Ball Catcher - Mini Project Report

Dylan Fu and Quentin Heng
The University of Auckland, dfu987@aucklanduni.ac.nz, qhen143@aucklanduni.ac.nz

ABSTRACT

The aim of this project is to develop a simple pong game implemented on a FPGA designed in VHDL. This report outlines the design specifications, game strategy, block diagram, final system design and implementation, design decisions, performance and optimisation, and conclusions and future improvements. The appendices include the extended block diagram.

INTRODUCTION

Ball Catcher is a simple 2D game that focuses on a player-controlled platform to collect as many balls as possible within a set time frame per level (game mode). The game has three levels with increasing difficulty. The player must move the platform at the bottom of the screen to collect the balls onscreen and consequently get a higher score. Once a ball has been collected the ball will reset pseudo-randomly at the top of the screen. To complete the game, one must finish all three levels.

DESIGN SPECIFICATION

The game was created using the following equipment:

- DEO Package
    - DE0-Board (see appendices)
    - USB Blaster Cable
- PS/2 Mouse
- VGA Cable

The DE0 board will act as the game console, the game will then be displayed on a monitor through a VGA cable at a resolution of 640 x 480 pixels. The user will interact with the game through the DIP switches and push buttons on the DE0 board and through a PS/2 mouse.

The ball(s) will move linearly where they will bounce indefinitely within the perimeter of the game screen, unless the ball is caught or the game ends. The platform is a user-controlled entity, where the user can use the PS/2 mouse to control the x-position of the platform to catch a moving ball(s). A ball is considered caught when the bottom of the ball touches the top edge of the platform. When the user catches a ball, the score will increase, and the ball will spawn randomly in a pre-defined region at the top of the screen, falling at a random angle. The current level will be complete once the score reaches 30. Otherwise when the time reaches 0, the game will be over. Once a level is completed, it will transition to intermediate screen until the player is ready to play the next level. After all, three levels have been completed, the user will be given the option to return to the main menu or to play again from level 1.

The levels will be designed, so that with each level, the difficulty will increase. This will be done by increasing the amount of moving balls, speed of the balls, and by decreasing the size of the platform with each level.

GAME STRATEGY

The main objective of the game is to complete all three levels by collecting balls. The game will consist of three levels with increasing difficulty. Each level will have a time limit where the player will try to reach the target score before the time limit has been reached. Once the player completes a level, the player will be shown on a transition screen when transitioning between levels. The player can then move on from the transition screen by interacting with the push buttons.

Practice: The Practice level contains one slow moving ball that the player wants to catch. There is no score limit in practice mode and the game only ends when the time runs out.

Level 1: The first level will also have one slow moving ball on the screen. However, the player needs to reach the target score to progress to the next level. The player will be able to move the platform across the screen faster than the ball; therefore, the player can easily catch the ball in this level.

Level 2: The second level will have two balls moving on screen. However, the ball speed will be faster than level 1. The player's platform will decrease in size, so the player needs to be more precise to catch the balls.

Level 3: The final level will be more challenging than the rest. The third level will feature three fast-moving balls on screen, moving faster than the ones in level 2. The platform will again decrease in size, so it is just larger than the diameter of a ball. The player needs to be precise and react quickly to catch multiple balls in succession in this level.

BLOCK DIAGRAM

The block diagram in Appendix C shows the relation between the components used. The paddle is controlled by mouse input. The movement of the ball is subject to the game screen and the paddle location. Catching the ball will increment the score register which in turns updates the score displayed dynamically. The ball spawn position is determined pseudo-randomly by the 'LFSR'. Game texts are controlled by the text gen component which outputs the address of each character to 'Char_ROM'. All visually

displayed components are synced with the game component before being displayed on the screen by 'VGA_SYNC'. The counter will track the time elapsed through the clock signal and update the text displayed.

System Design & Implementation

The design utilises a Finite State Machine (FSM) to control the multitude of screens in the game. The FSM has a total of eight states; 000, 001, 010, 011, 100, 101, 110, 111. These states represent the main menu, practice mode, level 1 of single player, level 2 of single player, level 3 of single player, level complete screen, lose screen, and win screen respectively.

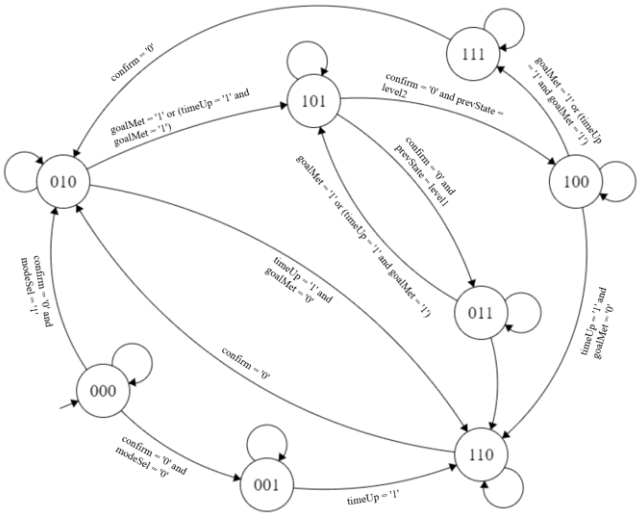


Figure 1: FSM Diagram

The FSM is used to determine which level the user is at and will output the specific state as an output of the FSM component. This output is then passed as an input into other components, and depending on the output mode, the game will then show the relevant text and graphics for the game.

The output mode is passed into the 'text_gen' component and depending on the mode 'text_gen' will determine what text to display on screen such as the menu text when the output mode is '000' or display the game variable when the output mode is either '001', 010', '011', '100'. The 'text_gen' component assigns a char address to the 'Char_ROM' component to handle display of text.
All the RGB signals from the components 'Char_ROM', ball, and platform are passed into AND gates then to the 'VGA_SYNC' component, so that when all the RGB signals are high then the 'VGA_SYNC' will display the colour white for that specific pixel, otherwise it will display another colour. Therefore, the game elements such as the ball/s, platform, text will be displayed on screen.

The ball component has two processes, 'RGB_Display' for handling the RGB display of the ball and the 'Move_Ball' for handling the game logic. The 'RGB_Display' process relies on signals 'Ball_X_Pos', 'Ball_Y_Pos', 'pixel_column', and 'pixel_row'. These signals are from the Move_Ball process and from the 'VGA_SYNC' component respectively, which are then used to determine when to display the ball on screen.
The 'Move_Ball' process is executed whenever the screen refreshes; the process will calculate the new 'x' and 'y' ball positions and when the ball collides with the platform, the ball will reset at the top of the screen at a pseudo-random 'x' position determined by the 'LFSR' component. It will also send status signals to the 'scoreAdder' component to increment the score. The movement of the ball is increased as we progress in levels, thus making the game more difficult as the player progresses in the game.

The 'platform' component handles the movement of the platform and the RGB signals for displaying it on the screen. The process handling the movement assigns 'Platform_X_Pos' the 'mouse_col' value. The 'Platform_Y_Pos' is a constant signal that is never changed. The 'platform' component has a 'RGB_Display' process which relies on 'Platform_X_Pos', 'Platform_Y_Pos', 'pixel_column', and 'pixel_row'. These signals are from the process handling the movement and from the 'VGA_SYNC' component respectively, which are then used to determine when to display the platform on screen.

The timer component is a basic down counter that decrements in time from 59:59 to 00:00. The timer has generic single digit sub-components which aren't specific to this project. The timer is modular and can be reused in other projects.

When in game, if the pause switch is switched on then all the processes and the timer is halted. However, the FSM will still be running in case the user wants to exit to the main menu.

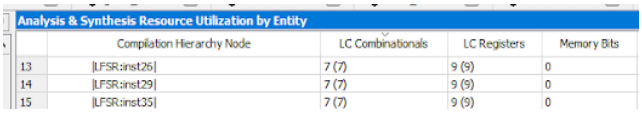
Design Decisions

Our design decisions affect the performance of the product and the reusability of the components. Additional features can result in more logic elements thus more expensive elements and possibly longer critical paths. Our design choices take into account these drawbacks whilst considering their overall benefits.

The game was split up into multiple components which each control its own functionality of the game. This meant we could easily change the functionality of any component without having to change the rest of our design. An example of this would be altering the properties of the balls. With our design, the modification of the speed of all three balls can be done by simply altering the ball component.
However, designing our game modularly results in more wiring in between the components thus more logic elements.

Initially, one LFSR was used to determine the spawn location and angle of all the balls. However, there was an

edge case where this caused gameplay issues. When two or more balls were caught at the same time, the spawn location and angle would be the same due to using the same generated seed from the LFSR.
To circumvent this issue, we decided to use a different LFSR for each ball. This means the spawn location and angle between all the balls will differ. Each LFSR also has a different clock signal so that the chances of the LFSR loop synchronising is lower.
Unfortunately, we are sacrificing more logic elements to implement these additional LFSR components. Figure… shows the logic elements that each LFSR uses.



Analysis & Synthesis Resource Utilization by Entity

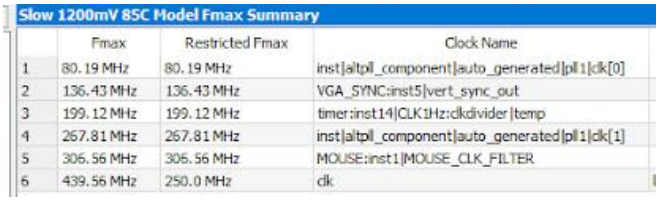
	Compilation Hierarchy Node	LC Combinationals	LC Registers	Memory Bits
13	|LFSR:inst26|	7 (7)	9 (9)	0
14	|LFSR:inst29|	7 (7)	9 (9)	0
15	|LFSR:inst35|	7 (7)	9 (9)	0

Figure 2: Resource Utilisation

The pause functionality of the game was implemented using a switch instead of a push button as specified in the brief. This was a conscious design decision made by us because it makes more sense to the end user. It is easier to tell the game is paused by looking at the position of the switch than pressing a push button. There is also a visual cue to indicate the pause state of the game if this is unclear.

PERFORMANCE & OPTIMISATION

The maximum restricted operating frequency of our components is 80.19MHz. Our critical path determines this frequency because the game needs a certain amount of time for the signals in the critical path to propagate.



Slow 1200mV 85C Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	80.19 MHz	80.19 MHz	inst|altpll_component|auto_generated|pll1|clk[0]
2	136.43 MHz	136.43 MHz	VGA_SYNC:inst5|vert_sync_out
3	199.12 MHz	199.12 MHz	timer:inst14|CLK1Hz:clkdivider|temp
4	267.81 MHz	267.81 MHz	inst|altpll_component|auto_generated|pll1|clk[1]
5	306.56 MHz	306.56 MHz	MOUSE:inst1|MOUSE_CLK_FILTER
6	439.56 MHz	250.0 MHz	clk

Figure 3: Max Freq. Summary

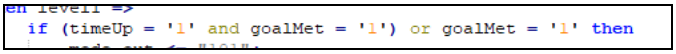
The resource utilisation table in appendices A shows the number of logic elements used by each component.
Most of the total logic elements used, are from the text_gen and ball components. A third of the total used are from just the text_gen component due to the abundance of 10-bit comparators in our implementation. Ball utilises around 175 logic elements each. This is sustainable for further game strategy changes such as adding more balls since each ball takes around 1% of the logic element capacity available in the FPGA.

The memory bits used are from storing the representation of characters or any other 8x8 pixel image. A MIF file containing the character data is read when the game is run initially. These are then used to help render text by reading the specified addresses.

Optimisation is important in terms of cost and efficiency for any RTL design. The use of logic elements can affect our critical path. The cost of logic elements may seem minuscule in singular designs but in mass productions can save thousands of dollars by simply reducing the number of elements used. In this project, this consideration is not applicable, but it is still important to reinforce good practices.

Our timer component uses a self-defined clock divider. It would make sense to replace that clock signal with one from the existing PLL since it does not use additional resources.

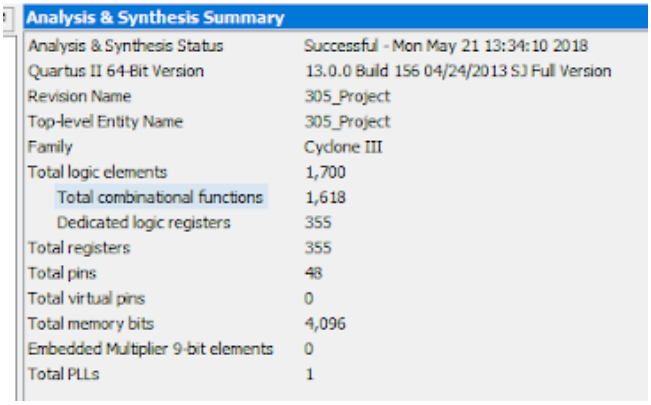
By minimising redundant implementations reduce our logic elements. In figure 5, the code in the parentheses is redundant thus can omitted and reduce our critical path since there is one less logic gates in its path. In previous iterations we had cascading adders, refer to Appendix B. After using correct parentheses in the VHDL design, this removes cascading adders and consequently reduces the critical path.



```
en level1 =>
  if (timeUp = '1' and goalMet = '1') or goalMet = '1' then
```

Figure 4: Redundant Code

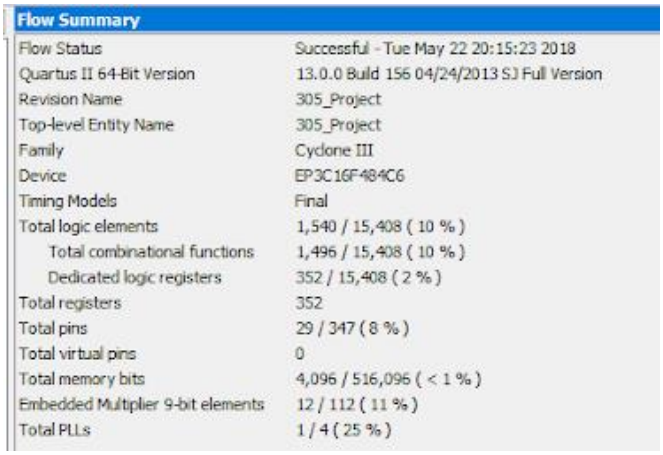
Figure 6 & 7 shows how simply removing latches can reduce the number of logic elements significantly.



Analysis & Synthesis Summary

Analysis & Synthesis Status	Successful - Mon May 21 13:34:10 2018
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Full Version
Revision Name	305_Project
Top-level Entity Name	305_Project
Family	Cyclone III
Total logic elements	1,700
Total combinational functions	1,618
Dedicated logic registers	355
Total registers	355
Total pins	48
Total virtual pins	0
Total memory bits	4,096
Embedded Multiplier 9-bit elements	0
Total PLLs	1

Figure 5: Flow Summary - Before Optimisation



Flow Summary	
Flow Status	Successful - Tue May 22 20:15:23 2018
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Full Version
Revision Name	305_Project
Top-level Entity Name	305_Project
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	1,540 / 15,408 (10 %)
Total combinational functions	1,496 / 15,408 (10 %)
Dedicated logic registers	352 / 15,408 (2 %)
Total registers	352
Total pins	29 / 347 (8 %)
Total virtual pins	0
Total memory bits	4,096 / 516,096 (< 1 %)
Embedded Multiplier 9-bit elements	12 / 112 (11 %)
Total PLLs	1 / 4 (25 %)

Figure 6: Flow Summary – After Optimisation

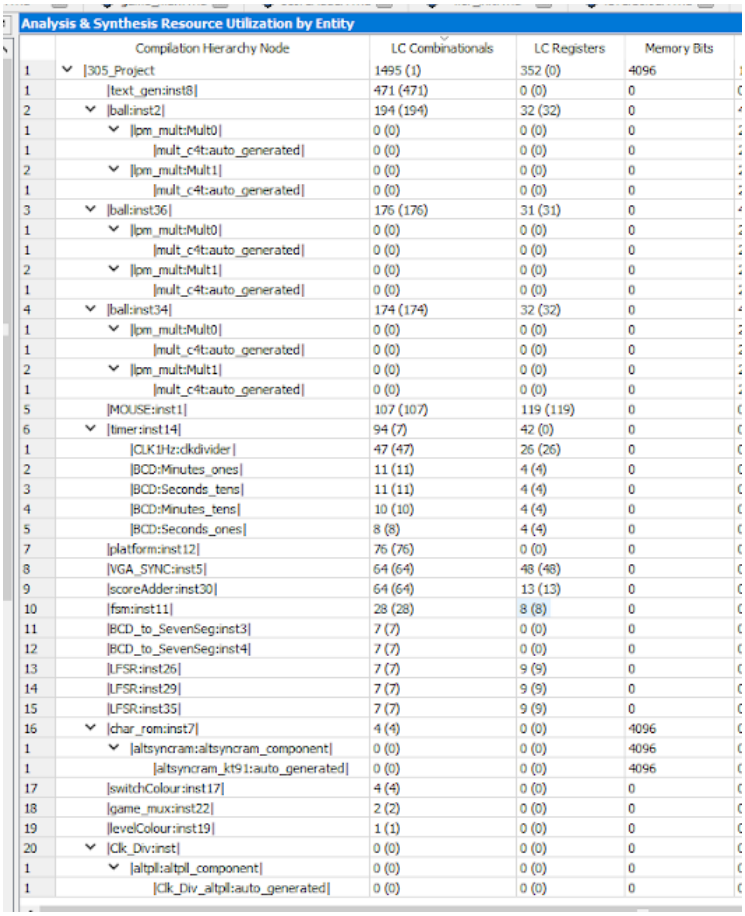
CONCLUSIONS & FUTURE IMPROVEMENTS

The aim of this project was to develop a pong inspired game on a simple game console using only digital logic and design. The game was implemented on a FPGA, while the digital logic and design was implemented in VHDL. Ball Catcher is a simple game involving balls and a platform. The game has a resolution of 640 x 480 and it displayed through a VGA cable. The final game includes three levels with increasing difficulty as the user progresses through the levels. In game, the user is shown important game variables such as score, level, and time remaining. The main objective of the game is to complete all three levels within the allotted time in each level.

With additional time to further iterate and refine the game, some future improvements can be made to further develop the game. A potential improvement could be to improve the graphics of the game by implementing 4bit graphics instead of 1bit graphics. Also, additional features could be added to make the game more fun and exciting to the user such as adding more balls, powerups, and increasing the amount of levels in the game.

APPENDICES

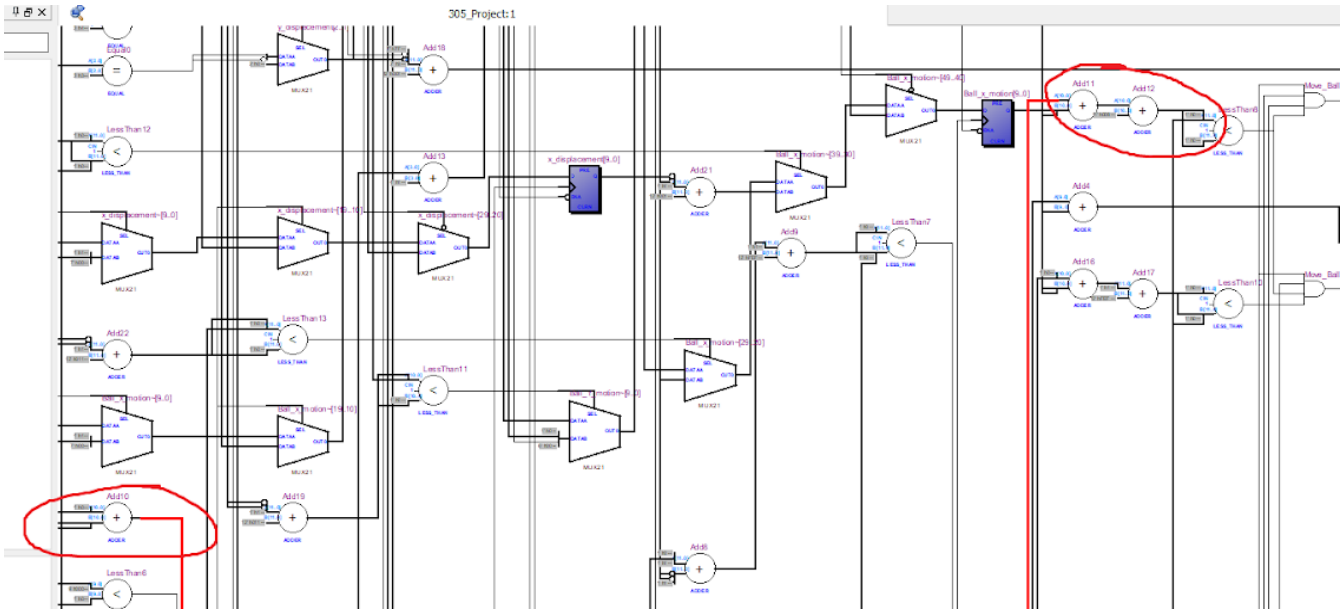
APPENDIX A: FULL RESOURCE UTILISATION



Analysis & Synthesis Resource Utilization by Entity

	Compilation Hierarchy Node	LC Combinationals	LC Registers	Memory Bits
1	|305_Project	1495 (1)	352 (0)	4096
1	|text_gen:inst8|	471 (471)	0 (0)	0
2	|ball:inst2|	194 (194)	32 (32)	0
1	|lpm_mult:Mult0|	0 (0)	0 (0)	0
1	|mult_c4t:auto_generated|	0 (0)	0 (0)	0
2	|lpm_mult:Mult1|	0 (0)	0 (0)	0
1	|mult_c4t:auto_generated|	0 (0)	0 (0)	0
3	|ball:inst36|	176 (176)	31 (31)	0
1	|lpm_mult:Mult0|	0 (0)	0 (0)	0
1	|mult_c4t:auto_generated|	0 (0)	0 (0)	0
2	|lpm_mult:Mult1|	0 (0)	0 (0)	0
1	|mult_c4t:auto_generated|	0 (0)	0 (0)	0
4	|ball:inst34|	174 (174)	32 (32)	0
1	|lpm_mult:Mult0|	0 (0)	0 (0)	0
1	|mult_c4t:auto_generated|	0 (0)	0 (0)	0
2	|lpm_mult:Mult1|	0 (0)	0 (0)	0
1	|mult_c4t:auto_generated|	0 (0)	0 (0)	0
5	|MOUSE:inst1|	107 (107)	119 (119)	0
6	|timer:inst14|	94 (7)	42 (0)	0
1	|CLK1Hz:clkdivider|	47 (47)	26 (26)	0
2	|BCD:Minutes_ones|	11 (11)	4 (4)	0
3	|BCD:Seconds_tens|	11 (11)	4 (4)	0
4	|BCD:Minutes_tens|	10 (10)	4 (4)	0
5	|BCD:Seconds_ones|	8 (8)	4 (4)	0
7	|platform:inst12|	76 (76)	0 (0)	0
8	|VGA_SYNC:inst5|	64 (64)	48 (48)	0
9	|scoreAdder:inst30|	64 (64)	13 (13)	0
10	|fsm:inst11|	28 (28)	8 (8)	0
11	|BCD_to_SevenSeg:inst3|	7 (7)	0 (0)	0
12	|BCD_to_SevenSeg:inst4|	7 (7)	0 (0)	0
13	|LFSR:inst26|	7 (7)	9 (9)	0
14	|LFSR:inst29|	7 (7)	9 (9)	0
15	|LFSR:inst35|	7 (7)	9 (9)	0
16	|char_rom:inst7|	4 (4)	0 (0)	4096
1	|altsyncram:altsyncram_component|	0 (0)	0 (0)	4096
1	|altsyncram_kt91:auto_generated|	0 (0)	0 (0)	4096
17	|switchColour:inst17|	4 (4)	0 (0)	0
18	|game_mux:inst22|	2 (2)	0 (0)	0
19	|levelColour:inst19|	1 (1)	0 (0)	0
20	|Clk_Div:inst|	0 (0)	0 (0)	0
1	|altpll:altpll_component|	0 (0)	0 (0)	0
1	|Clk_Div_altpll:auto_generated|	0 (0)	0 (0)	0

Appendix B: RTL Design Showing Cascading Design in Old Implementation



Appendix C: Block Diagram