Throttle Copter

Implementation on Xilinx Spartan3E FPGA

Gary Crum

Wes Edens

David DeTomaso

Justin McDowell

***Abstract*—Throttle Copter is a simple video game where the purpose of the game is to avoid crashing your helicopter into the top or bottom of a tunnel or into any of the flying obstacles by controlling the elevation of the copter with one button. The goal of this project was to develop an implementation of this game specifically for the Xilinx Spartan 3E FPGA. This implementation includes a single control button input, a VGA display, sound and LED effects. No processor was used for it's implementation. Instead, the game was instantiated entirely in hardware on the FPGA. The end result is a full implementation of the Throttle Copter video game utilizing 15% of the Spartan 3E FPGA resources.**

I. Introduction

The four engineers who developed this project are students seeking their Master's Degree in Electrical and Computer Engineering at Johns Hopkins University. This implementation of Throttle Copter is their final project for EN.525.442, VHDL/FPGA Design. This implementation of Throttle Copter utilizes various capabilities of the FPGA highlighted in this one semester course including user I/O, VGA interaction, use of intellectual property and CoreGen and basic FPGA design with VHDL.

This project was developed and tested on a NEXYS 2 development board. The NEXYS 2 development board contains a Xilinx Spartan 3E-500 FG320 FPGA along with numerous peripherals which make it well suited for this project. Also necessary for using this project is a simple RC low pass filter in order to play the sound output clearly.

II. Background

This implementation of Throttle Copter does not use a processor. Figure 1 gives a top level view of the VHDL modules instantiated for the design. The primary component of the design is the Game Controller module. The Game Controller module is the only module which takes user input, the Throttle which is tied to button 0 on the NEXYS 2 board. The Game Controller module is responsible for activating the Copter Position module, the Scene Generator Module, the Score Generator Module and the Sound Generator Module.

The Copter Position Module signals the Copter Generator. The Copter Generator, the Scene Generator and the Score Generator all output to the Frame Mixer which feeds the VGA Controller so that the proper image is displayed on the screen.

The final component of the system is the Collision Detector module. This module is able to detect whenever the position of the copter collides with the position of a wall or a flying object and when that collision occurs, a collision signal is send back to the game control and the Sound Generator Module to signal the end of the game. The game can be restarted once a collision occurs by pressing the throttle button again.

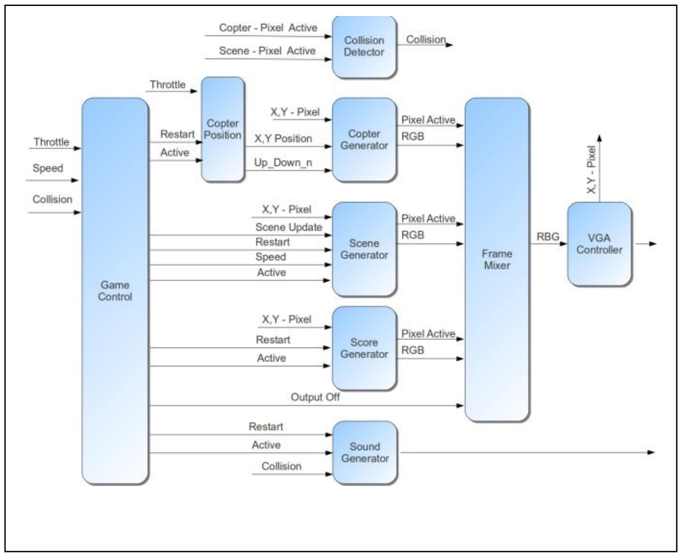


Figure 1. Throttle Copter System Diagram

III. Body

A. Game Controller Module

The Game Controller Module is built using a Finite State Machine (FSM) of seven (7) states as shown in Figure 2. After the asynchronous reset is release the state machine enters the 'IDLE' state. This state and the next state, 'SETUP', are responsible for allowing the other modules to initialize on reset and after a new game. During the initialization the FSM prevents output from being sent to the VGA Controller. This removes image artifacts from being displayed before they have been fully initialized. After the other modules in the game have been given time to initialize, the FSM enters the 'WAIT_FOR_START' state. In this state the FSM waits until the user presses the throttle button. When the throttle pressed condition is met the FSM enters the 'IN_PROGRESS' state and the game begins. In this state the 'GAME_IN_PROGRESS' signal set to '1' and the internal update image pulse generator is enabled allowing the game modules to run. The FSM will remain in this state until the Collision signal is set to '1'. When the Collision signal is asserted by an outside Collision Detection modules, the FSM transitions to the 'DISPLAY_END' state where is will remains for about two (2) seconds before transitioning into the 'WAIT_END_ACK_PRESS' state. The delay was used to allow the user's brain to process the fact that they have crashed and should stop toggling the throttle button. Without this state the user could unintentionally enter into a new game almost immediately after the current game has ended. After the display end delay, the FSM is waiting for the throttle to be pressed and then released. This acts as the continue after game over. When the throttle releases the FSM transitions from the 'WAIT_END_ACK_RELEASE' back into the 'IDLE' state allowing a new game to be setup.

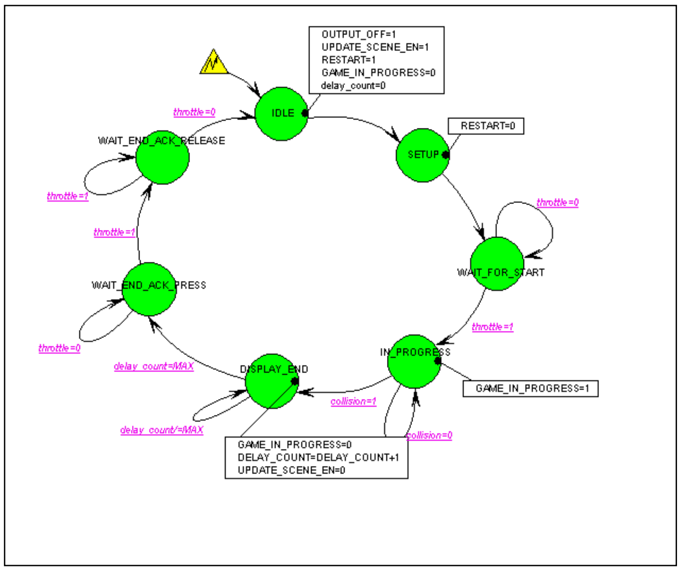


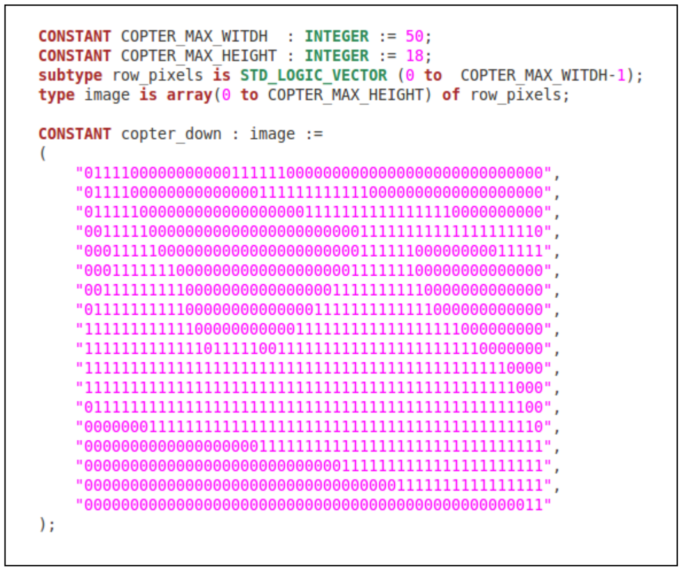
Figure 2. Game Controller State Diagram

B. *Copter Position Module*

The Copter Position Module is used to take the user throttle input and determine the proper VGA X and Y base coordinates to feed into the Copter Generation Module. It is also responsible for keeping track of the last direction of the copter (up or down) and feeding that into the Copter Generation Module. To prevent the copter from moving to quickly this module uses an internal pulse generator that slows the user input down. Without this pulse generator when the user attempts to press the throttle button once, that signal would be registered multiple times by the 50 MHz system clock result in extreme copter movement.

C. *Copter Generation Module*

The Copter Generation Module takes the current copter VGA X and Y base coordinates and the current VGA X and Y pixel coordinates and determines if the copter is present in the active pixel area. This is calculated using the known height and width and creating and imaginary box around the area on the screen that the copter will occupy. If the VGA X and Y pixel position is inside the copter box then this module will look into one of the two copter image lookup tables (Figure 3) to determine if it needs to drive a RGB value out. A '1' in the image lookup table indicates that color is present and a '0' indicates that there is not color present and pixel active should not be driven high.



```
CONSTANT COPTER_MAX_WITDH  : INTEGER := 50;
CONSTANT COPTER_MAX_HEIGHT : INTEGER := 18;
subtype row_pixels is STD_LOGIC_VECTOR (0 to  COPTER_MAX_WITDH-1);
type image is array(0 to COPTER_MAX_HEIGHT) of row_pixels;

CONSTANT copter_down : image :=
(
    "01111000000000001111110000000000000000000000000000",
    "01111000000000000001111111111110000000000000000000",
    "01111100000000000000000011111111111111110000000000",
    "00111110000000000000000000000000111111111111111110",
    "00011111000000000000000000000000111111000000000111111",
    "00011111110000000000000000000111111100000000000000",
    "00111111111000000000000000011111111100000000000000",
    "01111111111000000000000001111111111110000000000000",
    "11111111111100000000000111111111111111111000000000",
    "11111111111110111100111111111111111111111110000000",
    "11111111111111111111111111111111111111111111110000",
    "11111111111111111111111111111111111111111111111000",
    "01111111111111111111111111111111111111111111111100",
    "00000001111111111111111111111111111111111111111110",
    "00000000000000000001111111111111111111111111111111",
    "00000000000000000000000000001111111111111111111111",
    "00000000000000000000000000000000001111111111111111",
    "00000000000000000000000000000000000000000000000011"
);
```

Figure 3. Copter Generation Image Constant

D. *VGA Mixer Module*

The VGA Mixer module provides a mechanism to layer the four video components (background, scene, copter and score) into a single layer that feeds pixel data into the VGA Controller. Each source layer provides the mixer with their 8-bit Red Green Blue (RGB) value for the current pixel and a pixel active signal to indicate if they would like to drive a color out to the VGA controller. The background layer is an implied layer; in the absence of a pixel active from any of the source layers the default color is driven out to the VGA controller. The default color is set using a VHDL generic.

The following list is the order used for layering in the Throttle Copter game. The list is organized from the back layer to the top layer.

- 0 - Implied Background Layer
- 1 - Scene and Obstacles
- 2 - Copter
- 3 - Score

E. *Collision Detetection Module*

The Collision Detection module is responsible for detecting when the copter has crashed into the cave walls or the obstacle blocks. The detection is done by performing a logical AND with the copter pixel active signal and the scene pixel active signal. For proper handshaking of the collision signal back to the Game Controller module the Collision Detection modules also performs a rising edge detection of the collision signal and will only clear this flag when the Game Controller module triggers a Game Restart condition.

F. Scene Generation Module

To generate the cave walls, this module consists of two main branches which share access to a block ram. The block ram contains one 64-bit entry for each column of the cave which stores four coordinates: location of the cave ceiling, location of the cave floor, the top of the obstacle, and the bottom of the obstacle. During the rendering of the display, this information is read and through a series of comparators, a decision is made as to whether the current pixel is filled or empty. During the dead time of a frame, the “write” state machine shifts all of the columns to the left (by incrementing an offset variable) and adds new columns at the right of the screen. The overall motion of the cave wall is generated by using a linear feedback shift register (LSFR) to pick a new slope every 128 columns. The ceiling/floor locations are stored with 12 bits of decimal precision so that arbitrary slopes can be selected. In addition, a second LSFR generates a smaller (3-bit) noise signal which is added to the gentle sloping of the wall to create a jagged appearance. A third LSFR generates a new obstacle position at a fixed interval and based on a counter value. For columns that do not contain an obstacle, the locations for the top and bottom of the obstacle are written with the bottom above the top so that no pixel could evaluate as being “within” the block. At the beginning of the game, the state machine undergoes 640 iterations to write the entire first screen before entering a “waiting” state for its next update. During the game, the machine adds a variable number of columns each frame (set by the slider switches) to create the illusion of forward motion.

G. Sound and LED Generation Module

The Sound and LED Generation module is responsible for generating the background sound for the game and generating the LED scrolling pattern for the game. Also, this module is responsible for responding to a detected collision by flashing all LED's and playing an end of game sequence of notes.

This module utilizes a Digital to Analog Converter (DAC). This piece of intellectual property was provided for us by John Logue [1]. In order to use the provided DAC, one of the four Digital Clock Managers is utilized in order to generate a 100 MHz clock. All processes in the Sound and LED Generation module were run on this 100 MHz clock.

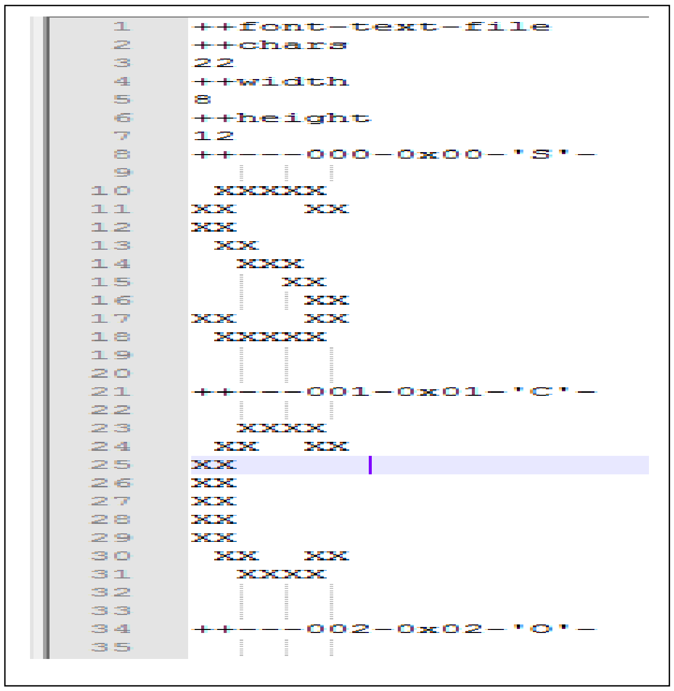
Also necessary for generating tones in this module was the implementation of a Sine Lookup Table. This was generated using Xilinx Coregen. The key component for this Sound and LED Generation module is it's six (6) state Finite State Machine which identifies at what frequency the Sine Lookup Table is incremented as well as setting the output pattern for the LED's.

The output of the DAC is fed into a simple RC low-pass filter in order to achieve a clean sound for the game.

H. Score Module

The purpose of the score module is to display text on the VGA screen that keeps a running count of how long/far the player has traveled during game play. To start off, the position on the screen will be a constant set by a generic parameter. The inputs to the module will need to be the current pixel location, the pixel clock, a game active signal, a score increment signal, and a reset for when a player crashes. The only outputs that we need to worry about are the enable and color lines that tell when the VGA signals are displaying the current pixels associated with text, and the color of the text.

The first thing that we need is a table of pixel masks for each letter. To create this, an open source c code file from Sourceforge [2] was used. This turns a text file, such as the one shown in figure X, into a postscript file. A perl script was written to reformat the postscript font file into a ‘.coe’ file that is used to initialize a ROM created in CoreGen.



```
++font-text-file
++chars
22
++width
8
++height
12
++---000-0x00-'S'-

 XXXXX
XX   XX
XX
 XX
  XXX
    XX
     XX
XX   XX
 XXXXX


++---001-0x01-'C'-

  XXXX
 XX  XX
XX
XX
XX
XX
XX
 XX  XX
  XXXX


++---002-0x02-'O'-

```

Figure 4. Font Text File

In addition to the font ROM, a font digit LUT was used to index the addresses of the start of each digit in the ROM. With this data available, all that is needed is a way to convert the pixel locations into addresses that can index what the text is supposed to be. For the incrementing score, it was a little more difficult. The ‘SCORE:’ was constant on the screen so no change was necessary. But the actual numbers needed to increment as the user progressed further through the level. To make this part, vector was created that was incremented from the outside by a strobe and indexed digits inside the module by groups of four (4). This caused a problem because when a digit exceeded nine (9) it still had to increment past 15 to register for a rollover in the regular digits. To solve this problem, this implementation counts in hex to have each of the four bits be meaningful. A block diagram can be seen in figure 5.

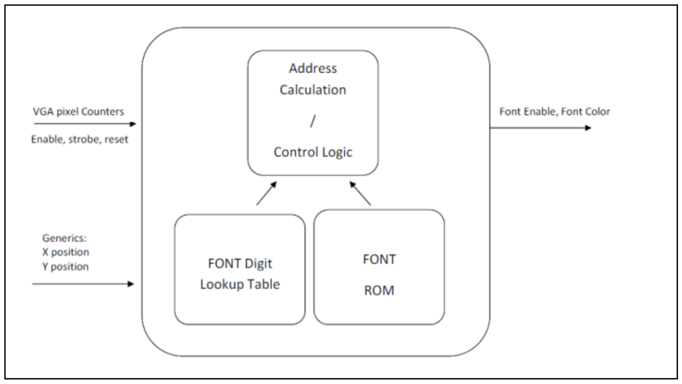
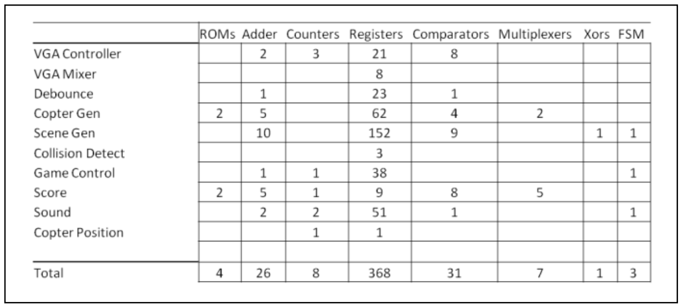


Figure 5. Displaying Font to the VGA

IV. Results and Analysis



	ROMs	Adder	Counters	Registers	Comparators	Multiplexers	Xors	FSM
VGA Controller		2	3	21	8			
VGA Mixer				8				
Debounce		1		23	1			
Copter Gen	2	5		62	4	2		
Scene Gen		10		152	9		1	1
Collision Detect				3				
Game Control		1	1	38				1
Score	2	5	1	9	8	5		
Sound		2	2	51	1			1
Copter Position			1	1				
Total	4	26	8	368	31	7	1	3

Figure 6. Resource Utilization

Minimum period: 19.420ns (Maximum Frequency: 51.493MHz)
Delay: 19.420ns (Levels of Logic = 20)
Source: vga_inst/pixel_y_2 (FF)
Destination: vgaMixer_inst/vgaGreen_2 (FF)
Data Path:vga_inst/pixel_y_2 to vgaMixer_inst/vgaGreen_2

Minimum input arrival time before clock: 6.927ns
Offset: 6.927ns (Levels of Logic = 14)
Source: sw<1> (PAD)
Destination: cave_gen_inst/colwrite/to_add_9 (FF)

Maximum output required time after clock: 11.217ns
Offset: 11.217ns (Levels of Logic = 6)
Source: vga_inst/horizontal_counter_5 (FF)
Destination: vgaRed<2> (PAD)

V. Conclusion

This project utilizes approximately %15 of the Spartan 3E FPGA resources and is a successful implementation of the Throttle Copter game. No processor was used and the final version includes a VGA display, sound and LED effects, and a one button throttle user input. Given more time for this project, the engineers would look to implement a scoring system in decimal instead of hexadecimal and implement a method of playing better background music possibly by playing sound files stored in ROM. Also given more time, future improvement to the Copter Generation could be made by utilizing the internal block-ram to store the Copter images. Using the block-ram would allow the amount of data stored to be increased from 1-Bit color per pixel to the full 8-Bit RGB color pallet available on our development platform.

This project exercises a few of the many capabilities of the Xilinx Spartan 3E on the NEXYS 2 Development board. Had a processor been used and software developed for the game rather than hardware, it would have been a much simpler task. However, by implementing it entirely on the FPGA, the engineers were able to demonstrate some of the many hardware capabilities of the FPGA. Although it may not actually be practical to implement this game on an FPGA when it can more simply be done in software, there are still practical purposes for all of the different components of this end system and this game serves as a good way of demonstrating and tying them together.

References

[1] J. Logue, "Virtex Synthesizable Delta-Sigma DAC", September 23, 1999

[2] http://nafe.sourceforge.net/