Pirate Cruiser

Technical Documentation

By
Thomas A. Deane
Marcello R. Pantuliano
Ambert C. Ho

Pirate Cruiser
Technical Documentation

Tunnel Cruiser is a 3D video game consisting of a ship flying through exponentially growing squares that give the impression of a tunnel. The objective of the game is to fly through as many squares as possible without crashing against one of the "walls". This document presents a detailed description of the system.

Outline

1. Interface tools
2. The big picture – Theory of Operation
3. Detailed module descriptions
4. Memory units
5. Implementation details
6. Extensions
7. User Manual

1. – Interface Tools:

a. Hardware:

The digital design was implemented on a Xilinx Spartan 2 Field Programmable Gate Array (FPGA). The Spartan 2 FPGA was mounted on an XSA board with a parallel port input coming from the PC and a VGA output. The XSA board was mounted on an XStend board with the purpose of connecting the Sega Gamepad as the user interface tool to play the game.

b. Software:

The hardware was described using Verilog. All of the modules were simulated, tested, and debugged using ModelSim.

2. – The big picture:

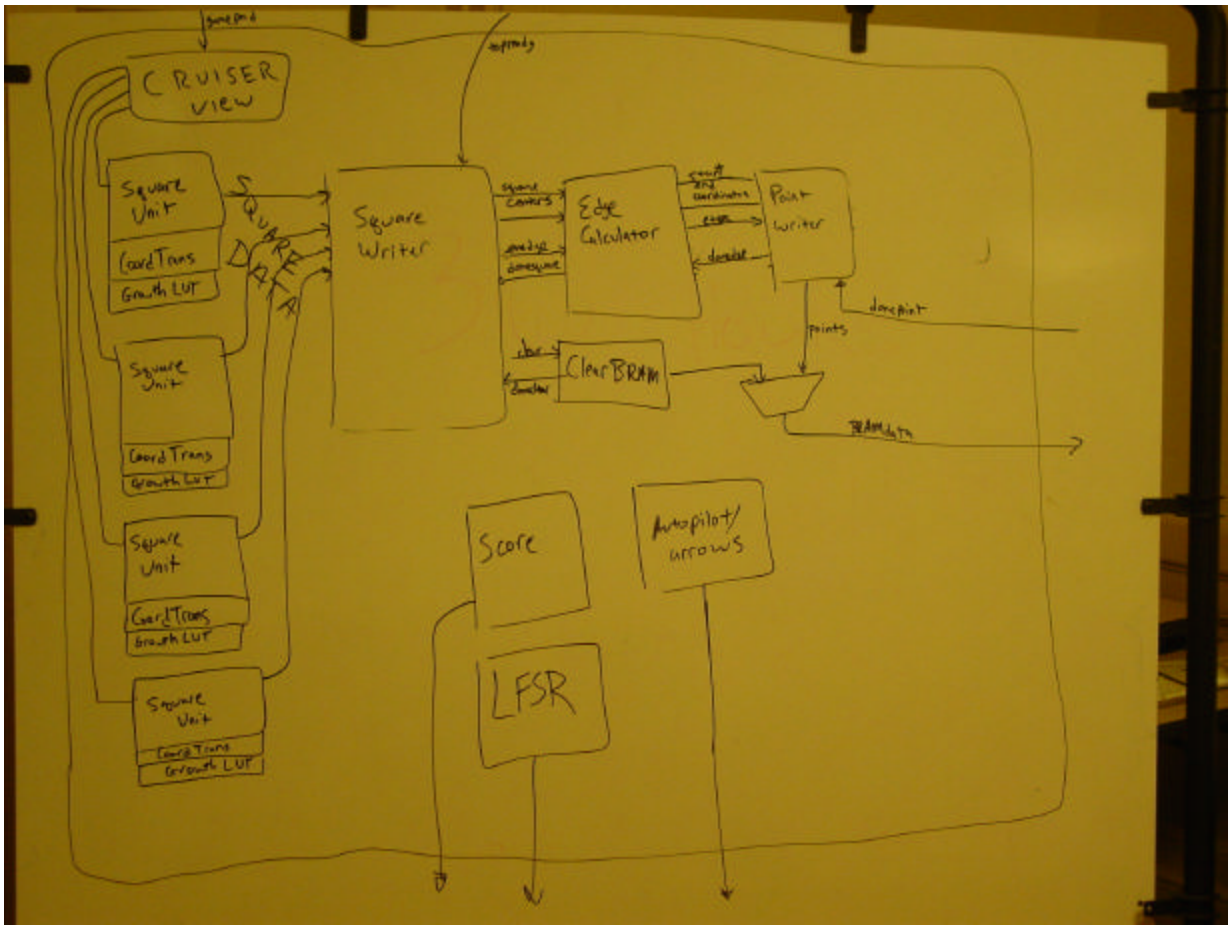
One of the greatest challenges of this project was translating the player's ideal conception of the game into a designer's problem. For example, whereas the player sees a tunnel in the screen, the designer thinks of it as 4 squares of different sizes. A sense of moving into the tunnel is achieved by making the squares grow faster as their depths decrease to give a pseudo 3-D feel to the game. Finally, whereas the players feel they are able to move around the tunnel, the designer understands that such is the result of shifting squares given a certain input.

The problem of drawing the squares was factored into hierarchical modules that communicate with each other. A state machine cycles through each of the 4 squares to be written and outputs their centers. Another module receives this information and then cycles through each of the four edges, outputting their start and end points. The

next module receives the start and end points and sends single coordinates to another module. Finally, a module updates the information of this single point and sends a done signal to the point-generating module. The point module then sends its next point until it is done with all the points of an edge, at which point it sends a ready signal to the edge module. The edge module sends the next edge, and similarly, sends a done signal to the square module once all four edges are drawn. The square module sends the next square until it is done drawing all the squares. At this point, the cycle repeats itself.

3. – Detailed Module Descriptions:

The design was factored into 5 main modules and a series of sub-modules. This section describes these modules in a hierarchical order, starting with the highest abstraction (top modules) and ending with the modules that deal with single square points.



a. Top Module: the top module acts as an interface between the VGA output and the game data contained in two BRAMs (Block Random Access Memory). Its functionality can be broken apart into two subcategories:
   i. VGA Display: all of the color, graphic, and character logic take place in this module. Two counters that represent the VGA electron guns sweep the screen at 50MHz. The X and Y value of these counters are mapped and used to retrieve data from the BRAMS, display characters stored in the TCGROM, and draw screen sections of a specific color.
   ii. Two Screen Buffer: To achieve a smooth frame transitioning, our design features a two-screen buffer. The top module contains a signal that determines which of two BRAMs will be used for reading the data and displaying it on the screen. While the VGA reads one of the BRAMs, the top module writes the next frame on the next BRAM. When the VGA counters reach the end of the screen, the two BRAMs swap their roles.

b. Tunnel: The tunnel module serves a secondary top module for all of the main modules that make up the game. Its main purpose is to establish a communication link between all the modules to ensure proper functioning. Its secondary purpose is to output a two-bit wire containing data to be written to the BRAMs. A finite state machine (FSM) controls the process of updating the BRAM data. Tunnel is responsible for checking if the user is within bounds to score by checking the position of the square center relative to the cruiser view with combinational logic. This same logic is applied to the arrow and autopilot features in later versions of the game. Tunnel also manages the Linear Feedback Shift Register (LFSR), which produces pseudorandom points to generate seemingly random topography. The Tunnel module inputs are the Gamepad buttons, and a ready signal from the top module that allows for the next frame to be written.

c. Square Unit and Cruiser View: these modules are responsible for storing the squares' X-Y centers, their respective depths, and the player's relative position within the game area. When the player presses the left button, the module shifts the squares to the right and the cruiser view (the player's vision) to the left by an amount that depends on the square's depth. The inputs to these modules are the Gamepad buttons. Square Unit uses CoordTrans to produce square centers relative to the upper-left corner of the Cruiser View, which all subsequent calculations are based on.

d. Square Writer: This module contains the master FSM of the work modules. For the majority of the running time, the FSM remains on an IDLE state. Once the top module is ready for a new frame, the FSM goes into the CLEAR state. In this state, the Square Writer module calls upon the ClearBram sub-module to remove any previous data in the BRAMs so that a new frame can be written. Once all addresses of the BRAMs are cleared, Square Writer sends the coordinates of Square 1 (obtained from the square unit) to the Edge Calculator module. Once this module finishes with all squares, its FSM goes back to the IDLE state.

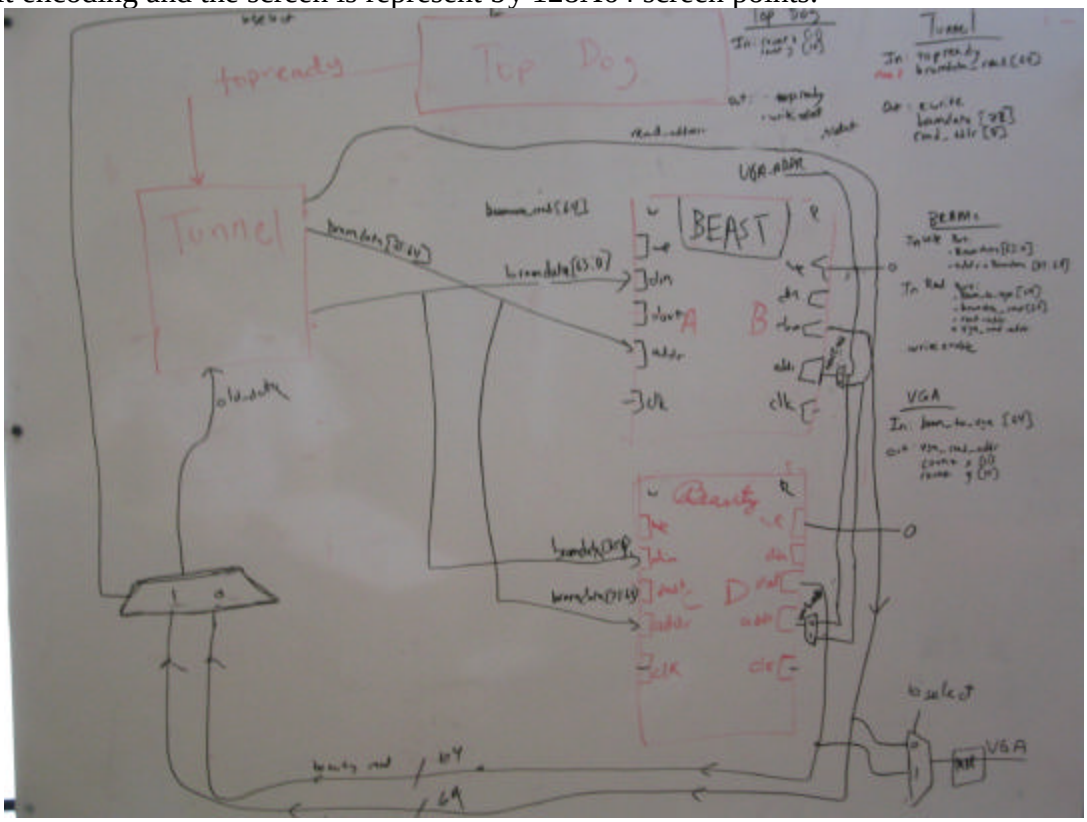
e. Edge Calculator: The Edge Calculator module takes in the X-Y coordinates of a square center and its respective depth. It then calculates the edges that need to be drawn to complete a square. An FSM cycles through each of the edges, outputting start and end coordinates, as well as the type of edge (vertical or horizontal) to the point writer module. Once all the edges are done, the edge calculator sends a signal to the square writer in order to obtain the information about the next square.

f. Point Writer: This module takes in the start and end coordinates of a single edge. Two counters (one for vertical and another for horizontal edges) run through all possible values that can be written to the BRAM. When the counters reach a point that is contained between the start and end coordinate of an edge, a valid signal goes high so that a point can be written to the BRAM. Tunnel gets the X-Y coordinate of the point and it is able to update the BRAM in 4 clock cycles. Once the process is completed, Tunnel sends a "done point" signal to the Point Writer module so that the next point can be considered.

g. Submodules: a series of basic, small sub-modules help larger modules operate properly. The slow clock module serves as the enable signal for the square unit (updating square's data). A look up table determines the growth rate of squares according to their depths. Among the most basic sub-modules are D-Flip-Flops, counters and an edge detector.

4. – Memory Units:

Two Dual-Port BRAMs were used to store the display data. The dual port feature of the BRAMs is utilized such that one port is used exclusively for writing, the other exclusively for reading. The BRAMs are each 2 bits wide and 8192 bits long. This dimension allows for convenient data updating since each screen point is described by a 2 bit encoding and the screen is represent by 128X64 screen points.



5. – Implementation Details:

The results seen in Table 1 (Appendix) reveal the size of this project. The first time we tried implementing the design, we were already above the maximum area allowed by the FPGA. We solved this problem by re-designing the shape of the BRAMs from 64X256 bits BRAMs to 2X8192 BRAMs. This new implementation allowed us to remove enough logic to place us within the maximum range. Nevertheless, the design did not meet timing requirements either. After pipelining most of the signals involving large amounts of logic gates we were able to reach a desirable clock frequency.

Mapping: A mapping system is used to translate the 128 by 64 point world into 8192 addresses by concatenating the seven-bit X position with the six-bit Y position, creating a 13-bit vector to describe the address. The reverse was used to extract the (X,Y) position from the address.

Timing and Optimization: In order to meet the timing constraint of 20ns, the design needed to be pipelined to account for disproportionate delays in modules with heavy gates delays or large amounts of data. Pipelining is accomplished by storing I/O signals in flip-flops to delay them one clock cycle in order to give the internal signals time to be calculated. If an I/O signal based on internal signals that face serious gate delays is sent out without being delayed, it could send out incorrect information until the gates have time to process the internal signals. The flip-flops give the gates time to catch up and process the data, so there is always a correct output. This often leads to problems between modules, especially between delayed handshaking signals and FSMs. For this reason, an extra idle state in Edgecalculator was used.

After the tunnel module was completed and pipelined, the area constraint on the Spartan II FPGA became a problem, which required copious optimization. Looking back on the logic used, it became necessary to simplify large comparators into single bit comparators that performed the same function. The LFSR was reduced from 32 bits to 11 bits, since the design did not need the extra 21 bits. Most of the optimization took place in the combinational logic, although the largest change was switching the BRAM configuration, as discussed above.

6. – Extensions:

Our project features a series of entertaining extensions:

- Pause button: halts square unit and cruiser view modules.
- High Score: separate reset button resets this value.
- 3 Speeds: 3 different slow clocks control the growth rates of the squares.
- Crazy mode: game area disappears and squares change colors giving the impression of taking crazy pills, or space travel. The tunnel is replicated 9 times to increase difficulty of the game.

- ✍ Navigation arrows: four arrows around the game area let the player know how to reach the squares centers.
- ✍ Autopilot: pressing a button disables Gamepad direction inputs. The system then uses the internal navigation arrows to control the player's position in the world.

7. – User Manual:

The objective of this game is to fly through as many squares as possible without crashing against one of the tunnel walls! The rest of this section explains how to use three different versions of Pirate Cruiser.

Backstory: Pirate Captain Jack Sparrow needed a ship. So he decided to steal the fastest ship in the empire, the starship Enterprise from his lifetime rival Captain Kirk. After beating Kirk ferociously with a stick, Sparrow finally succeeds into stealing the Enterprise. However, the evil Space Balls have erected a powerful force field designed to keep Sparrow from escaping. Luckily the Enterprise is equipped with an extremely big gun that can blast through the field, creating a tunnel Sparrow can use to escape. However, the gun is so extreme that it is inaccurate and the tunnel it creates is very erratic. That's where you come in! As an overworked, underpaid electrical engineer, you were liberated from prison Packard 128 to help Jack get out of the mess. Argh!

- ✍ **Pirate Cruiser: Extreme Treasure Hunter!** MSRP: $49.99
- ✍ Bitfile: pirate1.bit
- ✍ Move around the tunnel by pressing the Gamepad arrow keys. In case of crash, press the C button to restart the game. Features fancy graphics, such as skulls and crossed swords!

(*from pirate1)*

- **Pirate Cruiser 2: When All Exits are Blocked, the Only Thing to do is Turn Up the Speed!** MSRP: $59.99
- Bitfile: pirate2.bit
- Now try moving at different speeds across the tunnel! Press the C button once and the game goes twice as fast. Press the C button once more and you will go into Crazy mode (not recommended for beginners). If you need to take a break, press B to pause the game.

(from pirate2)

- **Pirate Cruiser 3: Argh! Captain Jack Sparrow Learns How to Fly with the Children** MSRP: $9.99 (under 4th grade only)
- Bitfile: pirate3.bit
- Features navigational tools and autopilot. This version of the game is recommended for beginners or young children. Follow the arrow directions to successfully pass through a square. In case of panic, press the B button to activate autopilot. The system will navigate the ship for you without risk of crashing.

(from pirate3)

Appendix

Table 1:

Device utilization summary:

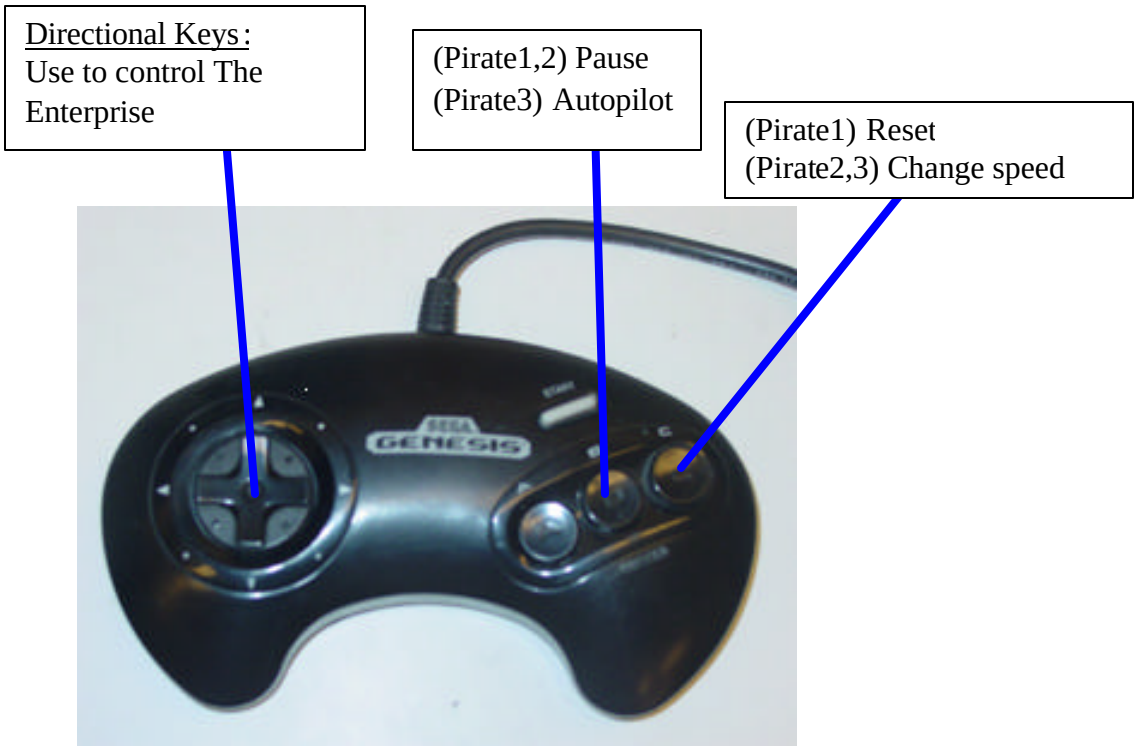
Number of External GCLKIOBs	1 out of 4	25%
Number of External IOBs	15 out of 92	16%
Number of LOCed External IOBs	15 out of 15	100%
Number of BLOCKRAMs	8 out of 10	80%
Number of SLICEs	1198 out of 1200	99%
Number of GCLKs	1 out of 4	25%

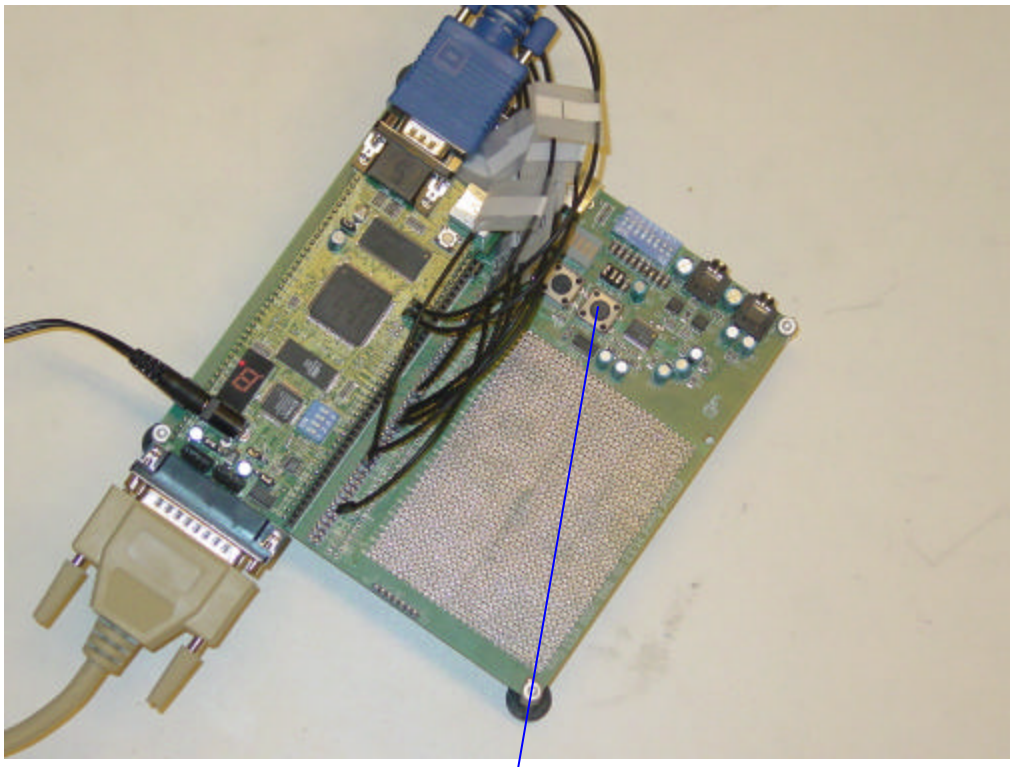
Total equivalent gate count for design: 151,254

Timing Summary:

Minimum period is 23.512ns.

Minimum period before extensions added: 19.6125ns.





(Pirate1) High Score Reset
(Pirate2, 3) Master Reset