FPGA: Fast Paced Gameplay Automator

6.111 Final Project

Jaeyoung Jung and Silvia Knappe

December 10th, 2020

Contents

1 Introduction

osu! is a popular ranked rhythm game in which the player must move their cursor to click on circles to the beat of the music. Players can choose from a selection of beatmaps, or community made game levels and songs to play from. Players also get 'performance points' after playing beatmaps that contribute to their overall osu! ranking, that are based on accuracy and ability to keep up a combo in game. There are 3 different game elements that the player must interact with in beatmaps: hit circles, sliders, and spinners. For hit circles, the player must only click on them at the right time. For sliders, the player must click and drag along the slider path. Lastly, for spinners, the player needs to click and spin their mouse in a circle, but these aren't seen nearly as often as hit circles or sliders. In order to assist players in clicking at the correct time, an outlined circle called the approach circle decreases in radius around hit circles, and when the approach circle becomes the same size as the hit circle, the player should click on the hit circle. A labeled screenshot of osu! gameplay is shown in Figure 1. As fun as osu! can be to play, gameplay gets very difficult with more advanced beatmaps. It's difficult to the point that more advanced players play with a tablet and keyboard instead of just mouse controls. However, it would be even easier to be good at osu! if we just had an FPGA that could play the game for us!

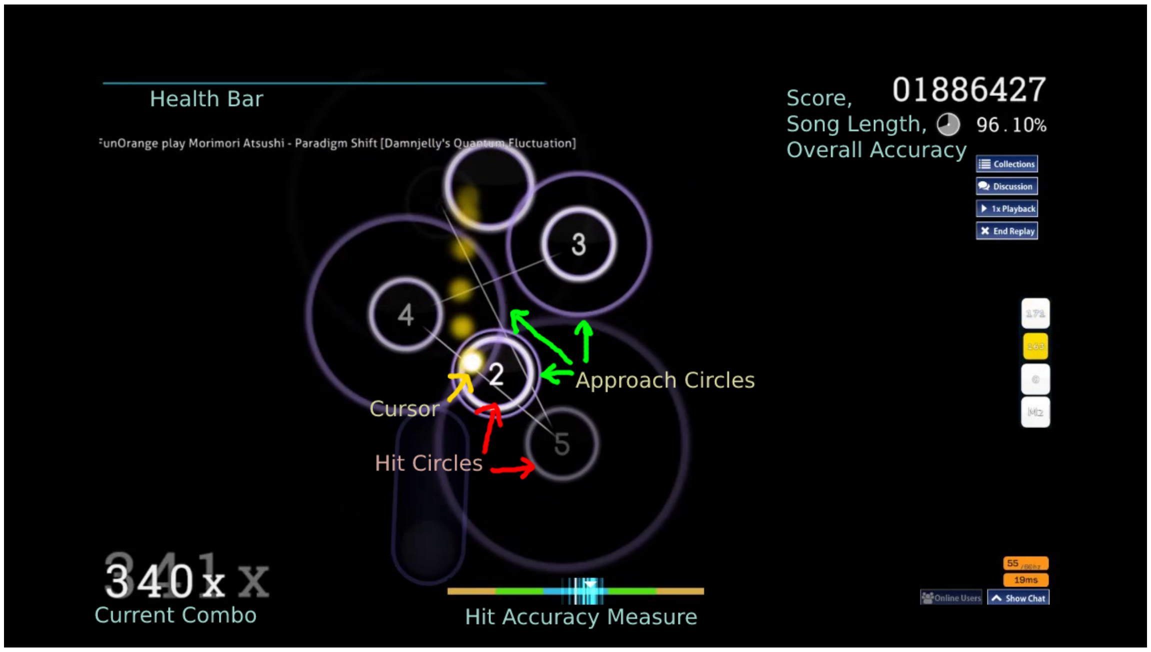

Figure 1: Labeled screenshot of osu! gameplay

This motivation to get a high ranking in osu! inspired us to create our project: Fast Paced Gameplay Automator, or FPGA for short. We wanted to have gameplay data inputted into our system, and have our system move the computer mouse to the correct position on the screen and click so that we can get a high score.

2 Background

We chose to focus only on detecting and clicking on hit circles for simplicity. In order to be able to click on the hit circles at the right time, we needed to be able to detect when the hit circles should be clicked on- which is the moment when the approach circle's radius is the same as the hit circle's radius. Since we wanted to detect only approach circles, and not hit circles, we made the approach circles a different color from the hit circles, since player can make their own osu! skins to customize the way that osu! looks. To detect the approach circles of the correct size, we decided to use the Circle Hough Transform, or CHT, which detects whether a circle of the specified radius is in an image.

2.1 Circle Hough Transform

As mentioned above, the CHT detects whether or not a circle of a given radius r is in a given image. This is done with the following steps:

1. Initialize an accumulator matrix
2. For each nonzero pixel in the image, add 1 to the accumulator in indices r away from the nonzero pixel (essentially drawing a circle around each nonzero pixel in the accumulator, incrementing existing values)
3. The maximum value in the accumulator is the center of the desired circle.

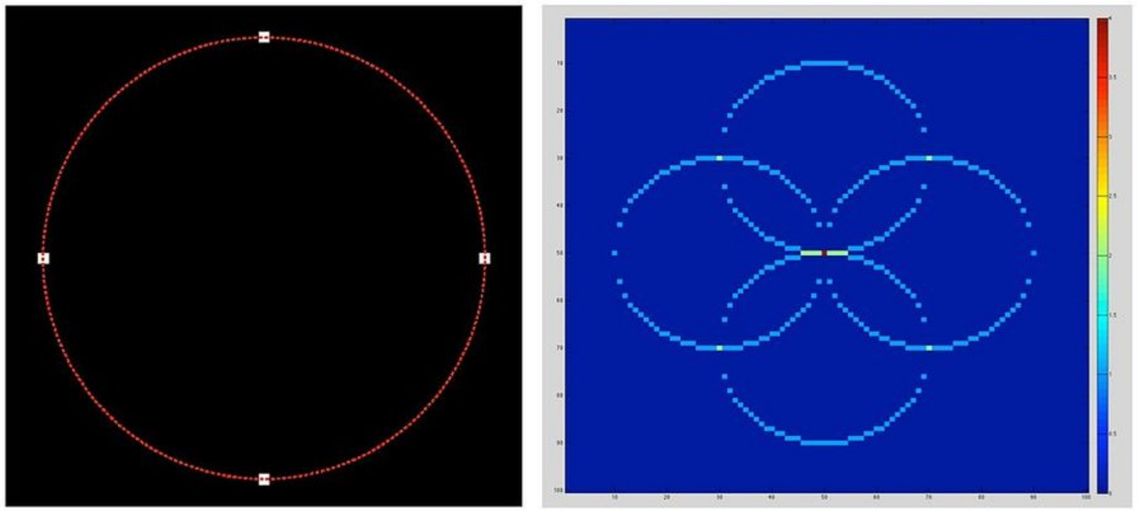

Figure 2: An example of the points used to detect a circle and the accumulator matrix [1]

An example of the points used in calculating the CHT are shown in Figure 2. Here we can see 4 points that form a circle of a certain radius on the left, and on the right, we can see a visualization of the accumulator matrix with lighter pixels where the value in the matrix is greater, and the lighter values being in the center where the 4 circles start overlapping more.

2.2 Fourier Transform

What essentially happens during the CHT is a convolution with the image and a kernel. The kernel has the circle of desired radius r that we are looking for. When the image and the kernel are convolved, the resulting image has the kernel replicated over every nonzero pixel in the input image. This is why kernels such as the Sobel operators give rise to interesting effects such as isolating edges in images. In this case though, we want to draw circles on all the nonzero pixels in our image, and find the maximum result. Since we now know that we can convolve an image and a kernel to obtain our desired result, we can use the knowledge that convolution in the spatial domain is multiplication in the frequency or Fourier domain. This is extraordinarily useful for us, since convolution in the spacial domain is extremely inefficient, as it requires many operations. If our image and the kernel have n pixels, the n it would take $O(n^2)$ time to perform the convolution in the spacial domain. We can reduce that to $O(n \log n)$ if we instead take the Fourier transform of the image and the kernel, multiplied them together, and took the inverse Fourier transform of the multiplied result.

Using the Fourier transform is also useful for us in this case because Vivado has IP for the Fast Fourier Transform (FFT). We were able to use 2 FFT IP modules in order to make the FPGA calculate the 2D FFT of our input image.

2.3 Functional Model

Before we went ahead to build the 2D FFT on the FPGA and using Verilog, we first implemented a simple Python replica of what we would be doing on the FPGA. In this replica, we used some old 6.003 Signal Processing code (specifically from this lab, in which we had to use a kernel to find the desired pattern within an image) as a framework to do the desired image processing. This code used the numpy FFT/inverse FFT functions to calculate the FFT of our desired image (in this case a screenshot from osu!) and a kernel (a black and white array with a circle drawn in it). Using the functional model, we were able to predict whether our desired approach using the CHT with the FFT would be feasible. We were able to simulate the results to be more like what the FFT on the FPGA would output by truncatng all the values to integers after each FFT computation, and by padding the image and kernel to dimensions that are powers of 2. Even when we did this, we were still able to find a peak in the image that was at the center of the approach circle.

Another useful aspect of the functional model is that we used that code to generate the kernel that used on the FPGA. Since we wanted to preserve as much precision as possible, we took the FFT of our ideal kernel (a circle of the desired radius as a numpy array of 1s and 0s) using numpy, and then stored that FFT as a `coe` file that already had the FFT precalculated.

2.4 Teensy Output

Since we wanted to send a click to the computer that is running the osu! game client, we needed a way for the FPGA to communicate with the computer to send the click. We decided to use a Teensy microcontroller to do this, as we had prior experience with Arduino code and the ability to simulate mouse inputs that way. We simulated touchscreen input into the computer using the Teensy, since touchscreen input allows us to use absolute coordinates, as compared to relative coordinates/mouse velocity, which is useful for us since we want to detect the exact coordinates of the center of the circle.

3 Methods and Modules

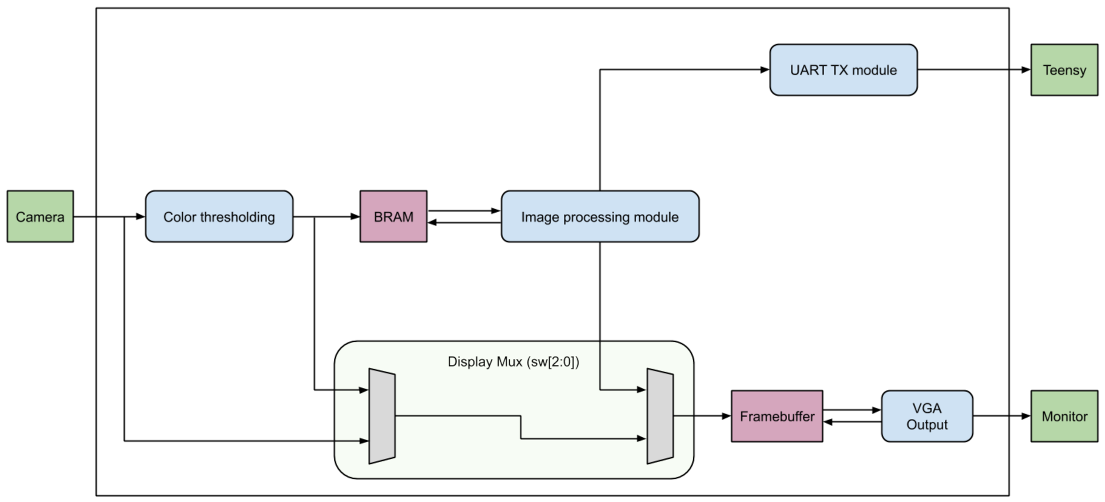

Figure 3: System Block Diagram

In this section, we will go over all the modules we made in order to get our system to work. The flow of the system can be seen in Figure 3. Here, we can see that the input to the FPGA comes from the camera data, which is passed to the color thresholding module to preprocess the image before storing it in the BRAM that the image processing module accesses to perform the CHT. The image processing module outputs a coordinate and click data which the UART TX module takes in and passes to the Teensy to send click information to the computer. The camera input and image data are also piped to a mux that decides what gets inputted to the VGA module. Here we can decide whether we want to see the raw camera data, the preprocessed camera data, or the convolved, inverse FFT-ed image.

3.1 Camera Module and Color Filtering

Author: Silvia Knappe

Since the starter camera code was already integrated into the `top_level` code, we didn't make a specific separate camera module, but rather edited the starter code to suit our needs, keeping the camera board connected to the `JA` and `JB` ports on the FPGA.

In the Background section, we mentioned that we were only looking detect approach circles. We made only the approach circles red, so that we could use the camera code to filter out the red part of the camera image. There was some color preprocessing already built in to the camera code, which separated color channels for us when we flipped switches, but we changed this in the following ways:

First, the filtered output image was kept as a 1-bit image to limit the size of the FFT outputs computed later on. So the pixels stored in the internal BRAM for image processing tell us either the pixel met the color threshold or it didn't. To filter out pixels of the wrong color, we made it so that the red bits of a pixel must be above a certain threshold, and the blue and green bits must be below a certain threshold. Our threshold values were set by pointing the camera at the computer screen showing the osu! screenshot and displaying the filtered pixels out from the FPGA on an external monitor. Using this method, we tuned the threshold values so that when pointed at a screen with the osu! screenshots, only the red approach circles would show up.

Second, a copy of the camera pixels were always filtered and stored in the internal BRAM. Another copy of the pixels were selectively filtered based on turning `sw[1]` on or off and sent to the frame buffer to be displayed on the external monitor.

3.2 Image Processing Module

Authors: Silvia Knappe & Jaeyoung Jung

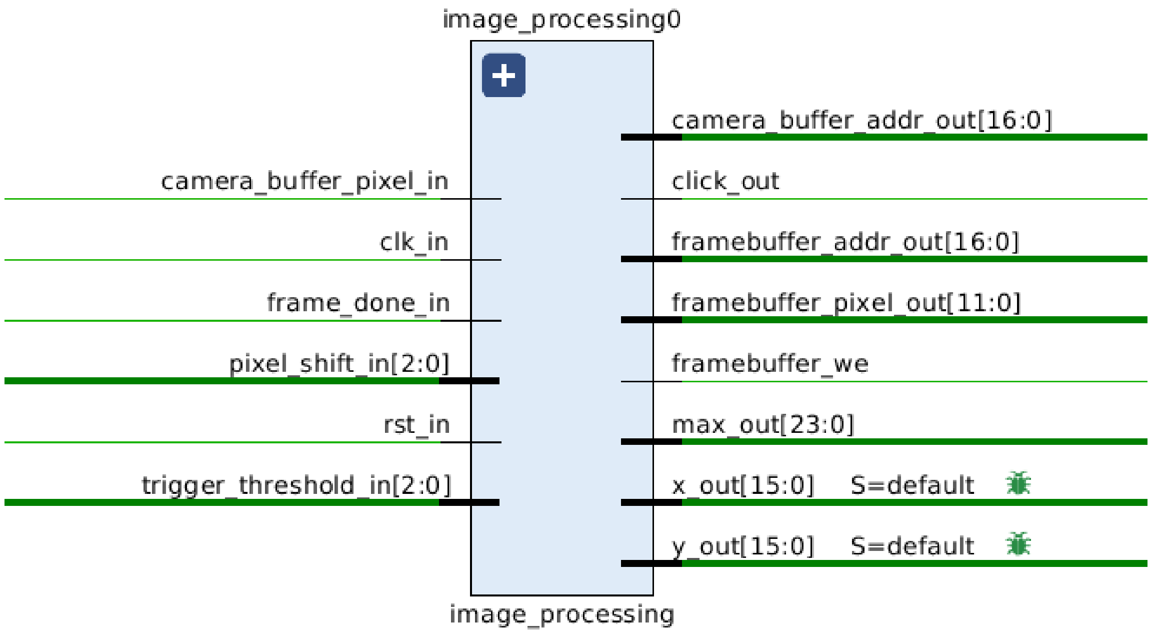

Figure 4: Image Processing Module

The image processing module reads the 1-bit image generated by the color filter, detects any circles of the right radius in the image, and if a circle of the right radius is detected, it asserts a signal (`click_out`) that tells the UART transmit module to send the coordinates of the center of the circle over UART.

This module contains a block RAM, which is used to store the matrix between each step of the image processing. All submodules in this module read from this RAM, and the submodules that produce results write them to this RAM. Because multiple submodules need to be able to read and write from this RAM, the memory address, write data, and write enable signal from all of the modules are multiplexed, and depending on which state the image processing module is in, the signals from the submodule that is currently working are passed to the RAM.

This module is structured as a state machine that performs the following tasks in this order:

1. `IDLE`: Wait for `frame_done_in` to be asserted.
2. `PIXEL_INPUT`: Copy the thresholded 1-bit image into the internal BRAM.
3. `FFT`: Perform the 2D Fourier transform of the image.
4. `MULTIPLY`: Perform element-wise multiplication of the resulting Fourier transform and the Fourier transform of the kernel, which is stored in a ROM.
5. `IFFT`: Perform the inverse 2D Fourier transform.
6. `PEAKFIND`: Find the peak of the convolved image.
7. `SEND`: Send the peak of the convolved image to the UART transmit module.
8. `PIXEL_OUTPUT`: Copy the convolved image into the output framebuffer.

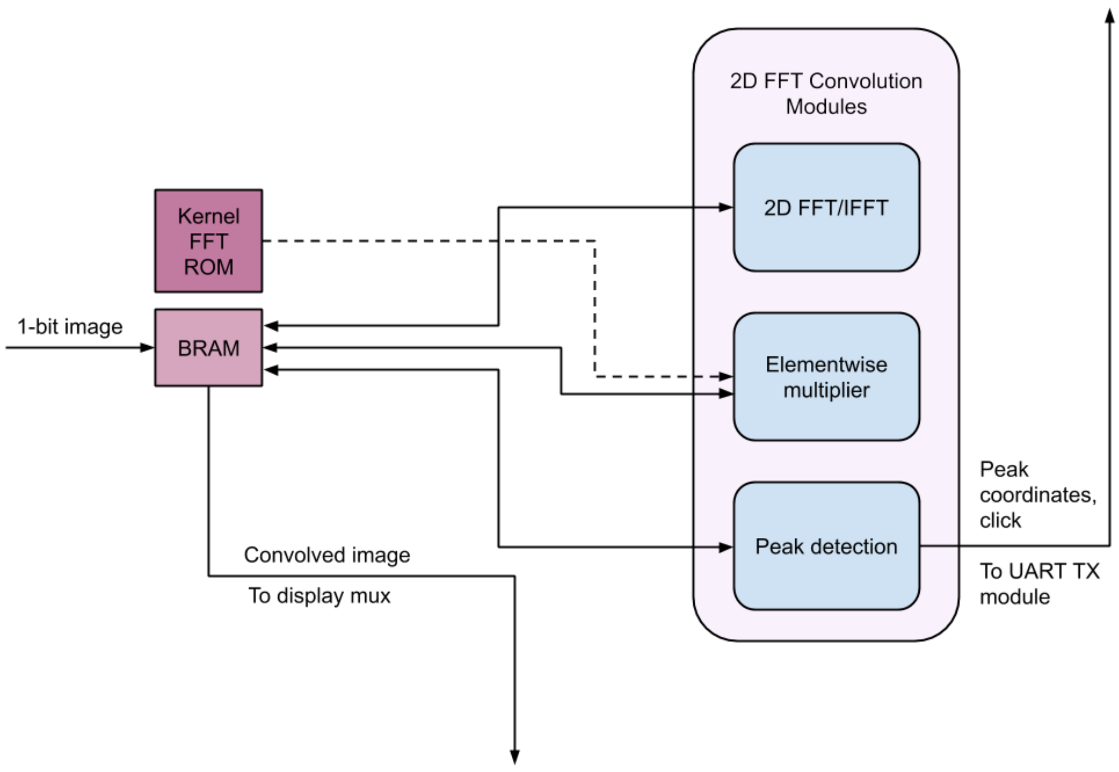

Figure 5: Block Diagram of the Image Processing Modules

3.2.1 Internal BRAM

The internal BRAM should be large enough to store the image after all image processing has been applied. The width should be large enough so that the largest possible value will not overflow, and the length should be large enough so that it can store all values of the Fourier transform. Because the Xilinx FFT IP only supports sample counts that are powers of 2, dimensions that are not powers of 2 need to be rounded up to the nearest power of 2. If the original resolution of 320x240 was used, the maximum possible value of the FFT would be 76800, assuming a matrix entirely made up of 1s. To store 76800, the width must be at least 36, since the FFT IP uses two's complement to represent numbers and uses as many bits to store the imaginary component as the real component. The length must be at least 131072, as the dimensions of the matrix must be 512x256, since dimensions have to be rounded up to the nearest power of 2. This would require a RAM that can store 4.7 megabits of data, which would use almost all of the block memory in the FPGA. Therefore, we decided to reduce the resolution of the image from 320x240 to 160x120.

For a resolution of 160x120, a BRAM that could store 256x128 values is required, so the length of the BRAM was set to 32768. The width was set to 50 so that values up to 16777215 could be stored, in order to make sure that overflow would not occur.

3.2.2 Pixel Input

In the pixel input state, the image processing module reads the contents of the BRAM containing the thresholded 1-bit image and copies it into the internal BRAM. In order to reduce the resolution of the image from 320x240 to 160x120, every other pixel in every other row was sampled and copied into the internal BRAM.

3.2.3 2D FFT/Inverse FFT Module

Author: Jaeyoung Jung

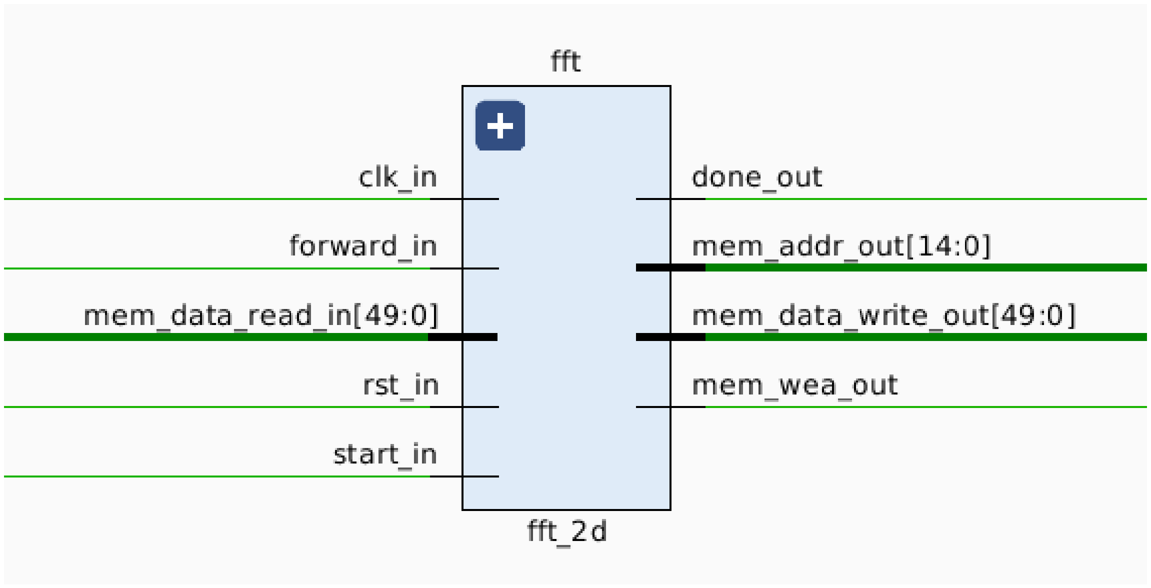

Figure 6: 2D FFT/Inverse FFT Module

The 2D FFT module performs 2D Fourier transform of the matrix in the BRAM in the image processing module. This module has two submodules: `fft_wrapper_x` and `fft_wrapper_y`. `fft_wrapper_x` performs the 1D Fourier transform of each row in the BRAM, replacing the contents of the row in the BRAM with the result of the FT. Similarly, `fft_wrapper_y` performs the 1D FFT of each column and replaces the column with the FT.

This module is structured as a state machine with three states:

1. `IDLE`: Wait for the `start_in` signal to be asserted, then transition to `FFT_X`.
2. `FFT_X`: Perform Fourier transforms in the x-direction. When done, transition to `FFT_Y`.
3. `FFT_Y`: Perform Fourier transforms in the y-direction. When done, transition to `IDLE`.

After the Fourier transforms of both the rows and columns are completed, the matrix stored in the BRAM is the 2D Fourier transform of the matrix that was stored before.

3.2.4 1D FFT/Inverse FFT Wrapper Module

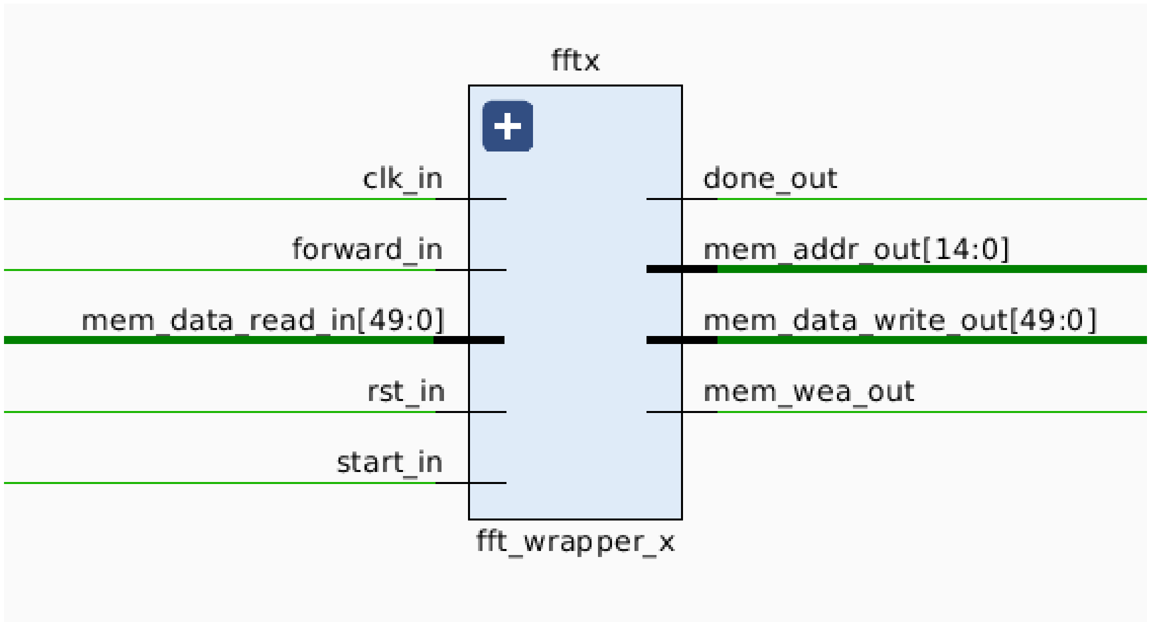

Figure 7: 1D FFT/Inverse FFT Wrapper Module

`fft_wrapper_x` and `fft_wrapper_y` are very similar modules that use the Xilinx FFT IP compute the 1D Fourier transform of each row and column respectively, and replace the original contents with the Fourier transform. These modules are structured as a state machine, whose state diagram is shown in figure 8.

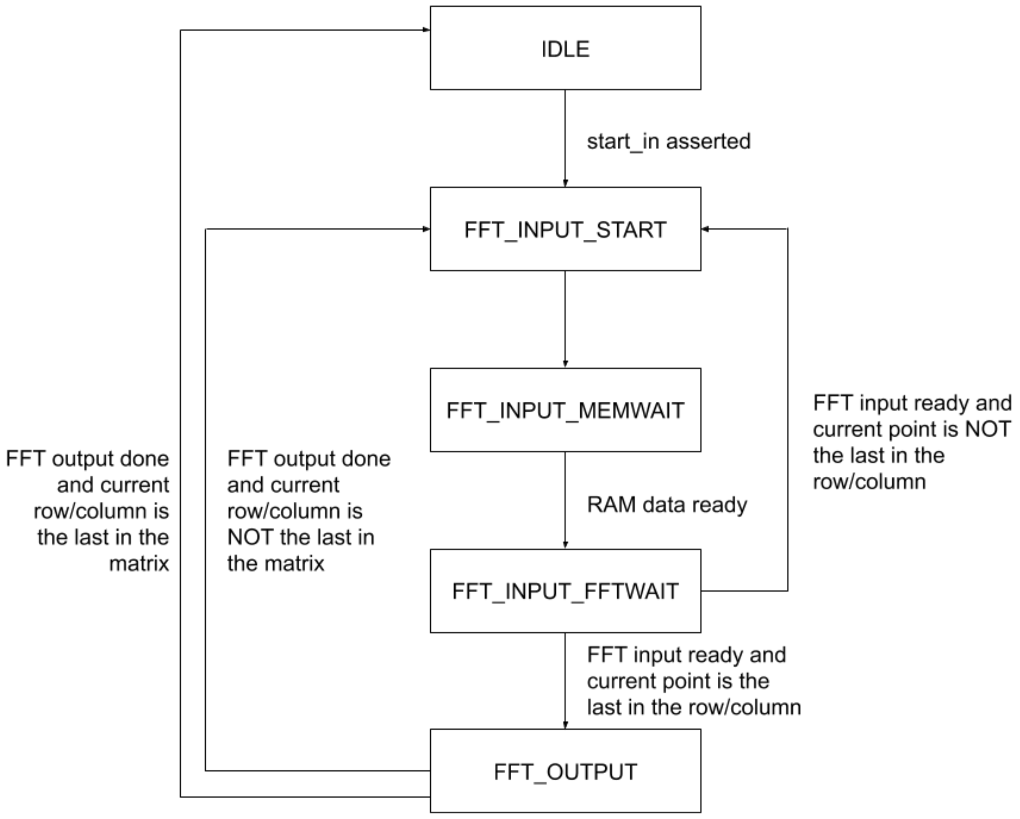
IDLE
start_in asserted
FFT_INPUT_START
FFT_INPUT_MEMWAIT
RAM data ready
FFT_INPUT_FFTWAIT
FFT input ready and current point is NOT the last in the row/column
FFT input ready and current point is the last in the row/column
FFT_OUTPUT
FFT output done and current row/column is the last in the matrix
FFT output done and current row/column is NOT the last in the matrix

Figure 8: 1D FFT/Inverse FFT Wrapper State Diagram

These modules can also calculate inverse Fourier transforms by replacing the inputs and outputs of the FFT IP with their complex conjugates. This result is not divided by the length of the transform like in a typical inverse Fourier transform because we want to avoid divisions, as the output of the FFT IP is truncated to be an integer.

3.2.5 Elementwise Multiplication Module

Author: Silvia Knappe

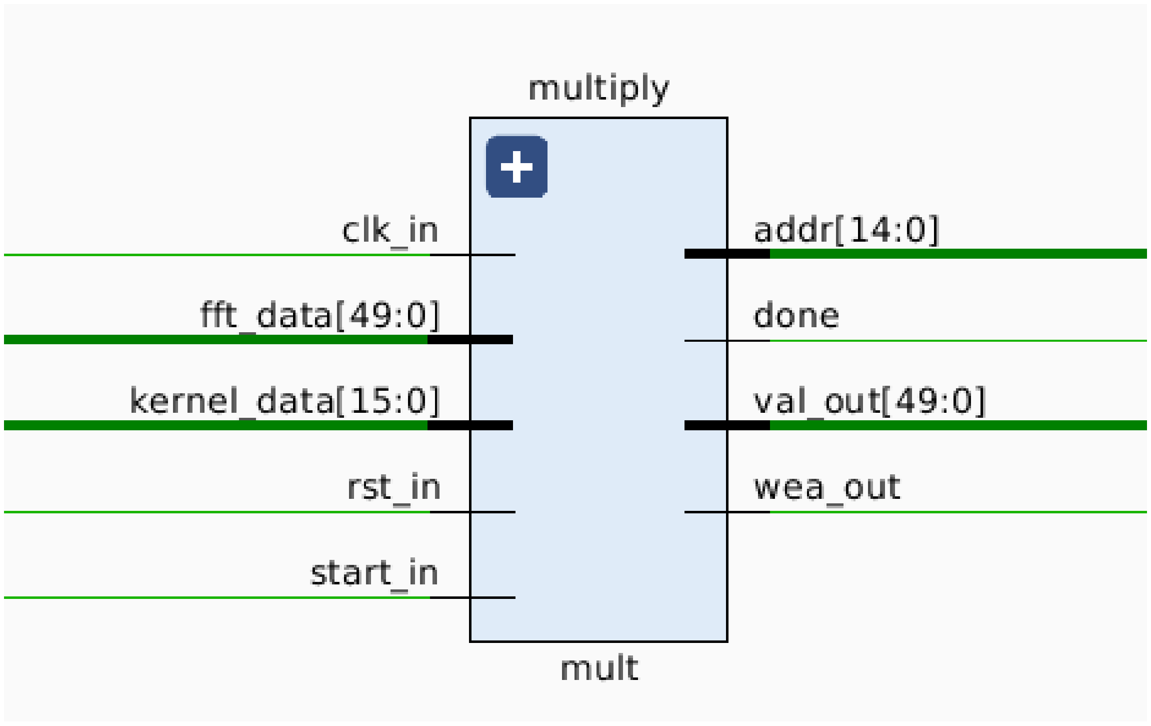

Figure 9: Multiplication Module

After performing the FFT on the input image, the image FFT and the kernel in the ROM need to be multiplied together to produce the convolved FFT. The multiplication module goes through every address in the 256×128 internal BRAM and multiplies the value at that address with the corresponding value in the kernel ROM. This module has a state machine with 4 states:

1. `IDLE`: The module is waiting for the `start_in` signal to come in to begin the multiplication process, with the starting address beginning at 0.

2. `READ_WAIT`: The module is waiting for the image and kernel data to be read in from the current address, since the block memory has a read delay.

3. `MULT_WRITE`: The module performs complex multiplication (which is $(a + bj)(c + dj)$) using the FFT image and FFT kernel values, and writes that result out in the internal BRAM to the same address as the address it read from. The `done` signal is asserted once the address has reached 256×128-1, and the state gets reset to `IDLE`

4. `ADDR_INC`: In this state, the address for the next read gets incremented. This step needed to be separate since we ended up only using a 1-port BRAM, otherwise we'd write to the wrong

address. Using a 2-port BRAM, this problem would be eliminated, but we ran short on time to implement that.

3.2.6 Peak Detection Module

Author: Silvia Knappe

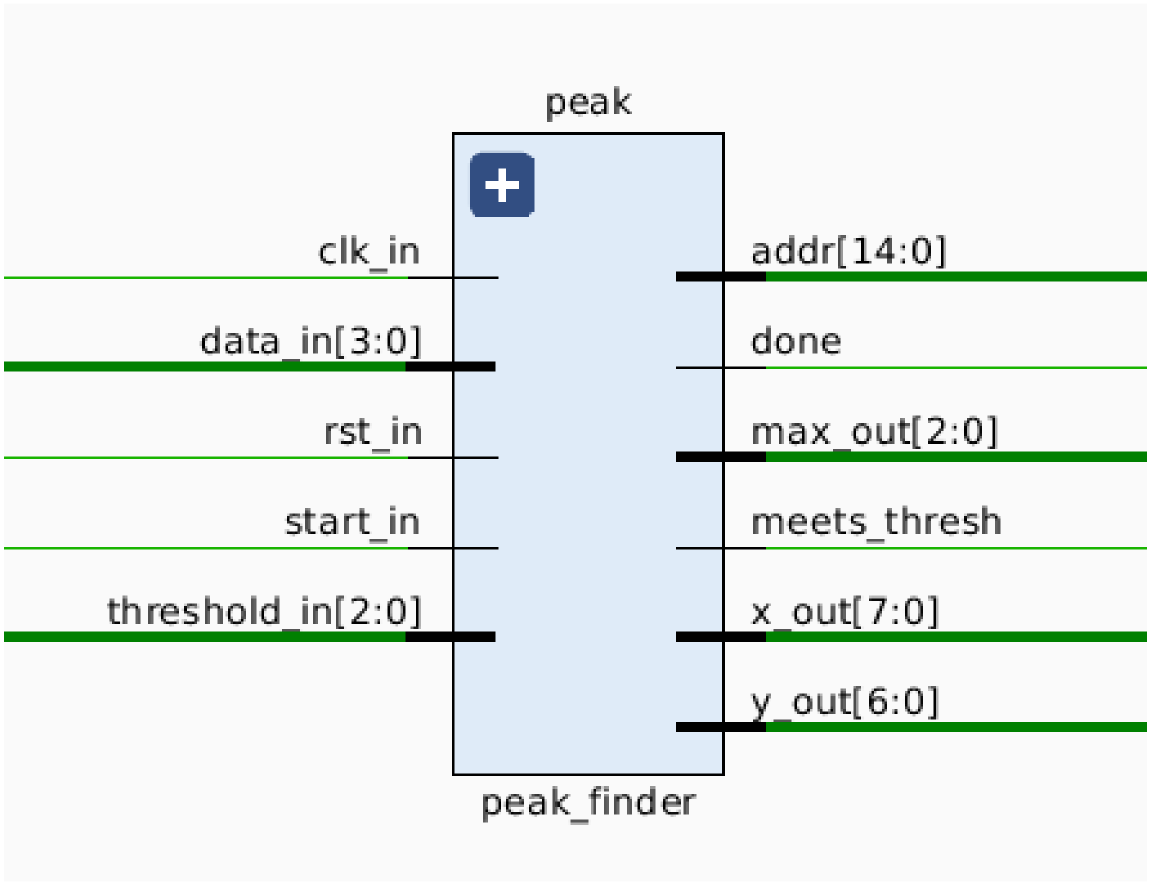

Figure 10: Peak Finder Module

After multiplying the image FFT with the kernel FFT, the next step is to detect the maximum value in the multiplied result. This module goes through every address in the internal BRAM and keeps track of the maximum value so far, as well as its location in the BRAM. To conserve clock cycles, this module only iterates over the 160×120 addresses corresponding with the image stored in the BRAM. This module is also implemented using a state machine with the following states:

1. `IDLE`: In this state, the module is waiting for `start_in` to be asserted so that it can start finding the maximum value of the BRAM.

2. `READ_WAIT`: In this state, the module is waiting for the data to be read in from the BRAM at the current address.

3. `COMPARE`: The module compares the value at the current address from the BRAM and sets it as the maximum value if it is greater than the previous maximum value. The address of the maximum value is also stored. The address the module uses to read from is also incremented in this state.

The address corresponding to the maximum value that this module stores is stored as an x-component and a y-component for easier translation into a click position on the screen.

3.2.7 Pixel Output

The last state in the image processing module copies the contents of the internal BRAM into the VGA framebuffer. In order to do this, the module iterates through every address in the framebuffer and writes `12'hFFF` if the corresponding point in the internal memory is above a threshold and `12'h0` otherwise. Because the internal BRAM stores a 160x120 image and the framebuffer stores a 320x240 image, one point in the internal BRAM corresponds to four points in the framebuffer. In order to correctly map the pixels, the least significant bits of the x and y coordinates used to access the framebuffer are dropped when accessing the internal BRAM.

3.2.8 Latency

For the latency, all modules used a clock with speed of 100MHz.

Calculation	Latency per operation	Number of operations	Total latency
Pixel input	1 cycle/pixel	19200 pixels	19200 cycles
FFT (rows/x-direction)	892 cycles/row + 5 cycles/pixel	128 rows + 32768 pixels	278016 cycles
FFT (columns/y-direction)	502 cycles/column + 5 cycles/pixel	256 columns + 32768 pixels	292352 cycles
Multiplication	4 cycles/pixel	32768 pixels	131072 cycles
IFFT (rows/x-direction)	892 cycles/row + 5 cycles/pixel	128 rows + 32768 pixels	278016 cycles
IFFT (columns/y-direction)	502 cycles/column + 5 cycles/pixel	256 columns + 32768 pixels	292352 cycles
Peak detection	3 cycles/pixel	19200 pixels	57600 cycles
Pixel output	1 cycle/pixel	76800 pixels	76800 cycles
Total latency	-	-	**1425408 cycles**
Total latency (ms)			**14ms**

3.3 UART Transmit Module

Author: Jaeyoung Jung

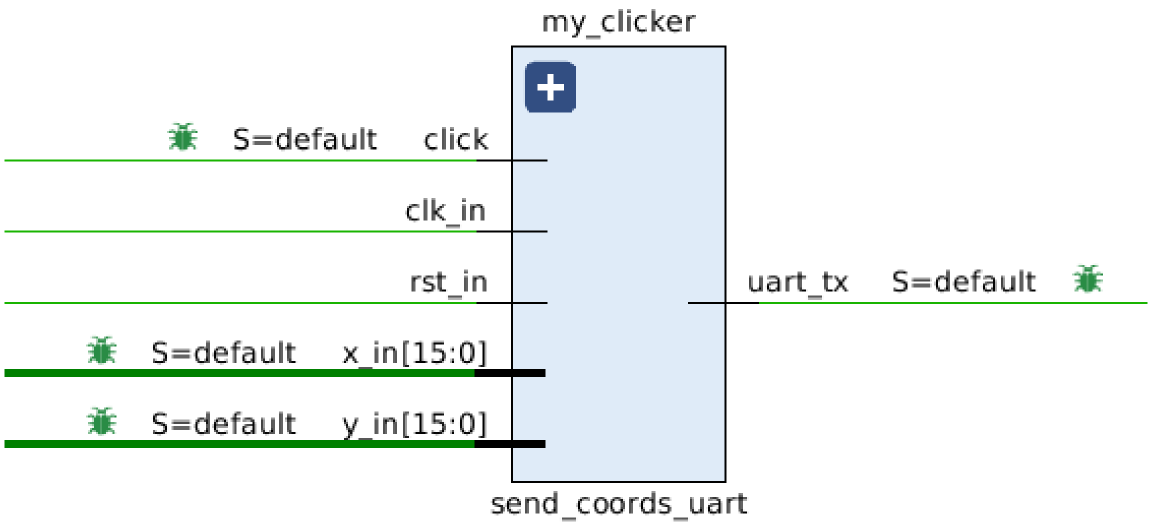

Figure 11: UART Transmit Module

The UART transmit module is a simple module that, when the "click" signal is asserted, sends the contents of the two 16-bit inputs `x_in` and `y_in` as a series of 4 UART packets. This module has two states:

1. `IDLE`: Wait for the `click` signal, then transition to `SENDING`.
2. `SENDING`: Send the coordinates over UART. When done, transition to `IDLE`.

3.4 VGA Output Module/Display Mux

Authors: Silvia Knappe & Jaeyoung Jung

We used the VGA output capabilities of the initial camera code and modified it so that we could display what we wanted to on an external monitor. We controlled what pixels got sent to the external monitor using `sw[2:0]`, using the `sw[0]` to enlarge the camera image, `sw[1]` to toggle whether the image or color filtered image is displayed, and `sw[2]` to toggle whether the resulting convolution from the image processing module is displayed. These switches controlled what information got sent to the frame buffer, which is what the VGA output was looking at to display.

4 Discussion and Conclusion

Overall, we chose a rather ambitious project to implement, and are pleased that we met our goal to have the FPGA system be able to click in the center of the hit circles when the approach circle around them was correctly detected by the image processing done on the FPGA.

The 2D FFT module was the most complex module of this project, and Jaeyoung spent much of the time working on this project on implementing the 2D FFT. This module was difficult to implement because it used the Xilinx FFT IP, which was not intuitive to interface with, as someone who was not very familiar with the AXI-Stream protocol. However, the structure of the module made it relatively simple to fully integrate the module once the submodules, which calculate 1D FFTs of the rows and columns, were implemented.

Some of the limitations of our project come from artifacts that arose in the FFT module. Some of these artifacts are just extra noise that come from the kernel and image being padded so that the dimensions are powers of 2. Other artifacts come from the image being circularly convolved, due to the FFT applied being discrete. Therefore, when the kernel and image are multiplied together, some of the results wrap around the edges of the resulting image. These artifacts did not affect our results very much, as those artifacts were always of a smaller magnitude than the peak in the image where the kernel and image matched. However, we did have some artifacts that did interfere with our system clicking in the correct place. We saw that around the point where the image and kernel matched, we also got a circle around it. This artifact most likely was caused by rounding/truncation to integers when performing the FFT.

An example of the image processing steps can be found in Figures 12-14. In Figure 14, some of the circular convolution artifacts can be seen to the left of the circle, along with the circle artifact around the point where the image and the kernel match. The point where the image and kernel match can also be seen very clearly.

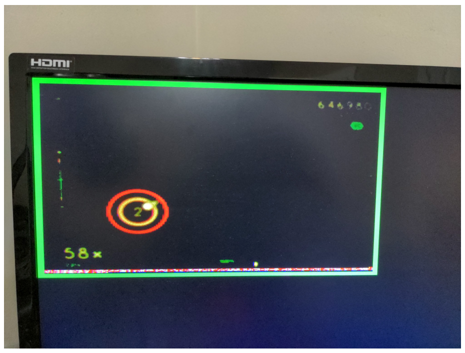

Figure 12: Image frame received from camera

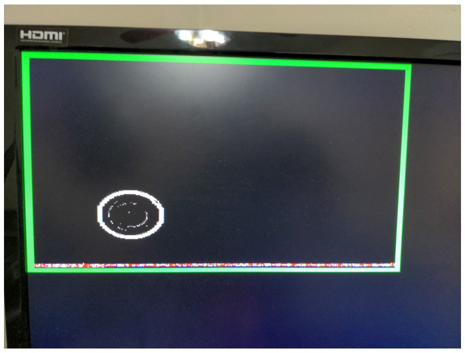

Figure 13: Image after thresholding

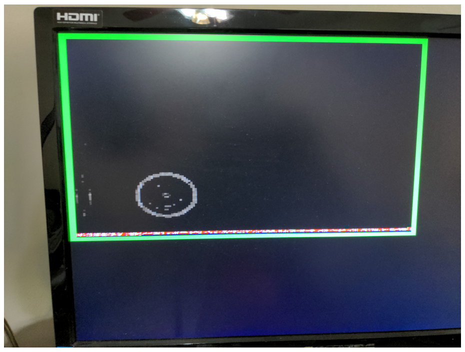

Figure 14: Image after going through image processing module

Some other features we implemented were an artificial delay before clicking, as well as a click cooldown. The reasoning for the artificial delay was that the approach circle was easiest to detect when it still hadn't converged with the hit circle. This can be seen in Figure 15 with the approach circle on the top right, since approach circle gets thinner as it shrinks, and the whiter part of the hit circle ends up displaying a lot better on the monitor, overpowering the red. An approach circle as seen in Figure 12 is detected with much more accuracy, so we made an adjustable delay using `sw[9:7]`, which increases the delay from 50ms to 350ms.

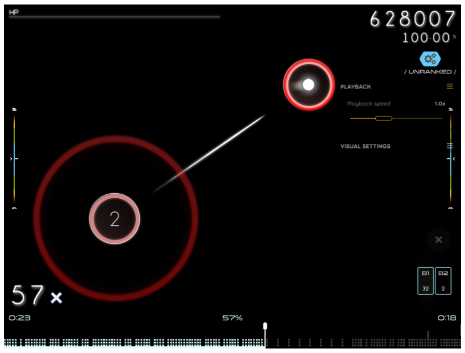

Figure 15: Approach circle that is hard to detect

Similarly, the click cooldown was made so that the system wouldn't keep clicking after it clicked once, and that cooldown can be adjusted with `sw[6:4]`.

One of the most fun demos we did with our system was display a screenshot image in a graphics editing program, and then have the FPGA send clicks to the computer which then drew on the screenshot. An example of that can be seen here, where Silvia had some trouble keeping the camera steady on the FPGA, but overall it can be seen that white circles are being drawn on the screenshot.

If we'd had more time on this project, it would've been good to dive further into getting the system to work on osu! in real time, not just on screenshots. It would've also been interesting to try and incorporate other methods of detection so that the FPGA system could handle the slider and spinner osu! game elements. We both had a fun time working on this project!

References

[1] *Circle Hough Transform.* URL: https://en.wikipedia.org/wiki/Circle_Hough_Transform.