Harris corner detection in an FPGA

Marc Cataford, McGill University - Matthew Beirouti, McGill University

Abstract

This project is aimed at implementing a Harris corner detection system in VHDL. The necessary background is first introduced. Design methodology and design decision are presented along with validation techniques and testing results. Major challenges encountered during the design and implementation phases are discussed. Finally, focus areas for future iterative improvement based on this project are suggested.

I. INTRODUCTION

Mobile sensing robots require a mechanism by which they can use their sensors to map new surroundings and localize themselves. ORB-SLAM is one particular method through which this can be done.

Our project will implement Harris corner extraction, the first step in the ORB-SLAM algorithm, on individual images in HDL. This extractor will run an image through various stages that will compute the values necessary to determine whether a sharp change in intensity is present at that position, signalling an edge (or corner) [1, 2]. Depending on time availability, we may expand the functionality of our system to include synthesizable VHDL along with proceeding steps in the ORB-SLAM algorithm.

Customarily, these algorithms run on a CPU with an operating system but can be accelerated using GPUs or dedicated hardware. Since ORB-SLAM is used in state of the art exploring robots and other autonomous vehicles, running it on dedicated hardware such as a programmed FPGA is interesting and can significantly improve performance. Whereas GPUs do provide significant increases in performance as well, they can also be quite expensive. FPGAs are relatively less expensive and have the potential to provide similar increases in performance, which makes them a preferred option. Given that FPGAs are more accessible to students and this particular algorithm can have profound applications (including AI and even search and rescue robots), this project was a worthy endeavour. We will produce the groundwork for an FPGA implementation of harris corner detection, upon which future endeavours can build.

II. BACKGROUND

In computer vision, ORB feature extraction allows machines to recognize visual elements regardless of how they are oriented and to track them in space. It is a process essential to the implementation of SLAM (Simultaneous Localization and Mapping) systems, which provide a way for autonomous robots to navigate unknown environments and assemble maps of their surroundings in real-time.

The ORB feature extraction process, outlined in *Figure 1* below, considers the image input in n x n pixel patches and calculates the xy gradient of each of them to obtain a *Harris score* representing the confidence that a corner is present in the patch.

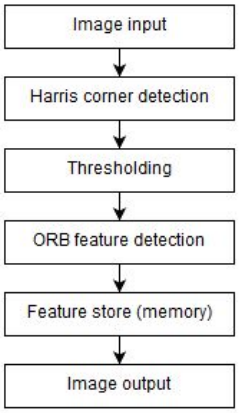


Figure 1: ORB feature extraction process

The patches with the highest score are then selected via *thresholding* and are forwarded to another layer of mathematical processing that will determine the *ORB descriptor*, a value that is unique to the patch and does not depend on orientation. Finally, this descriptor can be stored and tracked as the image input changes.

III. DESIGN METHODOLOGY

The first step in the design process of our Harris Corner Extraction circuit was to divide the extraction process shown in *Figure 1* into modules that can be designed, implemented and tested separately. This *divide-and-conquer* approach made the implementation of the feature extraction process much easier to plan and manage.

In addition to the stages included in *Figure 1*, two external scripts were added to emulate the input and output stages of our system, which would be ensured by third-party hardware in real-life applications.

At the end of the implementation of each module, a thorough testbench was designed to ensure the quality and reliability of the component. This allowed us to assess difficulties that may arise in the future (such as those related to timing, input/output value ranges and signal propagation).

A. Support scripting

In order to reduce the complexity of input and output processing in the context of simulation, two Python support scripts were used to handle reading source images (regardless of file format) and writing pixel data back to images for inspection. The pre- and post-processing scripts make use of the Python Imaging Library [4] to allow format-agnostic image processing.

The pre-processing script reads standard format image files (such as JPEG and PNG). It extracts the file's raw/uncompressed RGB pixel matrix. The pixel matrix is then grayscaled to obtain a single byte-wide pixel value, which allows us to use 8-bit VHDL vectors to handle them. This data is then rewritten into a binary file intended for the VHDL file input module of our system. The processed data only contains the grayscaled pixels; for memory design purposes, a static size of 800x600 was used for all images, allowing us to discard the image dimension data.

The post-processing script reconstructs the image file from the pixel data that has been processed by the corner extractor. This allowed us to visually inspect the resulting images and locate the detected features, which can then be compared to the features found by other implementations of the system. For further quality testing, automated comparisons between binary files were made using Linux's *diff* function, which can be used to track changes across different versions of the same file.

B. Memory blocks

Since the extraction process requires the full processing of the image in multiple steps, image-sized memory blocks were designed to act as buffers for partially-processed pixel matrices.

Our generic memory blocks were implemented as finite state machines so that they could handle being populated by file I/O (an instantaneous process during simulation) or by a continuous pixel-wide stream. Internally, the memory is a multidimensional vector array acted upon by the finite state machine. A single request signal is used to both read from the memory contents and write to the memory of stall operations, to allow for modules dependent on the memory's output to finish their task before modifying signals. This request signal has to be toggled in two consecutive clock cycles to load new data to the output.

The memory block provides access to three tri-byte words that represent a square patch of 9 pixels. As further operation requires access to any individual pixel in the patch being processed, an additional component was attached to memory: the pixel management unit.

The pixel management unit is a byte-addressable buffer that contains the 3x3 pixel patch given as output by our memory block. Being a purely combinational logic component, it doesn't affect the timing of the memory block's finite state machine and only acts as a *signal splitter*.

It accepts three rows of 3 pixels from the input stream as three 24-bit vectors, and breaks them down into byte-length segments representing individual pixels.

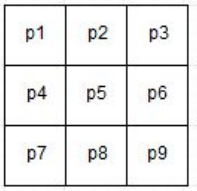


Figure 2: A pixel patch in the PMU and its signals

By convention, the top-left pixel is considered to be the first pixel. The PMU has 9 output lines corresponding to each of those pixels, giving complete and direct access of its content to the next stage.

The memory block formed by the finite state machine described earlier and the PMU is used as a stimulus source, taking the place of a hardware camera device that would feed the system with its own frame buffer. It also serves as intermediary buffer to hold the gradient matrices associated with the original image's pixel values, and the filtered values produced by the Gaussian blur stage.

C. Harris corner detector

The Harris corner detector is an implementation of the Harris corner score computation as defined below:

eq1:

$$har = g(I_x^2)g(I_y^2) - [g(I_xI_y)]^2 - \alpha[g(I_x^2) - g(I_y^2)]^2$$

This score is a combination of squared x, squared y and xy gradients of the pixel patch, on which we apply a gaussian blur in order to reduce noise. The computation itself is only considered in the last stage of the process (*Harris extractor*), and the preceding stages prepare the intermediate values involved in the computation. Note that the value of *alpha* is an empirical constant between 0.04 and 0.06.

The detection scheme was separated into 6 synchronous stages as illustrated in Figure 3 below. In the XY Grad component, the gradient of the image (that is the change in pixel values) in the x direction and the gradient of the image in the y direction are calculated. This gives us I_x and I_y. Following this, the Square stage takes I_x and I_y and produces $(I_x)^2$, $(I_y)^2$ and I_xI_y. Since the Gaussian filter (similar to the gradient operation) requires a patch of 9 pixels to work on, the memory needs to collect the results of the first two operations on the entire image, resulting in 3 new images before allowing the Gaussian filter to proceed. This dependency can be optimized away and is discussed in section **VI**.

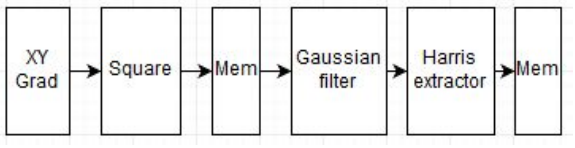


Figure 3: Harris corner detection process in 6 steps

The XY gradient calculation unit takes advantage of the *Sobel operator* [5], which can be used to obtain the gradient of the pixel patch via a convolution. For example, the x-gradient can be obtained by using the x-oriented Sobel kernel as follows, where **A** is the pixel patch:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A}$$

The convolution with the kernel simply reflects a weighted difference between pixel values oriented in the x direction. Therefore it simply reduces to the multiplication of the weights in the kernel by the corresponding pixels, with all the resulting values summed together. This was implemented behaviourally in the gradient component. It is important to note that the Gaussian filter operates in a similar manner. Namely, it uses a kernel (a matrix similar to the one above but with different weights) to operate on a patch of pixels at a time, producing a value for the pixel (which is now somewhat blurred as a result of the Gaussian filter) underneath the center of the kernel matrix.

The Square and Harris Extractor units were also implemented behaviourally, producing components of minor code complexity.

Once the first memory module has the entirety of the three semi-processed images (from the x gradient, y gradient and x gradient multiplied by the y gradient), the Gaussian filter can then be run over these images to produce three different filtered (or blurred) images which consequently provides $g(I_x^2)$, $g(I_y^2)$ and $g(I_xI_y)$. Finally, the Harris extractor uses these three images and plugs them into eq1 above to produce a Harris score for each pixel on the image. This is then output to the *thresholding unit* and an output file for validation purposes.

As mentioned before, each of these stages are synchronous. While this isn't entirely necessary, it was done to allow for flexibility with respect to optimization in future iterations of this design. A further optimized design with parallelized operations could conceivably use a clock to synchronize data flow. Moreover, if necessary, its removal is as simple as a change in a 3 lines of code for each component. Finally, a synchronous design simplifies manual performance analysis and removes any bottlenecks on clock cycle time, which is discussed in section **V.A.2**.

D. Thresholding unit

Following the *Harris score* computation, the *thresholding unit* serves as a filter to select *high-confidence points* based on their score. The threshold value is defined as an input to this module, as it can be either statically defined or computed through a feedback network. Our design allows for an extra module implementing the feedback properties required for the latter to function, to be integrated without major overhaul of this part of the system.

Internally, the *thresholding unit* is a simple combinational comparator that produces a binary

value determining whether the current patch is retained as a Harris corner.

IV. TESTING METHODOLOGY & RESULTS

A. Module-level validation

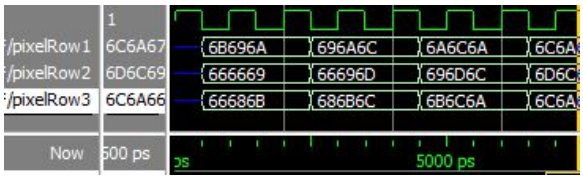
As mentioned in the *Design methodology* section, each module was tested separately to ensure that it met the benchmarks defined when it was designed. Modular testing was done on two types of modules: memory modules and arithmetic modules.

A.1 Memory validation

Memory modules were evaluated based on their ability to be populated by a preprocessed binary file or by a pixel stream, which would originate from different stages of the Harris score computation.

To evaluate the correctness of the writing process, binary files were fed to the memory, which was then dumped to a text file using Modelsim's *Memory viewer*. The memory dumps were compared to the original binaries using the *diff* function, mentioned in *section II.A*. No warping was observed (no different bytes across files). A similar test was carried out by generating a predictable pixel stream as pixel input to memory. An example of the files used can be found in the files appended to this report.

To ensure that the finite state machine at the core of the memory block behaved correctly, extensive logging was included in each of the states to allow for tracking from start to finish. Additionally, inspection of the output waveform was made to ensure that each "output step" was regular in length (as predicted since the FSM has a regular 1 cycle per state and the subcycle that outputs data has 2 states) and that the signals produced were sequential.



3x3 pixel patches being shifted every two cycles

Since the pixel patch travelled through the entire image one pixel at a time, it was also important to confirm that each update of the output would shift out a pixel to the left, and shift in a new one to the right. This can be seen in the figure above.

A.2 Arithmetic unit validation

Arithmetic modules were tested in a similar way. Since our design is implemented behaviourally and used solely in simulation, the propagation delay is assumed to be null. As such, all arithmetic modules are combinational. Their testbenches then only assert the validity of the output signal's value by comparing it to the result of the same calculation as executed by the testbench code.

Each component implementing the arithmetic operations of the global design was tested using three main cases: lower limit, upper limit and intermediary input test cases. The upper and lower limit input test cases were used to ensure that the functionality of the individual components was not thrown off in the case of overflow (numbers greater than 255, the maximum pixel value for an 8 bit image) or underflow (numbers less than 0, the minimum pixel value). The intermediary test cases included both trivial values and random values to cover a good-enough distribution of all possible input values.

Each component was tested using it's own testbench and the results of the simulations are displayed in appendix figures A1 through A6. These figures are also attached to the project submission. Note that annotations aren't necessary as the output waveforms should speak for themselves. Helper scripts were utilized to ensure fast collection of expected output values without the need to calculate them manually. Thus a small helper script was written in C for each arithmetic component and an example of how these scripts were used are shown in appendix figure A5 .

B. System-level validation

System level validation was applied incrementally as we assembled the final VHDL entity. This gave us the chance to revalidate each module by itself as well as in its interactions with the others.

The arithmetic units were first assembled into bigger units: one composed of the gradient calculators and the squaring units, one dedicated to the Gaussian blur filter, and finally the Harris score calculation unit.

These subdivisions stem from the arithmetic segments of our circuit that lie between memory blocks.

Summary validation was made by verifying that the primary memory module, meant to emulate camera output (and feed initial image data to the system) could interact with the gradient-square module. Using all-black and all-white source images (containing *ff* and *00* bytes exclusively), the output of the gradient-square module could be easily predicted (ie. null gradient), and non-trivial values (sourced from a photo taken and preprocessed during testing) were calculated and correlated with values calculated with the same set of operations, implemented in C.

The C implementation of the process followed closely the arithmetic involved in the VHDL implementation, we then assumed, with confidence, the correctness of the output we observed.

C. Support script validation

C.1 Pre-processing

Preprocessing validation has been executed by processing trivial images (all black or all white) into binaries and manually checking the results (the former would produce a zero-ed file, and the latter, a file in which all bytes are *ff*). This established basic correctness.

More elaborate tests have been run with a variety of images, the verification of correctness being done by using the *diff* command-line function to search for discrepancies between the initial and resulting pixel matrices.

Visual inspection was also done on the resulting images to ensure that no warping was introduced during the preprocessing, which should not change the image content beyond grayscaling it.

Left: original image, right: preprocessed image

C.2 Post-processing

The post-processing script testing used the results of the pre-processing testing and reconstructed the original image. A visual inspection was then done to ensure that little or no noise was introduced in the image, as the noisy pixels would affect the 3x3 patches analyzed by the feature extraction system.

Similarly to the preprocessed sample, a *diff*-based comparison was also made between the image generated by the preprocessing script (in parallel with the stripped binary file that was used for as VHDL input) and the *reconstructed* image.

Left: image produced by the preprocessing script, simple grayscaling of the original, right: image reconstructed from the binary file by the postprocessing script.

This testing method was also used to validate the memory writing/reading process, as the transition between the binary file and the VHDL process memory (and back) is also a potential source of corruption.

Images similar to the ones above were processed, passed through the memory modules and reconstructed to ensure that no distortion occurred along the way.

V. ANALYSIS

A. Module-level validation

A.1 Memory validation

The memory modules were all confirmed as functional given the benchmark that the unit must latch onto the input on the rising edge and update the output on the next rising edge. Since our design is bound to a simulation environment, we have no maximum operating frequency to report.

A.2 Arithmetic unit validation

The arithmetic simulation results all matched their helper script counterparts, proving the validity of our designs. Since the modules were implemented behaviourally, we expected this to happen as we were able to use VHDL's high-level types (*integer*) as well as type conversions to avoid implementing standard logic vector algorithms for those operations.

Similar to the tests run on the memory units, no maximum operating frequency is defined for our design as it is bound to simulation.

With that being said, we can make an educated guess for the performance of a synthesizable version of this code. Assuming (reasonably) a board clock period of about 6.5 nanoseconds. Given that each subcomponent operates synchronously, a single component operation takes 1 clock cycle of delay. There are 4 component operations in total therefore 4 clock cycles of delay for arithmetic per image pixel. For memory, if the images being operated on are 800*600 pixels large, and each writing and reading from memory takes 2 clock cycles we get a total of 6 clock cycles per image pixel. 2 of these result from reading the initial image from memory, 2 from putting it back into memory after the gradient squares are calculated, and 2 result from recovering it back from memory to apply a Gaussian filter. So in total we have 10 clock cycles of delay per image pixel (4 from arithmetic and 6 from memory). If we multiply this by the number of pixels per image and the period of a single clock cycle we get 800*600*6.5ns*10 = 0.0312 seconds per image. The reciprocal of this value gives us approximately 32Hz which roughly equals the framerate of most cameras. For a brute force, unoptimized first attempt with over exaggerations for memory access and storage times, this analysis does seem promising. Future iterations of this design could also be improved to gain performance as discussed below.

B. System-level validation

System-level validation proved to be our achilles heel. We were unable to fully validate our design using integration testing due to some mistakes made along the way. The main reason for this seems to be the fact that the tasks were divided perhaps a bit too cleanly. One team member was responsible for understanding and implementing pixel management functionality while the other was responsible for understanding and implementing arithmetic. This resulted in a laser like focus on individual tasks which proved difficult to execute given the challenging nature of image processing and computer vision techniques for the inexperienced (especially using VHDL). This also meant that less time was spent ensuring team and global module design cohesiveness. This ultimately resulted in a rough failure to integrate the components correctly. While the design does seem to be quite close to full functionality, small kinks with type conversions and system control still need to be resolved.

C. Challenges

One of the main challenges during this project was wrapping our heads around many of the subtleties involved with computer vision and image processing algorithms. While on the macroscale concepts are relatively graspable, microscale focus brings forward many practical issues that are glossed over in theoretical papers and guides. This created a constant need to go back to the drawing board to revise and retest existing modules to accommodate for newly discovered subtleties.

One example of such a subtlety is the possible output range of numbers between intermediary operations. While initially, this was thought to be purely based on the mathematics itself, it was later painstakingly discovered that these were fixed values and that it was normal to threshold any and all values that were out of range.

Another challenge faced was maintaining design cohesion between team members. At times, both members discovered they were implementing their understanding of the design even though the members' understanding of the design didn't match. This caused unnecessary delays that were uncovered relatively late.

Another challenge included the developing of manual test benches. This was time consuming and involved modifying every test bench when the design was modified.

Finally, it was sometimes difficult to make decisions with regards to optimization. On one hand a better optimized system makes for increased efficiency and therefore faster performance. However, on the other hand optimization with such design problems increases design complexity exponentially and thus it was challenging to find an acceptable balance between design complexity and performance.

VI. IMPROVEMENTS AND CONCLUSION

For future iterations of this project, it would be wise to draw detailed paper designs that are agreed upon by and accessible to all team members. These handwritten designs would be treated as the source for any implementation attempts.

As mentioned previously, certain sources of delay can be optimized away. For example, multiple arithmetic modules can work in parallel on different parts of the same image in memory to produce intermediary results more quickly. This adds a level of complexity however with regards to timing and the merging of obtained results. Additionally, delay resulting from providing pixels to modules through the pixel management unit can be reduced by using a memory buffer to provide data as soon as it is available (instead of waiting for the entire image to load) and by replacing only one row of pixel data at a time.

Automated test creation and verification scripts could be designed to allow for easy testing at the click of a button as opposed to requiring a large number of repetitive and tedious steps.

In conclusion, although we came quite close to a functional design, challenges and mistakes discussed prevented this from occurring. Our design does however provide good ground level framework for future teams to iterate upon and optimize, which is a win based on our stated objective.

VI. REFERENCES

[1] J. Weberrus, et al. *ORB Feature extraction and matching in hardware* [Online] Available: http://www.araa.asn.au/acra/acra2015/papers/pap150.pdf

[2] E. Rublee et al. *ORB: An efficient alternative to SIFT or SURF (2011)* [Online] Available: http://vision.cs.chubu.ac.jp/CV-R/pdf/Rublee_iccv2011.pdf

[3] R. Mur-Artal et al. *ORB-SLAM: A versatile and accurate monocular SLAM system* [Online] Available: https://arxiv.org/pdf/1502.00956v1.pdf

[4] F. Lundh, (2009, August) *Python Imaging Library Handbook* [Online] Available: http://effbot.org/imagingbook/

[5]O.R. Vincent, O. Folorunso *A descriptive algorithm for Sobel image edge detection* [Online] Available: http://proceedings.informingscience.org/InSITE2009/InSITE09p097-107Vincent613.pdf

VII. APPENDICES

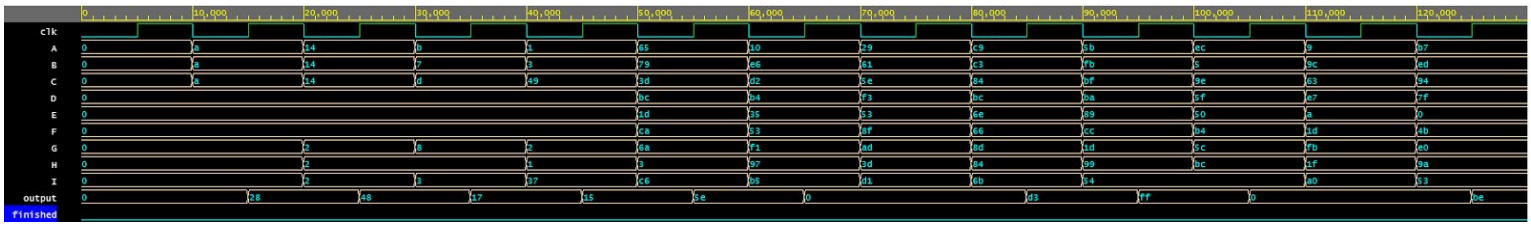


Figure A1: Gradient component unit tests for both X and Y directions. Since Sobel operator for Y is just a transpose of the operator for X, the inputs A, B, C and so on were also transposed, making the testbenches very similar. Attached as xGradientTest and yGradientTest.

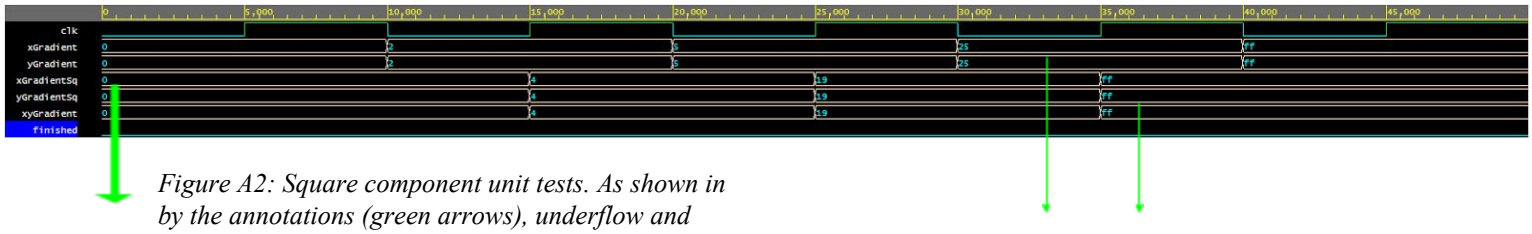


Figure A2: Square component unit tests. As shown in by the annotations (green arrows), underflow and overflow are capped at 0 and 0xFF or 255. Attached as squaresTest.

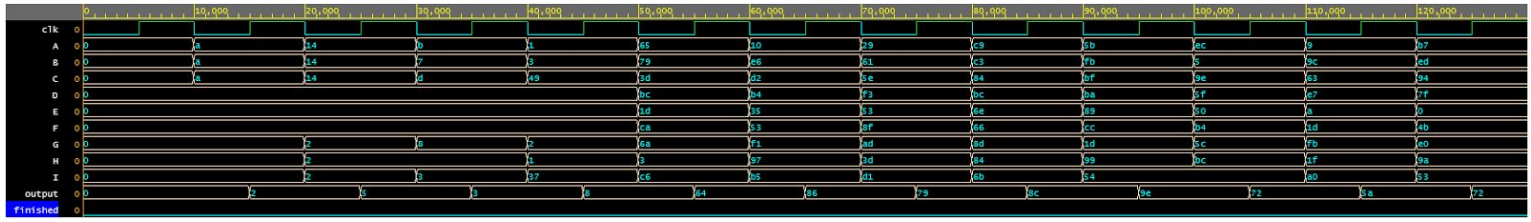


Figure A3: Unit tests for the Gaussian filter operation. Verified similar to previous operations. Attached as gaussianTest.

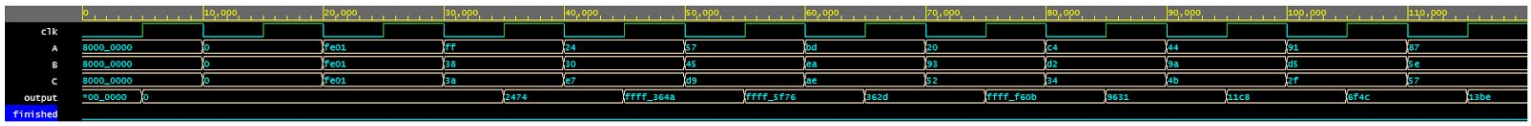


Figure A4: Unit tests for the final Harris operation. The verification procedure using the helper script is shown in Figure A5 below. Attached as harrisTest.

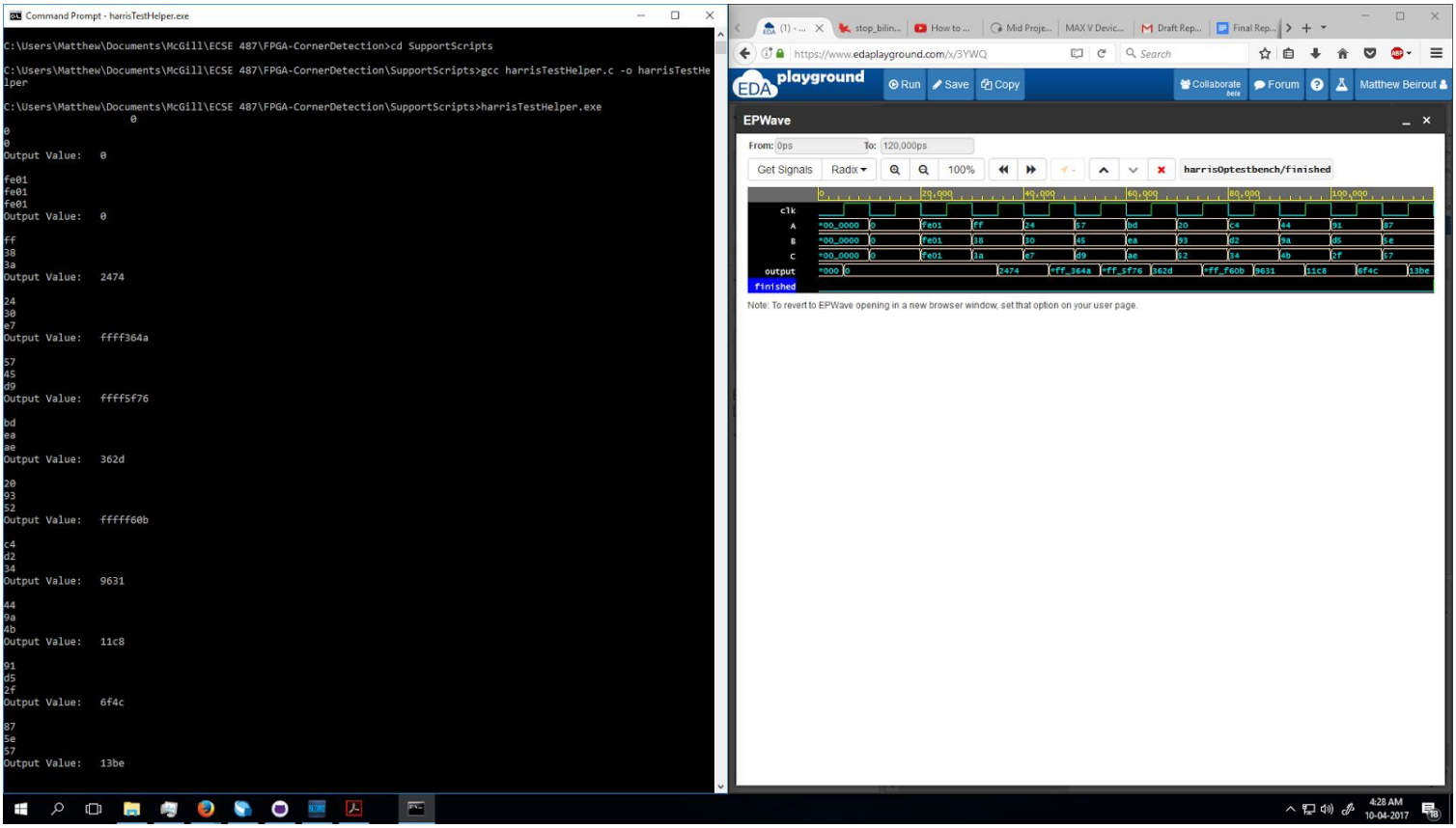


Figure A5: The helper script (left) was run to verify the output values obtained from the VHDL design on the right. Attached as testingWithSupportScriptExample.

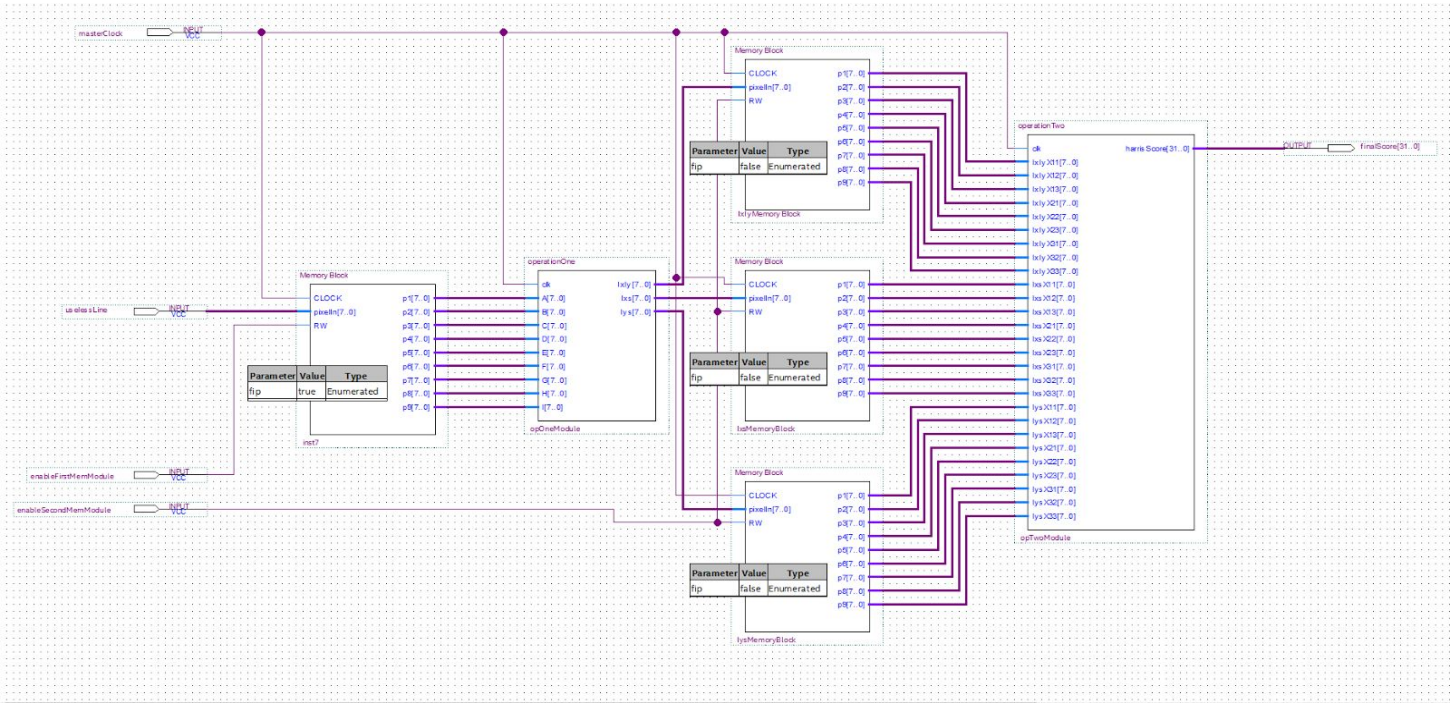


Figure A6: The design of our global module. The 6 component block diagram (Figure 3) resolves to this in practicality. Attached as globalModule.