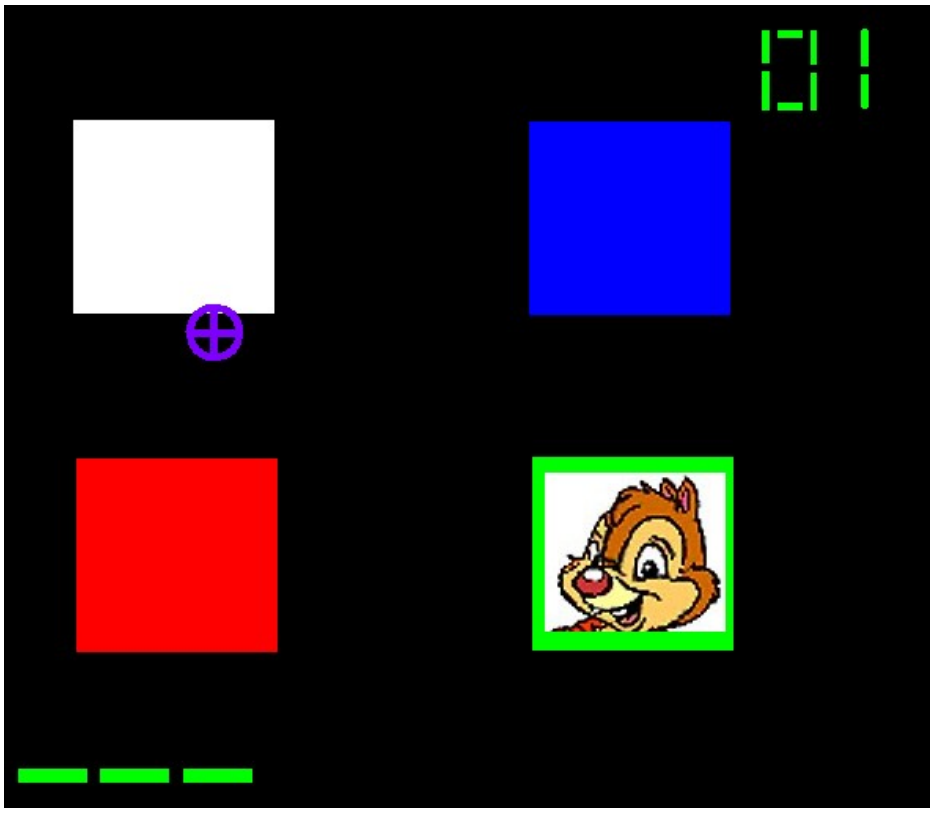
Squirrel Hunter

Balaji Sathyanarayanan (sx08bs4@student.lth.se)
Can Bilgin (sx08cb0@student.lth.se)
Rakesh.M.G (sx08rg5@student.lth.se)



Abstract

The aim of the project is to design a game on an embedded platform. We use a Digilent Nexys2 board with a Spartan 3E FPGA. The Microblaze CPU of the XILINX family is used as the processor to run the game software. The idea of the game is to shoot a squirrel when it pops up on the screen. The user scores every time the squirrel is shot correctly, and loses points for mis hits.
We managed to implement most of the ideas that we had thought of during the initial proposal. Many more improvements and add-ons could have been added but due to lack of time we could not implement those. Overall we were satisfied with our project and the main idea of Hardware-Software co-design in embedded system design was understood.

Table of Contents

1. Introduction

The idea of the game is to have the user shoot a squirrel. The VGA screen has four square holes. A squirrel pops up once in a while from one of the holes. The user can use a PS2 mouse to shoot the squirrel. The user starts with 3 “lives” and can score points by shooting the squirrel. If the user mis hits, the score decrements and every time the user mis hits the life decreases. Game ends when the user loses all the “lives”. Also, if the user fails to hit the squirrel 3 times in a row, user loses a point. If the user manages to score 10 points, the life increases by 1 and the user can have a maximum of 4 lives. The score is displayed on the right top of the screen and also on the 7 segment LEDs on the board, while the remaining lives is displayed by lines on the bottom of the screen.

The game progresses with time. Initially, the time for which the squirrel stays visible is longer, enabling the user to hit the squirrel easily, and as time goes by, the time duration for which the squirrel is visible goes on decreasing making it harder to hit the squirrel. The game ends after a programmable number of squirrels are displayed or if the user loses all the lives.

The game is developed on a Digilent Nexys-2 board with a Spartan3E FPGA on it. We have used the 7 segment LED and the VGA monitor interface available on the board.

The software is written in C and runs on a Microblaze processor. Since the software is very simple, only register reads and writes are needed to make the game work. The overall system control, game logic, scoring and life controls have been implemented on the software. Even though most parts of the game has been implemented on hardware, we still need the Microblaze since it is the master controller controlling all the other blocks and to run the software.

2. System Architecture

The game architecture is shown in the block diagram below in Figure 1. The Microblaze on the Digilent Nexys-2 board is used as the main processor for running the game logic. The VGA controller, the seven segment display and the PS2 mouse are controlled by hardware IPs. The *clock* and the *rst* block on the board are used to generate the clock and reset needed by the H/W IPs and the Microblaze processor. The communication between the Microblaze and the various H/W IPs is through the *PLB* bus. The user can control the various operations of the VGA controller by programming some registers in the VGA IP block and the 7 segment IP. These registers are also used to get information from the hardware to the software via the *PLB* bus. Detailed description of the different IPs and the application software is described below.

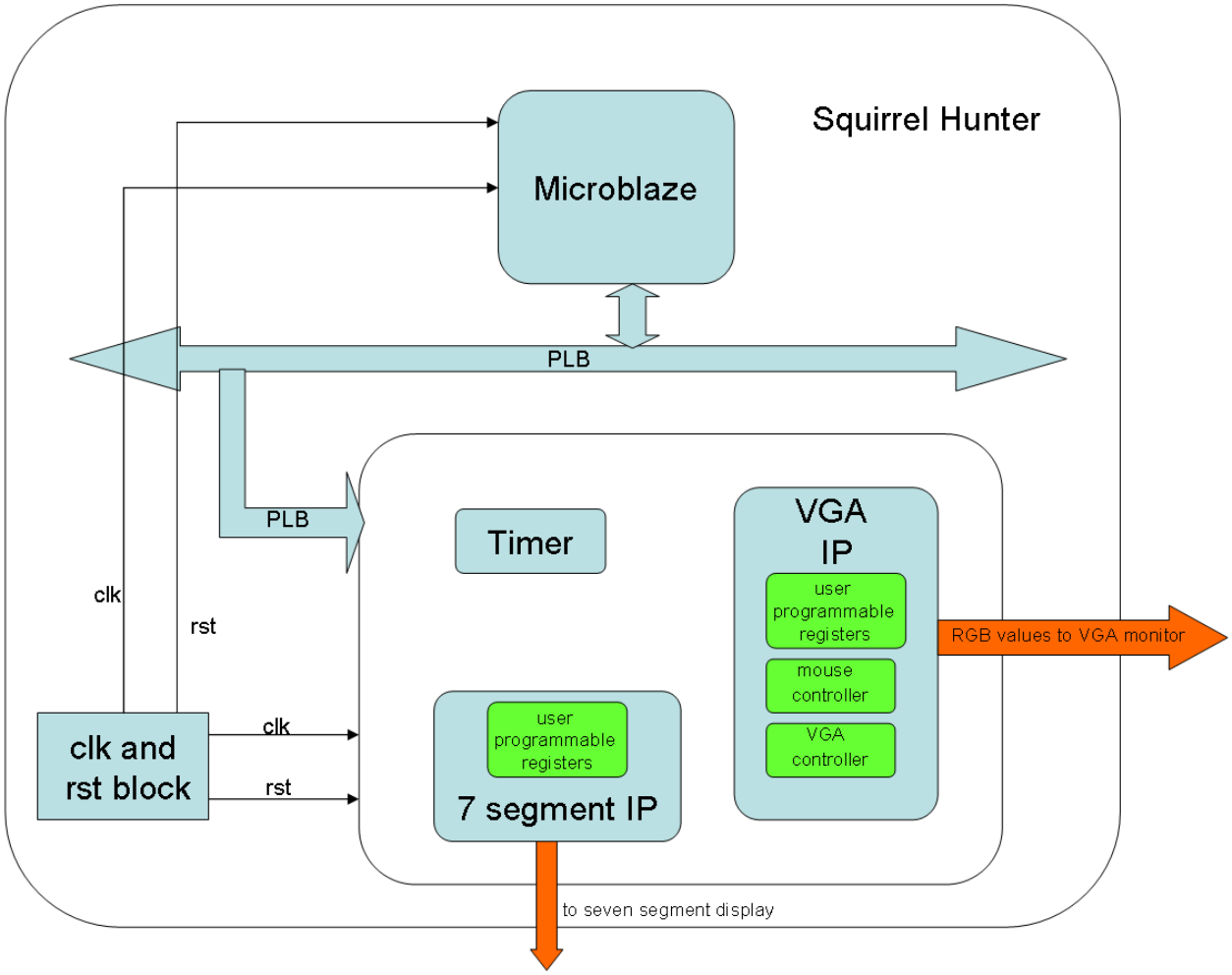


Figure 1 Top Level Block Diagram

The detailed description of each of the blocks have been discussed below.

2.1 VGA Controller

This block is designed to generate and display the graphics of the game on a 640*480 VGA monitor. As the VGA monitor works with 25 MHz synchronization signals, the 50 MHz on board clock is divided by two by using a simple D flip flop structure, the *DFF* and then supply it to the rest of the system. The block diagram is as shown below in Figure 2.

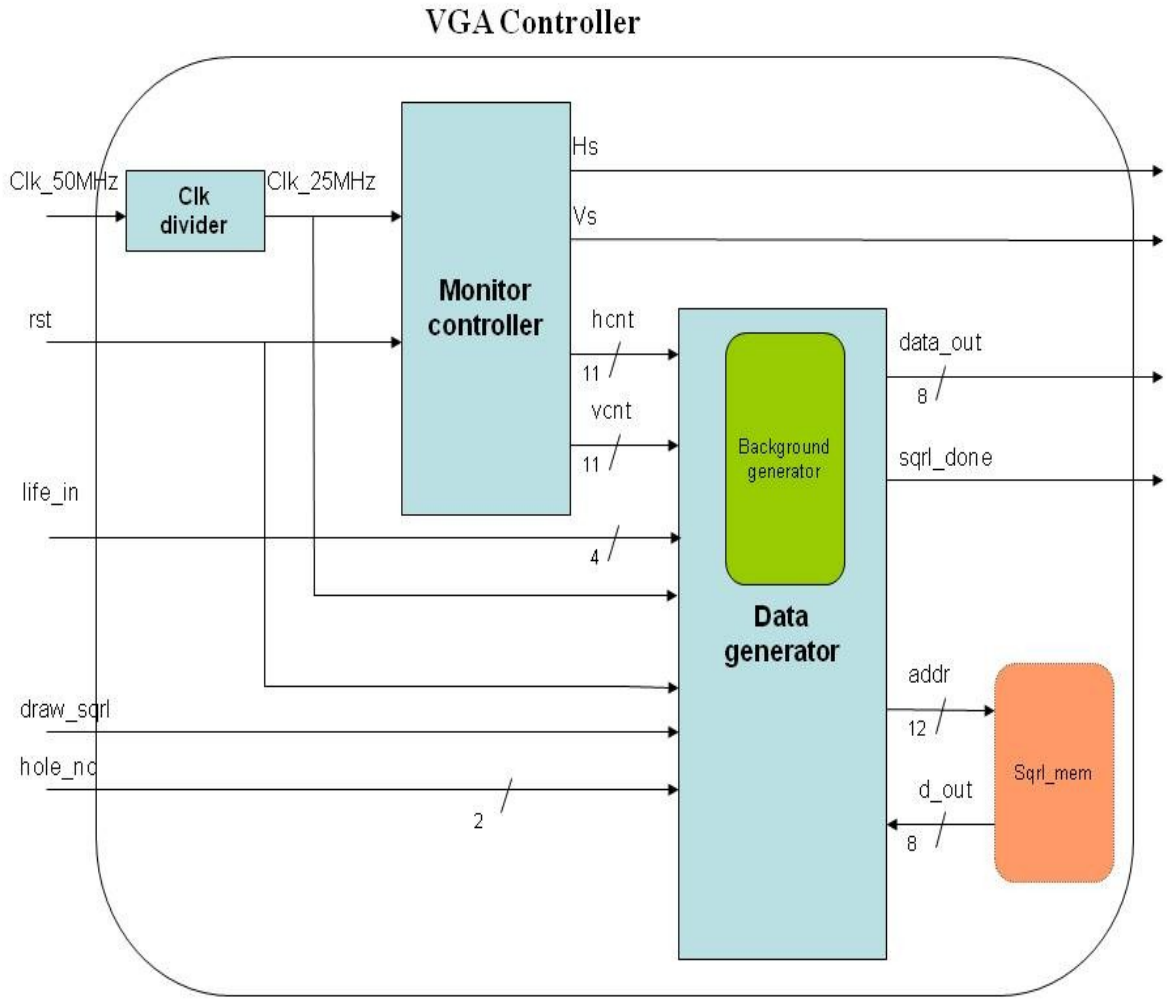


Figure 2 – VGA controller architecture

The block employs a *monitor_controller* block, which generates the necessary output horizontal and vertical synchronization *Hs* and *Vs* signals of the monitor together with the *hcnt* and *vcnt* signals to the *data_generator* block to generate the corresponding output data, *data_out*, to be displayed on the screen. The *monitor_controller* block is same as the *vga_controller_640_60* IP that was taken from the *VGA RefComp*[1] design of Digilentinc. The rest of VGA design is composed of custom designs.

The *data_generator* module is responsible for generating the output data synchronized with *hcnt* and *vcnt* signals coming from *monitor_controller*. Inside this module there is another module called *background_generator*, which generates the game background such as drawing the holes, showing the game score and the remaining lives also on the VGA screen.

The *data_generator* on the request of *draw_squirrel* signal draws the squirrel to the one of the 4 holes selected by the incoming *hole_number* signal. By making use of an internal ROM memory composed of logic cells, the image of the squirrel is hand-coded and stored there. So we are not using any BRAM for storing the squirrel image. After the squirrel is displayed, a signal called *sqrl_done*, which is a 1 bit simple signal, is sent to the Microblaze processor through one of the *PLB* registers in order to use it in the game logic. Even though we planned to use *sqrl_done* in the software for further processing, it was not needed in the end as we made sure that the squirrel images were written properly and completed by the *vga_controller*. Hence this signal was used only for debug purposes to begin with and later was not used in the software.

2.2 Seven Segment Display

This block is designed to control the seven segment display in order to show the game score on the board. As it can be seen in Figure 3 below, the block takes *clk*, *rst* and 15 bit input data, *d_in*, which contain the information for 4 different segments each represented by 4 bits and then it generates the segment data, *d_seg* and the corresponding 1 bit *d_anode* output signal which is needed to refresh the display:

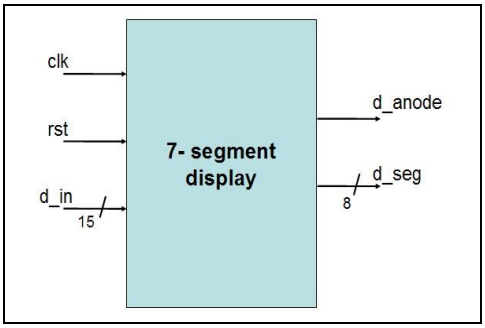


Figure 3 Seven segment display

Inside the block, there is a counter that counts until 50000 with 50 MHz on board clock signal in order to generate a refresh signal that is toggling with 1 ms. With this refresh signal, each segment is refreshed in sequence of *d_anode* signal which is the refreshing signal of each segment in each 4 ms, which is in the valid range of 1 ms to 16 ms of the display suppliers. For each segment, 4 bits of the digit to be displayed is read from the 16 bit input and then decoded into 8 bits to generate the score according to the connections that is required by the display supplier as illustrated in Figure 4 below which also can be seen in the Digilent Nexys2 Board reference[2] manual.

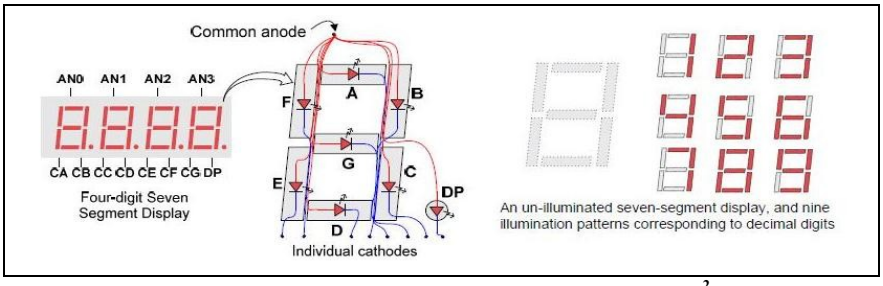


Figure 4 Illustration of the display of the digits[2]

2.3 Mouse Controller Implementation using PS2 Interface

The mouse controller is implemented using the PS2 interface. The basic mouse controller IP MouseRefComp[3], which was available at the Digilent website[3] was used as the reference. Few modifications had to be done to the code in order to make it compatible and integrate it with the XPS software. The block diagram below in Figure 5, gives the overview of the mouse controller.

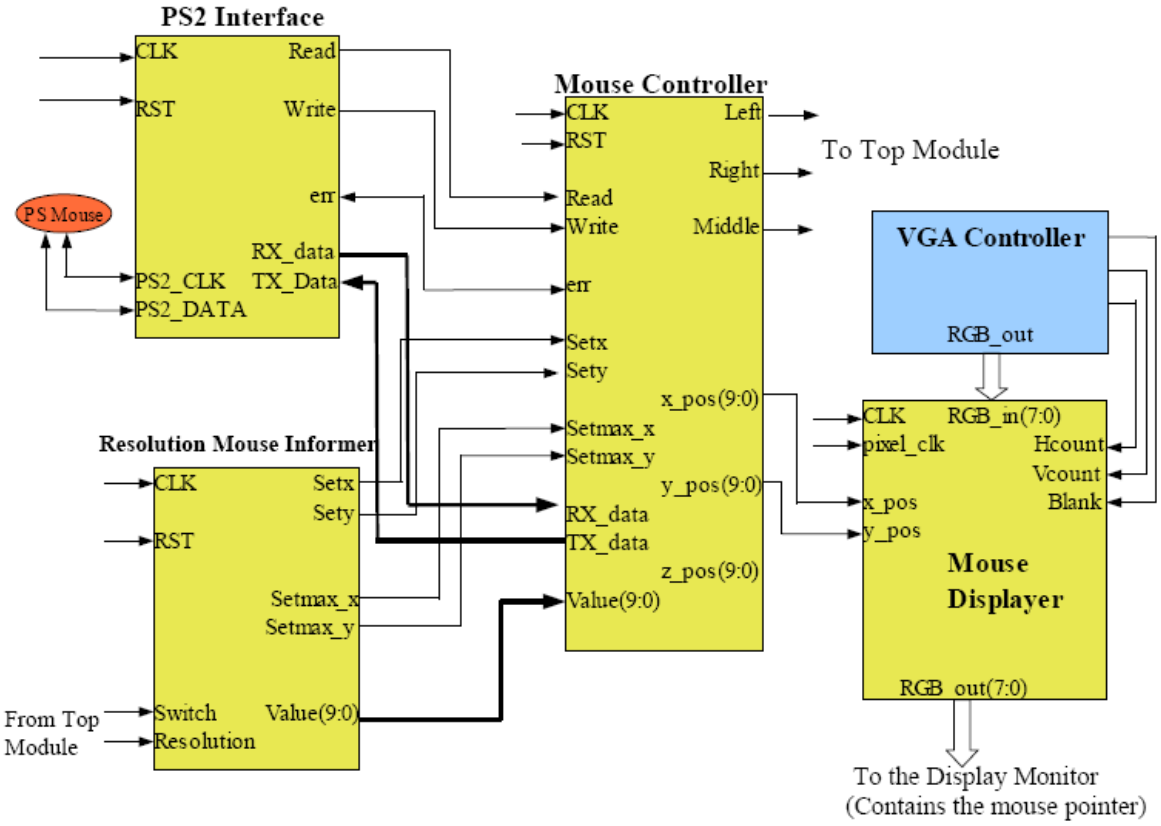


Figure 5 Block Diagram of the Mouse Implementation[3]

The function of various blocks in the above block diagram in Figure 5 have been discussed briefly below.

2.3.1 PS2 Interface

The PS/2 interface module is the one that directly interacts with mouse hardware. This module implements a generic bi-directional PS/2 interface. It can be used with any PS/2 compatible device and it offers its clients a convenient way to exchange data. The interface transparently wraps the byte to be sent into a PS/2 frame, generates the parity bit, and sends the frame one bit at a time to the device. Similarly, when receiving data from the PS2 device, the interface receives the frame, checks for parity, extracts the useful data, and forwards it to the *Mouse Controller* block. If an error occurs when sending or receiving a byte, the *Mouse Controller* block is informed by setting the err output line high. This way it can resend the data or issue a resend command to the device.

The CLK and RST signals are connected to the global clock and reset signals. PS2_clk and PS2_data are bi-directional signals that are mapped to the respective pins in the UCF file so that they are directly connected the mouse hardware.

2.3.2 Mouse Controller

The *Mouse Controller* module is connected between the *PS Interface* and *Mouse Displayer* modules. Based on the RX_data signal it receives from the *PS Interface* module and the other signals from the *Resolution Mouse Informer* module, this module generates the X and Y position of the mouse cursor. These 2 signals, the X and Y positions are the main signals required by the *Mouse Displayer* based on which it generates the mouse pointer on the appropriate position. This module is based on a state machine which starts with a reset state. When in reset state, the controller resets the mouse and begins an initialization procedure. After the initialization the module is ready to perform the calculation of the X and Y position of the mouse cursor. It also and sends information about mouse clicks to one of the user programmable registers. The decision of whether the user clicked in the correct square is done in hardware and the information whether the user was able to click at the right moment and in what square the user clicked is sent to one of the programmable registers. This register is read at the right time by the processor to make decisions about whether the mouse was clicked in the correct square or not.

2.3.3 Resolution Mouse Informer

The *Resolution Mouse Informer* implements the logic that sets the position of the mouse when the FPGA is powered-up or when the resolution changes. It also sets the bounds of the mouse according to the present resolution. The mouse is centered for the currently selected resolution which it gets from the *Top Module* (as per the signals required for a screen with resolution 640x480) and the bounds are set accordingly. This way, the mouse cursor will appear in the center of the screen at start-up, and when the resolution is changed it will not leave the screen. The position, and the bounds, are set by placing the coordinate of the center point on the output

value and activating the corresponding set signal (set_x for horizontal position, set_y for vertical position, setmax_x for horizontal maximum value, and setmax_y for the vertical maximum value). In our design, the values for the *Resolution Mouse Informer* are set so that it suitable for a resolution of 640x480 screen size.

2.3.4 Mouse Displayer

The *Mouse Displayer* module generates the mouse pointer based on the X and Y position it receives from the *Mouse Controller* module. The RGB signals along with horizontal and vertical counters (*hcount* and *vcount*) are received from *VGA Controller* module. If the counters are inside the mouse cursor bounds, then the mouse pointer is sent to the screen instead of the received pixels. The mouse pointer 16x16 pixels and uses 8 bit data. In order to implement the sniper image for the mouse pointer, a 256x8 bit ROM is used in which the data for the sniper image is stored. When mouse pointer is to be displayed, then the data from the ROM is sent out as the RGB_out signal to the monitor otherwise the RGB_in data from the *VGA Controller* is sent out without any changes.

2.4 Interface to Software

The interface to the software, meaning the Microblaze processor, is done via some user registers. The videos on the course homepage were very useful in learning how to create our own IP block and to interface it to the processor. Even though most of the game is written in hardware, we felt the need to use the Microblaze processor as the main processor controlling all the other hardware IPs. The display of the squirrel images is controlled by certain bits in the user programmable registers. Figure 1 shows the basic architecture of the design. It can be seen that the VGA controller IP contains some user programmable registers. One of the registers contains bits which can be used to enable the display of a squirrel in a particular hole.

Another important role of Microblaze was to randomize the display of the figure. The standard "C" *rand()* function call is used to get a random number between 1 and 4 which is then used to select which hole the display of the squirrel should be done.

The processor also controls the timer IP used. It programs the delays between each display of the squirrel image. It changes the time for which the squirrel is displayed by varying the amount of time for which the timer counts to the programmed. We have not used any interrupts to get the time when the timer counts down to zero. We have a simple delay routine, which accepts a number used to program the timer counter register. Once the "*delay(timer_count_value)*" routine is called, the timer starts counting until it reaches the programmed value. Once this is done, the timer is reset and the function returns to the *main()* function. Hence by programming the delay values passed into this function, we get a variable delay, which is then used to control the time for which the squirrel is visible.

One of the user programmable registers is also used to get information about the clicks the user does. If the user clicks in the correct square (i.e when the image of the squirrel is present) a single

bit is written to this register saying that the user did a correct click. Also we get the information as to which hole the user clicked in, but this information was not needed by game logic and was used only for debug purposes. The processor performs reads of this single bit register when it is inside the "*delay(timer_count_value)*" subroutine and increments the score if the bit is set, or decrements if the bit is not set. Hence by doing this, the processor is able to keep track of the score and the life of the player. These values are also written to another set of user programmable registers to be used for displaying life and score on the VGA monitor and the seven segment LED's available on the board.

There are basically two main IPs as described above. The *mouse controller* IP is embedded inside the VGA controller top level IP. The processor local bus (PLB) is used to control and interface all the modules in the "squirrel hunter". The hardware IP blocks are instantiated inside the IPs generated using the Xilinx custom IP design flow. There are a few user programmable registers created by when designing the custom IP, and these are used as registers to program the behavior of the IPs.

2.5 Timer

A timer from the Xilinx IP library is also used in the game. This block is needed to generate delays between which no squirrels are displayed in the squares. The timer is given different values to count down from, in order to generate different delays between displaying squirrels.

3. Game Logic

A simple flow chart of the user game logic is shown below in Figure 6.

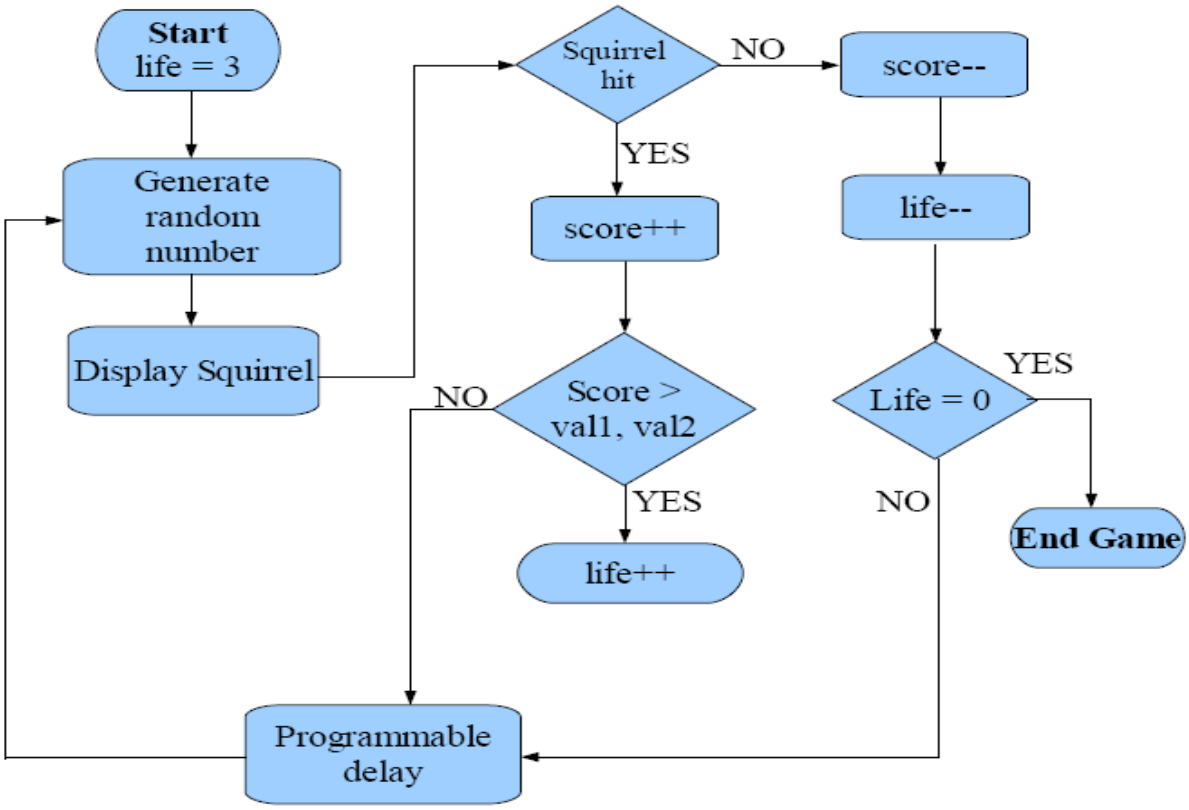


Figure 6 Game logic

The game starts when the system is brought out of reset. A user defined reset button on the board can be used to reset the game logic as well as the hardware. Once the processor is out of reset, the game software is executed as shown above.

The game depends mainly on the random number generator and the programmable delay. The standard *rand()* function in C is used to generate random numbers between 0 and 3 and these numbers are used to display a squirrel in a particular square. The delay block is done using a *Timer* block which is one the standard Xilinx IPs.

The game software is very simple in its nature and mainly has a call to a delay function. The delay function is called with a different delay value as the game progresses. Initially we start off with a big delay, giving the user a lot of time to be able to click on a squirrel. As the time goes on, the delay between each squirrel display decreases, making it harder for the user to score. For every squirrel missed, the user loses a life and also a point. Also if the user misses 3 squirrels in a row then, a point is lost and if the user is able to get 10 squirrels correctly, the user will be able to get one extra life and similarly one more life at 20 points. At anytime the user can have a maximum of 4 lives.

4. Design Synthesis

The design was synthesized using Xilinx XST. The synthesis results are shown below.

Number of External IOBs	65 out of 250	26%
Number of BUFGMUXs	2 out of 24	8%
Number of DCMs	1 out of 8	12%
Number of MULT18X18SIOs	3 out of 28	10%
Number of RAMB16s	16 out of 28	57%
Number of Slices	4132 out of 8672	47%
Number of SLICEMs	516 out of 4336	11%

Table 1: Device usage

Some problems were faced during synthesis of the design. We included a memory in the design, generated from Xilinx LogiCore IP generator tool. When this memory was included in the design, XST was not able to recognize the memory as a hard IP. We overcame the problem by hard-coding the memory as ROMs by using regular logic cells. The major part of the slices have been occupied by the ROMs for storing the images of the squirrel and mouse pointer.

5. Problems

Not many problems were faced during the design process. Initially we had problems interfacing the newly created user Hardware IPs with the Microblaze processor. But these initial hurdles were crossed once we looked at some demo videos.

We also had some problems integrating the hardware IPs like memories generated from Xilinx LogiCore software with other modules in XST design flow. We overcame this by hard-coding the memories as ROM arrays in the design.

While interfacing the PS2 mouse controller with XST flow, we had some issues with PS2_CLK and PS2_DATA lines which were *inout* ports. We had to make a few changes in the *.mpd* files and also change the ports to *in.* After this we got the mouse working with the XPS.

We faced some problems with having more than one hardware design (VHDL) files. But later we figured it out that the *.pao* file has to be modified in order to be able to use entities described in separate files. We could also have solved this by having all the entity-architecture pairs described in one file, but we found it very confusing and clumsy.

Finally, we faced some random errors in the squirrel image display. The image of the squirrel used to shift randomly inside the square. We could figure out that there was some issue with the addressing of the ROM in which the squirrel image was stored. Despite including conditions for resetting the address every time the squirrel image was read, we were not able to get over this issue. Due to lack of time and also since the problem occurs randomly and occurs very rarely, we left the issue unresolved and it still exits.

6. Contributions

Hardware:

VGA Controller and 7 Segment LED Controller
Can Bilgin

PS2 Mouse Interface, Score Display and Graphics
Balaji S

Software:

Register Programming, PLB Communication and Game Logic
Rakesh MG

Integration and Testing:
Everyone

Report:
Everyone

System Architecture:
Everyone

7. References

1. VGA RefComp

http://www.digilentinc.com/Products/Detail.cfm?NavTop=2&NavSub=451&Prod=NEXYS2

http://www.digilentinc.com/Data/Documents/Reference%20Designs/VGA%20RefComp.zip

2. Nexy2 Reference manual

http://www.digilentinc.com/Products/Detail.cfm?NavTop=2&NavSub=451&Prod=NEXYS2

http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf

3. MouseRefComp

http://www.digilentinc.com/Products/Detail.cfm?NavTop=2&NavSub=451&Prod=NEXYS2

http://www.digilentinc.com/Data/Documents/Reference%20Designs/MouseRefComp.zip