGA BASED VGA DRIVER AND ARCADE GAME

## 2.2 Hardware description languages

### VHDL

“VHDL is a language for describing digital electronic systems. It arose out of the United States government's Very High Speed Integrated Circuit (VHSIC) program. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of Integrated Circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed” (Ashenden, 1996) [2].

VHDL became an IEEE standard in 1987. Several revisions have been done since then.

VHDL can be used not only to describe digital hardware but also to create test benches for testing the designs.

VHDL supports many built-in and user-defined data types. Some of most often used are

- `std_logic` (single bit)
- `std_logic_vector` (bit vector)
- numerical types such as `integer`
- arrays, enumerated lists, etc.

Unlike imperative programming languages, such as C++, VHDL statements are concurrent (i.e. run in parallel). Concurrency is very useful in HDLs, as it resembles the way hardware works. Sequential statements can be implemented using a special construct called *process*.

Every VHDL design must have an *entity* and an *architecture*. The *Entity* contains a declaration of I/O ports while the actual design code resides in the *architecture* part.

Signals declared in the *entity* are referred to as external and are usually tied to I/O pins on the FPGA. The mapping of signals to pins is done in the User Constraints File (UCF).

Internal signals are declared in the architecture body and cannot be connected to the outside world.

### Digital systems theory

VHDL is a good example of practical application of digital circuit theory. Many of the basic concepts, such as gate and register-transfer level design, combinational logic and finite state machines have been used throughout the project.

### Other HDLs

As with the FPGAs, there are two predominant hardware description languages – VHDL and Verilog.

And just like with FPGAs, HDL usage is more a matter of preference. Capabilities are pretty much equal. The difference lies in how the hardware is described. Verilog is somewhat similar to C programming language, whereas VHDL is based on Ada.

According to Wakerly [3], it is best to “learn one well and, only if necessary, tackle the other later”.

## 2.3 Development board

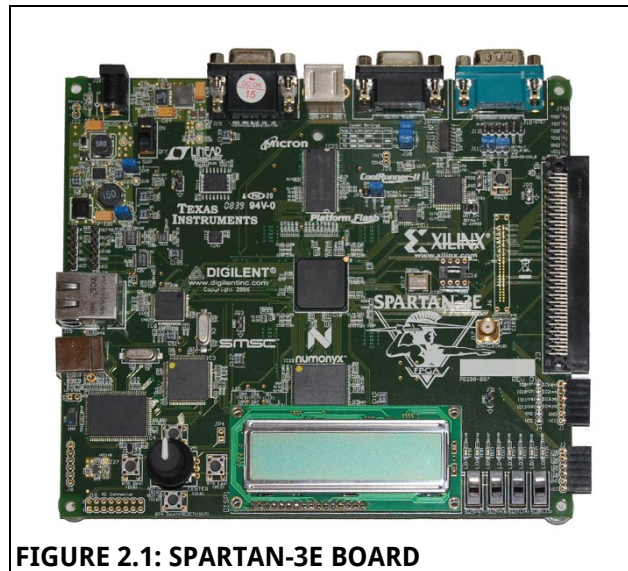


FIGURE 2.1: SPARTAN-3E BOARD

The development board used in this project is shown in Figure 2.1. It uses a Spartan-3E series FPGA (model XC3S500E) with the following characteristics [1]:

- 500,000 system gates, making over ten thousand equivalent logic cells.
- 20 RAM blocks totalling 360kb of memory.
- 20 dedicated multipliers.
- A maximum of 232 I/O pins.

The board has a 15-pin D-subminiature connector to plug in a VGA monitor.

Also, an on-board 100-pin expansion port is used to connect an adaptor board. Full list of features can be found in the user guide [4].

## 2.4 VGA

### Hardware

VGA is an analogue video standard, that is mostly used in personal computers. VGA can also refer to a piece of display hardware developed by IBM (Video Graphics Array) or a display mode, that uses 640 x 480 pixels resolution.

VGA connector uses a total of 15 pins, but only 5 signals are needed for operation:

- HSYNC – horizontal synchronization signal. This signal controls the horizontal position of the active pixel
- VSYNC – vertical synchronization signal. This signal controls vertical position of the active pixel. VSYNC rate can also be referred to as a refresh rate (i.e. number of times per second the screen is redrawn)
- RED – red colour channel
- GREEN – green colour channel
- BLUE – blue colour channel

Other pins include ground, return paths and I2C clock/data or are reserved [5].

HSYNC and VSYNC are TTL signals, so logic one is represented by 5V and logic zero is represented by 0V [5].

RED, GREEN and BLUE signals are analogue. The maximum voltage that can be used is 0.7V and will result in full intensity of that colour [5].

Xilinx Spartan-3E board is only capable of producing eight colours (3-bits) as a digital-to-analogue converter (DAC) is not used. The board uses 270Ω resistors, which form a potential divider with internal 75Ω termination. This divider scales the 3.3V signal from FPGA to required 0.7V [4].

### Timing

Pixels on the screen are drawn in sequence, one by one. Rows are arranged top to bottom, and columns go from left to right. The row and column addresses are constantly incremented thus changing the position of currently drawn pixel. Synchronisation signals are used to tell the monitor to return the pixel back to the first row (VSYNC) or the first column (HSYNC).

Sequence and duration of these signals are discussed below.

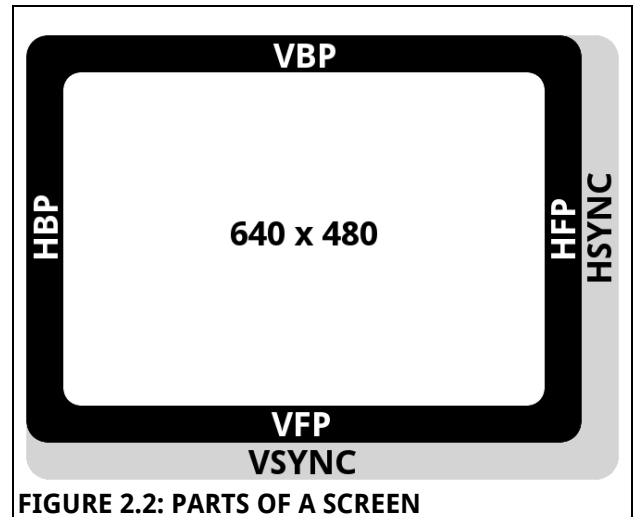


FIGURE 2.2: PARTS OF A SCREEN

Figure 2.2 shows parts that are required for an operation of a standard monitor.

Visible part of the screen is shown as the white area. It has a resolution of 640 by 480 pixels.

The black and grey borders denote parts of the screen that are not visible, but required for synchronization. With these parts, the total width of the screen is 800 pixels, and the total height is 524 pixels. Below is the short description of each part of the screen, followed by a table of dimensions.

**Active video.** This is the visible part of the screen, video output is enabled.

**Front porch.** When the trace reaches the end of the visible part of the screen, the video output is disabled. These areas are denoted as **VFP** (Vertical Front Porch) and **HPF** (Horizontal Front Porch) in Figure 2.2.

**Sync pulse.** In case of HSYNC, the trace goes back to column zero. If pulse is VSYNC, the trace goes back to row zero. This part is also known as the *retrace period*.

**Back porch.** This is the part that goes before the active video starts. These areas are denoted as **VBP** (Vertical Back Porch) and **HBP** (Horizontal Back Porch) in Figure 2.2.

Table 2.4.1 lists the dimensions for each part of the screen. These numbers are only valid for resolution of 640 x 480 and refresh rate of 60Hz. Other modes are described in the source [6].

# FPGA BASED VGA DRIVER AND ARCADE GAME

TABLE 2.4.1: VGA TIMINGS

25MHz pixel clock	Active video	Front porch	Sync pulse	Back porch
Horizontal	640	16	96	48
Vertical	480	11	2	31

## 2.5 Nintendo controllers

The classic *Nintendo Entertainment System* (NES) controller became a cultural icon among arcade game lovers and geeks. It therefore, seemed like a good idea to use these controllers as an input device.

The controllers use an 8-bit static shift register (4021B) to transmit user input as an 8-bit serial data. There are eight buttons on the gamepad, so each button has its own bit. Buttons and their corresponding values are listed in Table 2.5.1 below.

TABLE 2.5.1: GAMEPAD OUTPUTS

Button	Value
None	11111111
A	01111111
B	10111111
SELECT	11011111
START	11101111
UP	11110111
DOWN	11111011
LEFT	11111101
RIGHT	11111110

Note that the signal is active-low. When two buttons are pressed at the same time, two bits will be set to zero.

It is worth mentioning that the controllers and the development board cannot communicate directly with each other due to different voltage levels used. The FPGA can only go up to 3.3V (Low-Voltage TTL or CMOS), while the controllers need 5V. For this purpose, an adaptor board was designed and built. This is discussed in greater

detail in Chapter 7.

An equivalent schematic diagram of the controller is shown in Appendix D.

## 2.6 Principles of game implementation

### Logic blocks

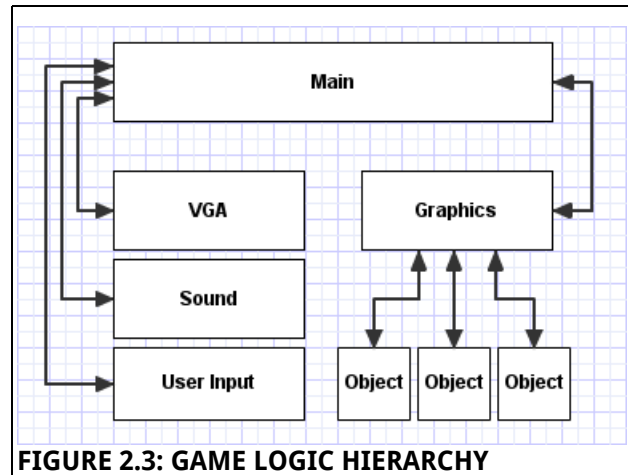


FIGURE 2.3: GAME LOGIC HIERARCHY

To make the code more manageable, it is a good idea to separate the source into logic blocks. Figure 2.3 shows one possible configuration of a game project.

At the very top of the hierarchy, is the **Main** block. Major logic parts are instantiated in this circuit. Signals shared between VGA, Sound, User Input and Graphics circuits all go through the main logic block.

**VGA synchronization** logic provides information about the pixel which is currently being drawn on the screen. It also outputs synchronization signals to the VGA port.

**Sound** circuit provides acoustic feedback to the user. This logic depends on signals provided by the graphics generation circuit.

**User input** logic is responsible for logging input events and providing the graphics logic with appropriate control signals.

At the bottom of the hierarchy, we have the **graphic objects**. They take pixel coordinates and control signals and, based on that information, generate RGB pixel values.

Finally, **graphics generation circuit**, in the middle of the diagram, can be thought of as a hub connecting all the objects. It passes pixel coordinates received from the VGA circuit, routes the data, sends control signals and enables or

disables the objects.

## Objects

In the game, each graphic element is treated as an object. The object has a source of pixel values, position coordinates, dimensions, logic and so forth.

The idea here is to provide another layer of logic separation. Each object is responsible for its graphical appearance, position on the screen, movement etc.


A ball in the game of Pong, is an example of an object. The graphical representation of the ball is stored in the ROM, the object knows its current location, and can update its coordinates to move or bounce of the wall.

Object generates a pixel value, which is used by graphics generation circuit to display the object on the screen. The value of the pixel is deduced from screen pixel coordinate. More on this in Chapter 3.3.

## 3 IMPLEMENTATION

### 3.1 VGA

#### Code

 vga\_sync.vhd

The VHDL code for driving a VGA monitor is taken from *FPGA prototyping by VHDL examples*, by Pong P. Chu [7].

It is one of the simplest parts in the project and, in this case, there is not much to gain from writing a custom implementation. This section will briefly explain the operation of the VGA controller logic. Comments in the code provide more detailed explanation.

Three major components are needed for the driver: a 25MHz clock, counters (for the vertical and horizontal pixel coordinates) and constants representing the timing values listed in Table 2.4.1.

The horizontal and vertical counters count to 799 and 523 respectively. These counters are updated at the speed of the pixel clock – 25MHz in this case.

During operation, the values in the two counters

are checked against the predefined constants. SYNC signals are produced during the sync stage (i.e. between the **front porch** and **back porch**).

The statement, therefore, can be described in pseudo-code like this:

```
sync_pulse = 1 when
    counter >= (display_area +
                front_porch)
and
    counter < (display_area +
                front_porch +
                sync_pulse)
```

Note that separate sync signals are used for vertical and horizontal synchronisation. The corresponding VHDL code can be found in the vga\_sync.vhd, lines 95 to 100.

The circuit also outputs *x* and *y* coordinates, named *px\_x* and *px\_y*, respectively. These coordinates are provided as 10-bit vectors.

## Results

To get a visual representation of the generated synchronization signals, the outputs were measured with an oscilloscope.

Channel 1 (top) in Figure 3.1 shows the HSYNC trace. The frequency needed to traverse 524 lines in  $1/60^{\text{th}}$  of a second is about 31kHz.

The VSYNC signal is shown as channel 2.

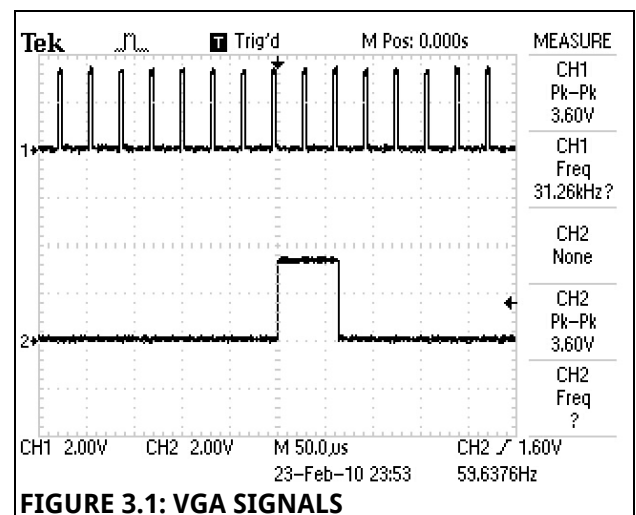


FIGURE 3.1: VGA SIGNALS

Figure 3.2 shows a zoomed-out version of the VSYNC trace. The HSYNC signal is left out, since no detail can be seen due to much higher frequency. Note the frequency of around 60Hz.



# FPGA BASED VGA DRIVER AND ARCADE GAME

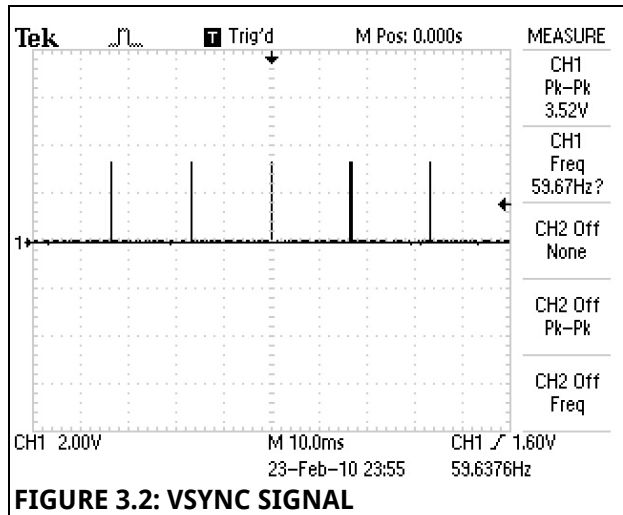



FIGURE 3.2: VSYNC SIGNAL

## 3.2 NES controller interface

### Code

 controller.vhd

The purpose of the driver is to provide control signals for game pads, as well as read the data from them.

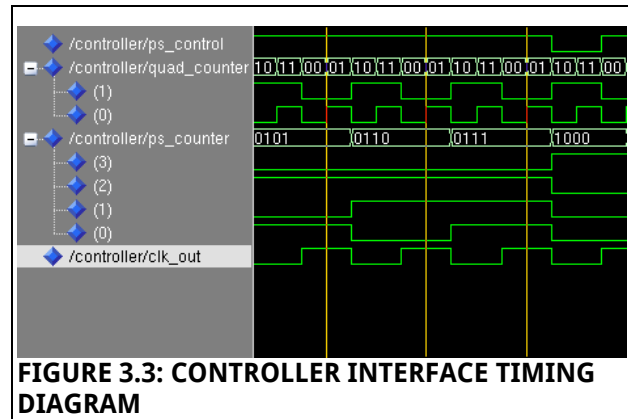
Controllers need external clock and latch signals. The clock frequency should not, under normal conditions, exceed 1.5MHz [8].

The latch signal must be eight times slower than the CLK in order to get all eight bits shifted out. When this signal is high, parallel data is loaded into the registers, when it is low, bits are shifted out.

Latch signal frequency was chosen to be 100Hz. This type of application does not require a very fast sample rate, so 100 samples per second seemed reasonable. Clock signal, therefore, needed to be 800Hz. This is much lower than the maximum given in the datasheet.

The CLK and latch signals are derived from the master counter, which is running at four times the CLK speed (3.2kHz). This is done so that the sampling point can be chosen with greater accuracy.

Figure 3.3 shows a part of simulated operation of the interface. Three vertical lines show the points at which data is sampled (master counter value is changing from zero to one). This point was chosen, as it is in the middle of high state of CLK signal (shown as clk\_out in the image) and, therefore, the voltages are stable.



The latch (ps\_counter in the image) value is also used to assign the received value to the right bit in the data byte.

When all bits have been received, the data is made available as an 8-bit logic vector.

### Results

Figure 3.4 shows the waveforms captured by an oscilloscope. Channel 1 shows CLK and channel 2 shows latch signal.

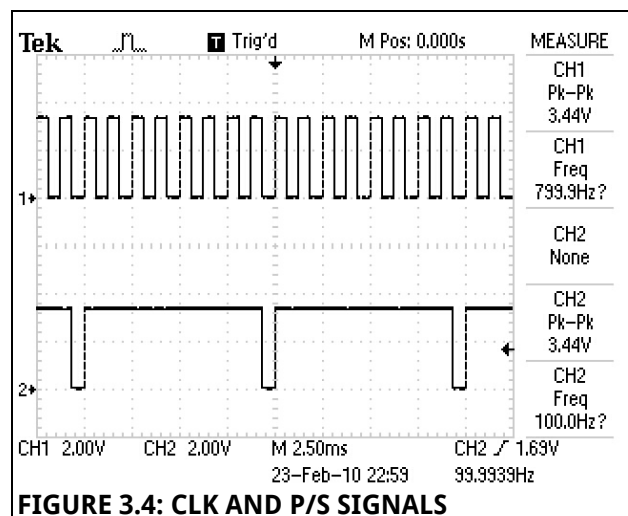


FIGURE 3.4: CLK AND P/S SIGNALS

Note the frequencies of 800Hz and 100Hz.

The output signals are inverted to match the operation of the adaptor board (described in Chapter 7).

## 3.3 Graphics

Section Principles of game implementation in Chapter 2, gave a brief introduction into the structure of a game. This chapter takes a deeper look into objects and graphics generation.


## Storing images

The graphics generation circuit gets a pixel coordinate from the VGA driver. Based on that coordinate, it generates a colour value for that pixel and outputs it to the RGB port.

For more complex graphics, an image for example, pixel values have to be stored in memory. One simple way of doing this, is to use the internal block RAM.

For storage of a simple monochrome image, a two-dimensional array is sufficient. Array rows represent image rows, and array indices represent individual pixels. In VHDL, this array can be described as follows:

```
type rom_type is array(0 to 1) of
    std_logic_vector(0 to 3);
constant SQUARES: rom_type :=
(
    "1010",
    "0101"
);
```

This array defines a four-by-two pixel image like this: . Note that *one* denotes the foreground (black) and *zero* denotes the background (white).

Even though, a 3-bit RGB scheme is used, we only need a single bit to describe monochrome images (same bit is used for Red, Green and Blue channels).

For simple cases, the synthesizer will infer a correct type of RAM based on the structure of the code. No special code to define RAM options was used in this project.

## Getting the pixels

Row and pixel addresses are derived from pixel coordinates. Lets do a simple task of displaying an image on the screen.

Figure 3.5 below shows ball graphics used in the Pong game. We want this image to appear in the top-left corner of the screen.

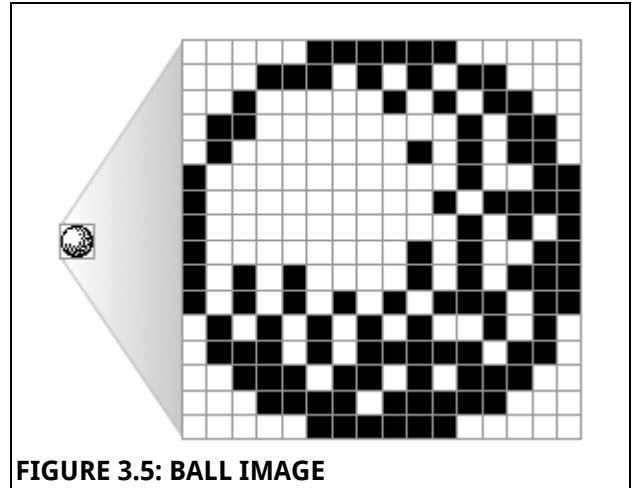


FIGURE 3.5: BALL IMAGE

The size of the example image is 16 x 16 pixels, therefore, we need the address to be 4-bits wide in order to be able to access each row and column.

The first pixel with coordinate (0, 0) is located at the top-left corner. Likewise, the last pixel is located at the bottom-right corner.

To begin with, we need to define the boundaries of the image. In our case, it is 16x16 square:

```
rgb <= img_data when px_x < 16 and
    px_y < 16 else
    "111";
```

So when the pixel coordinates are less than 16, image data is sent to RGB output. Otherwise, white colour is used as a background.

Now we need to set the image data. To do this, we need to derive the row and column addresses. Because the screen pixel coordinates correspond directly to the image pixel coordinates, it is sufficient to just take four least significant bits (LSBs) of the pixel coordinates provided by the VGA driver.

```
row_addr <= px_y(3 downto 0);
col_addr <= px_x(3 downto 0);
pixel <= pixel_data(col_addr);
img_data <= "000" when pixel = '1'
else "111";
```

The ROM instance (not shown here) automatically takes the row address and outputs the appropriate row as a `pixel_data` vector. The active pixel is then retrieved from this vector.

Lastly, single bit is converted to three bits for output at the RGB port.

*To keep the example as simple as possible, ROM instantiation and data type conversions are not*

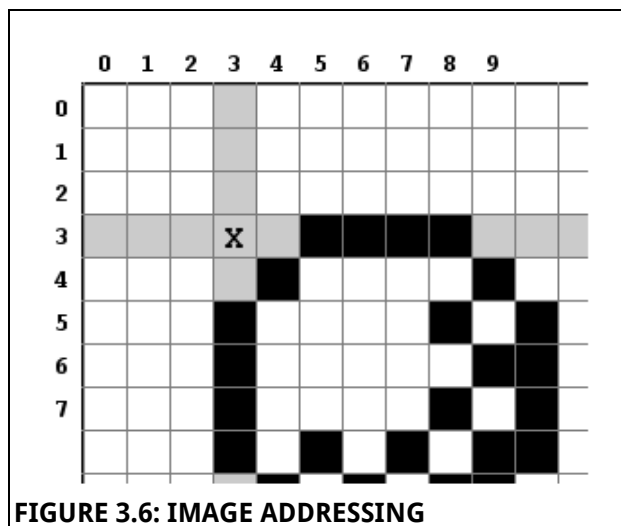
# FPGA BASED VGA DRIVER AND ARCADE GAME

*shown here. Reader should refer to code listing for full details. See the next section for a pointer.*


## Moving images

In order to be able to move images around the screen, we need to have vectors defining the position of an object. These will usually store x and y coordinates of the top-left corner of that image.

When the object is offset, we need a way to relate screen pixel coordinate to image pixel position. In Figure 3.6, we can see that, for example, when screen pixel is at (3, 3), corresponding image pixel is (0, 0).



The correct image pixel address is produced by simply subtracting the top coordinate of the ball from the current position of the screen pixel.

 Plong – graphics.vhd, line 360

```
row_addr <= px_y(3 downto 0) -  
            ball_top_y(3 downto 0);  
col_addr <= px_x(3 downto 0) -  
            ball_top_x(3 downto 0);
```


The code that enables the ball has to be updated as well.

```
rgb <= img_data when  
(px_x >= ball_top_x and  
px_x < ball_top_x + BALL_SIZE and  
px_y >= ball_top_y and  
px_y < ball_top_y + BALL_SIZE) else  
    "111";
```

Now whenever we change the top coordinate, the image will change its position on the screen. If the

coordinates are updated in a continuous manner, a motion will be created.

## Colour image storage

 FPGalaxy - alien\_1\_1\_rom.vhd

A different approach is needed to implement the storage of colour images. Since there are now three bits per pixel, array structure used before is not suitable.

The first challenge is: how to define a 3D array in VHDL? Readers, with experience in lower-level programming, may know the answer already – an array of arrays!

To begin with, we define a two-dimensional array type as before, and call it `rgb_array`:

```
type rgb_array is array(0 to 3) of  
std_logic_vector(2 downto 0);
```

This type will be used to describe rows of an image. There will be 4 pixels per row, and each pixel will be defined by 3-bit vector.

In the next step, we want to combine these row arrays to make an image. Structure definition is quite similar, but we use the type defined above instead of `std_logic_vector`:

```
type rom_type is array(0 to 1) of  
rgb_array;
```

Next, a constant with the pixel data is defined:

```
constant SQUARES_2: rom_type :=  
(  
    ("000", "001", "010", "011"),  
    ("100", "101", "110", "111")  
);
```

This image will show all eight colours that can be generated using 3-bit colour space.

Lastly, some updates to addressing are needed. In the monochrome image example, a whole row was made available at the output of the ROM. In the colour mode, the row type is an array, so it is better to return a value of a single pixel instead.

For this purpose, two addresses (row and column) have to be provided to the ROM. These addresses are joined together, so that externally the ROM looks the same.

To access each row of the example image, a single bit is sufficient. Likewise, two bits will cover all four columns. These addresses add up to a 3-bit



wide vector, which can be used like this:

```
rgb_row <= SQUARES_2(addr(2));  
data <= rgb_row(addr(1 downto 0));
```

## 3.4 Text generation

Often-times, games need to convey some textual information to the user. This chapter will describe the process of text generation in VGA applications.

### Font

 codepage.vhd

To begin with, a typeface has to be chosen. The font, used by the two games, was borrowed from Uzebox Project [9] as it resembles the typeface used by many old arcade games. The source image is shown in Figure 3.7.



When a suitable font is chosen, it has to be converted to a code page or, to say simply, a ROM.

The size of each symbol is 8 x 8 pixels, and there are 64 symbols in total. This results in 512 rows and 8 columns, or 4k bits of data in a ROM.

Entering this information by hand would be tedious and time-consuming. A simple Python script was written for this purpose and is provided in Appendix E.

The ROM is no different from that described in Chapter 3.3. This is how letter A looks after conversion.

```
"00111000", --   ###  
"01101100", --   ##  ##  
"11000110", --   ##   ##  
"11000110", --   ##   ##  
"11111110", --  #####  
"11000110", --   ##   ##  
"11000110", --   ##   ##  
"00000000", --
```

### Grid

 Arcade - menu.vhd

As the font has been chosen and the code page generated, we can now talk about placing the text on to the screen.

The easy way is to treat each letter as a tile. Our tiles are 8x8 pixels each, which means that 80 x 60 letter matrix can fit on a 640 x 480 resolution screen.

Each character has its own address within the ROM. The address of the first symbol, a space, is 0. The address of the second symbol, an exclamation mark, is 8 and so on. These addresses are used to define text.

For large amounts of text, a ROM storage is appropriate, but a simple multiplexer is sufficient when only a few letters are needed.

Addressing is very simple, provided the characters are placed strictly on the grid:

- To get a *row address*, we take the character address and add three lower bits of *px\_y* (the screen pixel coordinate) to it.
- Three lower bits of *px\_x* will select the right pixel in the row (i.e. the *column address*).

If text is moving or otherwise offset, a proper subtraction is needed, as described in Chapter 3.3.

Also, an additional logic may be required to select the character address if ROM structure is used to store the text.

## 3.5 Binary to BCD conversion

In order to display multi-digit numbers as a text, a binary-to-BCD (Binary Coded Decimal) converter is needed. This section will go through the theory and implementation of such converter.


### Theory

Binary numbers can be converted to BCD using a "Shift and add 3" algorithm, described in page 147 of source [7].

1. Bits are shifted to the left, so that the most significant bit (MSB) of a binary number becomes the least significant bit (LSB) of BCD0 digit, then the MSB of BCD0 digit becomes the LSB of BCD1 digit and so on.
2. After each shift, all BCD registers are checked; if value in the register is greater than four, three is added to that register.
3. Go to 1 until all bits have been shifted.

# FPGA BASED VGA DRIVER AND ARCADE GAME

## VHDL implementation

 FPGalaxy - bin2bcd.vhd

The conversion is controlled by a Finite State Machine with Datapath (FSMD). It has three states: start, shift and done.

*If you are unfamiliar with digital circuit theory, think of FSM as a flowchart. See Figure 3.8 for an example.*

**Start** state clears all the registers and the shift counter and preloads a new binary number.

**Shift** state checks the number of bits shifted, if all the bits have been shifted, the state is changed to done, otherwise, shift operation is performed again. Addition is done using concurrent statements outside of the FSMD.

**Done** state does not do anything explicitly. It, however, gives the output logic a signal that the conversion has been completed and the BCD digits can be made available at the output.

## 3.6 Sound

To make the games more complete, an audio feedback function was added. The sound is produced by a piezoelectric transducer which is driven directly by the FPGA.

The generation logic is divided in to three parts:

- Frequency generator
- Sound ROM
- Player

## Frequency generator

 sounds.vhd

The frequency generator is a low-level logic, that creates a square wave. It can produce different frequencies and duty cycles depending on the parameter values provided.

An internal counter is running at 50MHz. When the value of the counter matches the value of the *period* input, the counter will reset.

The duty cycle determines how long the output stays high during the period. The pulse width can be set as a fraction of the period by setting the

*volume* input.

The output can be switched on or off by setting the *enable* signal.

## Sound ROM

In order to create more complex sounds, a variety of frequencies, amplitudes and durations is needed. A ROM is used to store all this data.

To keep things simple, a *representation of data* is stored, rather than the actual data. The ROM contains a number of 9-bit vectors, which are later mapped to actual values.

Pitch, volume and duration are each allocated 3 bits, resulting in 7 different values (zeros are used for termination).

## Player

 player.vhd

A high-level logic, which takes data from sound ROM, transforms it and feeds the result to the frequency generator, is called a *player*.

The FSMD, shown in Figure 3.8, is the main control structure of the player logic.

This FSMD has two states – *Off* and *Playing*. When *start* signal is set, the FSMD state changes to *Playing*.

The **Playing** state serves two purposes: it checks for the end of the tune and also sets the address of the next note (datapath).

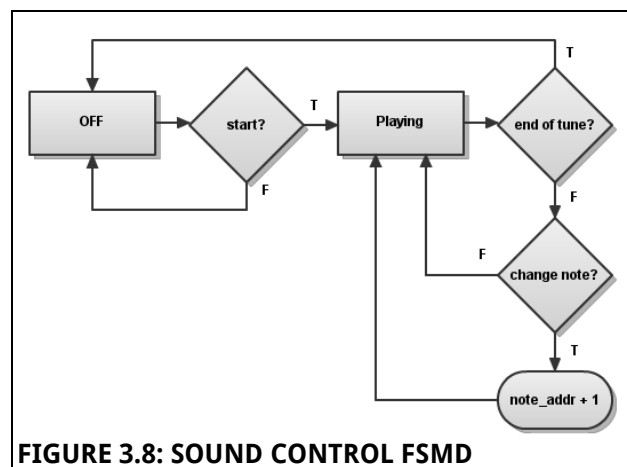


FIGURE 3.8: SOUND CONTROL FSMD

Output (enable signal) only depends on the state, and can be said to use the Moore model.

```
en <= '1' when state = playing else
      '0';
```

# FPGA BASED VGA DRIVER AND ARCADE GAME

Table 3.6.1 summarises operation of multiplexers used to transform the data from the sound ROM to appropriate values. The *Bits* column represents the data in the ROM, other columns show the corresponding output from the multiplexer.

TABLE 3.6.1

Bits	Frequency	Duration	Duty cycle
000	N/A		
001	110 Hz	1/64 s	50 %
010	220 Hz	1/32 s	25 %
011	440 Hz	1/16 s	12.5 %
100	880 Hz	1/8 s	6.25 %
101	1760 Hz	1/4 s	3.12 %
110	3520 Hz	1/2 s	1.56 %
111	7040 Hz	1 s	0.78 %

The set of values is far from comprehensive, but is sufficient for purposes of this project.

## 4 PONG

### 4.1 Structure

Development of the Pong game was a learning exercise in its way. There was not much planning involved and the main goal was to get it working and to grasp the basics game development.

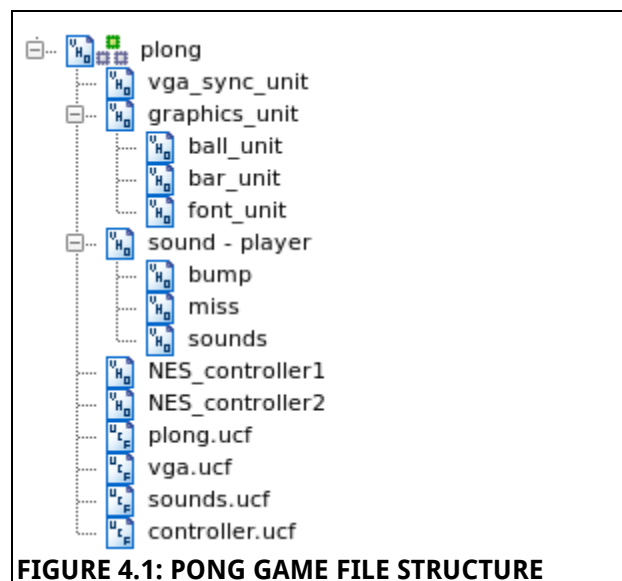



FIGURE 4.1: PONG GAME FILE STRUCTURE

Figure 4.1 shows the file structure of the Pong game. The structure follows the pattern described in Chapter 2.6, but the objects (ball and paddles) are not separated from the main graphics logic. This resulted in a large all-in-one file.

### 4.2 Graphics

#### Main logic

 Plong - graphics.vhd

The main graphics generation logic consists of two noteworthy parts: a game control FSM and a pixel routing logic.

The pixel routing is done with a conditional sentence. It basically defines object priorities, when two or more objects are at the same location. For example: the ball will go over the middle line and the score text, but will go under a paddle in case of the player missing the ball. Background has the lowest priority.

The control FSM is shown in Figure 4.2. It is responsible for keeping the scores, enabling or disabling motion of the ball and announcing the winner.

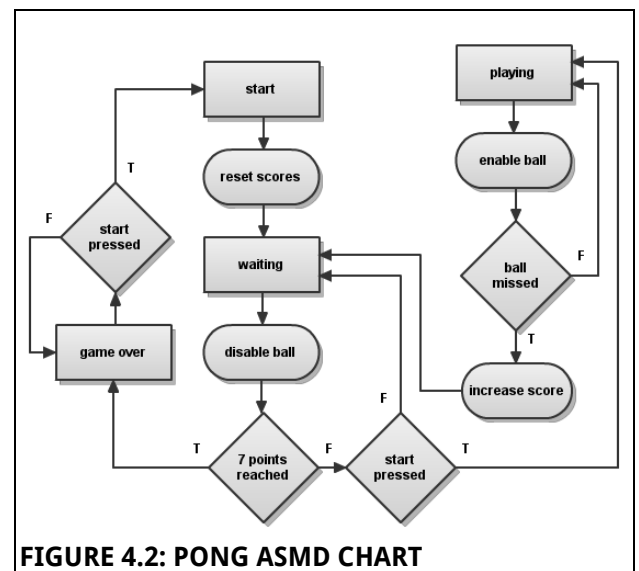


FIGURE 4.2: PONG ASMD CHART

**Start** state is used to reset the scores before the game begins.

**Waiting** state disables movement of the ball, changes state to game over, if one of the players has reached the seven-point limit, or waits for user input otherwise.

**Playing** state enables the ball. When the ball is missed, score count is updated and the state

# FPGA BASED VGA DRIVER AND ARCADE GAME

changes to waiting.

Winner is announced when the FSMD state changes to **Game over**.

## Paddle logic

This Pong implementation is a two-player game, so two paddles are needed. Dimensions and graphics are the same for both bars, but they have separate position coordinates.

Since the two paddles cannot appear in the same location, it was possible to use shared enable, ROM address, data and pixel signals.

Paddle control logic is fairly simple – it responds to user input by increasing or decreasing the top coordinate of each bar. When a paddle is at the top or bottom, no further movement is allowed in that direction.

## Ball logic

Ball logic is probably the most complex part of the game. One has to keep track of ball position, increase the velocity, enable or disable the motion and so forth.

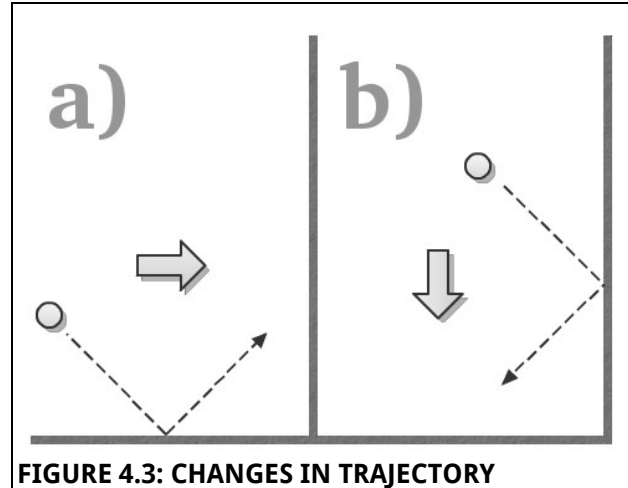
One of the most important features of the game is increasing the velocity of the ball. This allows a player to get used to controlling paddles and ensures that the game will come to an end at some point.

The acceleration of the ball is controlled by a counter. The counter counts to some value, and when it resets, the ball position will be updated. By reducing the value of the counter limit, ball velocity can be increased.

The counter limit is updated after a certain number of bounces from a paddle. There are only four levels of velocity, but it proved to be sufficient to make players concentrate more at greater speeds.

One issue with this acceleration approach is that sometimes the ball coordinate is updated more often than the screen itself. This results in small distortion of the ball (see Chapter 8), but is hardly noticeable during the game.

A separate process is dedicated to controlling the direction of the ball. Figure 4.3 illustrates the principles of ball movement.



**FIGURE 4.3: CHANGES IN TRAJECTORY**

The part a) of Figure 4.3 shows the ball going down and to the right. When the ball bounces, it starts going up and to the right. The bold arrow denotes the direction that does not change after the bounce.

Likewise, in the part b) the constant direction is downward, and the horizontal direction changes from right to left.

Using this principle, it is easy to implement a vertical direction control:

```
if ball_y = 0 then
    ball_v_dir_next <= '1';
elsif ball_y = SCREEN_HEIGHT -
    BALL_SIZE then
    ball_v_dir_next <= '0';
end if;
```

A horizontal direction control requires checking whether a paddle is in a right position, but the principle stays the same.

The position is updated according to the direction of the ball and is done like this:

```
if ball_h_dir = '1' then
    ball_x_next <= ball_x + 1;
else
    ball_x_next <= ball_x - 1;
end if;

if ball_v_dir = '1' then
    ball_y_next <= ball_y + 1;
else
    ball_y_next <= ball_y - 1;
end if;
```

As you can see, the x and y coordinates are updated independently.

## Scores

The pong game uses a text generation logic,

described in Chapter 3.4, to render points and a game-over message.

The score font address is acquired using this formula:

$$addr = 128 + (score\_value * 8)$$

Offset of 128 is needed because numbers start at this address in the memory. Multiplication by 8 is used to get the right digit (characters are 8 x 8 pixels in size).

The score value was deliberately limited to 7 for two reasons:

1. Simple implementation, no need for a binary-to-BCD converter.
2. Duration of a match is well balanced.

## 4.3 Sounds & User input

### Sounds

Two types of sounds are used in this Pong game. A short, high pitch sound is played when a ball bounces, and a longer, low pitch sound is played when the ball is missed.

Initially, only the square wave generator was used to make these sounds, so only single frequency was used. The tones remained after the sound logic was updated.

### User input

Initially, four on-board tactile switches were used to control the movement of paddles. At the end of the project development, the game was updated to make use of the Nintendo controllers.

There are three buttons, that the game uses: UP, DOWN and START. The purpose of these buttons should be obvious.

Since the functionality is very basic, there is no need for switch debouncing logic.

## 5 SPACE SHOOTER

The second game, a space shooter, was developed as a complement to the Pong game.

Space shooter is more complex than Pong – visual appeal improved by use of colour and animated graphics, enhanced sound logic is able to play simple tunes, proper game pads used to control the game.

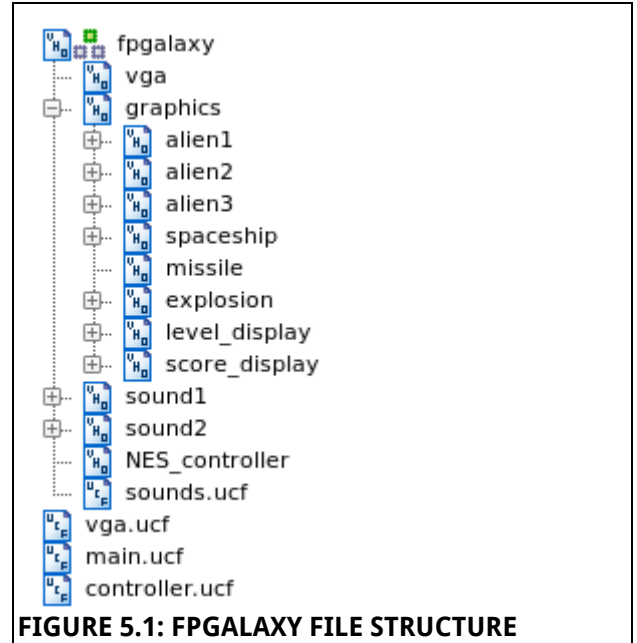


FIGURE 5.1: FPGALAXY FILE STRUCTURE

While being more more complex, the game is less complicated. More work has been done on the design side – the file structure was improved, each object is logically separated from all other objects. In fact, it would be possible to take the code that generates a spaceship, and put it into the Pong game with a minimum effort.

Figure 5.1 shows the improved file structure. As can be seen, graphics generation circuit now has many *child* files (objects). To conserve the space, files belonging to objects (e.g. ROM files) are not shown in the figure. Full file tree can be found in the source code listing.

## 5.1 Gameplay

The point of the game is to destroy the alien invaders. A player controls a spaceship located at the bottom of the screen; it can move sideways, but not up or down. The spaceship is able to shoot one missile at a time. When the missile hits the target or leaves the screen, the player can shoot another one.

The aliens are located at the top of the screen. There are three rows, with eight aliens in a row. Level 1 aliens hover in front (bottom row), they are weak and need only one hit to be destroyed. Level 2 aliens reside in the middle, they are stronger than Level 1, and need two shots to get killed. At the end, there are Level 3 aliens. It takes three hits to destroy a Level 3 alien. Moreover, they can turn invisible and avoid being hit by a missile.

When all the aliens are destroyed, another fleet


# FPGA BASED VGA DRIVER AND ARCADE GAME

shows up and the game continues.

## 5.2 Graphics

The two major improvements over the first game are use of colour (although limited to only 3 bits) and object animation. Word “animation” here means that the aliens are not only moving around the screen, but also change their shape.

### Main logic

 FPGalaxy - graphics.vhd

As most of the logic has been moved to other files, graphics generation circuit has somewhat less work to do than in the case of the Pong game. The logic is responsible for object control signals and interfacing with the VGA port.

The alien position coordinates are generated in the main file. The individual alien generators use signals generated by a simple FSM.

The score and level information is also updated here. Corresponding logic files are dedicated to displaying that information on the screen.

### Aliens


All the graphics were originally designed for the purpose of this project. Figure 5.2 shows the levels and states of the aliens.



FIGURE 5.2: ALIEN GRAPHICS

The frame size was chosen to be 32 x 32 pixels. The width of all aliens had to be the same, in order to make it impossible to destroy higher level aliens first. There is enough room for a missile to fly between two aliens.



These images were converted to ROM files for use by an alien generator logic. The logic takes master coordinate and generates pixel values based on that and some other signals.

 FPGalaxy - alien\_generator.vhd

Alien generation logic constantly checks the missile coordinates. When the missile is within the alien area, the logic checks which alien is being attacked and whether that alien is still alive. In case of destruction, appropriate signals are generated for use by other logic parts (such as explosion and score counter) and the alien is disabled. If alien was already destroyed, nothing happens and missile continues its journey.

An animation is created by changing the source of the alien graphics. The duration of each frame is controlled by a counter.

### Spaceship and missile

 FPGalaxy - spaceship.vhd  
 FPGalaxy - missile.vhd

The spaceship logic is very simple. Apart from pixel generation, it only has to react to user input to move sideways and output its centre coordinate for use by the missile generation circuit.

The missile generation logic also contains missile graphics (since the frame size is only 4 x 4 pixels).

When a launch button is pressed, the logic records the position of the spaceship and uses that value as its horizontal coordinate. Vertical coordinate is then updated at a certain frequency, which depends on the internal counter. The missile is disabled when it leaves the visible part of the screen or destruction signal is generated.

### Explosion logic

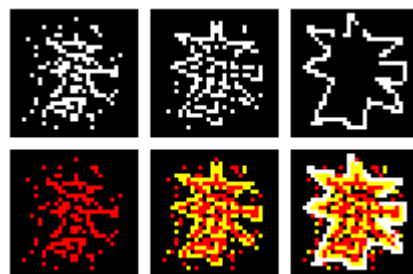


FIGURE 5.3: COLOUR MASKING

A more interesting design solution is implemented in the explosion generation logic. There is only a single frame in the ROM file and yet the explosion is animated in the game.



---

## FPGalaxy - explosion.vhd

---

The explosion image uses three colours: red at the centre, yellow in the middle and white on the outside. Top row in Figure 5.3 shows location of red, yellow and white pixels. By masking certain colours, three frames can be created, as shown in the bottom row. When these frames are shown one after another, the effect of animation is created.

The animation is controlled by a counter and a finite state machine. The pixel output depends on the state of that FSM.

### Score and level display

---

## FPGalaxy - score\_info.vhd FPGalaxy - level\_info.vhd

---

The score and level display generation circuits are also rather simple. They generate some text and use binary-to-BCD (see Chapter 3.5) converter for the numbers.

## 6 COMPLETING THE PROJECT

### 6.1 Game selection menu

---

## Arcade - menu.vhd

---

When both games were finished, it seemed to be a good idea to combine the two games into a single application and provide a nice menu to select a game.

The menu has to display some text, respond to the user input by changing the selection and provide an output signal, that indicates the current choice.

There are four pieces of text to be generated on the screen:

- title (text: 2 IN 1)
- game 1 (text: PLONG)
- game 2 (text: FPGALAXY)
- credits (author and date)

The title text is about eight times larger than normal letters, but is generated from the same source. Font scaling is very easy to implement and does not require any additional code.

Remember that pixel coordinates are stored in 10-bit vectors `px_x` and `px_y`. If we ignored the least significant bit, it would effectively reduce the resolution by 50%, or in other words, double the pixel size!

When generating the title text, three LSBs are discarded, making the pixel eight times as large.

### 6.2 Combining the applications

---

## Arcade - main.vhd

---

The next step is to combine all three applications together. Menu is the default one, and its RGB stream is selected after reset. When SELECT button is pressed (i.e. game is chosen), RGB output has to be fed from a game stream.

Since the games run in parallel, a way to disable on of them is needed. This was done by enabling/disabling user input for each game. When there is no input from controllers, a game will effectively be paused.

Then follows a standard procedure of component instantiation, definition of signals etc. This sort of combination may not be the most effective, but is definitely simple and highlights the benefits of modular design.

## 7 ADAPTOR BOARD

After the decision to use NES controllers was made, a couple of issues had to be resolved:

- how to connect Nintendo controllers to Xilinx development board and
- how to make two devices, using different voltage levels, communicate with each other?

The obvious solution was to design another piece of hardware that will have proper connectors to fit the development board and controllers and also host level conversion circuitry.

Extra functionality, such as reset button and speaker, was added to make better use of available board space. The list of parts can be found in Appendix B.

### 7.1 Reverse engineering the controllers

Some research into the internal construction of

# FPGA BASED VGA DRIVER AND ARCADE GAME

Nintendo controllers was done beforehand, in order to confirm the fitness for the purpose. After the game pads were acquired a proper schematic diagram of the internal construction had to be done.

At the heart of the device is a 4021B 8-bit static shift register used for parallel-to-serial data conversion. Each of eight parallel inputs has a button assigned to it. There are also power, clock, latch and serial data lines. The connections were found using a multimeter in a continuity test mode. Equivalent schematic diagram can be found in Appendix D.

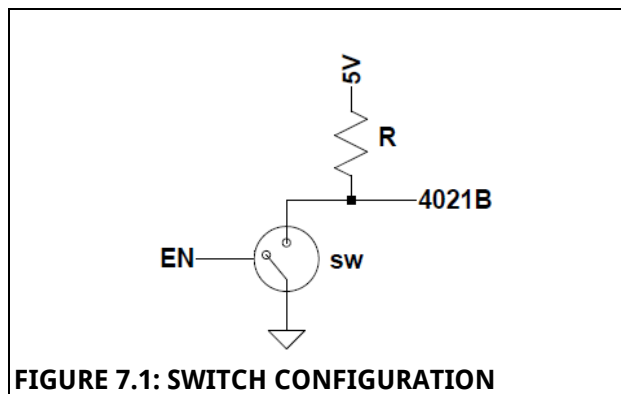
## 7.2 TTL / CMOS level conversion

The shift register inside the controller operates at TTL logic levels – 0V to 5V. The FPGA, however, uses low-voltage TTL/CMOS thresholds (0V to 3.3V).

While 3.3V from FPGA could be interpreted as a logic high by TTL circuitry, 5V from the controller could possibly damage the FPGA. In order to stay on the safe side, some sort of level conversion had to be implemented.

There are many ways to combine TTL and CMOS circuits. Approach used in this project is described below.

### FPGA to Shift register



The FPGA has to drive two lines – CLK and latch. Inside the controller, these lines are pulled high with pull-up resistors, so the signals are active-low.

A 74HCT4600 analogue switch is used to pull the line down: input is connected to GND and the enable signal is controlled by the FPGA. The state of the signal is controlled by enabling and disabling the switch.

### Shift register to FPGA

A 74LCX125 low-voltage buffer with 5V tolerant inputs was chosen to transform 5V output from a controller to 3.3V. This part is specifically made to interface 5V systems to 3V systems.

The data lines are not pulled high inside the game pads, so a 10k pull-up resistor was used on the board.

## 7.3 Board design

### Schematic capture

Schematic capture was done using *Easy-PC* EDA software. The resulting schematic diagram is shown in Appendix F.

On the left there is a 100-pin Hirose FX-2 connector, which plugs into the expansion port on Xilinx development board.

The FPGA I/O pins were chosen to make PCB design easier by reducing the length of tracks.

A piezoelectric transducer and a RESET switch are located at the top, near the FX-2 connector.

On the right side there are level conversion chips and on the left – NES controller ports.

Both Nintendo and Hirose connector design files were custom made. Design files for FX-2 connector were kindly provided by Philip Watson.

### PCB layout design

The circuit is rather simple, so there is not much to say about PCB design itself. Some practices and design considerations may be worth mentioning, though:

- Two-layer design, vias done using special pins.
- FPGA pins chosen to achieve minimum track length.
- Thicker tracks and “teardrops” used for power (3.3V and 5V) lines.
- Top and bottom ground planes connected in multiple points by “stitching” vias.

The top and bottom layer masks are shown in Appendix G.

### Manufacture

The board was milled using a Computer



Numerically Controlled (CNC) machine at university. Soldering was done by hand, using a hot air pencil for SMD components and soldering iron for through-hole parts.

The case was machined from a sheet of clear plastic with holes for controller ports and reset switch. As the switch is located on the board, extension rod was glued to improve accessibility.

## 8 DISCUSSION

### 8.1 Issues

During the course of the project, there has been a number of issues with various things. Some problems were minor, others required a bit more effort to solve. This chapter discusses issues that are most noteworthy.

#### Pong ball acceleration

In one of the final stages of development of the Pong game, a ball acceleration feature was being implemented. Two ways were considered to do it: increase step size or increase position update rate.

The step size increase means that instead of moving 1 pixel every time the position is updated, the ball would move 2, 3, 4, etc. pixels. This approach means that the ball position has to be calculated upfront or the ball may jump out of the visible portion of the screen.

A simple way to make the ball go faster is to increase the update rate. This approach does not require any extra logic, but due to limitations of VGA mode, introduces a slight graphical distortion of the ball.

The screen is updated sixty times per second. In order for graphics to be displayed properly, the image must not change while the pixels are being drawn on the screen. In other words, the refresh rate has to be faster than the image update rate. When ball position update rate is increased, this rule is violated and a slight distortion occurs.

Figure 8.1 shows an example distortion that could be perceived when the ball moves too fast towards the bottom-left corner of the screen. Note how the ball is divided into four parts.

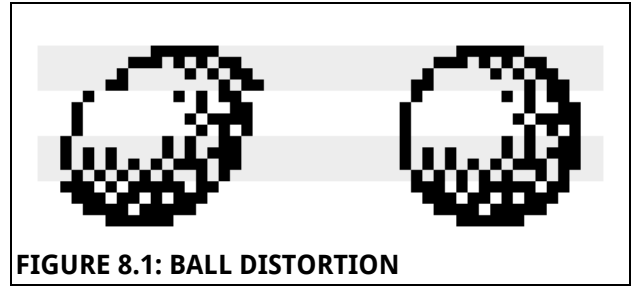


FIGURE 8.1: BALL DISTORTION

While the distortion can be spotted, it is not constant and so does not affect the game in any significant way.

#### Synchronisation problems

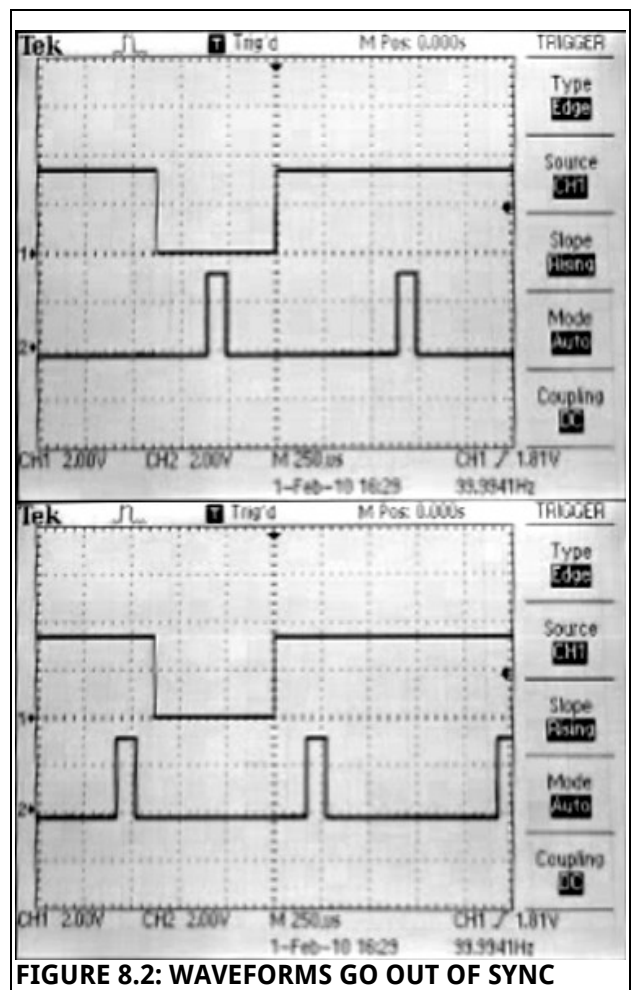


FIGURE 8.2: WAVEFORMS GO OUT OF SYNC

While implementing control of a spaceship, a bizarre behaviour was noted. From time to time, the spaceship would stop responding to any user input for a few seconds. Other functions, such as missile launch worked fine, so the controller interface problems were not considered.

Initial suspect was the enable signal, that was supposed to limit speed of the spaceship. After some “quick fixes” failed to solve the problem, signals were made available at the output for

# FPGA BASED VGA DRIVER AND ARCADE GAME

---

inspection with an oscilloscope.

Figure 8.2 shows two screen shots taken at different times. Top trace shows the data from the controller. Bottom trace is the enable signal. Note that the width of the enable signal was increased in order to be viewed on the oscilloscope. It is clear, that the two signals drift in and out of sync.

It was then, that it became clear, that the problem is really within the controller interface. User input was not supposed to be retained between samples. This caused malfunction and was the reason why the enable signal had to be added in the first place. When the problem was fixed, there was no need to have the enable signal any more and spaceship control started to work properly.

If there was still an need to limit the speed of the spaceship, a simple counter could be used to divide the sample rate to appropriate value.

## Adaptor board

This one can be written-off to silly mistakes. After the initial circuit was designed, time was not taken to double-check that the functionality of the adaptor matches the functionality of Nintendo game pads.

The clock and latch signals were internally pulled up, and, therefore, were active-low. It was falsely assumed, that the line was active-high and the switch in Figure 7.1 was tied to 5V instead of GND. Temporary fix was to cut the track and solder some wire between the pad and the ground plane.

The second mistake, was to omit the pull-up resistors on the data lines. While the absence of these resistors did not affect the operation of a controller, behaviour was undefined when a gamepad was not plugged in.

The second revision of the board was made to fix the remaining problems.

## Binary-to-BCD conversion

While implementing the “Shift and add 3” algorithm, described in Chapter 3.5, a wrong condition was chosen for stopping the conversion.

When the bits were shifted to the left, a '0' was appended in the place of the old LSB. This implied, that the conversion should be finished when all bits were zero. Such a condition meant that numbers like  $100000_2$  would not be

converted properly, since all bits would be zero after just one shift.

The problem could not be identified by just inspecting the code, so the Modelsim simulation software was used to track the bug down.

Step-by-step inspection made it trivial to notice the early termination of the conversion. A counter was added to track the number of bits shifted and the problem was gone.

## 8.2 Further work

While most of the set objectives have been achieved, there is still room for further improvement and experimentation.

If more time was given to work on this project, a number of different goals could be considered.

One possible approach to systems development on an FPGA is to use a soft-core microprocessor along with VHDL logic. Xilinx offers an 8-bit RISC microcontroller – PicoBlaze.

It would be possible to use a VHDL based VGA driver while implementing game logic on a soft-core processor. This could potentially give more flexibility and reduce the development time.

Another possibility is to focus more on improving the game features, or even develop an emulator able to run more than a couple of games.

## 9 CONCLUSION

The project was completed successfully – both aims and set objectives have been achieved.

The project was started by learning the basics of FPGA development, going on to produce simple applications, before moving on to work with the VGA graphics and game development.

A number of modules have been developed, to give extra functionality to the games. These include sound and text generation logic, a data converter and an interface for game pads.

An adaptor board, connecting the game pads to the development board, was designed and built. As well as providing a nice integration between two very different devices, this task gave an opportunity to work on circuit and PCB design.

The project provided valuable experience in FPGA based system design and was really enjoyable.

# References

1. Xilinx, Inc., 2008. Xilinx Spartan-3E FPGA Family Datasheet. Available at: [http://www.xilinx.com/support/documentation/data\\_sheets/ds312.pdf](http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf) [Accessed 1 March 2010]
2. Ashenden, Peter J., 1996. The Designers Guide to VHDL. San Francisco: Morgan Kaufmann Publishers, Inc.
3. Wakerly, John F., 2006. Digital Design: principles and practices. New Jersey: Pearson Education, Inc.
4. Xilinx, Inc., 2008. Xilinx Spartan-3E FPGA Starter Kit User Guide. Available at: [http://www.xilinx.com/support/documentation/boards\\_and\\_kits/ug230.pdf](http://www.xilinx.com/support/documentation/boards_and_kits/ug230.pdf) [Accessed 26 February 2010]
5. Myers, Robert L., 2002. Display interfaces: fundamentals and standards. Chichester: John Wiley and Sons
6. <http://martin.hinner.info/vga/timing.html> [Accessed 27 February 2010]
7. Chu, Pong P., 2008. FPGA prototyping by VHDL examples. New Jersey: John Wiley & Sons
8. ON Semiconductor, 2005. MC14021B: 8-Bit Static Shift Register. Available at: <http://www.onsemi.com/PowerSolutions/product.do?id=MC14021BCP> [Accessed 9 March 2010]
9. <http://code.google.com/p/uzebox/> [Accessed 26 March 2010]

# Bibliography

10. Holdsworth, Brian., Woods, Clive., 2002. Digital Logic Design. Newnes
11. Short, Kenneth L., 2008. VHDL for Engineers. Prentice Hall

# **APPENDIX A**

## **Initial technical proposal**

### **Project brief**

**Project Title: An FPGA Based VGA Driver and Arcade Game**

**Armandas Jarusauskas**

**Superisor: Dr Ahmet Aydin**

#### **AASP1**

Build a simple arcade game, using an FPGA development board. The project will involve exploring the capabilities of the board, and FPGAs in general, before moving on to develop a custom made arcade game. The project will involve the use of a Xilinx Spartan-3E development board and will be based around the use of VHDL as a tool to describe and implement digital hardware designs onto the development board.

The following objectives would form the bulk of the project

- To complete and fully understand the capabilities of the development board
- To implement a VGA interface module capable of displaying visual information on a computer screen
- To implement control i/o functionality capable of interfacing to a controller
- To design and implement a custom arcade game, using a VGA monitor and custom made controller

If time allows, the following objectives will be completed

- To design and build a simple controller
- To design and implement sound effect functionality
- To implement more complex game functions, including a non volatile high score table, two player mode, etc. etc.

## APPENDIX B

### Project parts list

Part	Unit price
NES controller x 2	£3.99
NES 2 to 4 adaptor (Connectors)	£12.00
Hirose FX-2 100-pin connector	£5.45
74LCX125	£0.44
74HCT4066	£0.42
TDK PS1720P02 Piezoelectronic Buzzer	£0.40
<b>Total</b>	<b>£22.70</b>

## Project plan

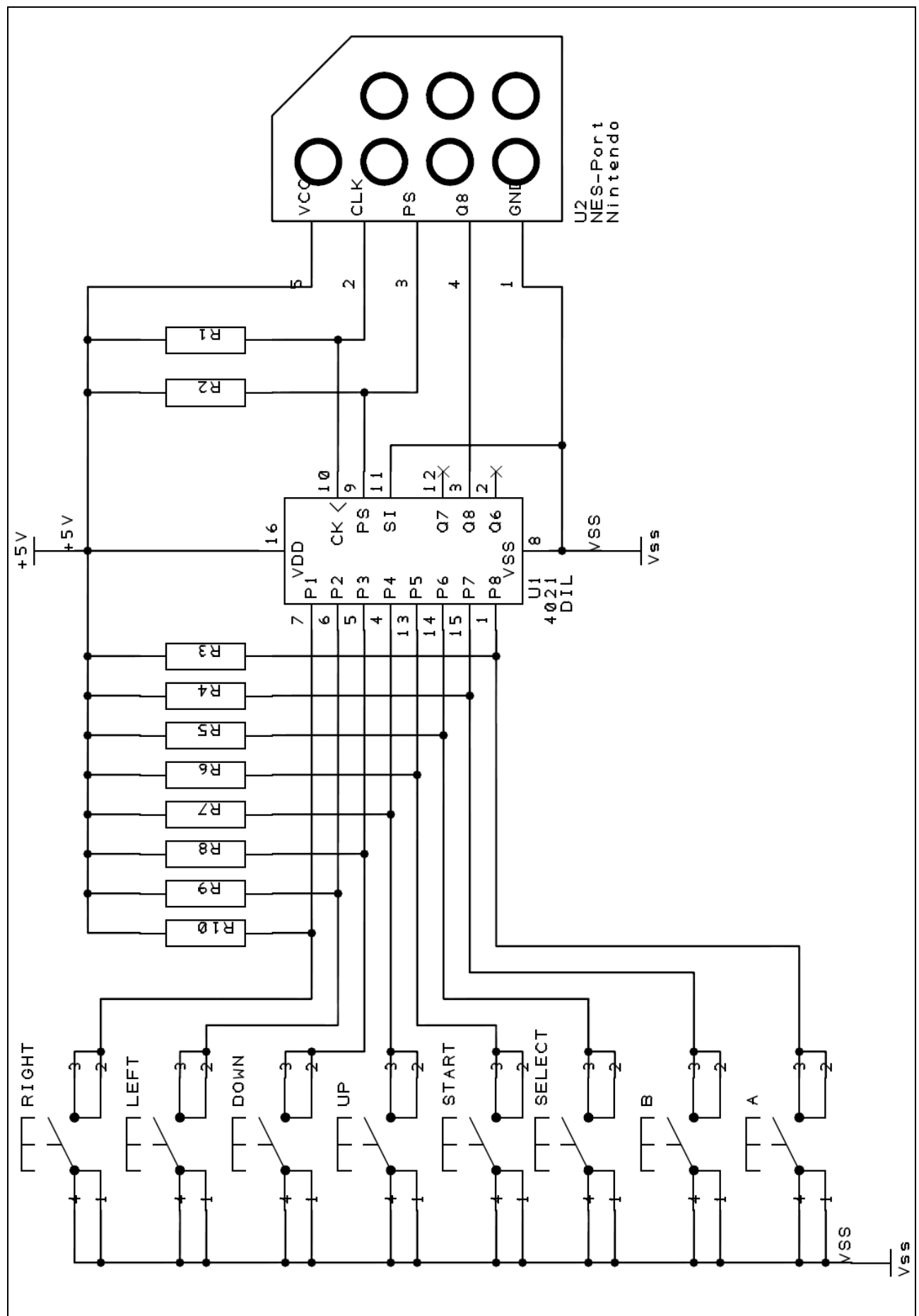
ID	Task Name	Jul '09	Aug '09	Sep '09	Oct '09	Nov '09	Dec '09	Jan '10	Feb
1	Learn VHDL basics	[Solid]							
2	Familiarise with ISE IDE	[Solid]							
3	Explore capabilities of development board	[Solid]							
4	Study principles of VGA	[Solid]							
5	Make working VGA test circuit	[Solid]							
6	Game development	[Hatched]							
7	2 Player pong game	[Solid]							
8	Pong graphics design	[Solid]							
9	Pong sound effects	[Solid]							
10	1 Player space shooter	[Hatched]							
11	Space shooter graphics design	[Hatched]							
12	Space shooter sound effects	[Hatched]							
13	Controller design and build	[Hatched]							
14	Testing & Debugging	[Hatched]							

Project: FPGA based VGA driver and arcade game.m  
Date: Wed 21/04/10

Page 1

## APPENDIX D

### NES Controller: equivalent schematic diagram



## APPENDIX E

### Python script for generating VHDL ROM from images

```
#!/usr/bin/env python
import os
import sys
from PIL import Image

def compare_names(a, b):
    """ compare function for sorting """
    n1 = a[:-4].split('_')
    n2 = b[:-4].split('_')

    if n1[0] == n2[0]:
        return cmp(int(n1[1]), int(n2[1]))
    else:
        return cmp(int(n1[0]), int(n2[0]))

files = os.listdir(sys.argv[1])
files.sort(cmp = compare_names)

output = ''

for i, file in enumerate(files):
    if file.endswith('.png'):
        img = Image.open(sys.argv[1] + file)
    else:
        break

    # temporary storage for bits
    tmp = ''
    # rom contents
    output += '-- %d\n' % i

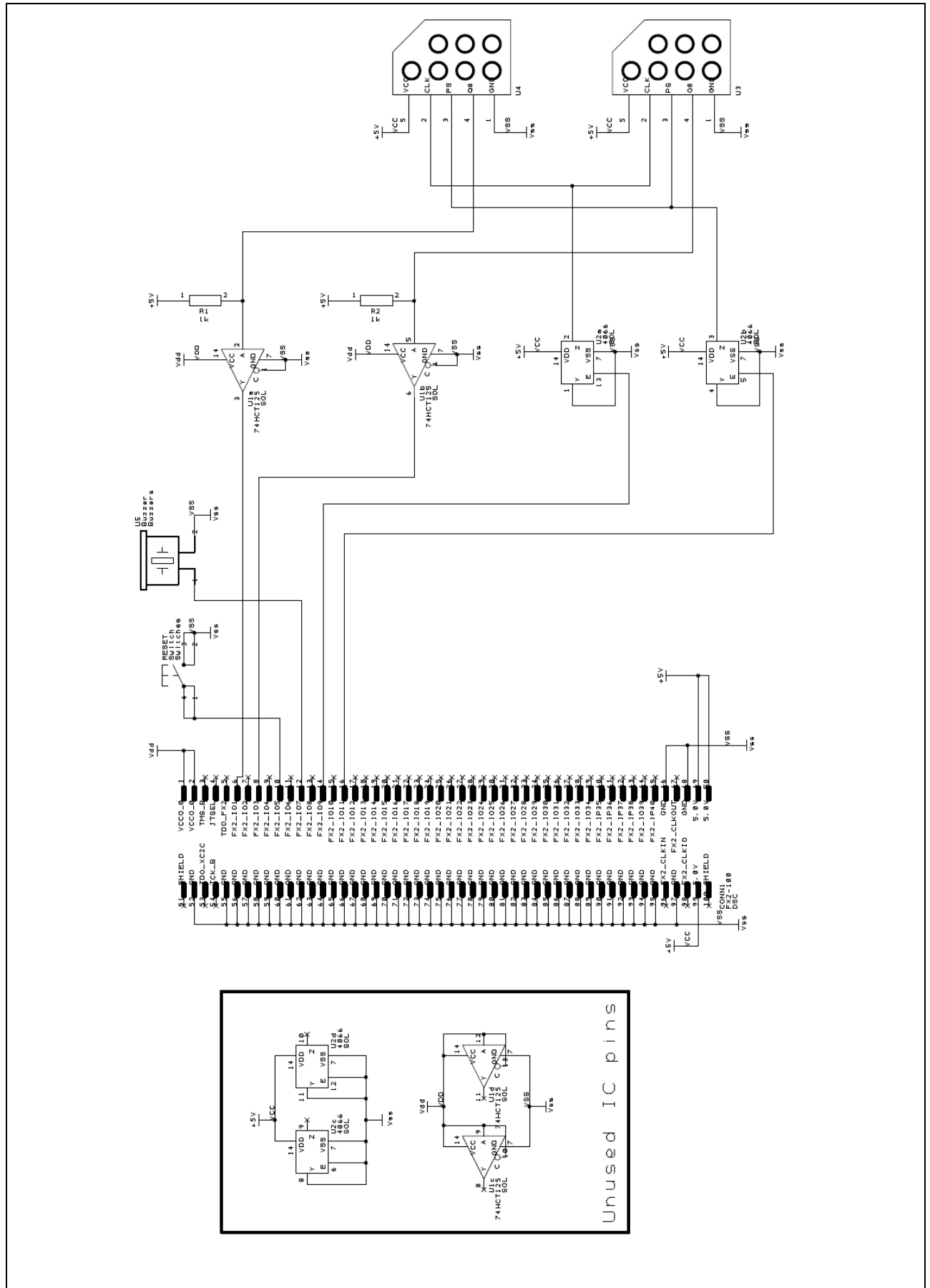
    for j, pixels in enumerate(list(img.getdata())):
        tmp += '1' if pixels[0] == 255 else '0'
        # we have 8 pixels in a row
        if (j + 1) % 8 == 0:
            # generate bit vector and comment
            t = "%s", -- %s' % (tmp, tmp.replace('0', ' ').replace('1', '#'))
            output += t.strip() + '\n'
            tmp = ''

sys.stdout.write(output)
```



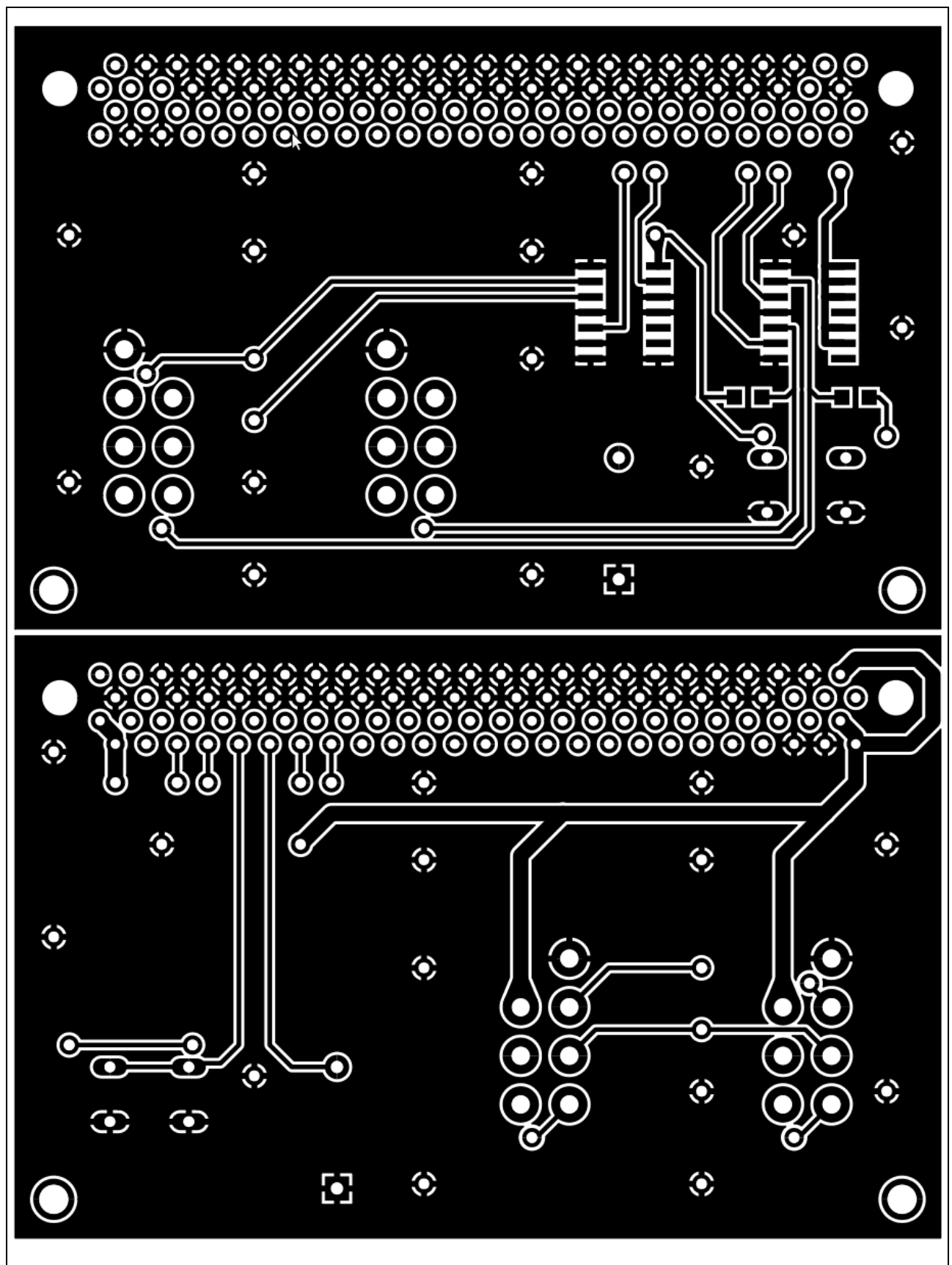
# APPENDIX F

## Adaptor board schematic diagram



## APPENDIX G

### Adaptor board PCB: top and bottom layer masks



## APPENDIX H

### Photographs

