Interactive arcade game development in a reconfigurable platform with hand motion recognition feature

Delacoura Angeliki

A Thesis presented for the degree of

Diploma of Science in
Computer and Communication Engineering

Supervisors: Nikolaos Bellas, Associate Professor
Christos Sotiriou, Associate Professor

University of Thessaly
Volos, Greece

September 2014

Dedicated to

My parents.

Interactive arcade game development in a reconfigurable platform with hand motion recognition feature

Ανάπτυξη διαδραστικού arcade παιχνιδιού σε επαναπροσδιοριζόμενη πλατφόρμα με δυνατότητα αναγνώρισης κίνησης χεριού

by

Delacoura Angeliki

Submitted to

the Department of Electrical and Computer Engineering

University of Thessaly, Volos, Greece

for the degree of Diploma of Sience in

Computer and Communication Engineering

September 2014

Acknowledgements

I would like to thank my advisors Dr. Nikolaos Bellas and Dr. Christos Sotiriou for the great collaboration, the ideas, the inspiring discussions and for their guidance.

I would also like to thank specifically my good friend and colleague Michalis Spyrou for all his help, insight and support during the development of my thesis.

In conclusion, I would like to thank my parents for the support they provided me through my entire life, for all the sacrifices they made on my behalf and for believing in me.

Abstract

Video games always fascinated people of all ages. In recent years, a very large industry has been developed that aims to create video games, which simultaneously has pushed the hardware industry to manufacture more and more specialized components for their reproduction. As a consequence, the handling of video games has escaped conventional ways, for example gamepads, and has progressed to more sophisticated and interactive media, such as Nintendo Wii, Xbox Kinect and more.

The purpose of this thesis is the development of an interactive arcade game in a reconfigurable platform, with hand motion recognition feature using an accelerometer. The game we implemented is Tetris, one of the earliest and most famous arcade video games. The game was implemented in Verilog Hardware Description Language.

Περίληψη

Τα ηλεκτρονικά παιχνίδια πάντα συναρπάζουν ανθρώπους κάθε ηλικίας. Τα τελευταία χρόνια έχει αναπτυχθεί μία πολύ μεγάλη βιομηχανία που έχει σκοπό τη δημιουργία ηλεκτρονικών παιχνιδιών, η οποία παράλληλα έχει ωθήσει τη βιομηχανία υλικού να κατασκευάζει ολοένα και πιο εξειδικευμένα εξαρτήματα για την αναπαραγωγή τους. Ως συνέπεια, ο χειρισμός των ηλεκτρονικών παιχνιδιών έχει ξεφύγει από τους κλασικούς τρόπους, για παράδειγμα gamepads, και έχει προχωρήσει σε πιο εξελιγμένα και διαδραστικά μέσα, όπως Nintendo Wii, Xbox Kinect και άλλα.

Στόχος αυτής της Διπλωματικής Εργασίας είναι η ανάπτυξη ενός διαδραστικού arcade παιχνιδιού σε επαναπρογραμματιζόμενη πλατφόρμα, με δυνατότητα αναγνώρισης κίνησης χεριού για το χειρισμό του, χρησιμοποιώντας αξελερόμετρο. Το παιχνίδι που υλοποιήσαμε είναι το Tetris® , ένα από τα πρώτα και πιο γνωστά arcade ηλεκτρονικά παιχνίδια. Η υλοποίηση πραγματοποιήθηκε στη Γλώσσα Περιγραφής Υλικού Verilog.

Declaration

The work in this thesis is based on research carried out at the University of Thessaly, Electrical and Computer Engineering Department, Greece. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Contents

List of Acronyms

CLB Configurable Logic Blocks

DSP Digital Signal Processor

FPGA Field-Programmable Gate Array

FSM Finite-State Machine

HDL Hardware Description Language

LFSR Linear Feedback Shift Register

LUT Look Up Tables

PAR Place and Route

RAM Random Access Memory

RGB Red Green Blue

VGA Video Graphics Array

XST Xilinx Synthesis Tool

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Purpose of This Thesis

Video games nowadays are more interactive than ever. Game developers take advantage of every contemporary technological feature in order to make video games more and more fascinating. There are many games that detect levers in players' hands, the movement of the hand alone, but some detect players' entire body and each and every move they make or even player's gender. Naturally, all these technologically modern games and features are developed in software, creating impressively accurate and detailed graphics, displaying even the player himself. We would like to develop such a modern and interactive game purely in Hardware Design, using a Hardware Description Language (HDL) to configure a Field Programmable Gate Array and implement our work. However, such precise graphics are impossible to be developed in Hardware Design, since a Graphics Processing Unit is needed to be developed first. Consequently, we chose to recreate an arcade video game in 2D graphics, Tetris®. In order for our video game to be more interactive and modern with a motion recognition feature, we are using accelerometers to control the game, by recognizing player's hand's motion.

1.2 Thesis Structure

This thesis is divided in three main parts.

The first part discusses background issues regarding every aspect of our work. More specifically, section 2.1 deals with FPGA devices, describing their architecture, the way they operate and the technical specifications of the FPGA device we used, ZedBoard™. In section 2.2 we analyze VGA protocol and all the information needed in order to drive a display monitor. section 2.3 talks about the game we developed, Tetris®, its game-play and facts regarding its development and licensing history. Finally, in section 2.4 we explain the way accelerometers work.

In the second part we present our design and its implementation, along with a summary report of FPGA's resources that were used and design tools' execution time. In subsection 3.1.1 we describe the implementation of the VGA driver and in subsection 3.1.2 we explain the game algorithm and how it was developed. subsection 3.1.3 and subsection 3.1.4 analyze the Linear Feedback Shift Register and the calculations required for displaying the score respectively. SUMMARY

Finally, chapter 4 we describe the conclusions that we came to and discuss possible future work.

Chapter 2

Background

In this chapter we describe basic information regarding FPGA technology, the VGA protocol and Arcade Video games for a better understanding of our work.

2.1 FPGA

An FPGA board is an integrated circuit based on tables of configurable logic blocks and designed to be configured using HDL [1]. Although there is the solution of one time programmable FPGAs, most common are FPGAs that can be reconfigured each time the design evolves [2] [3]. It is not restricted to a predetermined hardware function and allows the user to program applications and product features according to the needs of each design. Due to their programmability, FPGAs are ideal for a large variety of markets such as ASIC prototyping, such as Aerospace and Defence, Automotive, Communications, High Performance Computing, Industrial, Medical and Video and Image Processing.

2.1.1 Architecture

Most common FPGAs consist of Configurable Logic Blocks (CLBs), routing channels, SRAMs, Digital Signal Processing (DSP) modules, I/O circuitry and clock management blocks.

The CLB is the basic logic unit in an FPGA. Their number and size vary from device to device, but in general a CLB consists of some logical cells. A typical cell

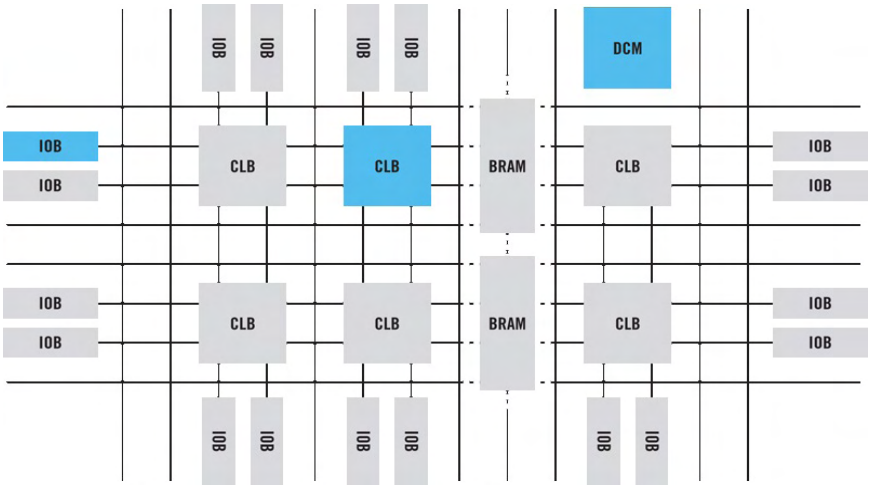


Figure 2.1: *This figure shows a common FPGA's block structure.*

consists of a configurable switch matrix, selection circuitry (Multiplexer (MUX), etc), Look-Up Tables (LUTs), full adders and flip-flops. Depending on the design mode,*normal* or *arithmetic*, the LUTs are either combined into a larger one or feed their outputs to the full adder [4] [5].

The routing channels are responsible for routing the signals between the clock, CLBs, RAMs and I/Os. In order for these routes to be optimal and fast, the routing task is hidden from the user and is completed solely by the tool, applying any optimization needed for the design.

The I/O features of an FPGA vary from device to device. Most of them support USB, video outputs; VGA or/and HDMI, audio lines in and out, Ethernet and connectors for many other features or devices such as cameras, sensors and many more. Digital clock management provides users the ability to manage the original clock generated from an oscillator on the FPGA and create new clocks, with lower or higher frequency.

Most contemporary FPGA devices are equipped with quite powerful processors, which make them suitable for Embedded Systems and Systems on Chip (SoC) development. With these abilities, these devices combine the software programmability of a Processor with the hardware programmability of an FPGA, resulting in outstanding system performance, flexibility and scalability, while also providing the great benefits of power reduction and lower cost.

2.1.2 ZedBoard™

Our work was developed for the ZedBoard™, which uses Xilinx Zynq®-7000 All Programmable SoC 7z020-CLG484. The device is equipped with an ARM® Processor of approximately 900 MHz and with a variety of Hardware Programmable Logic, allowing designers to add peripherals according to the desirable application [6].

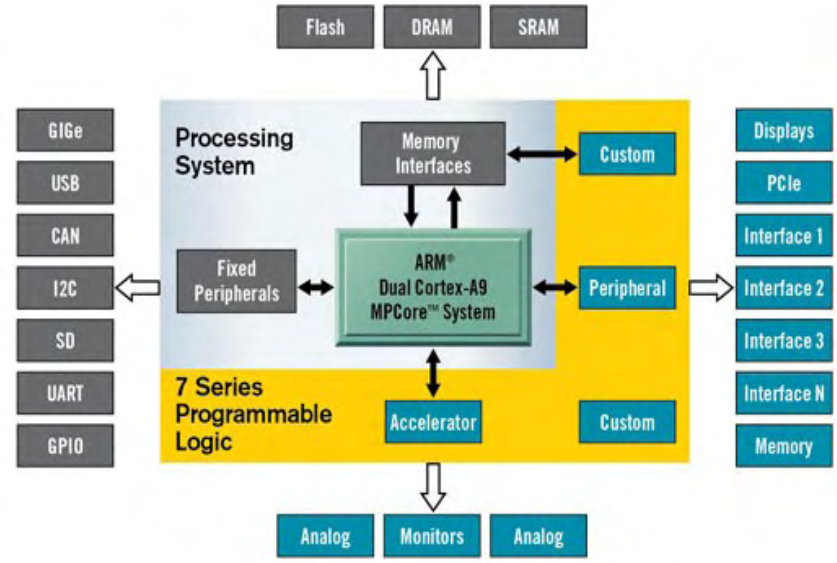


Figure 2.2: *System architecture's block diagram for Zynq-7000 AP SoC.*

The Zynq-7000 AP SoC provides us with optimized programmable logic and great computational capabilities. The device's technical features are provided below.

Device Name	**Z-7020**
Xilinx 7 Series Programmable Logic Equivalent	Artix-7 FPGA
Programmable Logic Cells (Approximate ASIC Gates(4))	85K Logic Cells (1.3M)
Look-Up Tables (LUTs)	53,200
Flip-Flops	106,400
Extensible Block RAM (# 36 Kb Blocks)	560 KB (140)
Programmable DSP Slices (18x25 MACCs)	220
Peak DSP Performance (Symmetric FIR)	276 GMACs

Table 2.1: Programmable logic of ZedBoard™.

2.2 VGA Protocol

VGA is a video standard mainly used for computer monitors introduced by IBM in 1987 and has also come to mean the 15-pin VGA connector or the 640x480 resolution itself, which is most commonly used [7]. VGA video is a stream of frames, where each frame is consisted of horizontal and vertical series of pixels which are transmitted from top to bottom and from left to right, like a beam is traveling through each pixel of the screen.

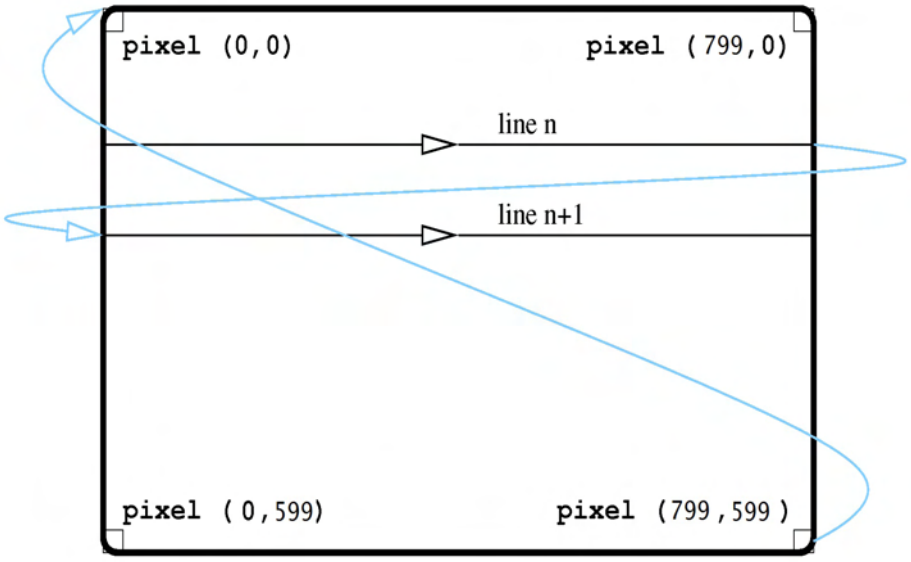


Figure 2.3: *This figure shows the horizontal and vertical inversion process.*

Each line of a frame begins with an active display region, in which RGB values are output for each pixel in the line. Then a blanking region follows in which a horizontal sync pulse is transmitted in the middle of the blanking interval. The interval before the sync pulse is known as front porch, after the sync pulse as back porch and the sync pulse itself as horizontal sync and shows when a full pixel line of the screen has been scanned. Respectively, each frame begins with an active display region, followed by the front porch, the vertical sync pulse and the back porch. Image is only displayed during the active display time and not during the front porch, back porch nor sync time. Depending on the resolution we want to display, hsync and vsync have different polarity and there are different pixel clocks, according to which each region has different timings [8].

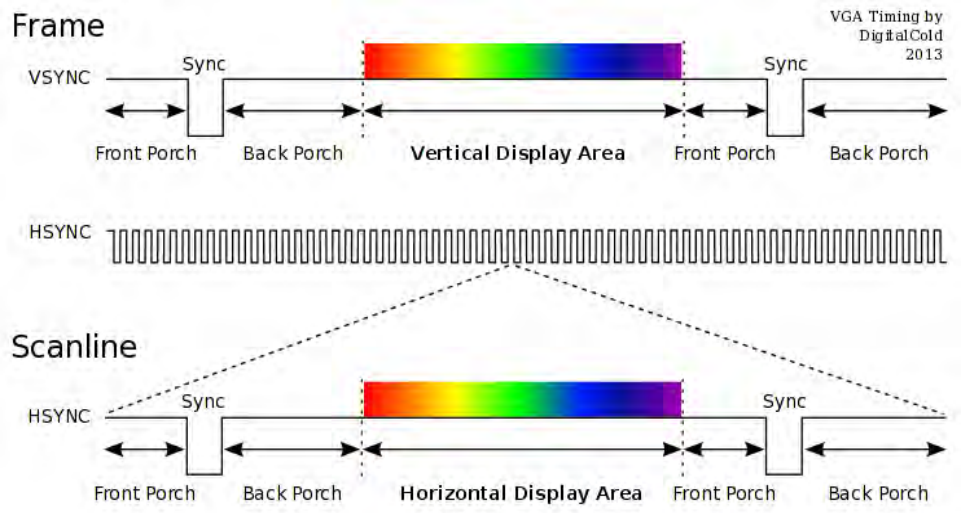


Figure 2.4: *The waveforms of hsync and vsync, which are identical regardless resolution.*

Each pixel's colour is a combination of red, green and blue, the size of which depends on the output device. When the colour of each pixel and all the other signals are set properly from the VGA controller, they are driven from the VGA Digital-to-Analog Converter to the correct pins of the connector. The connector consists of 15 pins. Six pins are used for the colours (RGB), and their respective ground signals two for the hsync and vsync signals, two for grounds and the remaining five are not used.

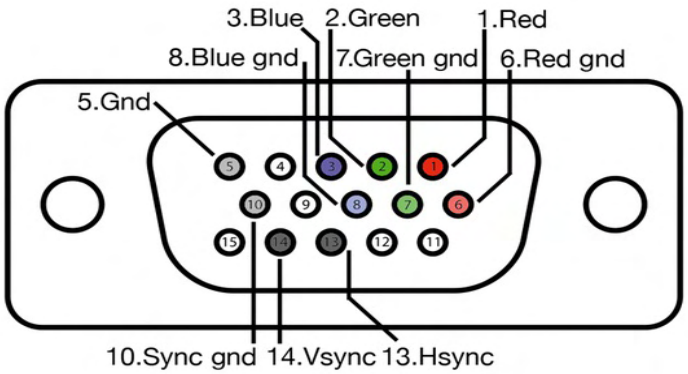


Figure 2.5: *The pins of the VGA connector (view from board side).*

2.3 Arcade Video Games

An arcade game is a coin-operated entertainment machine, usually installed in public businesses, such as restaurants, bars, and particularly amusement arcades. Most arcade games are video games, pinball machines, electro-mechanical games, redemption games, and merchandisers. The term "arcade game" is also, in recent times, used to refer to a video game that was designed to play similarly to an arcade game with frantic, addictive game-play. The golden age of arcade video games lasted from the late 1970s to the late 1990s. Arcade games saw a continuous decline in popularity around the world when home-based video game consoles made the transition from 2D graphics to 3D graphics. [9].

One of the few games that achieved ultimate popularity was Tetris®.

2.3.1 Tetris Game-play

Tetris® is a puzzle video game, where the objective is to manipulate random blocks that fall down the playing field, by moving them sideways and rotating them by 90 degrees, in order to create completed lines at the bottom of the playing field. When such a line is created, it disappears and all the above blocks fall to the bottom. For every ten lines that are cleared, the level increases and each new level makes the blocks fall faster. The game is over when the blocks are stacked up to the top of the playing field and no new blocks can be created [10].

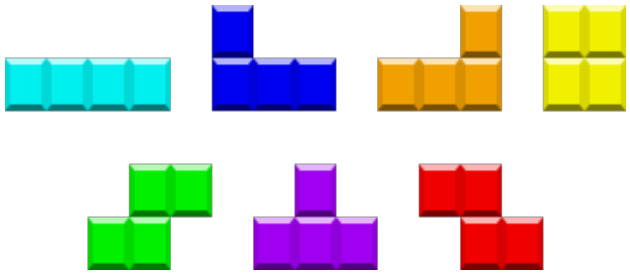


Figure 2.6: *The Tetriminos in their colours.*

These blocks, called *Tetriminos*, are created by four tiles which are combined in different ways to create each shape. Each Tetrimino is symbolized by a letter from

the English alphabet closer to its shape and has a specific color. Thus, we have I in cyan, J in blue, L in orange, O in yellow, S in green, T in purple and Z in red, as shown in Figure 2.5. All Tetriminos are able to clear single and double lines. I, J and L are able to clear triples and only the I Tetrimino is able to achieve a four-line clear, which is called "TETRIS". Depending on the level and the number of lines cleared, different points are awarded to the player.

Level / **Lines**	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Single	40	80	120	160	200	240	280	320	360	400
Double	100	200	300	400	500	600	700	800	900	1000
Triple	300	600	900	1200	1500	1800	2100	2400	2700	3000
TETRIS	1200	2400	3600	4800	6000	7200	8400	9600	10800	12000
For each level n greater than 9, the score is: *(n + 1)*40, (n + 1)*100, (n + 1)*300, (n + 1)*1200.*										

Table 2.2: The scoring system of Tetris® for each level and number of cleared lines.

When a number of lines are cleared, the above Tetriminos fall down the exact same distance to the cleared lines height. Contrary to the laws of gravity, this feature may leave blocks floating above gaps instead of falling all the way to the bottom as shown in Figure [11].

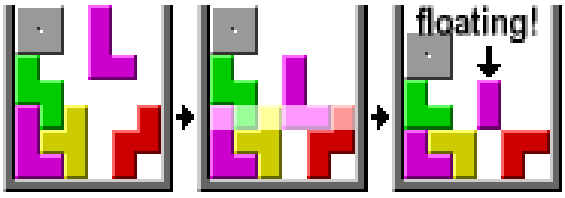


Figure 2.7: *Tiles of Tetriminos floating.*

2.3.2 History

Tetris® was introduced on June 6 1984 by Alexey Pajitnov, an artificial intelligence researcher working for the Soviet Academy of Sciences at their Computer Center in Moscow. Being responsible for testing the capabilities of new hardware, Alexey Pajitnov would create simple games in order to do so. The initial idea of Tetris® was the creation of a game around pentominoes [12], like many puzzle games he enjoyed as a child, but simpler since the variety of the shapes would make the game very complicated. Thus, instead of pentominoes he switched to tetrominoes, made of four tiles and creating only seven different shapes. The name of this new game *Tetris*, comes from the prefix tetra of the game's blocks and from tennis, which was Pajitnov's favourite sport. Since the Elektronika 60 that he was working on, supported only text based display, tetrominoes were initially formed of letter characters [13] [14].

Figure 2.8: *The very first version of Tetris®.*

Pajitnov's game was quite popular among his colleagues and along with Dmitry Pavlovsky and Vadim Garasimov, they ported the game to the IBM PC, which contained background graphics featuring Russian scenes. This version of the game, made its way to Budapest, Hungary, where it was ported to many different platforms and was noticed by the British software house Andromeda. While they made attempts to contact Alexey Pajitnov for acquiring the rights to the PC version of the game and before the deal was settled, the rights had already been sold to Spec-

trum HoloByte and Andromeda attempted to acquire license of this version from the Hungarian programmers.

Soon enough the same PC version acquired from Spectrum Holobyte made its way to the United States, where it became instantly popular and Computer Gaming World called the game "deceptively simple and insidiously addictive". Although the licensing issues were still unsolved, many new versions became available from Andromeda, Microsoft and Spectrum Holobyte. Unsure of how to publish the game, Pajitnov gave the rights to the Soviet government for ten years, which in 1988 began to market the rights to Tetris®.

Figure 2.9: *The picture that was on the front side of Tetris®'s packaging box.*

By 1989, many different companies claimed rights to create and distribute the Tetris software for home computers, game consoles, and handheld systems. In the meantime, Elorg organization signed the rights of the arcade version over to Atari and the non-Japanese console and handheld rights over to Nintendo. Tetris® was on show at the January 1988 Consumer Electronics Show in Las Vegas and from then on, Tetris® was bundled with every Game Boy.

Tengen, Atari's console software division, applied for copyright for their Tetris game

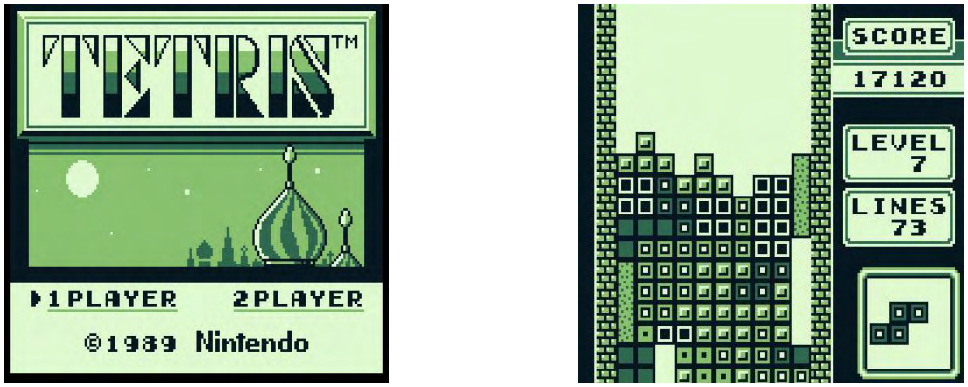


Figure 2.10: *Nintendo's version of Tetris® for Game boy: (a) The opening screen and (b) a screen-shot while playing.*

for the Nintendo Entertainment System and proceeded to market and distribute it under the name TETRIS: The Soviet Mind Game, disregarding Nintendo's license from Elorg. From then the lawsuits between Tengen and Nintendo over the NES version carried on until 1993.

2.4 Accelerometers

An accelerometer is an electromechanical device that measures acceleration forces. These forces may be static, like the constant force of gravity, or they could be dynamic caused by moving the accelerometer. There are different types of accelerometers depending on how they work. Some accelerometers use the piezoelectric effect; they contain microscopic crystal structures that get stressed by accelerative forces, which cause a voltage to be generated. Others implement capacitive sensing, that give as output a voltage dependent on the distance between two planar surfaces.

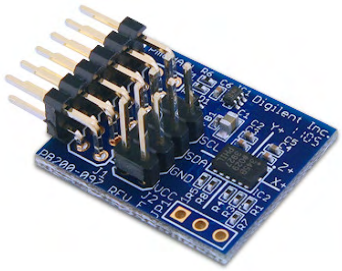


Figure 2.11: *The figure shows a Pmod 3-axis accelerometer.*

Chapter 3

Design and Implementation

In this chapter we introduce the design and implementation of our work; the implementation of the Tetris® game in an FPGA device purely in hardware using an HDL like Verilog. We describe how each module operates and the outputs that each one provides, but also all the essential optimizations for reducing XST and PAR execution time and area occupancy. Finally, we present the schematic design of the project and the summary reports from XST and PAR that the Xilinx ISE Tool provides.

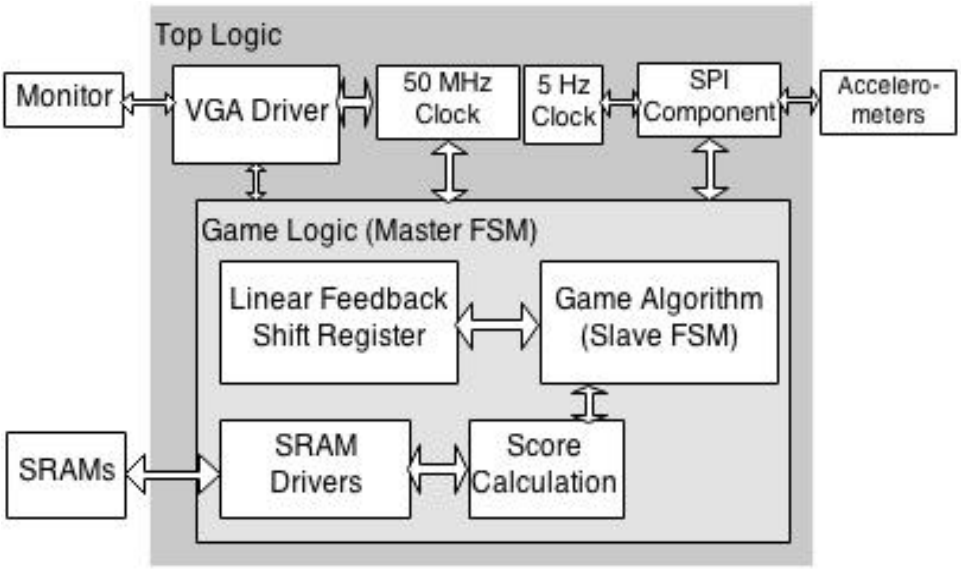


Figure 3.1: *This figure shows the block diagram of the project.*

3.1 Tetris Game

The implementation of this project was for ZedBoard Zynq 7z020 and consists of the VGA driver, the creation of a 50 MHz pixel clock and the main game logic, which controls the game algorithm, the Linear Feedback Shift Register for generating random blocks and the drivers for displaying images from block RAM. The game algorithm contains the movement and collision detection algorithm, the rotation algorithm and the completed lines detection and delete algorithm.

3.1.1 VGA Driver

For the 800x600 resolution that we used, a 50 MHz pixel clock is required and since Zynq 7z020 FPGA board oscillator provides an 100 MHz clock, we created a very simple frequency divider.
The timings for synchronising the display correctly are shown in Table 3.1.

General Timing	
Screen refresh rate	72 Hz
Vertical refresh	48.076923076923 kHz
Pixel freq.	50.0 MHz

Horizontal Timing (Line)			**Vertical Timing (Frame)**		
Scanline part	Pixels	Time [µs]	Frame part	Lines	Time [ms]
Visible area	800	16	Visible area	600	12.48
Front porch	56	1.12	Front porch	37	0.7696
Sync pulse	120	2.4	Sync pulse	6	0.1248
Back porch	64	1.28	Back porch	23	0.4784
Whole line	1040	20.8	Whole frame	666	13.8528
Polarity of hsync pulse is positive.			Polarity of vsync pulse is positive.		

Table 3.1: VGA Timings for 800x600 resolution.

The VGA driver that we implemented, is composed of two counters, one that counts the pixels of each line and one that counts the lines of the frame. As we can see above, the hsync pulse should be asserted 120 pixels after the front porch, that is including pixel zero (0) at the 975th pixel. Accordingly, the vsync pulse should be asserted at the 642nd line of the frame. At the end of each line, horizontal counter is zeroed and at the end of each frame, vertical counter is zeroed.

The Zynq 7z020 FPGA board that we used has an RGB output of 12 bits, that is 4 bits Red, 4 bits Green and 4 bits Blue, therefore a total of 4095 colours. Each foursome from each colour, as well as horizontal and vertical sync pulses, are driven to the corresponding pins of the VGA connector from the appropriate pins of the FPGA as shown in Table 3.2. In order to obtain a 50 MHz frequency clock from the

VGA Pin	Signal	Description	EPP Pin
1	RED	Red video	V20, U20, V19, V18
2	GREEN	Green video	AB22, AA22, AB21, AA21
3	BLUE	Blue video	Y21, Y20, AB20, AB19
4	ID2/RES	formerly Monitor ID bit 2	NC
5	GND	Ground (HSync)	NC
6	RED_RTN	Red return	NC
7	GREEN_RTN	Green return	NC
8	BLUE_RTN	Blue return	NC
9	KEY/PWR	formerly key	NC
10	GND	Ground (VSync)	NC
11	ID0/RES	formerly Monitor ID bit 0	NC
12	ID1/SDA	formerly Monitor ID bit 1	NC
13	HSync	Horizontal sync	AA19
14	VSync	Vertical sync	Y19
15	ID3/SCL	formerly Monitor ID bit 3	NC

Table 3.2: VGA Connector and FPGA Pins. [15]

100 MHz clock of the oscillator we needed a frequency divider. Since the IP-Core version of Digital Clock Management for the Zynq 7z020 was quite time consuming during the execution of XST and PAR, reaching almost one hour, we had to find an alternative solution. The simplest alternative was to create a module, where for every assert of the 100 MHz clock signal, the signal of the 50 MHz clock toggles. The source code can be seen in Listing A.1.

3.1.2 Game

Master FSM

The game is controlled by a master FSM consisted of four states as shown in Figure 3.1.

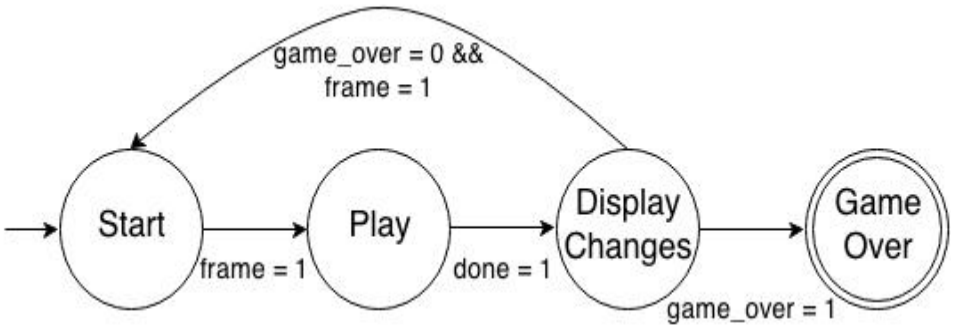


Figure 3.2: *This figure shows the master FSM that controls the game.*

During each state signals are asserted to control different functionalities and activate states of other FSMs.

Start

The image for the background is read from the Block RAM and driven to the monitor to be displayed and signal new_block is asserted in order for a new Tetrimino to be created. When frame is asserted after one second we move to state Play.

Play

Signal move is asserted in order to activate movement for the Tetrimino that

was created and the background image along with the new Tetrimino are driven to the display. When signal done is asserted we move to state DisplayChanges

DisplayChanges

The background image, the Tetrimino that reached bottom and changes such as the deletion of a line or lines, are driven to be displayed. When frame is asserted and game_over is not we return to state Start since we can keep on playing and the game is not over. If game_over is asserted we move to state GameOver.

GameOver

If the game is over, a picture with the according message is read from the Block RAM and driven to be displayed.

The source code can be seen in Listing A.2.

Game Algorithm

The master FSM controls and communicates with a secondary FSM which is the game algorithm that is responsible for the Tetriminos' movement and rotation, for collision detection, for detecting completed lines and deleting them. The same module displays the falling Tetrimino and the next one to come.

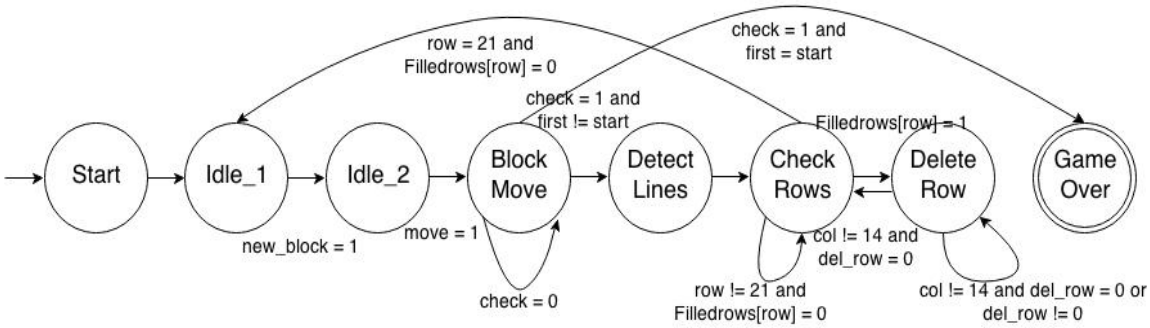


Figure 3.3: *This figure shows the slave FSM that controls the game.*

In order to be able to display the Tetriminos, detect collision but also display all the fallen Tetriminos we needed a collision buffer, a frame buffer and the images of all Tetriminos in Block RAMs. Since this was not a good design technique, we use only

one buffer for all the above. The buffer is a register of 368 words of 3 bits and represents a grid of the playing field surrounded by the walls and bottom that restrict Tetriminos' moves. Before the game begins the buffer is initialized to zeroes and to non-zero values at walls' and bottom's positions. Each Tetrimino is created by four grids, which are represented by four variables; first, sec, third and fourth. When a Tetrimino is created, its initial positions in the grid are written with its colour code in order to be displayed. An other optimization that was essential is that the codes of colours written in the buffer are not the actual hexadecimal colours, but each one of them correspond to a 3 bit number from one to seven. Thus, the displayed result and the equivalent state of the 14x23 buffer, are shown in Figure 3.3.

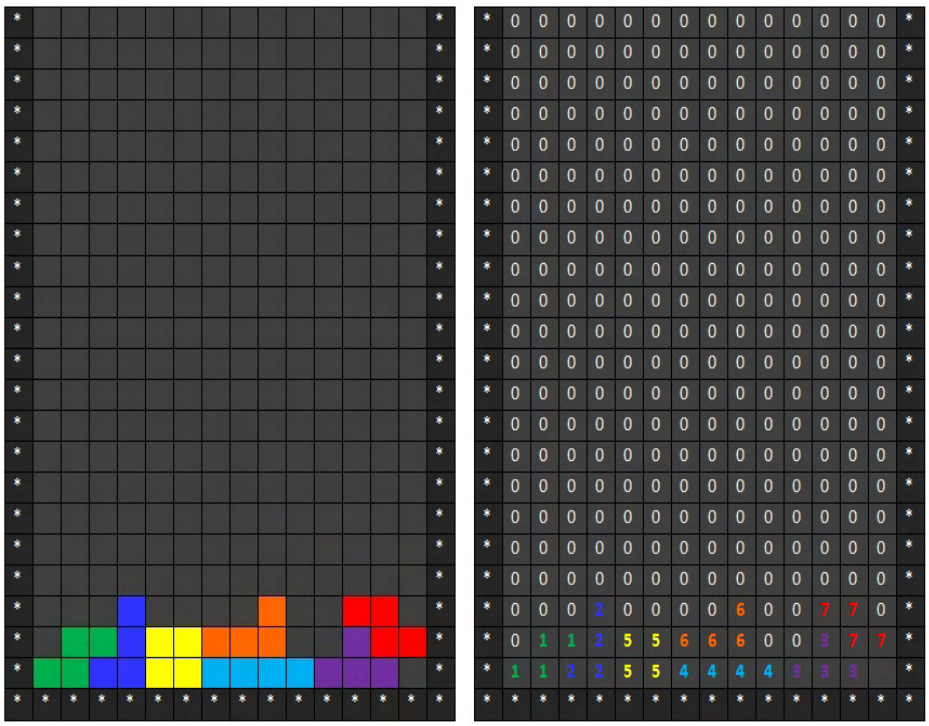


Figure 3.4: *This figure shows the displayed image and the actual changes of the buffer.*

For Tetriminos' movement, it is essential first to check that all future positions in the grid are not occupied by an other fallen Tetrimino or a wall. Therefore, as Tetriminos' fall, if the positions they are about to move in are zero then the move is completed, the previous positions are zeroed and the new ones are written with the corresponding colour code. This regards all possible movements, that are moving

left, right, falling down and rotating. All these moves are calculated through the four position variables first, sec, third and fourth. For left and right movement we have to check the previous and the next positions of the most left and the most right grids of moving Tetrimino and add -1 or +1 to the variables. For the falling movement we have to check the positions located below the bottom Tetriminos' grids in the next rows and add +16 for gravity falling or +16 again for moving down, but with a faster refresh rate of the buffer. Finally, Tetriminos rotate 90 degrees clockwise and their next positions depend on the Tetrimino and the previous rotation, thus they are calculated and certain values are added to the variables. Except for *O*, all other Tetriminos have four different rotation states as shown in Figure 3.4.

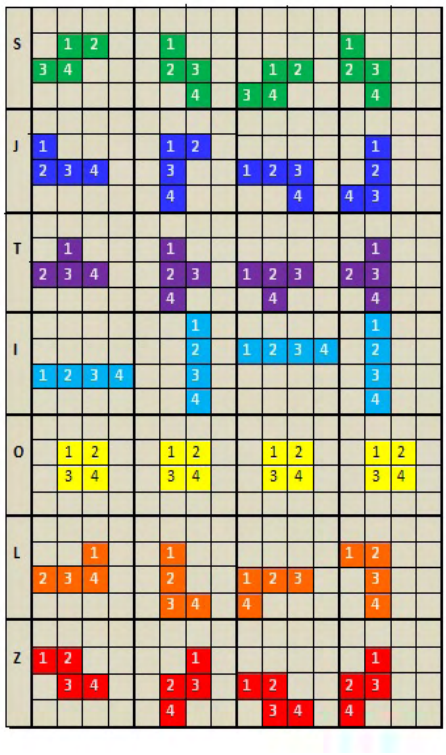


Figure 3.5: *This figure shows Tetriminos' different rotations.*

As Tetriminos are created, moved and rotated inside the playing field, they are finally placed on the bottom of the field. If a line is completed, it must be removed. Thus, we scan the buffer, from top to bottom and assign to a register of 22 positions zero if the line is not full and one if the line is full. Subsequently, we check the register of completed and non-completed lines from top to bottom. When a line is

full and must be removed, starting from that line and moving to the top of the buffer we replace the contents of each line with the contents of the previous one. This is repeated for every completed line in the buffer. Since there are no more completed

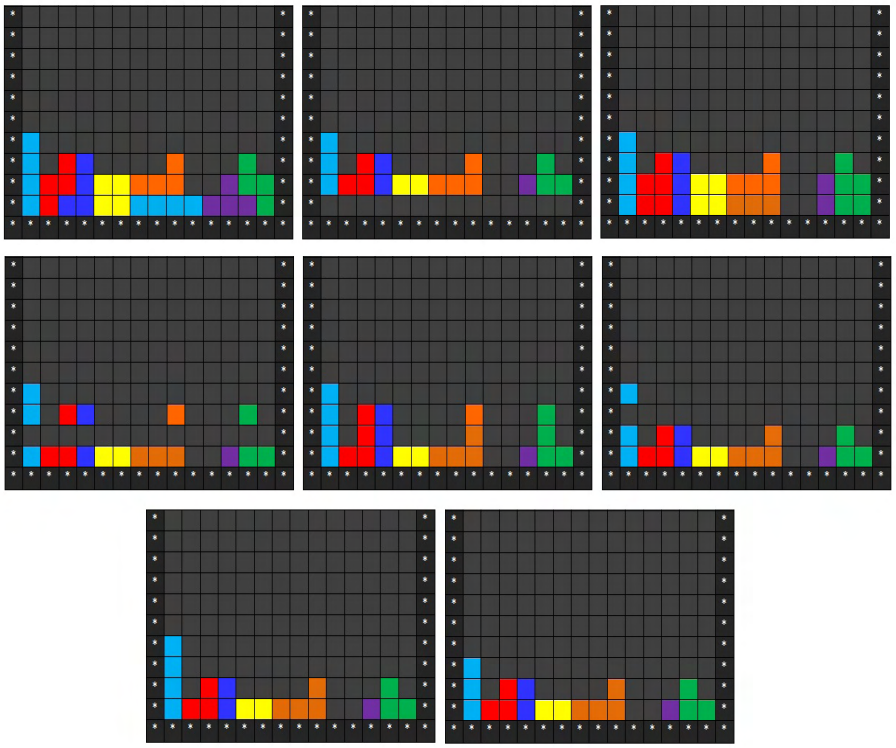


Figure 3.6: *This figure shows how completed lines are removed and their contents replaced appropriately.*

lines to delete and all necessary changes have been made, a new Tetrimino is created and the above process is repeated. The game ends when Tetriminos are stacked up to the top of the playing field and new ones can not be created. The "Game Over" message is displayed on screen as shown in Figure 3.7.

Figure 3.7: *This figure shows the game over message at the end of the game.*

Slave FSM

Each state of the slave FSM executes a specific part of the process described above as follows:

Start

All signals and buffers are initialized to their initial state.

Idle_1

When a new Tetrimino is created an LFSR determines which one it will be and also determines which one will come next.

Idle_2

Depending on the Tetrimino, its initial positions and colour code are defined. Also a variable keeps the initial position in order to detect game over. When signal move is asserted we move to state BlockMove.

BlockMove

Tetriminos move towards the bottom approximately one row every second. As level increases, so does the speed of Tetriminos. The calculations for each movement are activated with a corresponding button. If a button is pushed and the movement is legitimate, the values of first, sec, third and fourth are reduced or increased at a certain amount depending on the movement and the next positions. For moving left and right, all four of them are reduced by one and increased by one respectively. For simple falling due to gravity, all four of them are increased by sixteen and that is because the width of the playing field is fourteen grids but we have to include two more for the walls. Thus, for moving downwards and not simply falling same distance is covered, but with a faster rate, so all four of them are increased by sixteen. For the rotations, each one of the variables is either increased or reduced in order to achieve the 90 degree rotation according to Figure 3.5.As a better design technique and since the changes are numerical according to how many grids does each variable needs to be moved, they are summed up and added to the four variables. When signal check is asserted, which means that our active Tetrimino had an impact and is not able to move downwards anymore, it has

either reached bottom or an other Tetrimino. Therefore, we either resume the game and check for completed lines or the game is over if Tetriminos are stacked to the top and the appropriate message appears.

DetectLines

The buffer is scanned for completed lines. For every completed line the corresponding location of FilledRows register is assigned to one and for non-completed rows to zero.

CheckRows

During this state register FilledRows is scanned. When a completed line is found we move on to the deletion at state DeleteRows. Otherwise, and if we haven't reached the end of the register, the scanning continues. Eventually, by reaching the end of the register and if the last line is not full, we move back to state Idle_1 and game flow is resumed.

DeleteRow

Since a row is completed, we not only have to delete the whole row, but also move all the above rows downwards. Hence, the values of our completed row need to be replaced with the values of the previous row. Each grid of the row is written with the value of the above grid. At the end of the row we repeat the process for the previous line in the buffer but actually the next one as we move to the top. Finally, when we reach the top and we are at row zero, since there are not other lines above and the values can not be replaced, the whole row is initialized to zero.

GameOver

Signal game_over is activated and Game Over image is displayed.

The source code can be seen in Listing A.3.

In order for the contents of the buffer to be displayed and using the horizontal counter of pixels and the vertical counter for lines, we calculated the actual pixel coordinates of each grid of the playing field and determined a specific address for

each one. Hence, as the screen display is scanned and an address is assigned, if the corresponding position of the grid contains one of the colour codes, the appropriate colour is displayed.

3.1.3 Linear Feedback Shift Register (LFSR)

During the original game Tetriminos are created randomly, without following a distinctive pattern, but in reality nothing is absolutely random. Thus, in order to create Tetriminos in a seemingly random way we needed a pseudo-random number generator. The ideal pseudo-random number generator would use as seed outer parameters, such as time. Since our work is purely in hardware design and such parameters is not able to be used, we used an LFSR. LFSR is a shift register whose input bit is a linear function of its previous state. The most commonly used linear function of single bits is exclusive-or (XOR), therefore its input is driven by the XOR of some bits of the overall value of the shift register [16] [17].

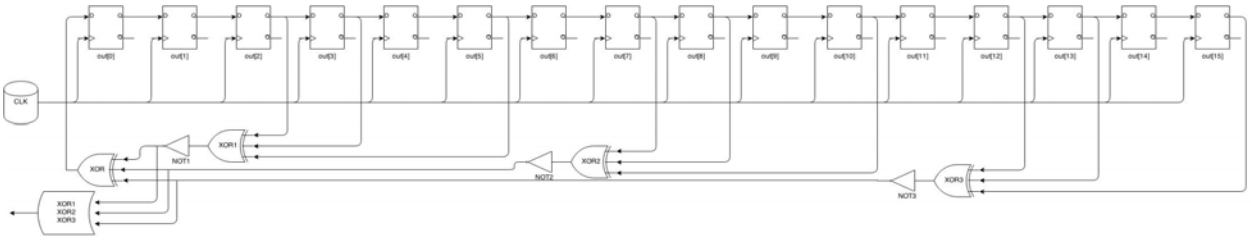


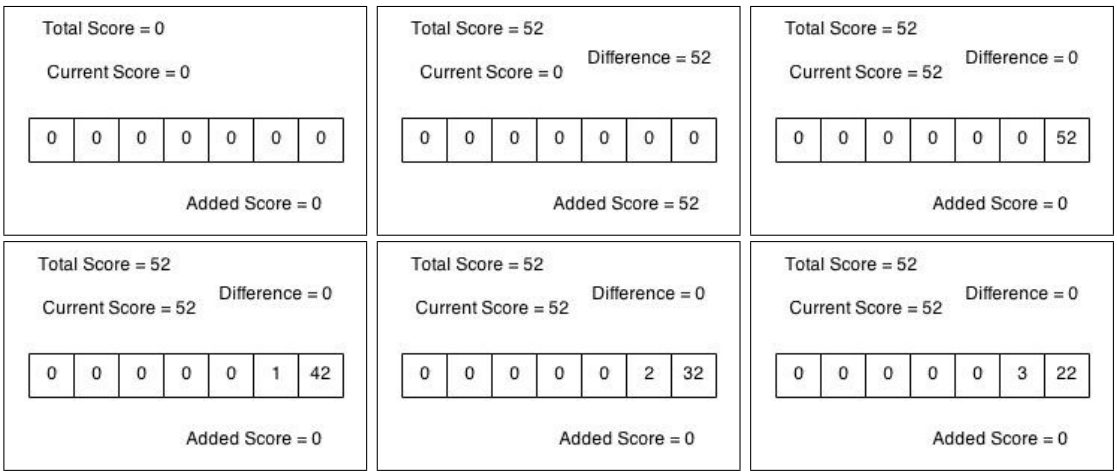
Figure 3.8: *This figure shows the linear feedback shift register that was used.*

Since Tetriminos are seven, we needed to compose three random bits in order to create all seven types. In order to obtain a less frequent pattern, we used a 16-bit LFSR with three XORs, which inverted give us our random numbers and a fourth one that drives the input each time a new Tetrimino is created as we can see in Figure 3.8. The source code can be found in Listing A.4.

3.1.4 Score, Completed Lines and Level Display

Points are awarded to the player according to the scoring system seen in Table 2.2. When a number of lines is erased, line counter is increased accordingly and for every

ten lines erased, level counter is increased by one. Level counter reaches up to number nine, which means we only need one digit to represent it. Images of the numbers from zero to nine are loaded in Block RAMs. For the ten possible values of level, each Block RAM is instantiated and according to the value of level the proper image is displayed. In order for score and completed lines to be displayed, we needed seven and five digits respectively. Consequently, from a value of seven or five digits, we needed to isolate each digit and display each one separately, thus we use variables to represent each digit. In the end, all together composed, form the entire sum. For each one of the two parameters there are two counters; one that keeps the total amount and one that keeps the current amount. When points are awarded or in the other case lines are erased, the total amount is increased and the difference between these two counters is not zero and the new sum must be calculated and displayed. The amount of difference is added to the value of units' digit and if its value is greater than nine, it is decreased by ten and dozens' digit is increased by one. This process is repeated for every digit and is continued until the digit has a maximum value of nine. Since we are not able to use one single instantiation of each number in Block RAM, as it is not possible to access Block RAM by multiple drivers, the instantiations for each number need to be as many as the digits in use.



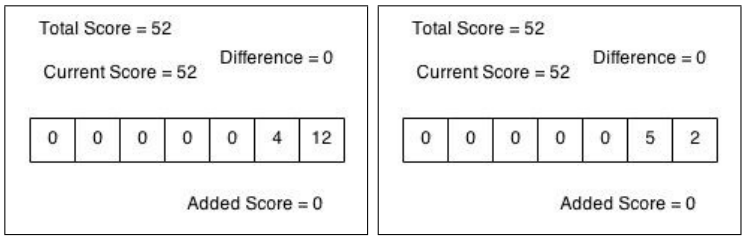


Figure 3.9: *This figure shows the way each digit of the score is isolated to be displayed.*

3.1.5 Accelerometers

In our implementation we used a 3-axis digital accelerometer, powered by the analog device ADXL345 and took advantage of the force of gravity on x and y axises, making Tetriminos move sideways by tilting the accelerometer right or left and down by tilting the accelerometer towards the floor. We connected the accelerometer through the SPI interface. SPI operates in full duplex mode and uses four signals: Slave select (SS), serial clock (SCLK), serial data out (SDO), to the accelerometer and serial data in (SDI), from the accelerometer. Devices communicate in master-slave mode, where master initiates the data frame.

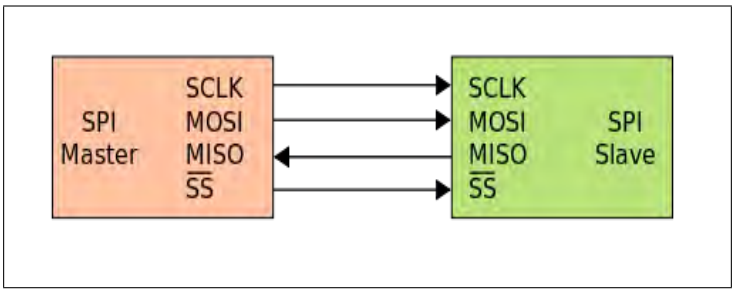


Figure 3.10: *This figure shows the master-slave communication for the SPI interface.*

Our setup contains two shift registers, one in the master and one in the slave and they are connected as a ring. Data is shifted out with the most significant bit first, while shifting a new least significant bit into the same register.

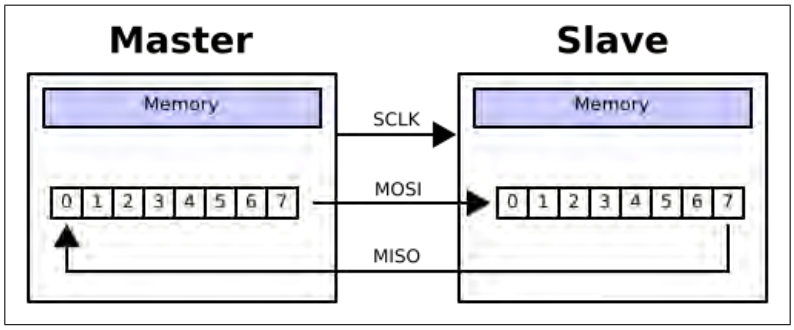


Figure 3.11: *This figure shows the SPI 8-bit circular transfer between the two shift registers.*

We initialize the transfer with a 5 Hz clock and we transmit and receive data at 22.4 kHz rate. The accelerometer is configured for +/- 2g operation. To convert the output to g we have to find the difference between the measured output and the zero-g offset and divide it by the accelerometer's sensitivity, which is expressed in counts/g or LSB/g. For our accelerometer in 2g sensitivity with 10-bit digital outputs, the sensitivity is 163 counts/g or LSB/g. The acceleration would be equal to: $\alpha = \frac{(Aout-zerog)}{163}g$. However, we did not calculate the acceleration as described above. We simply used the accelerometer raw output in order to move Tetriminos according to Table 3.3.

		Raw Value		
Axis	G Value	From	To	Movement & Tilting
+y Axis	0 through +0.5	0	175	Left
-y Axis	-0.5 through 0	250	375	Right
+x Axis	-0.5 through 0	250	375	Down

Table 3.3: Movements according to accelerometer's outputs.

3.2 Summary Report

The tool that was used for the development of our work, Xilinx ISE Design Suite, provides us with a summary report regarding slices, LUTs and generally how much

of the available logic was used. As it is observable in Table 3.4, apart from a small proportion that used device's RAMs, the rest of the project is entirely in hardware logic.

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	3,599	106,400	3%
Number of Slice LUTs	24,988	53,200	46%
Number of occupied Slices	7,855	13,300	59%
Number of RAMB36E1/FIFO36E1s	12	140	8%
Number of RAMB18E1/FIFO18E1s	132	280	47%

Table 3.4: Xilinx ISE summary report.

The above hardware logic that is occupied, corresponds to:

RAMs	: 2
Multipliers	: 4
Adders/Subtractors	: 93
Registers	: 1356
Comparators	: 102
Multiplexers	: 7350
FSMs	: 8
Xors	: 8

3.3 Design Issues

The basic design issue that we encountered during this project was the creation of a frame buffer; a buffer that would keep the position and the colours of Tetriminos that had reached bottom. The initial idea was to use a buffer in order to detect collision and the frame buffer. Since both of them were registers of more than three hundred addresses, but also there was the issue of synchronizing them, as when a Tetrimino reached bottom there should be a signal that activates the frame buffer to be written. For the frame buffer to be updated, a whole frame of the display should

be scanned. A frame buffer that keeps the RGB value for each pixel of a frame would be enormous, therefore we needed an optimization. The frame buffer would keep the RGB values of 26x26 pixel grids by scanning the display and keeping the RGB value of the center pixel of each grid. Although this was an optimal solution regarding area, it did not have the expected results, but also combined with the collision buffer occupied a large proportion of LUTs. An other attempt to solve this issue was the use of Block RAM instead of register for the frame buffer. Since we wanted at each frame the frame buffer to be read and displayed, even when its values were updated, Block RAM was not an efficient solution. Finally, we decided to combine the two buffers in one, without using Block RAMs for displaying Tetriminos and displaying them directly from the buffer according to a colour code for each one. Each time the falling Tetrimino moves, the buffer is updated and the updated values are displayed instantly on the monitor.

Chapter 4

Conclusion and Future Work

These days video games are developed with great and detailed graphics, requiring very efficient manipulation of memory and image processing, but also much power in order to be displayed. More and more technologically improved game consoles enter the markets, promising highly effective capabilities and the most contemporary interactive features.

We developed an arcade game in an FPGA device, applying many optimizations in order to occupy minimum area and for the minimum execution time of Synthesis and Place & Route tools. For our game to be modern and interactive, we used accelerometers to control the game by recognizing hand motion. In conclusion, our hardware design implementation in FPGA, requires low power since there are no cooling issues and the only thing that needs to be supplied with power is the FPGA chip.

However, there are some more features that we would like to address in the future.

Firstly, we would like to add sounds and music feature as the original Tetris® game has. Different sounds would be generated when Tetriminos move right, left or down, rotate and when lines are completed.

An other feature we would also like to include in our future work is hand gesture recognition to control the game. This could be achieved using a camera to recognise the player's hand gestures, making our game even more interactive.

Bibliography

[1] http://en.wikipedia.org/wiki/Field-programmable_gate_array, ***Field-Programmable Gate Array.***

[2] http://www.altera.com/products/fpga.html, ***Altera®: What is an FPGA?***

[3] http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm, ***Xilinx Inc©: What is an FPGA?***

[4] http://en.wikipedia.org/wiki/Field-programmable_gate_array#Architecture, ***Field-Programmable Gate Array: Architecture***

[5] http://www.xilinx.com/fpga/index.htm, ***Xilinx Inc©: Common FPGA features.***

[6] http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/use-cases-and-markets/index.htm, ***Xilinx Zynq®-7000 All Programmable SoC***

[7] http://en.wikipedia.org/wiki/Video_Graphics_Array, ***Video Graphics Array.***

[8] http://tinyvga.com/vga-timing, ***VGA Signal Timing***

[9] http://en.wikipedia.org/wiki/Arcade_game, ***Arcade Game.***

[10] http://en.wikipedia.org/wiki/Tetris#Gameplay, ***Tetris: Game-play***

[11] http://en.wikipedia.org/wiki/Tetris#Gravity, ***Tetris: Gravity***

[12] http://en.wikipedia.org/wiki/Pentomino, ***Pentomino***

[13] http://en.wikipedia.org/wiki/Tetris#History, ***Tetris: History***

[14] http://www.play-tetris.net/tetris-history.html, ***Tetris History***

[15] http://www.zedboard.org/sites/default/files/ZedBoard_HW_UG_v1_1.pdf, ***ZedBoard (Zynq™ Evaluation and Development) Hardware User's Guide***

[16] http://en.wikipedia.org/wiki/Linear_feedback_shift_register, ***Linear Feedback Shift Register***

[17] http://rijndael.ece.vt.edu/schaum/slides/ddii/lecture6.pdf, ***A Random Number Generator in Verilog, A Design Lecture***

Appendix A

Source Code

Listing A.1: 50 MHz Clock

```
module ClkDiv_50MHz(
    CLK,
    CLKOUT
);
  input     CLK;    // 100MHz onboard clock
  output      CLKOUT;  // New clock output 50 MHz
  reg   CLKOUT = 1'b0;

  always @(posedge CLK)
  begin
    CLKOUT <= ~CLKOUT;
  end

endmodule
```

Listing A.2: Master FSM

```
/* The FSM that controls the main logic of the game */
  always @( * )
  begin
```

```
4     pixel = pixel_board;
5     NextState = State;
6     move = 1'b0;
7     new_block = 1'b0;
8
9     case (State)
10      /* First the block to be shown is found pseudorandomly, *
11       * we initialize the Collision Buffer correctly and move*
12       * to the next state.                                   */
13      Start :
14        begin
15          new_block = 1'b1;
16          pixel = pixel_grid | pixel_board | frame_pixel | pixel_score;
17          if (frame)
18          begin
19            NextState = Play;
20          end
21        end
22      Play :
23        begin
24          move = 1'b1;
25          pixel = pixel_grid | pixel_board | frame_pixel | pixel_score;
26          if (done)
27            NextState = DisplayChanges;
28        end
29      DisplayChanges :
30        begin
31          pixel = pixel_grid | pixel_board | frame_pixel | pixel_score;
32          if (frame && !game_over)
33          begin
34            NextState = Start;
35          end
36          else if (game_over)
```

```
37          begin
38            NextState = GameOver;
39          end
40        end
41      GameOver:
42        begin
43          pixel = pixel_gameover;
44        end
45      default:;
46    endcase
47  end
```

Listing A.3: Slave FSM

```
1  /* FSM for the movement, rotation, completed *
2   * row check and completed row delete   */
3  case (State)
4    /* First initialize every signal */
5    Start :
6      begin
7        filled = 1'b0;
8        done = 1'b0;
9        check = 1'b0;
10       new_row = 5'd0;
11       new_col = 4'd1;
12       new_d_row = 5'd0;
13       diff_d = 9'b0;
14       diff_s = 9'b0;
15       color = 3'b000;
16       /* Initialize the CollisionBuf */
17       for (i = 0; i < 23; i = i + 1)
18       begin
19         CollisionBuf_new[i*16] = 1'b1;
20         CollisionBuf_new[i*16 + 15] = 1;
21         for (j = 1; j < 15; j = j + 1)
22         if (i == 22)
23           CollisionBuf_new[i*16 + j] = 3'b111;
24         else
25           CollisionBuf_new[i*16 + j] = 3'b000;
26       end
27       /* Initialize the FilledRows */
```

```
28        for (i = 0; i < 22; i = i + 1)
29        begin
30          FilledRows_new[i] = 1'b0;
31        end
32        NextState = Idle_1;
33      end
34    /* Idle state until a new block is created */
35    Idle_1 :
36      begin
37        if (ok)
38          lines_delfsm = 3'b0;
39        if (new_block)
40        begin
41          tetromino_new = next_tetromino;
42          next_tetromino_new = next_block;
43          NextState = Idle_2;
44        end
45      end
46    /* Depending on the Tetrimino assign the initial *
47    * values of first, sec, third, fourth and colour *
48    * code for the buffer            */
49    Idle_2 :
50      begin
51        case(tetromino)
52          S :
53            begin
54              start_n = 9'd8;
55              first_new = 9'd8;
56              sec_new = 9'd9;
57              third_new = 9'd23;
58              fourth_new = 9'd24;
59              color = 3'b001;
60            end
61          J :
62            begin
63              start_n = 9'd7;
64              first_new = 9'd7;
65              sec_new = 9'd23;
66              third_new = 9'd24;
67              fourth_new = 9'd25;
68              color = 3'b010;
69            end
70          T :
71            begin
72              start_n = 9'd8;
73              first_new = 9'd8;
74              sec_new = 9'd23;
```

```
75            third_new = 9'd24;
76            fourth_new = 9'd25;
77            color = 3'b011;
78          end
79        I :
80          begin
81            start_n = 9'd6;
82            first_new = 9'd6;
83            sec_new = 9'd7;
84            third_new = 9'd8;
85            fourth_new = 9'd9;
86            color = 3'b100;
87          end
88        O :
89          begin
90            start_n = 9'd7;
91            first_new = 9'd7;
92            sec_new = 9'd8;
93            third_new = 9'd23;
94            fourth_new = 9'd24;
95            color = 3'b101;
96          end
97        L :
98          begin
99            start_n = 9'd9;
100           first_new = 9'd9;
101           sec_new = 9'd23;
102           third_new = 9'd24;
103           fourth_new = 9'd25;
104           color = 3'b110;
105         end
106       Z :
107         begin
108           start_n = 9'd7;
109           first_new = 9'd7;
110           sec_new = 9'd8;
111           third_new = 9'd24;
112           fourth_new = 9'd25;
113           color = 3'b111;
114         end
115     endcase
116
117     if (move)
118       NextState = BlockMove;
119   end
120 /* Movement of the blocks */
121 BlockMove :
```

```
122     begin
123       if (frame)
124       begin
125         /* When block is falling or is driven down calculate next positions */
126         if (DOWN && (((tetromino == I && (rotate == 0 || rotate == 2)) && !CollisionBuf[first+32] &&
127               !CollisionBuf[sec+32] && !CollisionBuf[third+32] && !CollisionBuf[fourth+32]) ||
128             (((tetromino == T && rotate == 2) || (tetromino == Z && (rotate == 0 || rotate == 2))) &&
129               !CollisionBuf[first+32] && !CollisionBuf[third+32] && !CollisionBuf[fourth+32]) ||
130             ((tetromino == J && rotate == 2) && !CollisionBuf[first+32] && !CollisionBuf[sec+32] &&
                  !CollisionBuf[fourth+32]) ||
131             ((((tetromino == S || tetromino == L) && (rotate == 0 || rotate == 2)) || ((tetromino ==
                  J || tetromino == T) && rotate == 0)) &&
132               !CollisionBuf[sec+32] && !CollisionBuf[third+32] && !CollisionBuf[fourth+32]) ||
133             ((tetromino == L && rotate == 3) && !CollisionBuf[first+32] && !CollisionBuf[fourth+32])
                  ||
134             (((tetromino == S && (rotate == 1 || rotate == 3)) || (tetromino == J && rotate == 1) ||
                  (tetromino == T && rotate == 3)) &&
135               !CollisionBuf[sec+32] && !CollisionBuf[fourth+32]) ||
136             (((tetromino == J && rotate == 3) || (tetromino == T && rotate == 1) || tetromino == O ||
137               (tetromino == L && rotate == 1) || (tetromino == Z && (rotate == 1 || rotate == 3)))
                      &&
138               !CollisionBuf[third+32] && !CollisionBuf[fourth+32]) ||
139             (tetromino == I && (rotate == 1 || rotate == 3) && !CollisionBuf[fourth+32])))
140         begin
141           diff_d = 9'd32;
142           check = 1'b0;
143         end
144         else if(((tetromino == I && (rotate == 0 || rotate == 2)) && !CollisionBuf[first+16] &&
145               !CollisionBuf[sec+16] && !CollisionBuf[third+16] && !CollisionBuf[fourth+16]) ||
146               (((tetromino == T && rotate == 2) || (tetromino == Z && (rotate == 0 || rotate ==
                      2))) &&
147             !CollisionBuf[first+16] && !CollisionBuf[third+16] && !CollisionBuf[fourth+16]) ||
148           ((tetromino == J && rotate == 2) && !CollisionBuf[first+16] && !CollisionBuf[sec+16] &&
                !CollisionBuf[fourth+16]) ||
149               ((((tetromino == S || tetromino == L) && (rotate == 0 || rotate == 2)) || ((tetromino
                      == J || tetromino == T) && rotate == 0)) &&
150             !CollisionBuf[sec+16] && !CollisionBuf[third+16] && !CollisionBuf[fourth+16]) ||
151               ((tetromino == L && rotate == 3) && !CollisionBuf[first+16] &&
                      !CollisionBuf[fourth+16]) ||
152               (((tetromino == S && (rotate == 1 || rotate == 3)) || (tetromino == J && rotate == 1)
                      || (tetromino == T && rotate == 3)) &&
153                 !CollisionBuf[sec+16] && !CollisionBuf[fourth+16]) ||
154               (((tetromino == J && rotate == 3) || (tetromino == T && rotate == 1) || tetromino ==
                      O ||
155                 (tetromino == L && rotate == 1) || (tetromino == Z && (rotate == 1 || rotate ==
                        3))) &&
156                 !CollisionBuf[third+16] && !CollisionBuf[fourth+16]) ||
```

```
157                 (tetromino == I && (rotate == 1 || rotate == 3) && !CollisionBuf[fourth+16]))
158           begin
159             diff_d = 9'd16;
160             check = 1'b0;
161           end
162           else
163           begin
164             diff_d = 9'd0;
165             check = 1'b1;
166           end
167           /* When LEFT or RIGHT button is pushed calculate next positions */
168           if (LEFT && (((tetromino == I && (rotate == 1 || rotate == 3)) && !CollisionBuf[first-1] &&
169                       !CollisionBuf[sec-1] && !CollisionBuf[third-1] && !CollisionBuf[fourth-1]) ||
170                       (((tetromino == L && rotate == 1) || (tetromino == J && rotate == 3)) &&
171                           !CollisionBuf[first-1] && !CollisionBuf[sec-1] && !CollisionBuf[third-1]) ||
172                       (((tetromino == S && (rotate == 1 || rotate == 3)) || (tetromino == T &&
173                          (rotate == 1 || rotate == 3)) || (tetromino == Z && (rotate == 1 || rotate ==
                               3))) &&
174                           !CollisionBuf[first-1] && !CollisionBuf[sec-1] && !CollisionBuf[fourth-1]) ||
175                       (((tetromino == J && rotate == 1) || (tetromino == L && rotate == 3) ) &&
176                           !CollisionBuf[first-1] && !CollisionBuf[third-1] && !CollisionBuf[fourth-1])
                                ||
177                       (((tetromino == S && (rotate == 0 || rotate == 2)) || tetromino == O ||
                            (tetromino == Z && (rotate == 0 || rotate == 2))) &&
178                           !CollisionBuf[first-1] && !CollisionBuf[third-1]) ||
179                       (((tetromino == J && rotate == 0) || (tetromino == L && rotate == 0) ||
                            (tetromino == T && rotate == 0)) &&
180                           !CollisionBuf[first-1] && !CollisionBuf[sec-1]) ||
181                       (((tetromino == J && rotate == 2) || (tetromino == L && rotate == 2) ||
                            (tetromino == T && rotate == 2)) &&
182                           !CollisionBuf[first-1] && !CollisionBuf[fourth-1]) ||
183                       ((tetromino == I && (rotate == 0 || rotate == 2)) && !CollisionBuf[first-1])))
184           begin
185             diff_s = -9'd1;
186           end
187           else if (RIGHT &&(((tetromino == I && (rotate == 1 || rotate == 3)) &&
                    !CollisionBuf[first+1] &&
188                               !CollisionBuf[sec+1] && !CollisionBuf[third+1] &&
                                    !CollisionBuf[fourth+1]) ||
189                              (((tetromino == L && rotate == 1) || (tetromino == J && rotate == 3)) &&
190                               !CollisionBuf[first+1] && !CollisionBuf[sec+1] &&
                                    !CollisionBuf[fourth+1]) ||
191                              (((tetromino == S && (rotate == 1 || rotate == 3)) || (tetromino == T &&
                                   (rotate == 1 || rotate == 3)) ||
192                         (tetromino == Z && (rotate == 1 || rotate == 3))) &&
193                               !CollisionBuf[first+1] && !CollisionBuf[third+1] &&
                                    !CollisionBuf[fourth+1]) ||
```

```
194                     (((tetromino == J && rotate == 1) || (tetromino == L && rotate == 3) ) &&
195                         !CollisionBuf[sec+1] && !CollisionBuf[third+1] &&
                                !CollisionBuf[fourth+1]) ||
196                     (((tetromino == J && rotate == 0) || (tetromino == L && rotate == 0) ||
                             (tetromino == T && rotate == 0)) &&
197                         !CollisionBuf[first+1] && !CollisionBuf[fourth+1]) ||
198                     (((tetromino == J && rotate == 2) || (tetromino == L && rotate == 2) ||
                             (tetromino == T && rotate == 2)) &&
199                         !CollisionBuf[third+1] && !CollisionBuf[fourth+1]) ||
200                     (((tetromino == S && (rotate == 0 || rotate == 2)) || tetromino == O ||
                             (tetromino == Z && (rotate == 0 || rotate == 2))) &&
201                         !CollisionBuf[sec+1] && !CollisionBuf[fourth+1]) ||
202                     ((tetromino == I && (rotate == 0 || rotate == 2)) &&
                             !CollisionBuf[fourth+1])))
203         begin
204           diff_s = 9'd1;
205         end
206         else
207         begin
208           diff_s = 9'd0;
209         end
210
211         if (ROTATE)
212         begin
213         /* When ROTATE button is pushed, if the rotation can occur *
214         * assign the next positions of first,sec, third and fourth */
215           case (tetromino)
216             S :
217               begin
218                 if (rotate == 0 && !CollisionBuf[fourth+1] && !CollisionBuf[fourth+17])
219                 begin
220                   diff_r_first = 9'd0;
221                   diff_r_sec = 9'd15;
222                   diff_r_third = 9'd2;
223                   diff_r_fourth = 9'd17;
224                   rotate_new = rotate + 2'd1;
225                 end
226                 if (rotate == 1 && !CollisionBuf[fourth-2])
227                 begin
228                   diff_r_first = 9'd16;
229                   diff_r_sec = 9'd1;
230                   diff_r_third = 9'd14;
231                   diff_r_fourth = -9'd1;
232                   rotate_new = rotate + 2'd1;
233                 end
234                 if (rotate == 2 && !CollisionBuf[first-1] && !CollisionBuf[first-15])
235                 begin
```

```
236                 diff_r_first = -9'd17;
237                 diff_r_sec = -9'd2;
238                 diff_r_third = -9'd15;
239                 diff_r_fourth = 9'd0;
240                 rotate_new = rotate + 2'd1;
241               end
242               if (rotate == 3 && !CollisionBuf[first+2])
243               begin
244                 diff_r_first = 9'd1;
245                 diff_r_sec = -9'd14;
246                 diff_r_third = -9'd1;
247                 diff_r_fourth = -9'd16;
248                 rotate_new = rotate + 2'd1;
249               end
250             end
251           J :
252             begin
253               if (rotate == 0 && !CollisionBuf[third+16])
254               begin
255                 diff_r_first = 9'd1;
256                 diff_r_sec = -9'd14;
257                 diff_r_third = 9'd0;
258                 diff_r_fourth = 9'd15;
259                 rotate_new = rotate + 2'd1;
260               end
261               if (rotate == 1 && !CollisionBuf[third-1] && !CollisionBuf[third+1] &&
                      !CollisionBuf[fourth+1])
262               begin
263                 diff_r_first = 9'd15;
264                 diff_r_sec = 9'd15;
265                 diff_r_third = 9'd1;
266                 diff_r_fourth = 9'd1;
267                 rotate_new = rotate + 2'd1;
268               end
269               if (rotate == 2 && !CollisionBuf[first+16] && !CollisionBuf[sec+16])
270               begin
271                 diff_r_first = -9'd15;
272                 diff_r_sec = 9'd0;
273                 diff_r_third = 9'd14;
274                 diff_r_fourth = -9'd1;
275                 rotate_new = rotate + 2'd1;
276               end
277               if (rotate == 3 && !CollisionBuf[sec+1])
278               begin
279                 diff_r_first = -9'd1;
280                 diff_r_sec = -9'd1;
281                 diff_r_third = -9'd15;
```

```
282                 diff_r_fourth = -9'd15;
283                 rotate_new = rotate + 2'd1;
284               end
285             end
286           T :
287             begin
288               if (rotate == 0 && !CollisionBuf[third+16])
289               begin
290                 diff_r_first = 9'd0;
291                 diff_r_sec = 9'd1;
292                 diff_r_third = 9'd1;
293                 diff_r_fourth = 9'd15;
294                 rotate_new = rotate + 2'd1;
295               end
296               if (rotate == 1 && !CollisionBuf[sec-1])
297               begin
298                 diff_r_first = 9'd15;
299                 diff_r_sec = 9'd0;
300                 diff_r_third = 9'd0;
301                 diff_r_fourth = 9'd0;
302                 rotate_new = rotate + 2'd1;
303               end
304               if (rotate == 2)
305               begin
306                 diff_r_first = -9'd15;
307                 diff_r_sec = -9'd1;
308                 diff_r_third = -9'd1;
309                 diff_r_fourth = 9'd0;
310                 rotate_new = rotate + 2'd1;
311               end
312               if (rotate == 3 && !CollisionBuf[third+1])
313               begin
314                 diff_r_first = 9'd0;
315                 diff_r_sec = 9'd0;
316                 diff_r_third = 9'd0;
317                 diff_r_fourth = -9'd15;
318                 rotate_new = rotate + 2'd1;
319               end
320             end
321           I :
322             begin
323               if (rotate == 0 && !CollisionBuf[third+16])
324               begin
325                 diff_r_first = -9'd30;
326                 diff_r_sec = -9'd15;
327                 diff_r_third = 9'd0;
328                 diff_r_fourth = 9'd15;
```

```
329                 rotate_new = rotate + 2'd1;
330               end
331               if (rotate == 1 && !CollisionBuf[sec-2] && !CollisionBuf[sec+1])
332               begin
333                 diff_r_first = 9'd14;
334                 diff_r_sec = -9'd1;
335                 diff_r_third = -9'd16;
336                 diff_r_fourth = -9'd31;
337                 rotate_new = rotate + 2'd1;
338               end
339               if (rotate == 2 && !CollisionBuf[sec+32])
340               begin
341                 diff_r_first = -9'd15;
342                 diff_r_sec = 9'd0;
343                 diff_r_third = 9'd15;
344                 diff_r_fourth = 9'd30;
345                 rotate_new = rotate + 2'd1;
346               end
347               if (rotate == 3 && !CollisionBuf[third-1] && !CollisionBuf[third+2])
348               begin
349                 diff_r_first = 9'd31;
350                 diff_r_sec = 9'd16;
351                 diff_r_third = 9'd1;
352                 diff_r_fourth = -9'd14;
353                 rotate_new = rotate + 2'd1;
354               end
355             end
356           O :
357             begin
358               diff_r_first = 9'd0;
359               diff_r_sec = 9'd0;
360               diff_r_third = 9'd0;
361               diff_r_fourth = 9'd0;
362               rotate_new = rotate + 2'd1;
363             end
364           L :
365             begin
366               if (rotate == 0 && !CollisionBuf[third+16] && !CollisionBuf[fourth+16])
367               begin
368                 diff_r_first = -9'd1;
369                 diff_r_sec = 9'd1;
370                 diff_r_third = 9'd16;
371                 diff_r_fourth = 9'd16;
372                 rotate_new = rotate + 2'd1;
373               end
374               if (rotate == 1 && !CollisionBuf[sec-1] && !CollisionBuf[third-1] &&
                      !CollisionBuf[sec+1])
```

```
375                 begin
376                   diff_r_first = 9'd15;
377                   diff_r_sec = 9'd0;
378                   diff_r_third = -9'd15;
379                   diff_r_fourth = -9'd2;
380                   rotate_new = rotate + 2'd1;
381                 end
382                 if (rotate == 2 && !CollisionBuf[first-16] && !CollisionBuf[sec+16])
383                 begin
384                   diff_r_first = -9'd16;
385                   diff_r_sec = -9'd16;
386                   diff_r_third = -9'd1;
387                   diff_r_fourth = 9'd1;
388                   rotate_new = rotate + 2'd1;
389                 end
390                 if (rotate == 3 && !CollisionBuf[sec+1] && !CollisionBuf[third+1] &&
                        !CollisionBuf[third-1])
391                 begin
392                   diff_r_first = 9'd2;
393                   diff_r_sec = 9'd15;
394                   diff_r_third = 9'd0;
395                   diff_r_fourth = -9'd15;
396                   rotate_new = rotate + 2'd1;
397                 end
398               end
399             Z :
400               begin
401                 if (rotate == 0 && !CollisionBuf[sec+1] && !CollisionBuf[third+16])
402                 begin
403                   diff_r_first = 9'd2;
404                   diff_r_sec = 9'd16;
405                   diff_r_third = 9'd1;
406                   diff_r_fourth = 9'd15;
407                   rotate_new = rotate + 2'd1;
408                 end
409                 if (rotate == 1 && !CollisionBuf[sec-1] && !CollisionBuf[fourth+1])
410                 begin
411                   diff_r_first = 9'd14;
412                   diff_r_sec = 9'd0;
413                   diff_r_third = 9'd15;
414                   diff_r_fourth = 9'd1;
415                   rotate_new = rotate + 2'd1;
416                 end
417                 if (rotate == 2 && !CollisionBuf[third-1] && !CollisionBuf[sec-16])
418                 begin
419                   diff_r_first = -9'd15;
420                   diff_r_sec = -9'd1;
```

```
421                 diff_r_third = -9'd16;
422                 diff_r_fourth = -9'd2;
423                 rotate_new = rotate + 2'd1;
424               end
425               if (rotate == 3 && !CollisionBuf[first-1] && !CollisionBuf[third+1])
426               begin
427                 diff_r_first = -9'd1;
428                 diff_r_sec = -9'd15;
429                 diff_r_third = 9'd0;
430                 diff_r_fourth = -9'd14;
431                 rotate_new = rotate + 2'd1;
432               end
433             end
434           default :
435             begin
436               diff_r_first = 9'd0;
437               diff_r_sec = 9'd0;
438               diff_r_third = 9'd0;
439               diff_r_fourth = 9'd0;
440             end
441         endcase
442       end
443       else
444       begin
445         diff_r_first = 9'd0;
446         diff_r_sec = 9'd0;
447         diff_r_third = 9'd0;
448         diff_r_fourth = 9'd0;
449       end
450
451       CollisionBuf_new[first] = 3'b000;
452       CollisionBuf_new[sec] = 3'b000;
453       CollisionBuf_new[third] = 3'b000;
454       CollisionBuf_new[fourth] = 3'b000;
455
456       CollisionBuf_new[first+ diff_d + diff_s + diff_r_first] = color;
457       CollisionBuf_new[sec + diff_d + diff_s + diff_r_sec] = color;
458       CollisionBuf_new[third + diff_d + diff_s + diff_r_third] = color;
459       CollisionBuf_new[fourth + diff_d + diff_s + diff_r_fourth] = color;
460
461       first_new = first + diff_d + diff_s + diff_r_first;
462       sec_new = sec + diff_d + diff_s + diff_r_sec;
463       third_new = third + diff_d + diff_s + diff_r_third;
464       fourth_new = fourth + diff_d + diff_s + diff_r_fourth;
465
466       /* If the block can't move down any more */
467       if (check)
```

```
468         begin
469           rotate_new = 2'd0;
470           block_score = 4'd12;
471           /* Detect game over if at the *
472           * initial position.        */
473           if (first != start)
474           begin
475             NextState = DetectLines;
476           end
477           else
478           begin
479             done = 1'b1;
480             NextState = GameOver;
481           end
482         end
483         else
484           NextState = BlockMove;
485       end
486     end
487   /* Find all completed rows */
488   DetectLines :
489     begin
490       for (i = 0; i < 22; i = i + 1) //detect filled lines
491       begin
492         filled = 1'b1;
493         for (j = 1; j < 15; j = j + 1)
494         begin
495           if (!CollisionBuf[j + i*16])
496             filled = 1'b0;
497         end
498
499         FilledRows_new[i] = filled;
500       end//for i
501
502       new_row = 5'd0;
503       new_col = 4'd1;
504       NextState = CheckRows;
505     end
506   /* Check if a row is completed */
507   CheckRows :
508     begin
509       /* If at the last row */
510       if (row == 5'd21)
511       begin
512         if (FilledRows[row] == 1'b0)
513         begin
514           new_row = 5'd0;
```

```
515           new_col = 4'd1;
516           lines_delfsm = lines_delcount;
517           lines_delcount_n = 3'd0;
518           done = 1'b1;
519           NextState = Idle_1;
520         end
521         else
522         begin
523           FilledRows_new[row] = 1'b0;
524           new_d_row = row;
525           lines_delcount_n = lines_delcount + 3'd1;
526           NextState = DeleteRow;
527         end
528       end
529       else
530       begin
531         if (FilledRows[row] == 1'b0)
532         begin
533           new_row = row + 5'd1;
534           NextState = CheckRows;
535         end
536         else
537         begin
538           FilledRows_new[row] = 1'b0;
539           new_d_row = row;
540           new_row = row + 5'd1;
541           lines_delcount_n = lines_delcount + 3'd1;
542           NextState = DeleteRow;
543         end
544       end
545     end
546   /* Remove the completed row and replace *
547   * its values with the values of the *
548   * previous row                       */
549   DeleteRow :
550     begin
551       if (!del_row) //if at row 0
552       begin
553         CollisionBuf_new[col] = 3'b000;
554         if (col == 4'd14) //if at last grid write zero and check rows again
555         begin
556           new_col = 4'd1;
557           NextState = CheckRows;
558         end
559         else
560         begin //assign row 0 with zeroes
561           new_col = col + 4'd1;
```

```
562           NextState = DeleteRow;
563         end
564       end
565       else //at any other row
566       begin
567         CollisionBuf_new[col + del_row*16] = CollisionBuf[col + (del_row-1)*16];
568         if (col == 4'd14) //if at last grid change the value and go to the smaller row
569         begin
570           new_col = 4'd1;
571           new_d_row = del_row - 5'd1;
572         end
573         else //at any other grid change the value and go to next grid
574         begin
575           new_col = col + 4'd1;
576         end
577         NextState = DeleteRow;
578       end
579     end
580   /* The game is over and master FSM *
581   * is informed                    */
582   GameOver :
583     begin
584       game_over = 1'b1;
585       NextState = GameOver;
586     end
587   default:;
588 endcase
```

Listing A.4: Linear Feedback Shift Register

```
1  parameter     A = 3'b000, B = 3'b001, C = 3'b010, D = 3'b011,
2                E = 3'b100, F = 3'b101, G = 3'b110, H = 3'b111;
3
4    /* The Linear Feedback Shift register provides *
5    * us with pseudo-randomly generated numbers *
6    * and create pseudo randomly Tetriminos.  */
7    always @(posedge vclk or posedge rst)
8    begin
9      if (rst)
10     begin
11       out = 16'b1000_1111_0010_0010;
```

```
12        Random = 3'b0;
13        next_block = 3'd3;
14        count = 1'd0;
15      end
16      else
17      begin
18        if (new_block && !count)
19        begin
20        /* Drive the input at bit 0 and shift the rest */
21          out = {out[14],out[13],out[12],out[11],out[10],
22                 out[9],out[8],out[7],out[6],out[5],out[4],
23                 out[3],out[2],out[1],out[0],linear_feedback};
24
25          Random = {linear_feedback1,linear_feedback2,linear_feedback3};
26          count = 1'd1;
27        end
28        else if (move)
29        begin
30          count = 1'b0;
31        end
32        case (Random)
33          A : next_block = 3'd4; //I
34          B : next_block = 3'd1; //S
35          C : next_block = 3'd2; //J
36          D : next_block = 3'd3; //T
37          E : next_block = 3'd4; //I
38          F : next_block = 3'd5; //O
39          G : next_block = 3'd6; //L
40          H : next_block = 3'd7; //Z
41          default:;
42        endcase
43      end
44    end
```

```
45  assign linear_feedback1 = !(out[15]^out[13]^out[12]);
46  assign linear_feedback2 = !(out[10]^out[8]^out[7]);
47  assign linear_feedback3 = !(out[5]^out[3]^out[2]);
48  assign linear_feedback =
        (linear_feedback1^linear_feedback2^linear_feedback3);
```