Pong Power Player

Imara van Dinten
imvan13@student.sdu.dk
University of Southern Denmark

September 1, 2014

Abstract

This is the report of the Pong Power Player project. For this project an implementation of a Player for a game of Pong is made. This in combination of an FPGA-based vision system on which the game Pong is run.
The goal is to process the incoming video signal and have the FPGA control one of the bats. A video of the system can be found on YouTube [2].

Contents

Chapter 1

Pong Power Player

1.1 Handling

There are two options on this board to play a game of Pong. One is by playing yourself, the other is enabling the implemented A.I. to play for you. This option can be enabled by switching switch 8 on the 8-bit DIP switch.

By switching between switch 3 and 4 on the 4-bit DIP switch on the main-board you enable different video options. Switch 4 will show the filtered output, switch 3 will show the tracking lines.

1.2 Hardware

The board used for this project is based on an Xilinx Spartan6, as can be seen in Figure 1.1. There are multiple board add-ons attached to the main board, this makes it possible to, for instance, attach a screen and keyboard on it. Which are nice add-ons for a game of Pong.

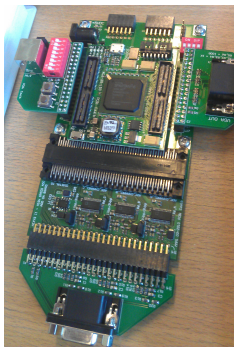

Figure 1.1: Spartan6 Development board

Chapter 2

Pong Power System

2.1 System Design

Before implementing, a system design was made, see Figure 2.1. This makes it easier to write a coherent piece of VHDL and spot any difficulties.

As can be seen in the Figure, various signals will pass through several modules before reaching as output. The module "ObjDet" contains three different detection modules, two for the two different bats and one for the ball. The PPG stands for the Pixel Point Generator.

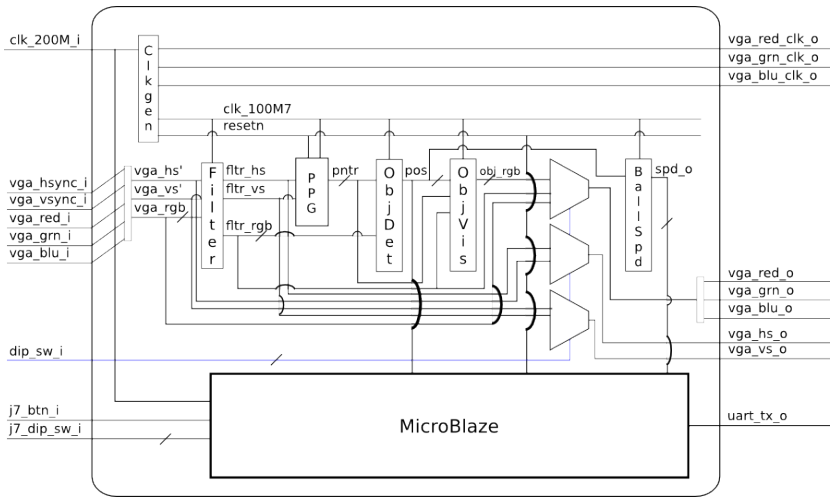

Figure 2.1: System Block Diagram

2.2 System Implementation

2.2.1 ADC Sampling

The processing pipeline starts out by being fed data from a set of Analog to Digital Converters (ADCs). These ADCs sample the colour information as being transmitted by the VGA source. The ADCs are strobed at 100.7MHz, which provides a 4 times oversampling. The regular pixel clock for the industry standard VGA (640x480 @ 60Hz) is 25.175MHz.

To strobe the ADCs, pseudoclocks are generated from the 200MHz system clock. This is needed, because the FPGA does not have external clock pins. Driving regular I/O pins as clock pins will lead to skew and stability issues.

Because the ADCs buffer and average the values coming in, the colour data coming from the ADCs are not synchronous with the VGA synchronization signals. Using a simple shift register, we compensate for 5 clock ticks and make sure they line up again. These signals are then fed into the VGA filter.

2.2.2 Video filter

The filter is designed as a sliding average filter. This will give us an average colour reading over several measurements. To eliminate a lot of the initial noise, the input signals are thresholded first. This gets rid of a lot of the salt noise that can be seen in Figure 2.2.

The averaging is done by a set of 16-bit shift registers, combined with arithmetical logic. The resulting colour signals are thresholded again to get rid of the last specs of noise.

Figure 2.2 gives us a visual representation of the noise we want to remove. Upon initial visual inspection, it is a case of salt noise. The thresholding operations remove a lot of the spurious pixel data, as they are replaced by black.

This processing procedure cleans up the video signal by a lot, leaving only a few small holes in the on-screen objects. The holes do not interfere with the rest of the processing pipeline.

To make sure we still have a valid VGA signalling, the synchronization signals are delayed by the same factor as the length of shift registers.

2.2.3 Pixel Pointer Generator

The Pixel Pointer Generator (PPG), uses the synchronization signals to produce a row- and column-based raster. By doing so, each individual 'pixel' can be analysed by the rest of the pipeline.

The module makes use of a state machine which counts pixel clocks. The counters are synchronized to the visible area of the signal. The values are then put out as a vector of twenty bits, 10 bits are used for the horizontal position and 10 bits for the vertical position. The state machine is drawn out in Figure 2.3.

The state machine keeps track of synchronization signals, going active-low. VGA is a specific standard that dates back to the days of the old CRT monitors.

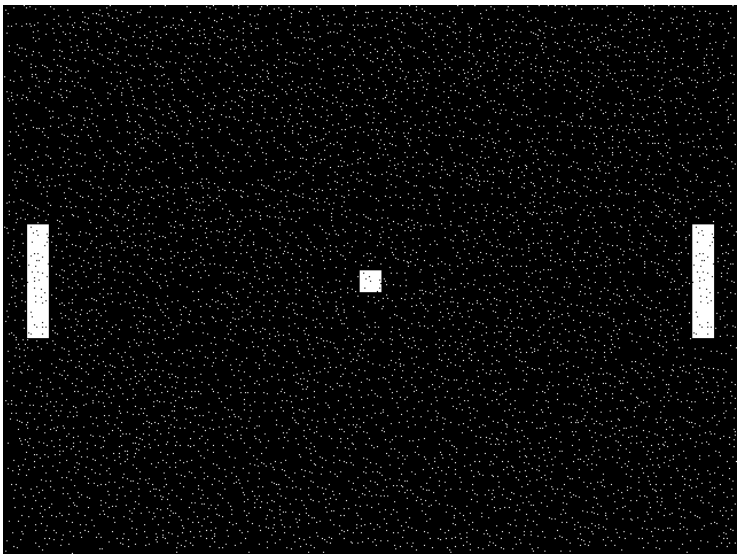

Figure 2.2: Noisy Pong game

The various non-visible timeslots are used for the cathode ray to travel back to the beginning of the next row or frame.

A new frame is signified by the vertical sync signal, which pulses after 525 lines. Likewise, the horizontal sync is used to signify the beginning of a new line. One line is made up out of 800 pixel clock ticks, of which only 640 are visible. The various VGA frame details are shown in Figure 2.4.

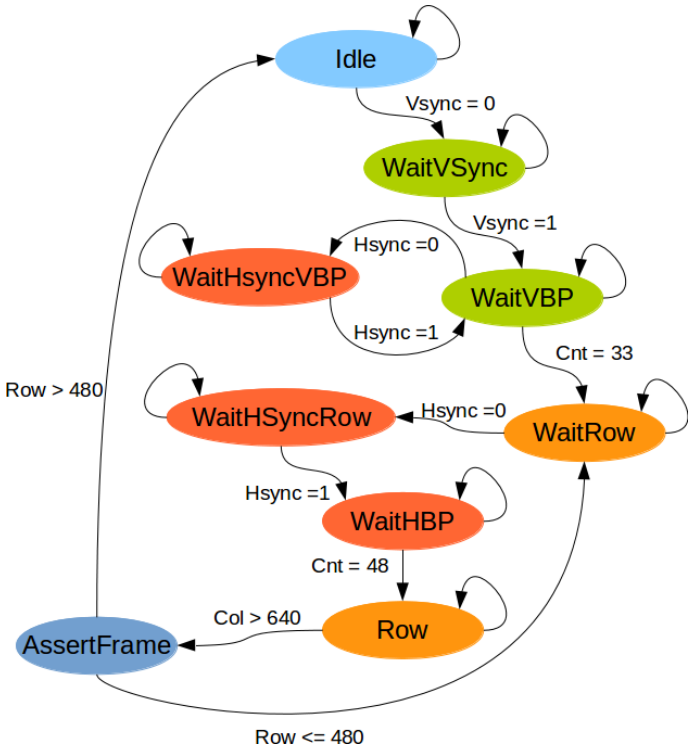

Figure 2.3: Pixel Pointer Statemachine

2.2.4 Object Tracker

The system block diagram (Fig. 2.1) shows a single Object Tracker. In the implementation, each object is given its own tracker.

By going over the screen, the object tracker looks at its X region and width to determine whether a white pixel is part of that object.

By setting an offset and width, the tracker is limited to a vertical band in which it may detect an object. This is shown in Figure **??**.

By use of combinatorial logic, the positions are found and form a 20-bit vector.

2.2.5 Object Visualizer

The Object Visualizer was developed as a means to debug the Object Tracker's output. It intercepts the filtered RGB signals and replaces it with a unique colour for each object, where needed.

It draws lines intersecting at the top-left pixel of the object. Its result can be seen in Figure 2.6.

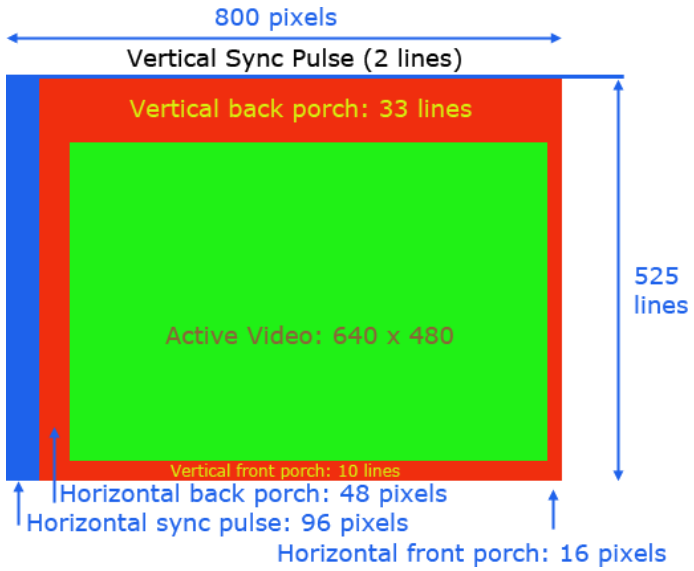

Figure 2.4: VGA Frame visualization [1]

Figure 2.5: Object search regions

2.2.6 Ball Speed Calculator

Ball Speed is a factor that is essential for trajectory calculations. The ball's speed can be gained from the tracker's output, compared to the position in the previous frame.

To get an accurate representation of speed, the Manhattan Distance is used.

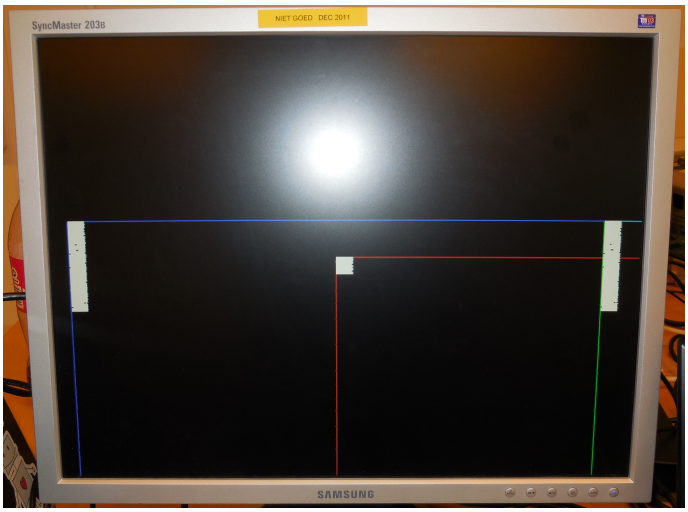

Figure 2.6: Object tracking superimposed on a filtered VGA signal

Since it is based on the sum of absolute displacements in both directions, it will keep increasing as the ball's actual speed increases.

2.2.7 Ball Speed Colourizer

Even though it was not formally implemented, the Ball Speed Colourizer can be designed similarly to the Object Visualizer. The drawing logic would have to define an inner region, defined by the width, height and ball position.

The ball's new colour could be put inside a lookup table to match a speed against the desired colour.

2.2.8 MicroBlaze

This module implements a Xilinx-designed RISC processor. The MicroBlaze is used to process positional data to provide control for one of the bats.

The inputs given to the module are represented to the processor as Memory Mapped I/O registers. This makes it simple and elegant to use when combined with the Xilinx EDK. A description of the control algorithm can be found in section 2.4.

2.3 MicroBlaze EDK

By making use of the EDK, a piece of software is written. This software is designed to handle control for the left bat.

Based on a single register, the softcore decides whether the software should control the bat or not. If this is not the case, the player may use the physical tactile buttons on the board to control the bat.

2.4 Softcore Bat Control

2.4.1 A.I.

Many different algorithms exist to implement a Smart Pong Player. Initially, we planned out an algorithm that would predict where the ball was supposed to end up by making use of the ball's position and speed. An example of such trajectory prediction is visually described in Figure 2.7.

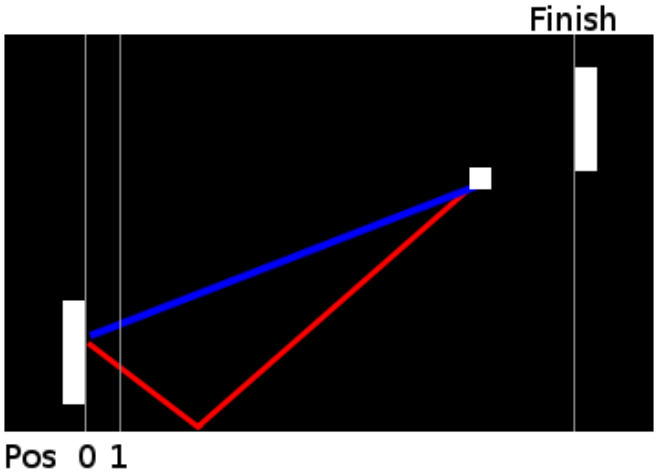

Figure 2.7: A.I.

2.4.2 Simple Control

A simple manner of controlling the bat is to change the bat's Y position as the ball's Y position changes. This can be easily written and easily executed as the bat only has to mimic the ball.

For the first few hits, the bat will be successful. As time goes on, however, the ball will outspeed the bat and it will lose time and time again.

Chapter 3

FPGA Design

3.1 ChipScope Utilization

A ChipScope was not implemented in our system. Looking back on the whole design experience, implementing one would have saved a lot of time.

Doing an analysis using ChipScope would have showed that one of the colour channels wasn't returning any data from the VGA link.

3.2 Floorplanning and I/O Planning

When doing a digital design on an FPGA, effective floorplanning makes a world of difference. Even though the design is the same, a different floorplan can mean making your timing constraints.

One general rule of thumb is to keep interconnects and signal paths as short as possible. In doing so, the total signal propagation delay is reduced by a significant amount.

Another good rule is to try and design using native FPGA blocks. This can trim a lot of excess logic and speed up designs, even on a device with a low speed grade. The use of native blocks will also reduce the number of slices and LUTs used, so it is a good way to fit more into an FPGA.

Our floorplan was generated by PlanAhead by using the Default ISE Strategy. The long signal paths in Figure 3.1 show that this is not an optimal strategy. However, the timing analyses show that no serious timing violations are made.

3.3 Design Hierarchy

A design's hierarchy shows how various modules are layered. A module's width shows it's relative usage compared to the total usage of the Top Level block.

PlanAhead drew us the hierarchy found in Figure 3.2. ISE did not generate netlist files for all of the modules that were designed. As such, they do not show

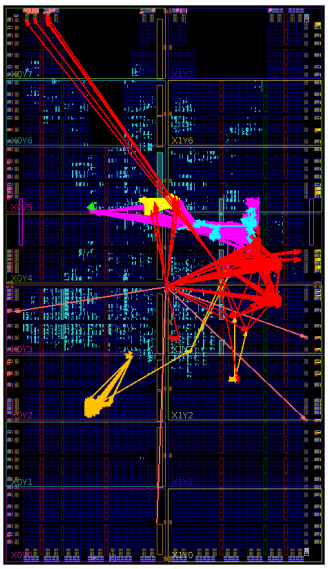

Figure 3.1: Power Player floorplanning in PlanAhead

up in the hierarchy and seem to have been made into primitives and nets as a part of the TopLevel module.

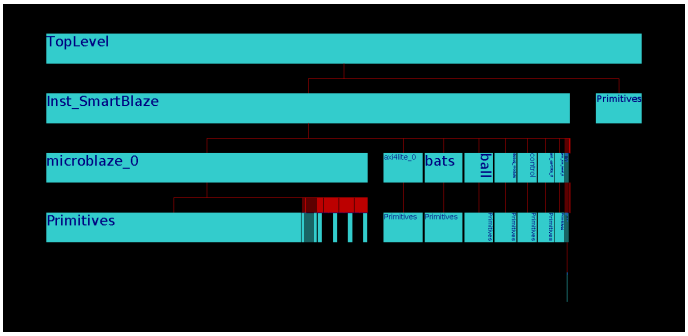

Figure 3.2: FPGA Design Hierarchy

3.4 Resource Utilization

Because PlanAhead could not recognize all our system modules, the resource estimates for the ROOT block have been used. These estimates can be found in Table 3.1.

The design is not big, by any standards as the Spartan6 still has plenty of room to fit more logic. Given a few optimization passes, the current logic could

Site Type	**Available**	**Required**	**% Utilization**
LUT	27288	3007	12
FD_LD	54576	2719	5
SLICEL	1809	200	12
SLICEM	1602	177	12
SLICEX	3411	376	12
BSCAN	4	1	25
BUFGMUX	16	7	44
DCM	8	1	13
DSP48A1	58	3	6
PLL_ADV	4	1	25
RAMB16BWER	116	8	7

Table 3.1: Physical Resource Estimates

also be shrunk down and optimized for timing.

Chapter 4

Wrapping Up

4.1 Conclusion

In the end the filter works. There were some issues with one of the ADCs, which contributed to a colour channel not working properly. The A.I. implemented is quite rudimentary, but works reasonably well up until a certain speed.

4.2 Acknowledgements

I would like to express my thanks to Xander Bos, as my partner for this project. Also special thanks to Patrick Stolc for discussing various ideas.

Bibliography

[1] Nick Gammon. *VGA Frame visualization*. 2012. URL: http://www.gammon.com.au/images/Arduino/VGA_Output_6.png.

[2] Imara van Dinten Xander Bos. *EMB3 Pong Power Player Spring 2014*. 2014. URL: http://youtu.be/cGOVO1OtH0c.