Real-Time Feature Detection and Tracking on FPGA Hardware

Final Year Project
Final Report

Brad Taylor

n9294775

Supervised by

Dr. Jasmine Banks

Executive Summary

The aim of this project is to develop a robust, real-time feature tracking design capable of implementation on low-end FPGA embedded platforms. Due to the high computational throughput required for real-time feature tracking, high performance, high cost FPGAs are typically required to implement most designs. This project aims to remedy this problem by assessing currently proposed feature tracking designs to streamline/compromise on their core feature detector/descriptor/matcher components in order to produce a more compact implementation.

This report describes the creation of a fully-functional, real-time feature tracking system for an FPGA. Realising this system required the design and implementation of numerous components. Firstly, a feature detection system was developed using a FAST detector, due to its very low complexity and ability to be pipelined. A small 9x9, 128-bit binary descriptor, based on BRISK, was designed and implemented as it demonstrated strong performance while using minimal system resources. A complete feature tracking module was developed from the ground up, allowing the $\mathcal{O}(n^2)$ complexity problem to be fully pipelined with acceptable use of system resources.

Testing of this system demonstrated its ability to track features throughout a video, currently capable of 720p at over 100fps. However, limitations were found regarding the system's lack of scale and rotational invariance, currently limiting its viability in a real-world scenario. Further work is required to improve this system, which includes the addition of scale and rotational invariance to the descriptor, along with the further optimisation and pipelining of existing components. This will result in an extremely fast and robust system that is completely viable for real-world feature tracking scenarios.

Contents

List of Figures

Glossary

BRAM Block ram.

BRIEF Binary Robust Independent Elementary Features.

BRISK Binary Robust Invariant Scalable Keypoints.

Contiguous Sharing a common border; connected.

DSP Digital signal processor.

FAST Features from Aaccelerated Segment Test.

Feature Descriptor Method of capturing the contents of a small image section.

FF Flip flop.

FIFO First-in, first-out, meaning the oldest contents are output first.

FPGA Field-programmable gate array.

FREAK Fast Retina Keypoint.

LUT Look up table.

LUTRAM Look up table random access memory.

Non-Maximal Suppression Suppression of all components which are not a local maxima.

SLAM Simultaneous Localisation and Mapping.

Vivado FPGA integrated development environment.

1 Introduction

With the advent of embedded platforms with increasing autonomy, robust, real-time visual sensory information is imperative for the performance of many tasks (e.g. Visual Odometry, SLAM, etc). This commonly requires the utilisation of visual feature trackers which are fundamentally a highly computationally expensive task. The low power constraints typically required on embedded platforms has motivated research towards the optimisation and implementation of these trackers on low power computing devices, namely FPGAs. However, these implementations are primarily focused towards high performance, high cost systems. This leaves open a highly demanded place in the market for low cost, strongly performing feature trackers for use in real time embedded applications.

The task currently undertaken in this project is to provide a remedy to this issue by proposing a feature tracking design and implementation capable of use on low end FPGA devices such as Xilinx Artix/Spartan. This poses a number of research questions regarding the components required to realise this system. Firstly, an investigation of feature detection, description, and matching processes is required to determine if they can designed in a manner that allows them to be efficiently pipelined on an FPGA. Furthermore, analysis of feature description methods is imperative to determine if smaller, lower complexity descriptors are able to be utilised while maintaining acceptable performance. Finally, as feature tracking generally has $\mathcal{O}(n^2)$ complexity, extensive investigation and design will be necessary to determine if this component can be realised on an FPGA without compromising the system's performance.

This report outlines the analysis, design, and implementation goals that have been achieved throughout this project, presenting descriptions of all designed and implemented components. Furthermore, an analysis of the currently implemented system is performed, followed by discussions of its current limitations. Finally, this report discusses how well the research questions were answered and what future work would be necessary to improve the system.

2 Literature Review

The primary objective of this project is to achieve real-time feature tracking, requiring the review of proposals and theories demonstrated in the literature. Feature tracking can be broken down into three main components: feature detection, feature description and feature matching. The most prevalent and innovative methods of implementing these components will be discussed. Furthermore, how these methods are mapped to FPGA hardware will be investigated.

2.1 Feature Detection

Feature detection, in the context of feature tracking, describes the process of finding discrete points within an image that demonstrate a strong local gradient variability (Harris & Stephens, 1988). Various feature detection methods exist, ranging significantly in mathematical complexity and accuracy. One of the most popular of these methods is known as FAST which demonstrates high computational efficiency while also retaining acceptable accuracy and repeatability (Gauglitz, Hllerer & Turk, 2011).

2.1.1 FAST

The FAST method iterates over an image and devises simple logic tests to determine the adequacy of a candidate pixel as a feature point (Rosten & Drummond, 2006). The candidate pixel is compared to the intensities of 16 surrounding pixels that lie on a $r = 3$ Bresenham circle. This is demonstrated in Figure 1.

A feature is detected when the Bresenham pixels $I_{p \to x}$ have at least 9 contiguous pixels that are all sufficiently darker or all sufficiently brighter than the center pixel I_p (Rosten, Porter & Drummond, 2010). This piecewise test is shown in the following equation where t represents a tuned threshold value.

$$S_{p \to x} = \begin{cases} \text{Darker} & I_{p \to x} \leq I_p - t \\ \text{Brighter} & I_{p \to x} \geq I_p + t \end{cases} \qquad x = 1, ..., 16 \tag{1}$$

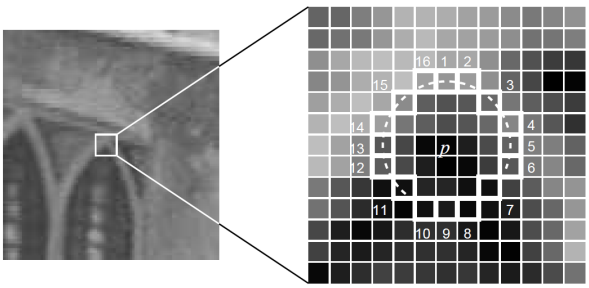


Figure 1: Bresenham circle demonstration (Rosten & Drummond, 2006)

Non-Maximal Suppression

To prevent the detection of multiple feature points within a close proximity, non-maximal suppression is utilized. This leaves only the strongest feature point for each local region. To achieve this, a weight must be assigned to each local feature. The equation below calculates a weight based on the sum of absolute differences between the center pixel and pixels on the Bresenham circle. $S_{\text{bright}}/S_{\text{dark}}$ are pixels belonging to the brighter and darker sets.

$$V = max\left(\sum_{x \in S_{\text{bright}}} |I_{p\to x} - I_p| - t, \sum_{x \in S_{\text{dark}}} |I_p - I_{p\to x}| - t\right) \tag{2}$$

2.1.2 Harris

The Harris feature detector firstly computes the gradients of an image I via discrete approximation using the following formulae.

$$X = I \otimes [-1, 0, 1]$$
$$Y = I \otimes [-1, 0, 1]^T$$

Matrix values A, B, C are then computed by convolving X and Y combinations with an $N \times N$ window, w.

$$A = X^2 \otimes w$$
$$B = Y^2 \otimes w$$
$$C = XY \otimes w$$

$$M = \begin{bmatrix} A & C \\ C & B \end{bmatrix}$$

The eigenvalues (α, β) of matrix M are computed and their values allow for the categorization of each image pixel into that of 3 categories (edge, corner, flat). These categories are visually demonstrated in Figure 2.
An optimization proposed eliminates the requirement of directly computing eigenvalues and instead approximating the response with the following formula.

$$R = |M| - k(M_{11} + M_{22})^2 \qquad 0.04 \leq k \leq 0.06$$

The resulting value R is then able to be used with a threshold value T to categorize the feature using the following piecewise.

$$\text{Category} = \begin{cases} \text{Corner} & R > T \\ \text{Edge} & R < -T \\ \text{Flat} & |R| < T \end{cases}$$

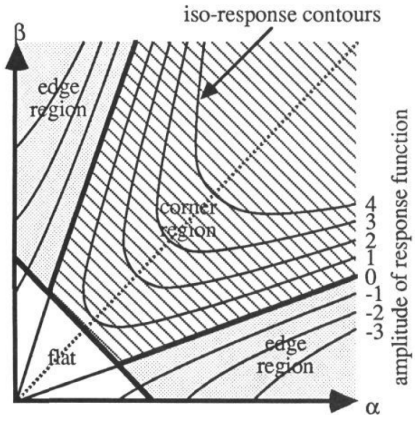


Figure 2: Relationship between eigenvalues and category (Harris & Stephens, 1988)

2.2 Feature Description

The past 15 years has seen an influx of feature descriptor proposals, which drastically range in performance. Recent advances in feature descriptors have given rise to binary descriptors due to their computational efficiency and accuracy, the most fundamental descriptor being BRIEF (Calonder et al., 2010).

Binary descriptors employ a pixel comparison method where pixel pairs at predefined positions within an $S \times S$ window w have the following logic applied

$$\lambda_i(w; p_i, q_i) = \begin{cases} 1 & w(p_i) < w(q_i) \\ 0 & \text{otherwise} \end{cases} \qquad i = 1, ..., 256 \tag{3}$$

Where p_i and q_i denote the positions of 2 predefined pixels within patch w. This logical operation is applied to 256 pixel pairs producing a 256-bit description of the feature.

Examples of predefined pixel pairs are shown in Figure 3 where lines connect pixel pair positions within a 33×33 window, centered on a pixel recognized as a feature point.

The FREAK and BRISK proposals built upon this methodology, each proposing their own predefined, 'radially symmetric' pixel pair layouts which promote pairs closer to the center (Alahi et al., 2012; Leutenegger et al., 2011). An example of these layouts is shown in figure 4 where a BRISK pixel pair pattern is demonstrated. This methodology provides a markedly more robust descriptor with no increase in computational cost. Furthermore, each author also proposed an efficient method of implementing rotational invariance by using already tested pixel pairs, drastically improving the descriptors robustness.

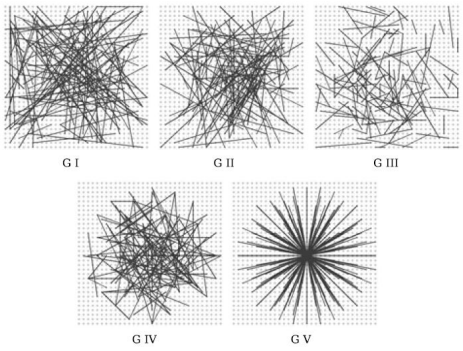


Figure 3: Potential BRIEF pixel pair patterns (Calonder et al., 2010)

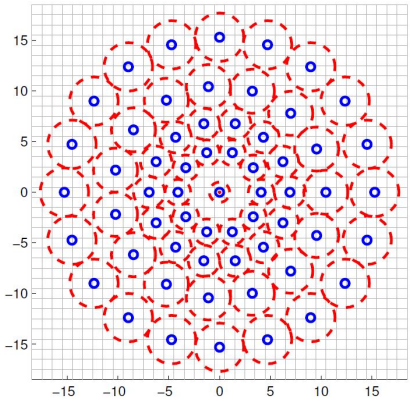


Figure 4: Improved BRISK pixel pairs (Leutenegger, Chli & Siegwart, 2011)

2.3 Feature Matching

Feature matching, in the context of feature tracking is the method of comparing descriptors against a library of descriptors (created in subsequent video frames) in an effort to find the same feature across multiple video frames. For binary descriptors, a simple method can be deployed to determine the acceptability of a match by calculating the Hamming distance H between the new descriptor λ and a library descriptor entry τ (Calonder et al., 2010). A match is recognized if the hamming distance between λ and τ does not exceed a predetermined threshold value T. This is shown in the following piecewise.

$$H = \sum_{i=0}^{N} \lambda_i \oplus \tau_i \qquad R = \begin{cases} \text{Match} & H \leq T \\ \text{Fail} & \text{otherwise} \end{cases} \tag{4}$$

It is to be noted that although binary descriptors are significantly faster than conventional floating-point descriptors, feature matching still has $O(n^2)$ complexity which requires significant computational effort when high feature counts are tracked.

2.4 FPGA Implementation

Recent design proposals have endeavored to couple FAST detection along with binary descriptors such as BRIEF to achieve real time feature tracking. The basic overview of how these modules are implemented in hardware will be discussed.

FPGA Feature Detection
Brenot et al. (2015) proposed an FPGA implementation which utilized the FAST based detector. Figure 5 shows a block diagram of this implementation in which line buffers and register arrays are utilized to feed a 7x7 section of the image into the FAST detector to be tested for contiguity. For non-maximal suppression, a CornerScore module (see figure 6) is implemented in parallel with the thresholding module, applying weights to potential features (see eq. 2). This design provides simple and strongly performing feature detection with relatively low resources.

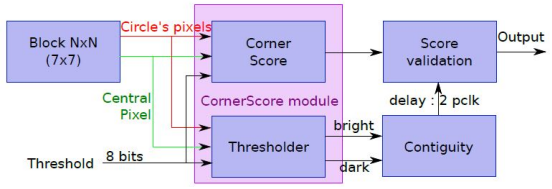


Figure 5: FAST Detector Module (Brenot et al., 2015)

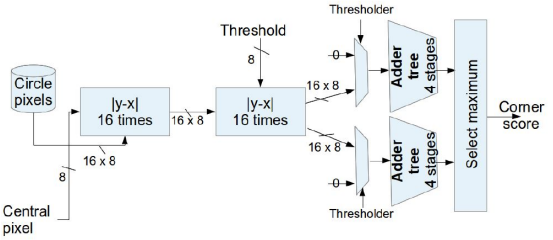


Figure 6: CornerScore Module (Brenot et al., 2015)

FPGA Feature Description

Due to the simplistic nature of binary feature descriptors, most implementations map very cleanly to hardware. Typically, to form feature descriptors, a 33×33 image window of pixels is held in registers and logic tests are applied (see section 2.2) to predefined pixel positions within the window. The binary results of these logic tests are concatenated to form a 256-bit descriptor of the feature. This design is fast, typically requiring only a few clock cycles (varies between designs), however the storing of an image window in 33×33 registers uses significant resources.

As descriptors are highly sensitive to noise, image smoothing filtering is commonly applied. Rather than Gaussian filters, most common FPGA designs such as that of Fularz et al. (2015) utilize simple averaging filters to remove noise as they maintain acceptable performance while requiring only simple addition and bit-shift operations.

An effective, generic implementation of a FAST/BRIEF feature detector/descriptor is shown in appendix 1 in which the block diagram demonstrates all of the core components required for feature tracking. Further detail of these core components will be provided in the project overview.

FPGA Resource Usage

The respective footprints of recent proposals vary significantly in DSP/LUT/Register/BRAM usage, however typically require higher cost/performance ZYNC FPGAs to fit these footprints (see table 1).

	Ulusel et al. (2016)	Wang et al. (2014)	Zhu et al. (2016)
Approach	FAST/BRISK	FAST/BRIEF	FAST/BRIEF
FPGA	ZYNC	ZYNC	ZYNC
DSP	0	52	0
LUTs	25575	18437	12368
Registers	7115	13007	10156
BRAMs	11	230	100

Table 1: Recent FPGA implementation resource usages

3 Progress

The aim for this project is to implement a functioning feature tracking system on an FPGA. This has required various levels of design from broader design decisions such as feature detection/description methods, to FPGA specific designs including pipelining and resource optimisation. The following sections document this design process as well as provide FPGA implementations for all designed components. Furthermore, the effectiveness of the completed system is demonstrated in section 3.10.

3.1 System Overview

In order to track a feature across frames of a video, numerous processes are required including feature detection, description, indexing, and matching. These processes are shown below in figure 7 where they have been designed as separate modules which connect together to perform the full feature tracking process.

This system receives two inputs, a stream of pixels from a camera or equivalent source, and a flag, denoting when a new video frame begins. Using this data, the feature tracking system detects features (see section 3.4), generates their binary descriptors (see section 3.5), and determines their position in the frame (see section 3.7). These modules are carefully delayed and pipelined as to cause their outputs to be synchronised, allowing a rich set of information to be constructed for each feature (see section 3.8).

The information provided by these modules is utilised by a feature tracking module, capable of simultaneously tracking numerous features throughout a video (see section 3.9). This is achieved with the use of a feature comparator module (see section 3.6) which allows two descriptors to be compared to determine if they are the same feature. This system outputs the movements of tracked features, including their current position, where they were first detected, and how many frames have since elapsed; $[n_f, x_1, y_1, x_2, y_2]$. An example of this output is shown in figure 8 where a simulation was performed, showing a feature being tracked across five consecutive frames.

Code for this module can be found in appendix A4. All simulations of this module were performed with the code found in appendix A3.

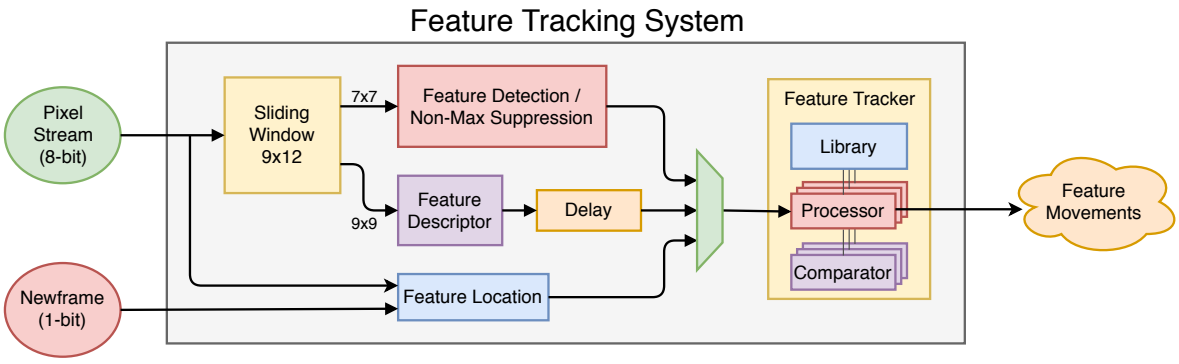


Figure 7: Feature Tracking System Design Overview

```
1,  242,   14,  241,   14
2,  242,   14,  239,   14
3,  242,   14,  236,   14
4,  242,   14,  233,   13
5,  242,   14,  230,   12
```

Figure 8: Feature tracking system output

3.2 Sliding Window

To provide the feature detection and description modules access to the contents of a video frame, a sliding window was designed and implemented on the FPGA. This module is responsible for converting a serial stream of pixels into an accessible $N \times M$ window which progressively moves across an image, row by row, until the bottom of the image is reached. The architecture of this sliding window implementation is shown in figure 9, however displayed as a 3×3 window for simplicity. The arrows demonstrate the flow of data at each clock cycle.

To provide the parallel output from a pixel stream input, multiple lines of the image must be stored in FIFO type memory, known as line buffers. For an $N \times M$ window, $M - 1$ line buffers are required and their depth is that of the image width.

The source code for this module is listed in appendix A5. Additionally, the source code of the line buffers required by this module is listed in appendix A8.

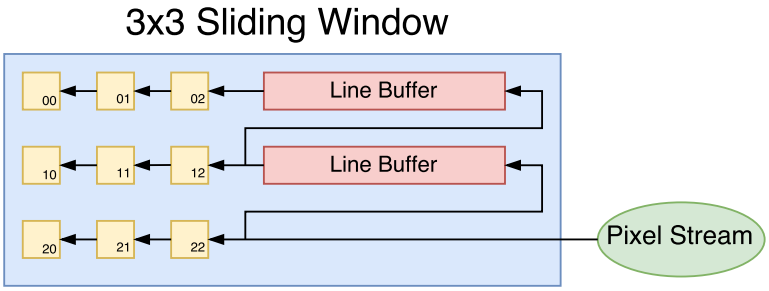


Figure 9: 3x3 sliding window diagram

3.3 Feature Detection Method Analysis

Through significant research of feature detectors, the choice of detector was narrowed down to two: FAST and Harris.

The first feature detection method considered was the Harris detector. This is due to its strong repeatability performance and as well as robustness to different types of image scenery (Harris & Stephens, 1988). As stated in the literature review, this method uses discrete approximations of local gradients to form an $M_{2\times2}$ auto-correlation matrix. The eigenvalues of this matrix can be computed and used to categorize the current region into either: corner, edge, flat.

The second feature detection method considered was the FAST detector. This is due to FAST demonstrating high computational efficiency while also retaining acceptable accuracy and repeatability (Gauglitz, Hllerer & Turk, 2011). As stated in the Section 2.1.1 of the literature review, to perform

feature detection, FAST performs simple logical arithmetic on the intensities of 16 surrounding pixels that lie on a $r = 3$ Bresenham circle about the pixel of interest (center pixel). If the differences between 9 or more contiguous Bresenham pixels and the center pixel exceed a predetermined threshold, the center pixel is considered a feature.

When comparing these two detectors, the FAST detector was found to have many distinct advantages over the Harris detector in terms of FPGA suitability. The FAST detector requires only low level arithmetic to implement $(+, -, <, >)$ whereas the Harris detector requires numerous multiplication stages throughout its detection process which can require significant resources and limit performance. Although some FPGAs combat this issue by offering DSP blocks, this FPGA feature detector is targeted towards lower end FPGA models which may not have enough or any of these resources. Additionally, the FAST detector can be efficiently pipelined with minor memory consumption, whereas the Harris detector requires a number of buffered stages throughout its pipeline that consumes 2-3 times more memory than that of FAST. Therefore the design decision was made to implement FAST as the method of feature detection.

Nearly all features detected in images span across numerous pixels. This causes feature detectors to detect features not only at the pixel which best represents the feature's position, but likely at surrounding pixels. This results in image features with multiple detections that cluster around the true location of the feature. To prevent this, a non-maximal suppression module was added to the feature detection pipeline which will remove all detections which are not the true feature. For non-maximal suppression to work, each feature must be assigned a weight that is representative of the feature's strength. Non-maximal suppression pairs well to the FAST feature detector as this weight can simply be calculated by summing up all of the previously calculated absolute differences between the Bresenham pixels and the center pixel (see Section 2.1.1 of Literature Review).

3.4 Feature Detection Module

The designed feature detection module contains two main components, the feature detection logic as well as the non-maximal suppression sub-module. The method in which the module's components interconnect is depicted in figure 10. This module receives a 7×7 window of the current image and outputs a z^{-n} delayed binary flag which denotes if a feature exists at the respective pixel. The input/output pixel/flag delay is shown in the following formula

$$D = R_{\mathrm{NM}}(W + 1) + P$$

where R_{NM} is the non-maximal suppression window radius (see section 3.4.2), W is the image width, and P is the pipeline delay.

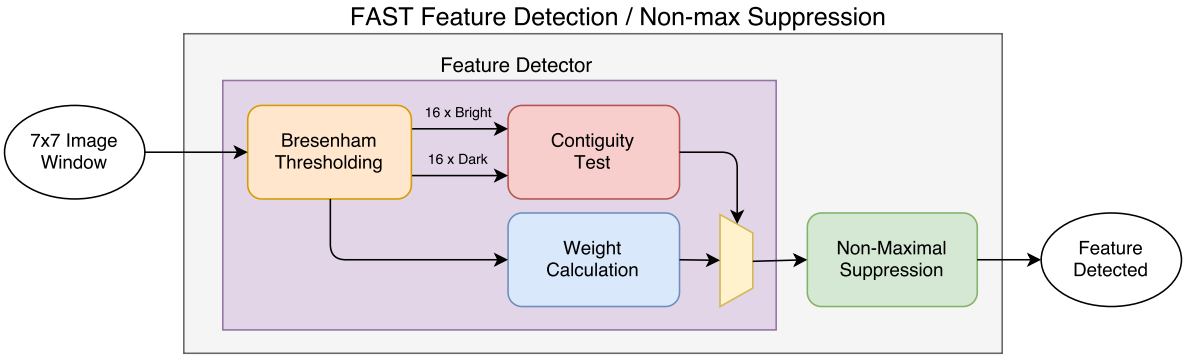


Figure 10: Feature detection module diagram

Feature detection results using Vivado simulation of this module can be found in appendix A1. Additionally, the code used for simulation of this module and all other modules can be found in appendix A3. The FPGA specific design and implementation of the module's sub-components are detailed below.

3.4.1 Feature Detector

The design for the feature detector implements a pipelined version of FAST. This module receives a 7×7 window of an image and outputs a z^{-n} weight value, denoting the strength of a detected feature (see fig. 10). Pixel positions with no features detected have their respective feature strength weights set to 0.

The data flow of the pipelined FAST logic is shown in figure 11. Firstly, this module remaps the Bresenham pixels (shown in red) in the 7×7 window to a 16-byte array. The absolute differences of these pixels is taken against the window's center pixel and the result is thresholded. Pixels which pass the thresholding have their respective bit set in either the 'Bright' or 'Dark' array depending on whether the pixel was brighter or darker than the center pixel in the initial window. Lastly, contiguity is assessed by searching the 'Bright' and 'Dark' arrays for 9 consecutive high bits (accounting for wrapping from bit 15 to bit 0). If contiguity exists in either array, a feature has been detected.

As previously discussed, if a feature is found, the feature weight is output from this module, passing into the non-maximal suppression module. This feature weight is computed by taking the sum of absolute differences between the pixel intensities of the Bresenham pixels and center pixel.

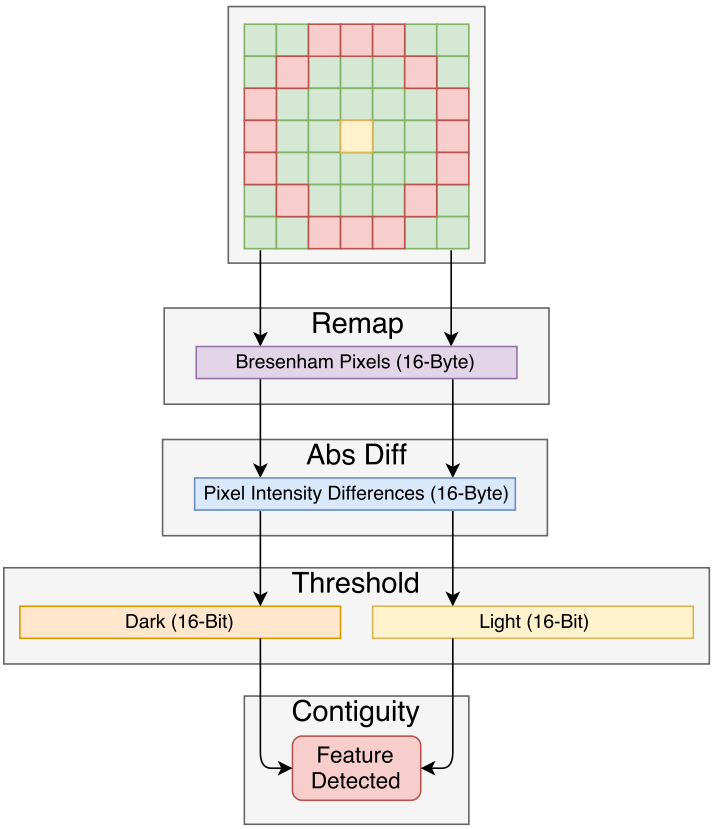


Figure 11: FAST feature detection logic

Shown in figure 12 is the result of a Vivado simulation of the described feature detection module which has performed feature detection on a grayscale image. As can be seen in the zoomed-in component of this image, multiple features have been detected for the same feature point in the scene. This demonstrates the need for non-maximal suppression.

The source code for this module is listed in appendix A6.

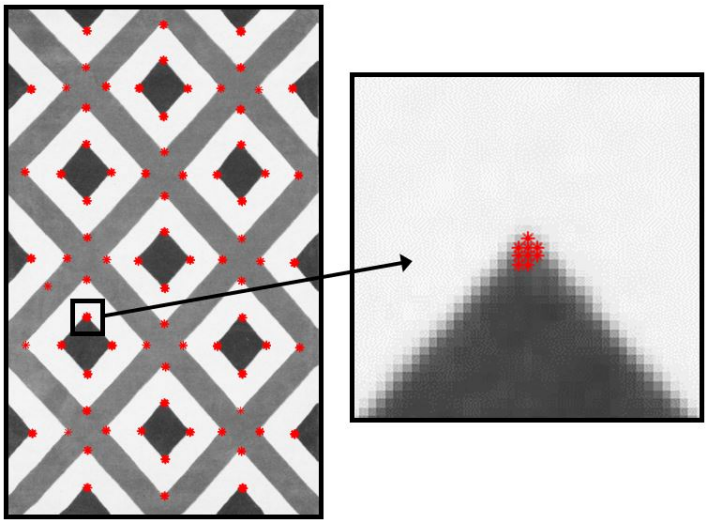


Figure 12: Feature detector results

3.4.2 Non-Maximal Suppression

The non-maximal suppression module is designed to remove all false-positive feature detections. More specifically, this module removes features detected whose feature strength is not a local maxima within a specified $N \times N$ sized window.

Given a minimum distance R_{NM} desired between detected features, the window size can be found.

$$S_{\text{NM}} = 2R_{\text{NM}} + 1$$

The design for this module utilizes a modified sliding window architecture in which window values are compared and edited to perform the non-maximal suppression. As shown in the block diagram of the module in figure 13, the module receives a stream of feature weights corresponding to each pixel and outputs a z^{-n} delayed boolean flag to denote a feature. This module has been designed to support non-max suppression of arbitrary window sizes, however, a 7×7 window was found to be sufficient for most applications.

Through experimentation, it was found that the module could be implemented in a way that only relies on the knowledge and editing of current and past weight values, not future. This results in approximately half the window being required, saving resources, as weight values can enter the 'center' of the window. This is depicted in figure 13 where non-required window registers are faded.

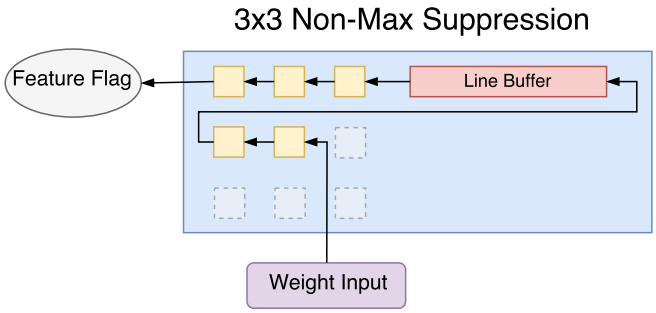


Figure 13: Non-maximal suppression module

This Non-Max suppression works by applying a simple rule to weights in the window at each clock cycle:

if *center weight* > *all other weights* **then**
  set all other weights to 0
else
  set center weight to 0
end

All weights that exit the last (top left) register are tested to determine if they are non-zero values and if so, the output feature flag is set to true.

Shown in figure 14 is the result of a Vivado simulation of this module using a 7×7 non-maximal suppression window. As can be seen in this figure, the non-max suppression has removed all but one detected feature. Additionally, the respective weights for the detected features are shown, where pixel intensity is indicative of feature strength.

The source code for this module is listed in appendix A7. Additionally, the source code for the line buffer used by this module is presented in appendix A8.

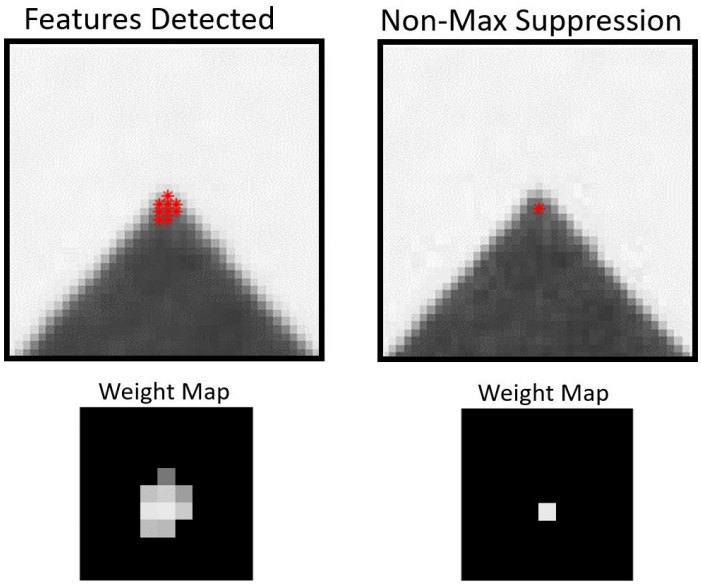


Figure 14: Non-max suppression results

3.5 Feature Descriptor

The binary descriptor module implemented utilises the BRIEF architecture (see section 2.3) and is based upon work of Azimi et al. (2017), where an efficiently pipelined version was implemented on an FPGA. Although BRIEF is highly effective, it requires a 33×33 feature window along with a 256-bit binary descriptor, consuming significant FPGA resources. In order to reduce this consumption, compromises were made to the sizes of both the feature window and descriptor.

The feature window was reduced to a size 9×9 (see fig. 15) as Brenot, Fillatreau & Piat, (2016) demonstrated that this size was still able to maintain acceptable tracking stability. Furthermore, the descriptor size was reduced to 128-bit as this only results in a feature matching performance degradation of $0\% - 10\%$, acceptable for this application (Calonder et al., 2010).

The resulting module is demonstrated in figure 17. This module receives a 9×9 pixel window and maps the pixels chosen for analysis into a vector. These chosen pixels, labelled in figure 15, were based on BRISK where an approximately radially symmetric pixel pattern was found to be markedly more robust than arbitrarily chosen pixels (Leutenegger et al., 2011). Furthermore, however outside of this project's scope, this pattern allows for the implementation of rotationally invariant descriptors.

From the vector of mapped pixels, 128 comparisons between elements within this vector are performed, producing a 128-bit descriptor of the feature (see section 2.3 of Literature Review). These vector element comparisons are arbitrarily chosen and the comparisons used for this descriptor are shown in figure 16 where the lines span between the pixels being compared. A basic example of the system's operation is shown in figure 18 where three arbitrary pairs of pixels are being compared, producing part of the feature descriptor.

The source code for this module can be found in appendix A10.

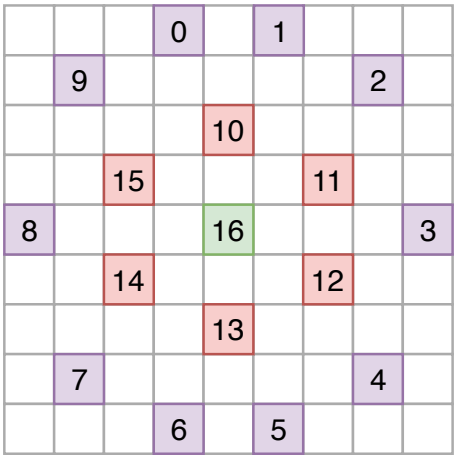


Figure 15: BRISK Based Feature Descriptor Template

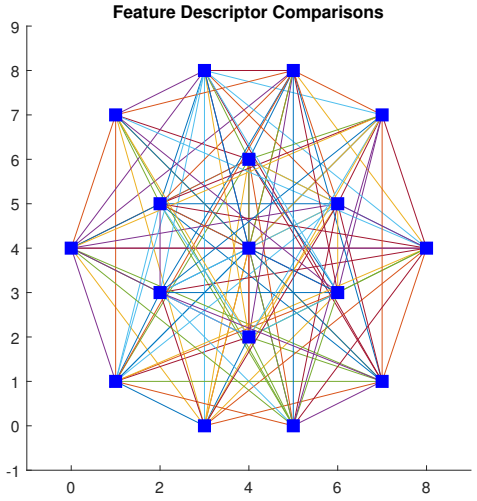


Figure 16: Feature Descriptor Comparisons

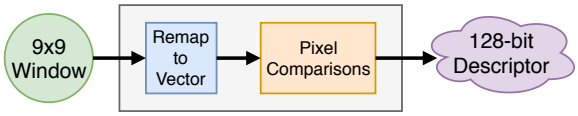


Figure 17: Feature Descriptor Module

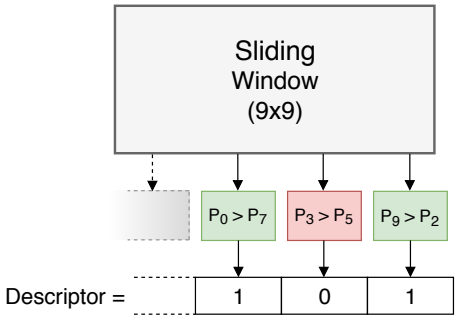


Figure 18: Feature Descriptor Process

3.6 Descriptor Comparator

The descriptor comparator module is responsible for receiving two binary feature descriptors and quantifying how different they are to one another. This is achieved by computing the descriptors' Hamming Distance (see section 2.3), where the number of bits that differ between descriptors are counted. This module is used by the Feature Tracking module to determine if two features' descriptors are similar enough to be considered the same feature. As can be seen in Figure 19, computation of the Hamming Distance requires two main steps: An XOR of the descriptors, followed by the computation of the result's Hamming Weight, which involves counting the number of bits it has set.

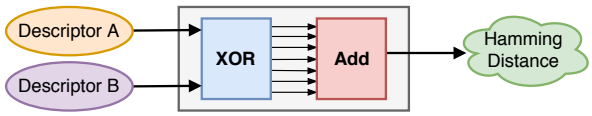


Figure 19: Descriptor Comparator Module

The logic used to determine the Hamming Weight is depicted in figure 20. Although adding together each bit within the 128-bit XOR result would successfully produce the Hamming Weight, this process can be more efficiently achieved with the use lookup tables (LUTs) and an adder tree. As most modern FPGAs contain 6-input LUTs, each group of 6 bits can be mapped to 3 LUTS, which together, will output the number of set bits in the group as a 3-bit number. These results are then accumulated to find the Hamming Weight of the XORed descriptors, resulting the the Hamming Distance. This is achieved through the use of a pipelined adder tree, significantly increasing the speed at which this module may run over a conventional N input adder. The depth, and consequently, delay of this adder tree is six clock cycles, as shown is equation 5.

This feature comparator code can be found in appendix A11.

$$\begin{aligned} D &= \left\lfloor \log_2 \left(\left\lceil \frac{W_{\text{Descriptor}}}{W_{\text{LUT}}} \right\rceil \right) \right\rfloor \\ &= \left\lfloor \log_2 \left(\lceil \tfrac{128}{6} \rceil \right) \right\rfloor \\ &= 6 \end{aligned} \tag{5}$$

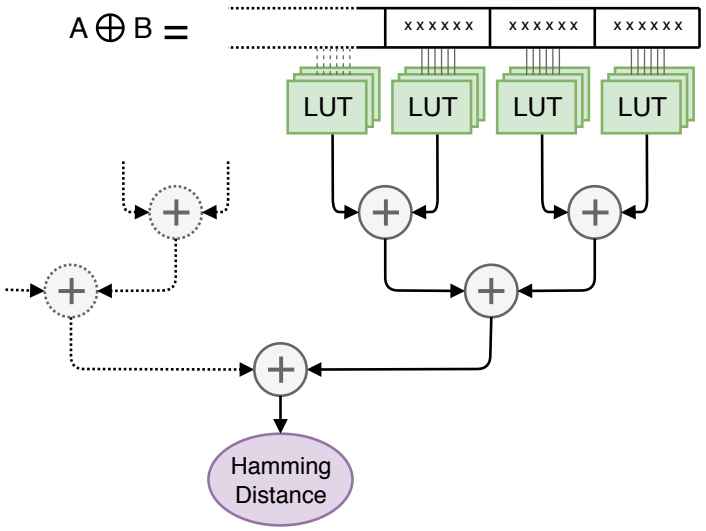


Figure 20: Hamming Weight Adder Tree

3.7 Feature Location Computer

In order to track the movements of features throughout a video, the locations of all detected features are required to be known. To achieve this, a feature location module is implemented in parallel to the feature detector and is responsible for assigning $[x, y]$ locations to features exiting the detector module. These locations are found by determining which pixel indices each detected feature falls on within it's respective video frame. This module is illustrated blue in figure 21 where it is shown in relation to relevant components within the feature tracking system.

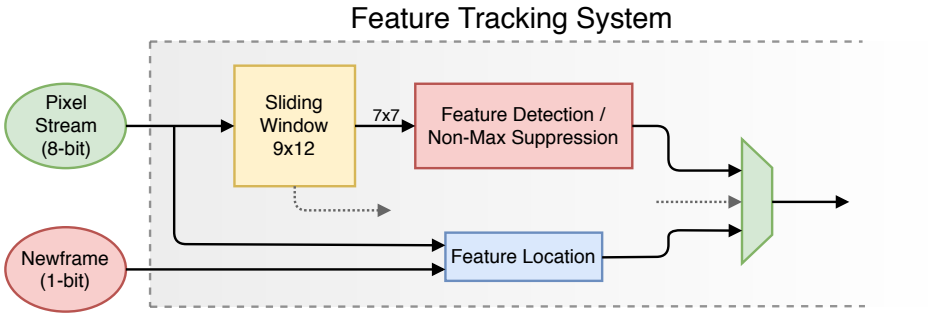


Figure 21: Feature Location Computer

Assigning these correct locations to features requires meticulous accounting of all the delays experienced, from when a pixel first enters the feature tracking system to when a feature exits the detector. This delay is modelled with the following equation

$$D = \frac{1}{2}(W+1)\cdot\left(W_{\mathrm{SW}} + S_{\mathrm{NM}} - 4\right) + P$$

where W_{SW} is the sliding window width, S_{NM}, the nom-maximal suppression window size, W, the video frame width, and P, the pipelining delay. With this model, this module is able to account for the feature detection delay and assign to features the exact position within the video frame they were detected. Code for this module can be found in appendix A9.

3.8 Matching Descriptor/Detector Output Delays

As both the feature detector and descriptor modules have very different output delays, an efficient method was developed to ensure the outputs from both modules were synchronised. Due to the simplicity of the feature descriptor module, it contains only a very small pipeline delay. However, as the feature detector module contains a non-maximal suppression sub-module, its output delay is approximately $WR_{NM}+P$, where W is the image width, R_{NM} is the non-maximal suppression radius (see section 3.4.2), and P is a small pipeline delay. Therefore, the descriptor's output is required to be delayed by approximately WR_{NM}, or R_{NM} image lines.

A standard FIFO delay buffer is completely impractical as the descriptor module's output is 128-bit, requiring more memory than is available on most mid-tier FPGAs. Instead, the delay was applied to the module's input by using a rectangular sliding window and configuring the descriptor to access pixels R_{NM} lines behind the feature detector (see fig. 22). Additionally, a small pipeline was added to the descriptor module's output to make up the small remaining pipeline difference. This implementation reduced the required memory by 16× over the conventional output delay buffer.

Source code for the small pipeline delay buffer is provided in appendix A12. The remaining components provide source code in their respective dedicated sections.

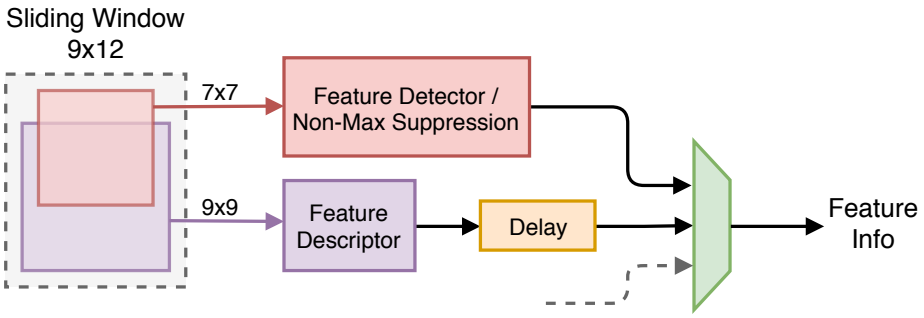


Figure 22: Descriptor delay design

3.9 Feature Tracking Module

The feature tracking module is responsible for tracking the movement of numerous features across multiple frames of a video. This module achieves this by receiving a stream of features from the current video frame and searching for each in an internally updated list of features found in past frames. If the same feature was found in a past frame, this module provides information as to tracked feature's current location, how far it has moved, and across how many frames of the video.

Section 3.9.1 provides a description of the design and implementation of this feature tracking module. Additionally, Section 3.9.2 provides a more efficient, space-saving design yet to see implementation.

3.9.1 Design and Implementation

Achieving this feature tracking capability requires three main sub-modules: Feature Library, Shift Register, and Processor (see fig. 23). The Feature Library is responsible for holding a number of feature entries found in past frames of a video. The shift register provides a means of allowing each newly detected feature to be analysed against each element within the feature library. Finally, the processor sub-module is responsible for utilising the information stored in both the feature library and shift register entries in order to find matches as well as remove and update library entries.

Each newly detected feature is placed into the shift register where its descriptor is systematically, by means of the processors, compared to every feature within the library (see section 2.3 of Literature Review). If a match if found, the output buffer, and subsequently the module output, is provided with the $[x, y]$ locations of the matching features along with the number of video frames spanning between them.

Any feature found to be similar to one in the library, yet not similar enough to be considered a match, will be flagged as non-unique. Unique and unmatched features which exit the shift register are re-entered for the purpose of being assessed by the processors for potential placement within the library. If the processor's associated library entry is empty, any feature looking for library placement will automatically be accepted. However, if a stronger feature (see section 2.1 of Literature Review) from the same video frame comes along, the processor will swap out its feature with that of the stronger feature. This causes the weaker feature to be placed back into the shift register where it will continue to search for a position to be placed within the library. This allows library to maintain a list of only the strongest possible features found within a video frame.

Throughout a video, features of past frames may become undetectable, go out of frame, or possess descriptors which are unable to find matches. This causes the library to rapidly build up with stale features. This is solved by instructing the processors to remove their associated library entries if they are not matched to any feature within N consecutive frames of a video, making room for newer features. This removal mechanism in combination with the processor's similarity flagging and feature strength prioritisation process causes the library to be up-kept with only the strongest active and unique features available.

The code for this module can be found in appendix A13.

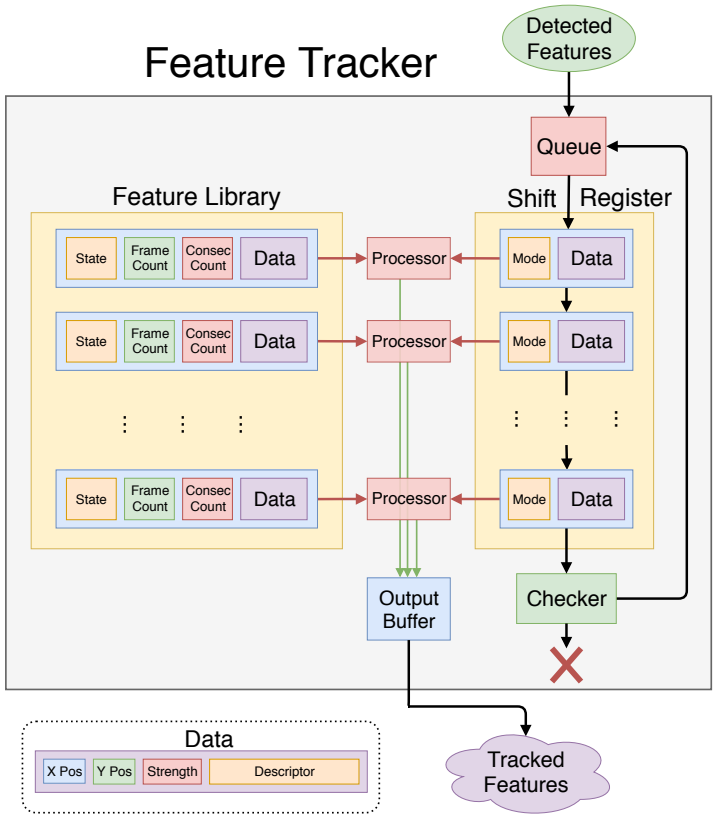


Figure 23: Feature tracking module

3.9.2 Future Space Saving Design

Although the basic design outlined in section 3.9.1 would be possible, it is an inefficient use of the FPGA resources. This is due to both the feature library and shift register being created completely out of LUTs and FFs, not components designed to store data such as BRAMs. The updated design is shown in figure 24 where the feature library has been hybridised. This new library utilises a FIFO for the majority of feature storage while simultaneously using a small number of registers to perform feature comparisons.

This design is possible due to the limited rate at which features enter the feature tracking module. Given a specified minimum allowable distance between detected features, R_{NM} (see section 3.4.2) the

most frequently a feature can enter the feature tracker and ultimately the shift register is once every R_{NM} clock cycles. Additionally, accounting for the shift register's dual purpose (see section 3.9.1), the maximum speed the shift register shifts is once every $R_{\text{NM}}/2$ clock cycles. Further accounting for the fact that this module is able to run at $2N$ times faster than the rest of the system (limited by the implementation), this leaves a number of clock cycles completely unutilised. Therefore, for every clock cycle the shift register is not shifting, the features within the improved feature library will be instead shifted, causing feature comparisons to now occur every clock cycle. This design allows for the number of registers required to be reduced by a factor of $NR_{\text{NM}}/2$ while still maintaining the full functionality previously outlined.

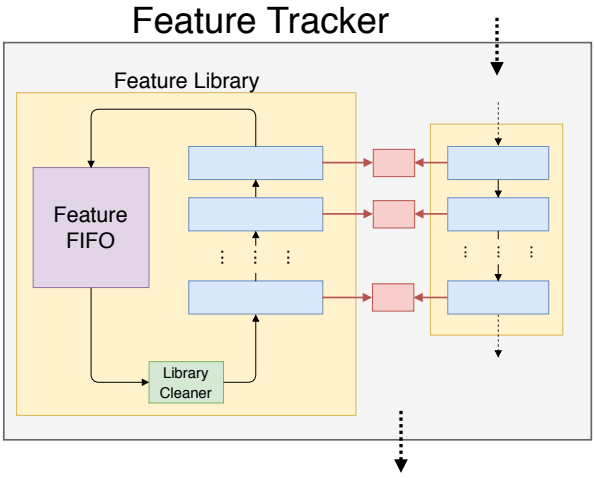


Figure 24: Improved feature tracking module

3.10 Feature Tracking System Results

Presented in this section is an analysis of the feature tracking system which has been implemented on an FPGA. Section 3.10.1 presents a simulation of the system, demonstration how it performs in a real feature tracking task, and identifying its current limitations. Additionally, section 3.10.2 reviews the performance of the system as well as its resource implementation, identifying areas for future improvement.

3.10.1 Feature Tracking Test

Presented below is the results of the feature tracking system, showing how 50 features are tracked across numerous frames of a video. These below figures were generated by superimposing the output $[n_f, x_s, y_s, x_e, y_e]$ feature matches (see section 3.1) over the video frames provided to the system. The image in the left of the figures is a larger reference image, allowing the true location of features within the library to be shown in a fixed position. The right side of the figure presents the current frame of a video, where the video is shown to be a small window, panning about the larger reference image.

Presented in figure 25 is the second frame of the video, where all 50 that were found in the first video frame are being tracked successfully.

Figure 25: Feature tracker results (frame 2)

It can be seen in figure 26 that as the video pans to the right, a number of features previously tracked in the left of the library (see figure 25) have disappeared. This is because the video has panned enough to cause these library features to go out of frame, preventing any matches from occurring. This has caused some of these features to be removed from the library as features which go unmatched for a configured number of consecutive frames (currently 10) are removed. As some of these features have been removed, the library has available space which allows for new features to be added, as shown in green.

Figure 26: Feature tracker results (frame 19)

This same process occurs below in figure 27, where as the video pans back left, new features, potentially ones they were previously deleted, have been added to the library and tracked.

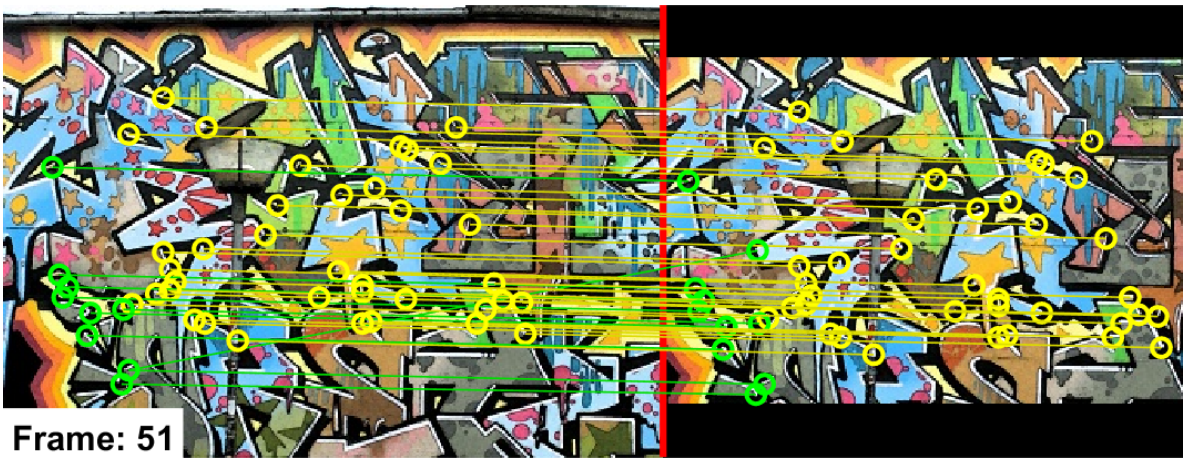


Figure 27: Feature tracker results (frame 51)

The results of this simulation show the feature tracker's effectiveness at tracking features throughout frames of a video. Further demonstrated was the feature tracker's ability to maintain an up-to-date list of powerful features, where stale features were removed and the newest, strongest features were added as the camera panned about the scene.

However, further testing has shown that as a camera performs other movements such as rotation or zooming, tracked features which are still in frame are unable to be found. This is due to the feature descriptor's current lack of scale and rotational invariance, as adding such exceeded the scope of this already large project.

3.10.2 Performance and Resource Utilisation

This system is currently capable of running at 120MHz before timing violations occur, allowing for the support of 1080p video at 60fps. Additionally, this allows 640x480 video to be run up to 300fps for high speed feature tracking applications. With further refinement of the pipelining currently implemented in the system's modules, this system is expected to be capable of 150MHz.

Shown below in figure 28 is the current utilisation of system resources on an Arty Artix-7 FPGA development board. The majority of the LUT (30%) and FF (20%) resources are utilised by the feature comparator modules, responsible for determining if two features are the same. This is because one is required for every feature within the library, allowing the $\mathcal{O}(n^2)$ feature tracking complexity to be linearised. This implementation size could be drastically reduced by running the feature tracking module at $2n$ system speed (limited by timing constraints) and decreasing the comparators used by a factor of $2n$, maintaining the same data throughput. Alternatively, implementing the improved feature tracking design (see section 3.9.2) would also alleviate this issue.

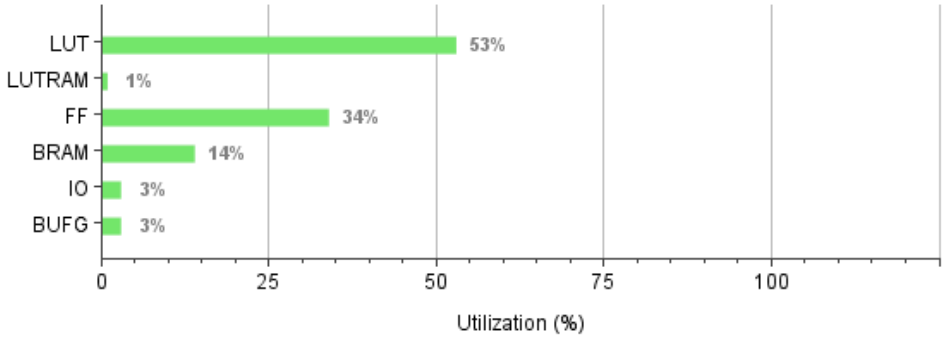


Figure 28: FPGA resource utilisation

3.11 Camera Choice and Interface Design

As this project aims to provide real-time feature detection, interfacing to a camera with a live video output stream is required. Two main output types exist, serial and parallel. A serial camera interface is relatively simple to implement however, it typically requires clock speeds of over 500Mhz for 720p video. Conversely, parallel camera connections require approximately a 12-wire data interface and for 720p video require clock speeds of less than 100Mhz. Therefore, the FPGA implementation was designed to support a camera with a paralell interface as its lower clock speeds will allow for implementation on lower-end FPGAs and the several wire data interface is no issue as FPGAs generally have high IO counts.

The two cameras considered for this design are the OV7670 and OV5640. The OV7670 is under $10 and has a sensor size of 640×480 capable of 60fps. The OV5640 is approximately $40 with a 1440p sensor capable of 30fps 1440p, 60fps 720p and 120fps 640×480.

As these cameras contain identical IO (except 2 more LSBs for OV5460) the interface designed supports both cameras. However, the register settings required for each camera to work is significantly different. As the OV7670 has more supporting documentation, the implementation has been designed to support this. However in future, the OV5640 will be supported as the higher resolution and lower noise profile will provide substantially better feature detection performance.

In order to transfer the camera pixel bitstream onto the FPGA, a camera interface module was designed and implemented. This module consists of four main components: clock generator, shift register, buffer, and pixel indexer.

Clock Generator: Feeds the FPGA's core clock into a clock manager module and outputs the specific clock frequency required of the camera (varies for FPS and resolution).

Shift Register: As the camera transmits pixel data (RGB565) across 2 clock cycles (8 MSBs then 8 LSBs), a shift register was implemented to reconstruct the pixel data as it was received. Every second clock cycle the contents of this register is passed to the buffer.

Buffer: Because the camera utilizes its own timing prescalers and PLLs, the data received from the camera is not guaranteed to be in phase with the FPGA's clock. This required a FIFO style buffer to be implemented that allows the data to cross the camera/FPGA clock domains without timing violations.

Pixel Indexer: This component tracks the cameras VGA style vertical/horizontal sync signals to track the x/y index of the current pixel being stored in the shift register. These indices will be used in future work to aid in the tracking of features.

4 Conclusion and Further Work

The goal for this project was to design and implement a full feature tracking system, capable of detecting features and tracking their movements across frames of a video. This required significant research and development of numerous components, including a feature detector, descriptor, comparator, and tracker. This posed a variety of research questions regarding how all of these components are able to be implemented on an FPGA in a compact and efficient manner.

Firstly, I conducted feature detection research which resulted in my implementation of the FAST detector, a very effective, low resource feature detection method. Feature description methodologies were then researched which led to the discovery and subsequent implementation of a minimised BRISK binary descriptor, whose implementation size and speed was found to be exceptional, compared to more common floating-point methods. A feature tracking module was designed and implemented from the ground up, allowing the $\mathcal{O}(n^2)$ complexity problem to be fully pipelined with acceptable use of system resources. The successful resolution of my research questions resulted in a fast and effective feature tracking system being implemented on an FPGA. A Gantt chart outlining my progression throughout this project, with respect to the system's individual components, is presented in Appendix A2.

Due to time constraints and the scope of this project, a number of improvements have been identified which would significantly improve the systems size and performance. Adding scale and rotational invariance to the feature descriptor would make the system robust to camera zoom, forward/backwards movement, and rotation, something of which is common in a real-world scenario. Currently, the system's clock is tied to that of the incoming data's pixel clock. Decoupling these clocks would allow for components such as the feature tracker run at higher speeds, drastically reducing the implementation size. Additionally, this high speed would lend well to the future implementation of the designed 'improved feature tracker' which reduces the footprint of the currently implemented module.

5 References

Alahi, A., Ortiz, R., & Vandergheynst, P. (2012). FREAK: Fast Retina Keypoint. 2012 IEEE Conference On Computer Vision And Pattern Recognition. doi:10.1109/cvpr.2012.6247715

Alahi, A., Vandergheynst, P., Bierlaire, M., & Kunt, M. (2010). Cascade of descriptors to detect and track objects across any network of cameras. Computer Vision And Image Understanding, 114(6), 624-640. doi:10.1016/j.cviu.2010.01.004

Azimi, E., Behrad, A., Ghaznavi-Ghoushchi, M., & Shanbehzadeh, J. (2017). A fully pipelined and parallel hardware architecture for real-time BRISK salient point extraction. Journal Of Real-Time Image Processing. doi:10.1007/s11554-017-0693-4

Bailey, D. (2011). Design for embedded image processing on FPGAs. Singapore: Wiley.

Banks, J. (2017). Report on How to Write Reports. QUT

Brenot, F., Fillatreau, P., & Piat, J. (2015). FPGA based accelerator for visual features detection. 2015 IEEE International Workshop Of Electronics, Control, Measurement, Signals And Their Application To Mechatronics (ECMSM). doi:10.1109/ecmsm.2015.7208697

Brenot, F., Fillatreau, P., & Piat, J. (2016). FPGA based hardware acceleration of a BRIEF correlator module for a monocular SLAM application. Proceedings Of The 10Th International Conference On Distributed Smart Camera - ICDSC '16. doi:10.1145/2967413.2967426

Calonder, M., Lepetit, V., Strecha, C., & Fua, P. (2010). BRIEF: Binary Robust Independent Elementary Features. Computer Vision ECCV 2010, 778-792. doi:10.1007/978-3-642-15561-1_56

Canclini, A., Cesana, M., Redondi, A., Tagliasacchi, M., Ascenso, J., & Cilla, R. (2013). Evaluation of low-complexity visual feature detectors and descriptors. 2013 18th International Conference on Digital Signal Processing (DSP). doi:10.1109/icdsp.2013.6622757

Chiu, L., Chang, T., Chen, J., & Chang, N. (2013). Fast SIFT Design for Real-Time Visual Feature Extraction. IEEE Transactions On Image Processing, 22(8), 3158-3167. doi:10.1109/tip.2013.2259841

de Lima, R., Martinez-Carranza, J., Morales-Reyes, A., & Cumplido, R. (2015). Accelerating the construction of BRIEF descriptors using an FPGA-based architecture. 2015 International Conference On Reconfigurable Computing And Fpgas (Reconfig). doi:10.1109/reconfig.2015.7393285

Fischer, J., Ruppel, A., Weisshardt, F., & Verl, A. (2011). A rotation invariant feature descriptor O-DAISY and its FPGA implementation. 2011 IEEE/RSJ International Conference On Intelligent Robots And Systems. doi:10.1109/iros.2011.6094813

Fularz, M., Kraft, M., Schmidt, A., & Kasiski, A. (2015). A High-Performance FPGA-Based Image Feature Detector and Matcher Based on the FAST and BRIEF Algorithms. International Journal Of Advanced Robotic Systems, 12(10), 141. doi:10.5772/61434

Gauglitz, S., Hllerer, T., & Turk, M. (2011). Evaluation of Interest Point Detectors and Feature Descriptors for Visual Tracking. International Journal Of Computer Vision, 94(3), 335-360. doi:10.1007/s11263-011-0431-5

Harris, C. & Stephens, M. (1988). A Combined Corner and Edge Detector. Proceedings of the 4th Alvey Vision Conference, 147-151.

Leutenegger, S., Chli, M., & Siegwart, R. (2011). BRISK: Binary Robust invariant scalable keypoints. 2011 International Conference On Computer Vision, 2548-2555. doi:10.1109/iccv.2011.6126542

Li, Y., Wang, S., Tian, Q., & Ding, X. (2015). A survey of recent advances in visual feature detection. Neurocomputing, 149, 736-751. doi:10.1016/j.neucom.2014.08.003

Mikolajczyk, K. (2004). Scale & Affine Invariant Interest Point Detectors. International Journal Of Computer Vision, 60(1), 63-86. doi:10.1023/b:visi.0000027790.02288.f2

Possa, P., Mahmoudi, S., Harb, N., Valderrama, C., & Manneback, P. (2014). A Multi-Resolution FPGA-Based Architecture for Real-Time Edge and Corner Detection. IEEE Transactions On Computers, 63(10), 2376-2388. doi:10.1109/tc.2013.130

Rosten, E., & Drummond, T. (2006). Machine Learning for High-Speed Corner Detection. Computer Vision ECCV 2006, 430-443. doi:10.1007/11744023_34

Rosten, E., Porter, R., & Drummond, T. (2010). Faster and Better: A Machine Learning Approach to Corner Detection. IEEE Transactions On Pattern Analysis And Machine Intelligence, 32(1), 105-119. doi:10.1109/tpami.2008.275

Schulz, V., Bombardelli, F., & Todt, E. (2016). A Harris Corner Detector Implementation in SoC-FPGA for Visual SLAM. Communications In Computer And Information Science, 57-71. doi:10.1007/978-3-319-47247-8_4

Szeliski, R. (2011). Computer vision: Algorithms and Applications. New York: Springer.

Ulusel, O., Picardo, C., Harris, C., Reda, S., & Bahar, R. (2016). Hardware acceleration of feature detection and description algorithms on low-power embedded platforms. 2016 26Th International Conference On Field Programmable Logic And Applications (FPL). doi:10.1109/fpl.2016.7577310

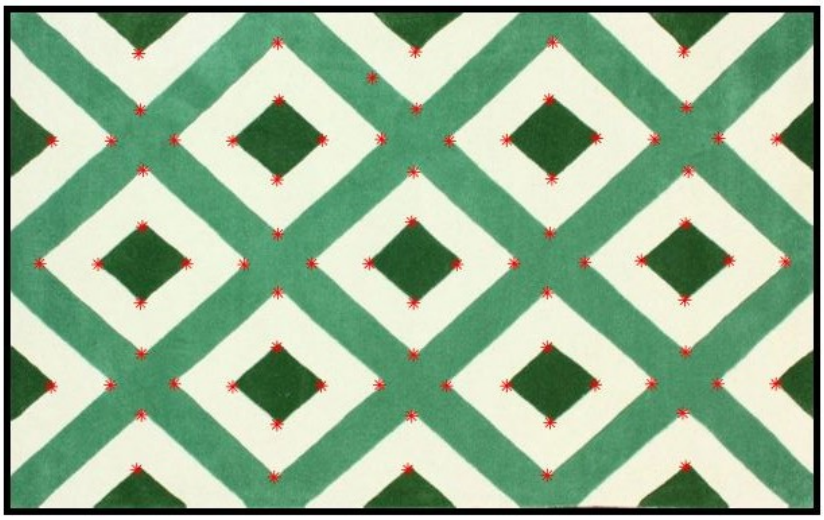
Wang, J., Zhong, S., Yan, L., & Cao, Z. (2014). An Embedded System-on-Chip Architecture for Real-time Visual Detection and Matching. IEEE Transactions On Circuits And Systems For Video Technology, 24(3), 525-538. doi:10.1109/tcsvt.2013.2280040

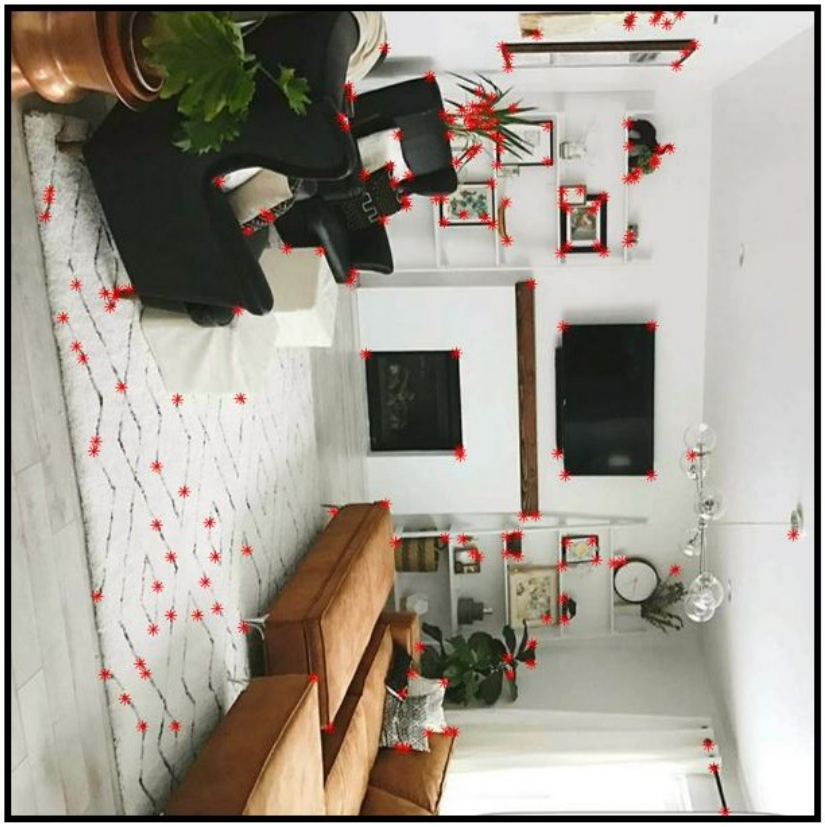
Xilinx. (2017). Vivado Design Suite User Guide. Retrieved from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug901-vivado-synthesis.pdf

Zhu, W., Liu, L., Jiang, G., Yin, S., & Wei, S. (2016). A 135-frames/s 1080p 87.5-mW Binary-Descriptor-Based Image Feature Extraction Accelerator. IEEE Transactions On Circuits And Systems For Video Technology, 26(8), 1532-1543. doi:10.1109/tcsvt.2015.2469116

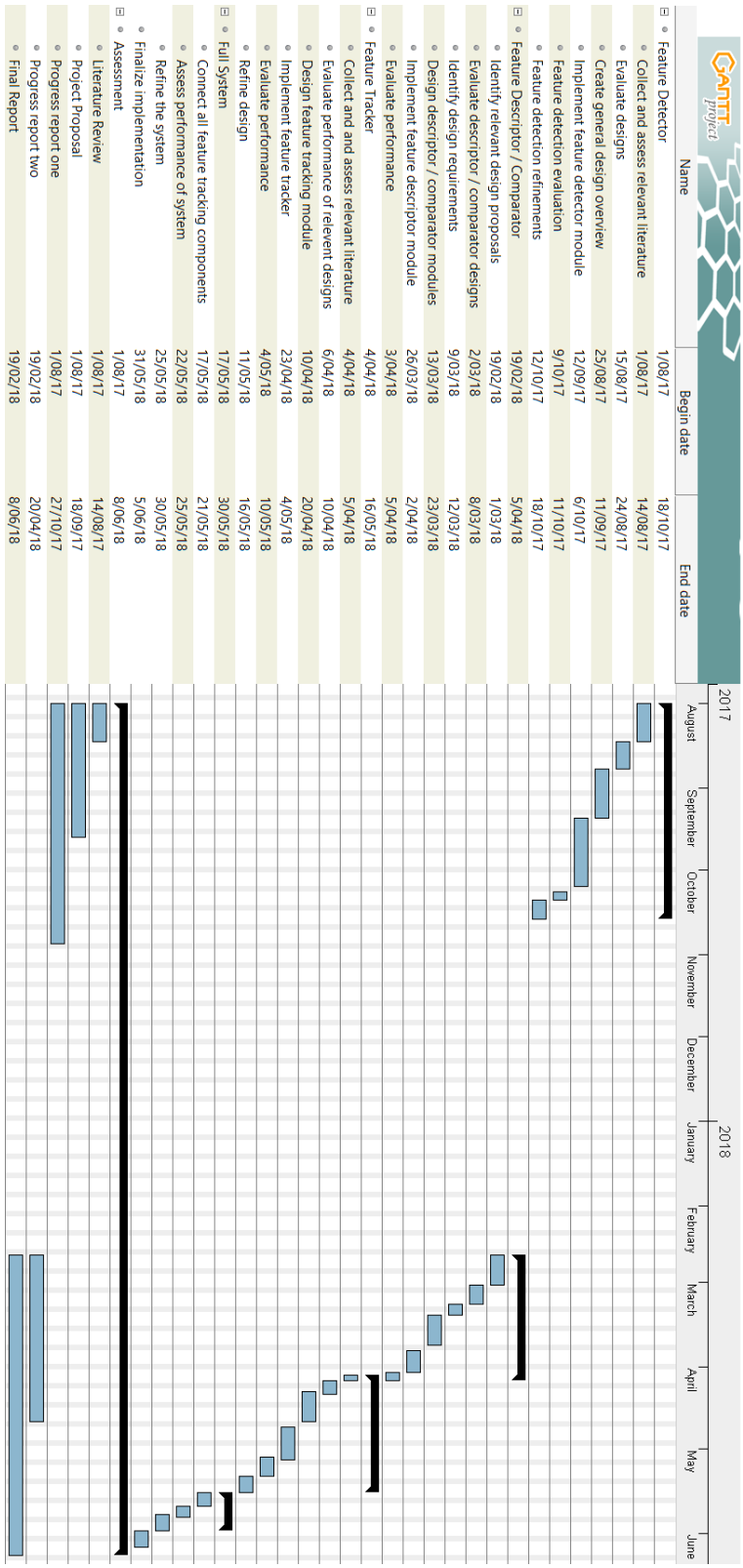
Appendix

A1 Feature Detection Results





A2 Project Gantt Chart



A3 Simulation Code

```
`timescale 1ns / 1ps

/* This module is used to feed an 8-bit pixel stream in from a source text file and output tracked features
in the format [frame_number, n_frames, x_start, y_start, x_end, y_end], to a destination text file */

module sim_top();

    reg clk;
    reg [15:0] temp_input;
    reg [15:0] data_input;
    reg new_frame;

    wire feature_flag;
    wire [11:0] feature_x;
    wire [11:0] feature_y;
    wire [127:0] descriptor;
    wire [7:0] feature_strength;
    wire delayed_new_frame;
    integer frame_counter = 0;

    wire match;
    wire [11:0] xs;
    wire [11:0] ys;
    wire [11:0] xe;
    wire [11:0] ye;
    wire [9:0] span;

    integer f_in, f_features, f_matches;
    integer count = 0;

    /* Run sim at 100MHz */
    always #10 clk = !clk;


    /* Function to end the simulation */
    task stopsim();
        begin
            $fclose(f_in);
            $fclose(f_features);
            $fclose(f_matches);
            $display("Stopping Simulation");
            $stop;
        end
    endtask


    /* Reads in new data every clock */
    always@(posedge clk) begin
        // Read in new data, either pixel value or new frame flag
        count = $fscanf(f_in, "%d", temp_input);
        // Set new frame flag if new frame detected
        new_frame = temp_input == 16'hFFFF;
```

```
53          // Read in another value if last one was the new frame flag
54          if(temp_input == 16'hFFFF) begin
55              count = $fscanf(f_in, "%d", temp_input);
56          end
57          // Assign input data
58          data_input = temp_input;
59          // Stop sim if end of file
60          if(count==-1) stopsim();
61      end
62
63
64      /* Write appropritate data to file */
65      always@(posedge clk) begin
66          /* Keep track of current frame for exiting features */
67          if(delayed_new_frame) begin
68              frame_counter = frame_counter + 1;
69          end
70          /* Print out the features to the feature file */
71          if(feature_flag) begin
72              // The descriptor is broken up into two 64-bit values
73              $fwrite(f_features, "%d,␣%d,␣%d,␣%d,␣%d,␣%d,␣%d,␣%d\n",
                    frame_counter, feature_x, feature_y, feature_strength,
74                  descriptor[127:96], descriptor[95:64], descriptor[63:32],
                        descriptor[31:0]);
75              //$fwrite(f_features, "%d, %d, %d, %x\n", frame_counter, feature_x
                    , feature_y, descriptor);
76          end
77          /* Print out the match to the match file */
78          if(match) begin
79              $fwrite(f_matches, "%d,␣%d,␣%d,␣%d,␣%d,␣%d\n", frame_counter, span
                    , xs, ys, xe, ye);
80          end
81      end
82
83
84      /* Prepare input/output files */
85      initial begin
86          clk = 1'b0;
87          temp_input = 16'b0;
88          data_input = 16'b0;
89          new_frame = 1'b0;
90
91          f_in = $fopen("input_data.txt", "r");
92          f_features = $fopen("feature_data.txt", "w");
93          f_matches = $fopen("matches_data.txt", "w");
94
95          $display("Starting␣Simulation");
96      end
97
98
99      system #(.IM_WIDTH(640), .IM_HEIGHT(480), .LE(50), .MATCH_THRESHOLD(15), .
            SIMILAR_THRESHOLD(15))
100     system1(
101         .clk(clk),
102         .data_in(data_input[7:0]),
103         .new_frame(new_frame),
```

```
104
105         .feature_flag(feature_flag),
106         .feature_x(feature_x),
107         .feature_y(feature_y),
108         .feature_strength(feature_strength),
109         .descriptor(descriptor),
110         .delayed_new_frame(delayed_new_frame),
111
112         .match_xs(xs), .match_ys(ys), .match_xe(xe), .match_ye(ye), .
                match_flag(match), .match_span(span)
113     );
114
115 endmodule
```

A4 Top Level Processor Module

```
`timescale 1ns / 1ps

module system #(
    parameter IM_WIDTH = 640,           // Width of the image
    parameter IM_HEIGHT = 480,          // Height of the image
    parameter LE = 20,                  // How many features to track
    parameter MATCH_THRESHOLD = 15,     // Mat dist between features to be
        concidered a match
    parameter SIMILAR_THRESHOLD = 20,   // Max dist between features
        concidered similar
    localparam IND_WIDTH = 12,          // Width of the pixel index registers
    localparam BW = 8,                  // Bit width of data
    localparam DW = 128,                // Descriptor width
    localparam MFW = 08,                // Matcher - Missed frames counter
        width
    localparam FCW = 10                 // Matcher - Frame counter width
    )(
    input clk,
    input [BW-1:0] data_in,
    input new_frame,

    // Detected features debug output
    output reg feature_flag,
    output reg [IND_WIDTH-1:0] feature_x,
    output reg [IND_WIDTH-1:0] feature_y,
    output reg [BW-1:0] feature_strength,
    output reg [DW-1:0] descriptor,

    // Matched features output
    output wire match_flag,
    output wire [IND_WIDTH-1:0] match_xs,
    output wire [IND_WIDTH-1:0] match_ys,
    output wire [IND_WIDTH-1:0] match_xe,
    output wire [IND_WIDTH-1:0] match_ye,
    output wire [FCW-1:0] match_span,

    // Used for dubugging with frame numbers
    output wire delayed_new_frame
    );

    localparam NMR = 3;                 // Radius of non-maxima supression
    localparam WS = 9;                  // Width of the sliding window
    localparam HS = WS + NMR;           // Height of the sliding window
    localparam RS = (WS-1)/2;           // Radius of the sliding window
    localparam THRESHOLD = 50;          // Bresenham threshold

    // Clock cycles taken for input pixel to reach feature output (delay ~=~ (
        W+1)/2 (S_sw + S_nm - 4) + P) , P approx 7
    localparam PIPELINE_DELAY = 7;
    localparam PROP_DELAY = (IM_WIDTH + 1) * (WS + (2 * NMR + 1) - 2) / 2 +
        PIPELINE_DELAY;

    // Clock cycles to delay descriptor by to match the non-max output flag
    localparam D_PROP_DELAY = 9;
```

```
50
51      /* Feeds signals between modules */
52      wire [BW-1:0] feature_strength_raw;
53      wire [HS-1:0] [WS-1:0] [BW-1:0] px;
54      wire [6:0] [6:0] [BW-1:0] fast_px;
55      wire [WS-1:0] [WS-1:0] [BW-1:0] desc_px;
56      wire [DW-1:0] descriptor_initial;
57
58      /* Feeds features to the edge rejector */
59      wire [IND_WIDTH-1:0] feature_x_prelim;
60      wire [IND_WIDTH-1:0] feature_y_prelim;
61      wire [BW-1:0] feature_strength_prelim;
62      wire [DW-1:0] descriptor_prelim;
63      wire feature_flag_prelim;
64
65      /* Assign the bottom part of the sliding window to the feature descriptor
            */
66      assign desc_px[WS-1:0] = px[HS-1:NMR];
67
68      /* Assigning the top centre 7x7 of the sliding window to wires fed into
            the FAST detector */
69      // Parameters used to locate the 7x7 FAST window within the sliding window
70      localparam fw_min = RS - 3;
71      localparam fw_max = RS + 3;
72      generate
73          for(genvar i=0; i<7; i=i+1) begin
74              assign fast_px[i][6:0] = px[fw_min+i][fw_max:fw_min];
75          end
76      endgenerate
77
78      /* Creates a 7x7 sliding window to hold pixels for bresenham feature
            detection */
79      sliding_window #(.W(WS), .H(HS), .BW(BW), .IM_WIDTH(IM_WIDTH))
80          sliding_window_1 (.clk(clk), .data_in(data_in), .px_window(px));
81
82      /* Creates the fast feature detector which inputs an NxN window and
            outputs a feature weight */
83      fast_detector #(.THRESHOLD(THRESHOLD), .BW(BW), .IM_WIDTH(IM_WIDTH))
84          fast_detector_1 (.clk(clk), .px_window(fast_px), .feature_strength(
                feature_strength_raw));
85
86      /* NxN winow used to remove features whose sum of absolute differences are
             not local maximas */
87      nonmax_suppression #(.WR(NMR), .BW(BW), .IM_WIDTH(IM_WIDTH))
88          nonmax_suppression_1(.clk(clk), .strength_in(feature_strength_raw),
89          .feature_flag(feature_flag_prelim), .strength_out(
                feature_strength_prelim));
90
91      /* Feature descriptor which takes a 9x9 sliding window and returns a 128-
            bit binary descriptor */
92      feature_descriptor #(.BW(BW)) feature_descriptor_1(.clk(clk), .px_window(
            desc_px), .descriptor(descriptor_initial));
93
94      /* Delays the descriptor to match the output from the nonmax_suppression
            unit */
95      delay_buffer #(.WIDTH(DW), .DELAY(D_PROP_DELAY))
```

```
96          descriptor_delay(.clk(clk), .data_in(descriptor_initial), .data_out(
                descriptor_prelim));
97
98      /* Determines the index of the pixels exiting the nonmax_suppression
            module */
99      pixel_indexer #(.IM_WIDTH(IM_WIDTH), .PROP_DELAY(PROP_DELAY), .IND_WIDTH(
            IND_WIDTH))
100         pixel_indexer_1(.clk(clk), .new_frame(new_frame), .ind_x(
                feature_x_prelim),
101         .ind_y(feature_y_prelim), .delayed_new_frame(delayed_new_frame));
102
103     /* Rejects features that have been detected too close to the edge of the
            frame */
104     edge_rejector #(.WIDTH(IM_WIDTH), .HEIGHT(IM_HEIGHT), .BW(BW), .DW(DW), .
            IND_W(IND_WIDTH), .RADIUS(RS))
105         edge_rejector_1(.clk(clk), .detected_in(feature_flag_prelim), .
                strength_in(feature_strength_prelim),
106         .descriptor_in(descriptor_prelim), .x_in(feature_x_prelim), .y_in(
                feature_y_prelim), .detected_out(feature_flag),
107         .strength_out(feature_strength), .descriptor_out(descriptor), .x_out(
                feature_x), .y_out(feature_y));
108
109     /* Keeps track of quality features and outputs their movement */
110     feature_matcher #(.DW(DW), .PW(IND_WIDTH), .LE(LE), .STRW(BW), .FCW(FCW),
            .MFW(MFW), .MATCH_THRESHOLD(MATCH_THRESHOLD), .SIMILAR_THRESHOLD(
            SIMILAR_THRESHOLD))
111         feature_matcher_1(.clk(clk), .new_frame(new_frame), .feature_flag(
                feature_flag), .feature_d(descriptor),
112         .feature_x(feature_x), .feature_y(feature_y), .feature_s(
                feature_strength),
113         .match_xs(match_xs), .match_ys(match_ys), .match_xe(match_xe), .
                match_ye(match_ye), .match_flag(match_flag), .match_span(
                match_span));
114
115 endmodule
```

A5 Sliding Window

```
`timescale 1ns / 1ps

/* An BW-bit WxH sliding window based on an input pixel stream */
module sliding_window
    #(
        parameter W = 9,            // Window width
        parameter H = 9,            // Window height
        parameter BW = 8,           // Bit Width of the values
        parameter IM_WIDTH = 640    // Width of line buffer
    )
    (
        input clk,
        input [BW-1:0] data_in,
        output reg [H-1:0] [W-1:0] [BW-1:0] px_window // WxH sliding window
    );

    /* Generic iterator */
    integer i;

    /* Connects line buffers together */
    wire [H-1:0] [BW-1:0] line_feed;

    /* Attach input to first line buffer in addition to window */
    assign line_feed[0] = data_in;


    /* Generates line buffers */
    genvar j;
    generate
        for(j=0; j<H-1; j=j+1) begin : gen_loop
            fifo_buffer #(.WIDTH(BW), .DEPTH(IM_WIDTH))
                fifo_x (.clk(clk), .data_in(line_feed[j]), .data_out(line_feed
                    [j+1]));
        end
    endgenerate


    /* Initialize window registers */
    initial begin
        for(i=0; i<H; i=i+1) begin
            px_window[i][W-1:0] = {W{{BW{1'b0}}}};
        end
    end


    /* Feeds line buffer values into sliding window */
    always @(posedge clk) begin
        for(i=0; i<H; i=i+1) begin
            px_window[i][W-1:0] <= {px_window[i][W-2:0], line_feed[i]};
        end
    end

endmodule
```

A6 FAST Detector

```
`timescale 1ns / 1ps

/* Fast detector with configurable data-width and threshold  */

module fast_detector #(
        parameter THRESHOLD = 50,   // Bresenham threshold
        parameter BW = 8,           // Pixel value Bit Width
        parameter IM_WIDTH = 640    // Width of image
    )(
        input clk,
        input wire [6:0] [6:0] [BW-1:0] px_window,
        output reg [BW-1:0] feature_strength = {BW{1'b0}}     // Sum of
            absolute differences of pixel of interest (0 if not feature)
    );

    integer k;

    reg [15:0] [BW-1:0] bres_t = {16{{BW{1'b0}}}};  // Contains the absolute
        differences
    reg [15:0] sign_t = 16'b0;                      // Preserves the sign of
        the absolute differences
    reg [1:0] [15:0] bd = {2{16'b0}};               // Denotes 16 [bright,
        dark] pixels
    reg [1:0] contiguity = 2'b0;                    // Flags if pixels are
        contiguous [bright, dark]

    wire [15:0] [BW-1:0] px_circle;                 // Remaps the 7x7 window
        to the 16 bresenham pixels
    wire [BW-1:0] px_center;

    localparam SW = BW + 4;                         // Bits required to sum
        all pixel values
    reg [SW-1:0] weight_sum = {(SW){1'b0}};         // Measure of feature
        strength (for nonmax suppression)
    reg [BW-1:0] weight_sum_scaled = {BW{1'b0}};    // Scales the feature
        strength down


    /* Bresenham pixel circle */
    assign px_circle[15:0] = {
        px_window[0][3], px_window[0][4], px_window[1][5], px_window[2][6],
        px_window[3][6], px_window[4][6], px_window[5][5], px_window[6][4],
        px_window[6][3], px_window[6][2], px_window[5][1], px_window[4][0],
        px_window[3][0], px_window[2][0], px_window[1][1], px_window[0][2]};


    /* Center pixel (pixel of interest) in the 7x7 window */
    assign px_center = px_window[3][3];


    always @(posedge clk) begin

        /* Iterates through all Bresenham pixels */
        for(k=0; k<16; k=k+1) begin
```

```
46
47              /* Computes the absolute difference between bresenham and center
48                  and preserves the 'sign' of the result */
49              if(px_circle[k] > px_center) begin
50                  sign_t[k] <= 1'b1;
51                  bres_t[k] <= px_circle[k] - px_center;
52              end else begin
53                  sign_t[k] <= 1'b0;
54                  bres_t[k] <= px_center - px_circle[k];
55              end
56
57              /* Sets the [bright, dark] bits to denote a threshold has passed
                    */
58              bd[1][k] <= (bres_t[k] > THRESHOLD) && sign_t[k];
59              bd[0][k] <= (bres_t[k] > THRESHOLD) && !sign_t[k];
60          end
61
62      end
63
64
65      always @(posedge clk) begin
66          /* Determines if 9 contiguous bright/dark pixels exist */
67          for(k=0; k<2; k=k+1) begin
68              contiguity[k] <= !(
69                  (~bd[k][15:07]) && (~bd[k][14:06]) && (~bd[k][13:05]) && (~bd[
                        k][12:04]) &&
70                  (~bd[k][11:03]) && (~bd[k][10:02]) && (~bd[k][09:01]) && (~bd[
                        k][08:00]) &&
71                  (~{bd[k][07:00], bd[k][15:15]}) && (~{bd[k][06:00], bd[k
                        ][15:14]}) &&
72                  (~{bd[k][05:00], bd[k][15:13]}) && (~{bd[k][04:00], bd[k
                        ][15:12]}) &&
73                  (~{bd[k][03:00], bd[k][15:11]}) && (~{bd[k][02:00], bd[k
                        ][15:10]}) &&
74                  (~{bd[k][01:00], bd[k][15:09]}) && (~{bd[k][00:00], bd[k
                        ][15:08]}));
75          end
76
77          /* The sum of absolute differences of the bresenham pixels */
78          weight_sum[SW-1:0] <= (
79              bres_t[00] + bres_t[01] + bres_t[02] + bres_t[03] +
80              bres_t[04] + bres_t[05] + bres_t[06] + bres_t[07] +
81              bres_t[08] + bres_t[09] + bres_t[10] + bres_t[11] +
82              bres_t[12] + bres_t[13] + bres_t[14] + bres_t[15] );
83
84          /* Weight sum converted to scaled down to represent the average
                difference */
85          /* Note - (>>4) is equivalent to (/16) */
86          weight_sum_scaled[BW-1:0] <= weight_sum[SW-1:0] >> 4;
87
88          /* Only applies weights to that which are features */
89          feature_strength[BW-1:0] <= contiguity ? weight_sum_scaled[BW-1:0] : {
                BW{1'b0}};
90      end
91
92  endmodule
```

A7 Non-Max Suppression

```
`timescale 1ns / 1ps

/* Performs non-maximal suppression of features based on their associated
    strenghts  */

module nonmax_suppression
    #(
        parameter WR = 3,           // 'Radius' of the square non-max
            supressor window
        parameter BW = 8,           // Bit Width of the feature strengths
        parameter IM_WIDTH = 640    // Width of the image
    )(
        input clk,
        input wire [BW-1:0] strength_in,
        output reg [BW-1:0] strength_out,
        output reg feature_flag
    );

    integer i;
    genvar j, k;

    /* Size of the (WS x Ws) non-maxima window */
    localparam WS = 2*WR + 1;

    /* First row of the window (half+1) */
    reg [WR:0] [BW-1:0] px_line;
    /* Top R rows of the window */
    reg [WR-1:0] [WS-1:0] [BW-1:0] px_window;

    /* Connects line buffers together */
    wire [WR-1:0] [BW-1:0] line_feed_in;
    wire [WR-1:0] [BW-1:0] line_feed_out;

    /* Used for input/window maxima logic */
    wire [WR-1:0] [WS-2:0] maxima_window;
    wire [WR-1:0] maxima_line;
    wire [WR-1:0] maxima_feed;

    /* Feeds the end of the first line into a line buffer */
    assign line_feed_in[0] = px_line[WR];


    generate
        /* Assigns the line buffer input as the last element of each window
            row */
        for(j=0; j<WR-1; j=j+1) begin : wire_gen_loop
            assign line_feed_in[j+1] = px_window[j][WS-1];
        end

        /* Generates line buffers */
        for(j=0; j<WR; j=j+1) begin : gen_loop
            fifo_buffer #(.WIDTH(BW), .DEPTH(IM_WIDTH-WS)) fifo_x (.clk(clk),
                .data_in(line_feed_in[j]), .data_out(line_feed_out[j]));
        end
```

```
51
52          /* Generates all wires required for maxima comparison */
53          for(j=0; j<WR; j=j+1) begin : maxima_gen_loop
54              assign maxima_line[j] = strength_in <= px_line[j];
55              assign maxima_feed[j] = strength_in <= line_feed_out[j];
56
57              for(k=0; k<WS-1; k=k+1) begin
58                  assign maxima_window[j][k] = strength_in <= px_window[j][k];
59              end
60          end
61      endgenerate
62
63      /* Initialize output registers */
64      initial begin
65          feature_flag = 1'b0;
66          strength_out[BW-1:0] = {BW{1'b0}};
67      end
68
69      /* Initialize window registers */
70      initial begin
71         px_line[WR:0] = {WR+1{{BW{1'b0}}}};
72
73          for(i=0; i<WR; i=i+1) begin
74              px_window[i][WS-1:0] = {WS{{BW{1'b0}}}};
75          end
76      end
77
78
79      /* Features that reach the last register in the window are successful if
            they are a non-zero value */
80      always @(posedge clk) begin
81         feature_flag <= px_window[WR-1][WS-1] > 0 ? 1'b1 : 1'b0;
82         strength_out <= px_window[WR-1][WS-1];
83      end
84
85
86      /* Feeds line buffer values into sliding window */
87     always @(posedge clk) begin
88          /* If the maxima is not the new input value */
89          if(maxima_line || maxima_feed || maxima_window) begin
90              /* ----- Standard window shift but with the input as 0 ----- */
91              px_line[WR:0] <= {px_line[WR-1:0], {BW{1'b0}}};
92              for(i=0; i<WR; i=i+1) begin
93                  px_window[i][WS-1:0] <= {px_window[i][WS-2:0], line_feed_out[i
                        ]};
94              end
95          end else begin
96              /* ----- Set everything to 0 but the input weight ----- */
97              px_line[WR:0] <= {{WR{{BW{1'b0}}}}, strength_in};
98              for(i=0; i<WR; i=i+1) begin
99                  px_window[i][WS-1:0] <= {WS{{BW{1'b0}}}};
100             end
101         end
102     end
103
104 endmodule
```

A8 Line Buffer

```
`timescale 1ns / 1ps

 /* FIFO buffer which is inferred to a block ram (BRAM) */
 /* NOTE: Accessing addresses larger than the depth has unknown behaviour */

module fifo_buffer
    #(
    parameter WIDTH = 8,    // Width of values held in buffer
    parameter DEPTH = 640,  // Depth of the buffer
    parameter ADDR_W = 11   // Width (bits) required to hold address
    )
    (
    input clk,
    input [WIDTH-1:0] data_in,
    output reg [WIDTH-1:0] data_out = {WIDTH{1'b0}}
    );

    /* Current index of the buffer */
    reg [ADDR_W-1:0] addr_in  = {ADDR_W-1{1'b0}};
    reg [ADDR_W-1:0] addr_out = {{ADDR_W-2{1'b0}}, 1'b1};

    /* Buffer */
    reg [WIDTH-1:0] data [DEPTH-1:0]; //2**ADDR_W-1:0

    /* Clears the block ram */
    integer i;
    initial begin
        for(i=0; i<DEPTH; i=i+1) data[i] = {WIDTH{1'b0}};
    end

    /* Increments the address counters, resetting at DEPTH */
    always @(posedge clk) begin
        addr_in <= addr_out; // Follows the out addr with z^-1 delay
        addr_out <= addr_out==(DEPTH-1) ? 0 : addr_out+1;
    end

    always @(posedge clk) begin
        /* Stores input data */
        data[addr_in] <= data_in;
        /* Assigns output data */
        data_out <= data[addr_out];
    end

endmodule
```

A9 Feature Location Computer

```
`timescale 1ns / 1ps

/* Determines the (x,y) index of detected features */
/* WARNING - new_frames cannot occur faster than the propogation delay */

module pixel_indexer #(
        parameter IM_WIDTH = 640,   // Frame width
        parameter PROP_DELAY = 0,   // Number of clocks for input pixel to
            reach output
        parameter IND_WIDTH = 12,   // Number of bits in the pixel index x/y
            counters
        parameter COUNT_WIDTH = 16  // Number of bits in the delay counter
    )(
        input clk,
        input new_frame,
        output reg [IND_WIDTH-1:0] ind_x = {IND_WIDTH{1'b0}},
        output reg [IND_WIDTH-1:0] ind_y = {IND_WIDTH{1'b0}},
        output reg delayed_new_frame = 1'b0 // New frame flag delayed by the
            propogation delay
    );

    typedef enum {NONE, RUNNING, NEW_FRAME} State;

    /* Register used for counting propogation delays */
    reg [COUNT_WIDTH-1:0] delay_counter = {COUNT_WIDTH{1'b0}};

    /* Only account for propogation delay if one is configured */
    State state = PROP_DELAY ? NONE : RUNNING;

    /* Delays the pixel index counting and new_frame flag by the specified
        propogation delay */
    always @(posedge clk) begin
        /* If RUNNING as usual */
        if(state == RUNNING) begin
            delayed_new_frame <= 0;
            delay_counter <= 1; // Pre-empt the propagation delay
            if(new_frame) begin
                state <= NEW_FRAME;
            end
        /* If NONE or NEW_FRAME and delay has been reached */
        end else if(delay_counter == PROP_DELAY - 1) begin
            state <= RUNNING;
            delayed_new_frame <= NEW_FRAME ? 1 : 0;
        /* If NONE or NEW_FRAME and delay not yet reached */
        end else begin
            delay_counter <= delay_counter + 1;
        end
    end

    /* Count pixel (x,y) indicies once propogation delay has occurred */
    always @(posedge clk) begin
        if(delayed_new_frame || state == NONE) begin
            ind_x <= 0;
            ind_y <= 0;
```

```
51          end else begin
52              if(ind_x == IM_WIDTH - 1) begin
53                  ind_y <= ind_y + 1;
54                  ind_x <= 0;
55              end else begin
56                  ind_x <= ind_x + 1;
57              end
58          end
59      end
60
61  endmodule
```

A10 Feature Descriptor

```
`timescale 1ns / 1ps

/* Generates a 128-bit binary descriptor from a 9x9 pixel window */

module feature_descriptor
    #(
        parameter BW = 8,           // Pixel value Bit Width
        localparam DW = 128         // Descriptor width
    )(
        input clk,
        input wire [8:0] [8:0] [BW-1:0] px_window,
        output reg [DW-1:0] descriptor
    );

    localparam NODES = 17;
    wire [NODES-1:0] [BW-1:0] node;    // Vector containing all relevent
        pixels


    /* Remap the relevent points in the 9x9 window to a vector */
    assign node = {
        /* Outer Nodes */
        px_window[00][03], px_window[00][05], px_window[01][07], px_window
            [04][08],
        px_window[07][07], px_window[08][05], px_window[08][03], px_window
            [07][01],
        px_window[04][00], px_window[01][01],
        /* Middle Nodes */
        px_window[02][04], px_window[03][06], px_window[05][06], px_window
            [06][04],
        px_window[05][02], px_window[03][02],
        /* Center Node */
        px_window[04][04]
    };

    /* Clear the descriptor */
    initial begin
        descriptor[DW-1:0] = {DW{1'b0}};
    end

    /* Generate the descriptor */
    always @(posedge clk) begin
        descriptor[DW-1:0] <= {
            node[16] > node[05], node[07] > node[00], node[15] > node[07],
                node[03] > node[04],
            node[14] > node[04], node[01] > node[05], node[02] > node[10],
                node[02] > node[08],
            node[12] > node[15], node[09] > node[02], node[08] > node[11],
                node[05] > node[12],
            node[13] > node[16], node[10] > node[11], node[13] > node[04],
                node[06] > node[11],
            node[08] > node[03], node[09] > node[00], node[10] > node[07],
                node[01] > node[08],
            node[13] > node[07], node[04] > node[10], node[05] > node[09],
```

```
                node[00] > node[12],
46              node[09] > node[07], node[11] > node[13], node[15] > node[04],
                    node[14] > node[02],
47              node[11] > node[16], node[08] > node[05], node[14] > node[13],
                    node[01] > node[10],
48              node[15] > node[08], node[00] > node[14], node[07] > node[02],
                    node[16] > node[09],
49              node[08] > node[09], node[15] > node[06], node[06] > node[04],
                    node[03] > node[14],
50              node[07] > node[14], node[10] > node[09], node[06] > node[09],
                    node[01] > node[14],
51              node[09] > node[03], node[11] > node[07], node[01] > node[02],
                    node[08] > node[13],
52              node[03] > node[15], node[14] > node[16], node[05] > node[04],
                    node[08] > node[04],
53              node[00] > node[11], node[11] > node[02], node[01] > node[04],
                    node[00] > node[01],
54              node[03] > node[05], node[11] > node[14], node[07] > node[08],
                    node[13] > node[05],
55              node[00] > node[15], node[03] > node[02], node[06] > node[14],
                    node[15] > node[16],
56              node[11] > node[05], node[15] > node[13], node[09] > node[04],
                    node[08] > node[14],
57              node[06] > node[12], node[04] > node[00], node[02] > node[05],
                    node[07] > node[05],
58              node[06] > node[13], node[07] > node[16], node[11] > node[01],
                    node[16] > node[03],
59              node[07] > node[12], node[12] > node[14], node[02] > node[06],
                    node[16] > node[00],
60              node[09] > node[01], node[00] > node[06], node[00] > node[13],
                    node[07] > node[03],
61              node[09] > node[13], node[06] > node[07], node[02] > node[00],
                    node[02] > node[16],
62              node[11] > node[09], node[16] > node[01], node[11] > node[04],
                    node[01] > node[13],
63              node[09] > node[12], node[00] > node[05], node[15] > node[11],
                    node[10] > node[03],
64              node[15] > node[09], node[12] > node[02], node[00] > node[08],
                    node[13] > node[03],
65              node[06] > node[01], node[04] > node[07], node[09] > node[14],
                    node[10] > node[15],
66              node[12] > node[04], node[03] > node[00], node[01] > node[15],
                    node[13] > node[12],
67              node[02] > node[15], node[05] > node[06], node[05] > node[10],
                    node[10] > node[06],
68              node[13] > node[10], node[05] > node[15], node[13] > node[02],
                    node[01] > node[03],
69              node[12] > node[01], node[11] > node[03], node[12] > node[16],
                    node[06] > node[08],
70              node[15] > node[14], node[10] > node[00], node[07] > node[01],
                    node[16] > node[08],
71              node[10] > node[14], node[08] > node[10], node[04] > node[16],
                    node[14] > node[05]
72          };
73      end
74
```

```
75 endmodule
```

A11 Descriptor Comparator

```
`timescale 1ns / 1ps

/* Computes the hamming distance between two descriptors */
module descriptor_comparator #(
        localparam DW = 128, // Descriptor Bit Width
        localparam SW = 8    // Width of register that holds hamming distance
            (suggested log_2(DW)+1)
    )(
        input clk,
        input wire [DW-1:0] a,
        input wire [DW-1:0] b,
        output reg [SW-1:0] distance
    );

    /* XOR of the two descriptors */
    reg [DW-1:0] hxor = {DW{1'b0}};

    /* 6-input lookup table */
    reg [2:0] hlut [63:0] = '{
        6, 5, 5, 4, 5, 4, 4, 3, 5, 4, 4, 3, 4, 3, 3, 2,
        5, 4, 4, 3, 4, 3, 3, 2, 4, 3, 3, 2, 3, 2, 2, 1,
        5, 4, 4, 3, 4, 3, 3, 2, 4, 3, 3, 2, 3, 2, 2, 1,
        4, 3, 3, 2, 3, 2, 2, 1, 3, 2, 2, 1, 2, 1, 1, 0
    };

    /* Adder tree rows in format [count][bits] */
    reg [10:0][3:0] ra = {11{4'b0}};
    reg [05:0][4:0] rb = {06{5'b0}};
    reg [02:0][5:0] rc = {03{6'b0}};
    reg [01:0][6:0] rd = {02{7'b0}};

    /* Perform the XOR operation */
    always @(posedge clk) begin
        hxor <= a ^ b;
    end

    /* Computes hamming distance using 22 LUTS followed by an adder tree */
    always @(posedge clk) begin
        /* Top row using 6-input LUTS */
        ra[10] <= hlut[hxor[127:122]] + hlut[hxor[121:116]];
        ra[09] <= hlut[hxor[115:110]] + hlut[hxor[109:104]];
        ra[08] <= hlut[hxor[103:098]] + hlut[hxor[097:092]];
        ra[07] <= hlut[hxor[091:086]] + hlut[hxor[085:080]];
        ra[06] <= hlut[hxor[079:074]] + hlut[hxor[073:068]];
        ra[05] <= hlut[hxor[067:062]] + hlut[hxor[061:056]];
        ra[04] <= hlut[hxor[055:050]] + hlut[hxor[049:044]];
        ra[03] <= hlut[hxor[043:038]] + hlut[hxor[037:032]];
        ra[02] <= hlut[hxor[031:026]] + hlut[hxor[025:020]];
        ra[01] <= hlut[hxor[019:014]] + hlut[hxor[013:008]];
        ra[00] <= hlut[hxor[007:002]] + hlut[hxor[001:000]];
        /* Row 2 */
        rb[05] <= ra[10];
        rb[04] <= ra[9] + ra[8];
        rb[03] <= ra[7] + ra[6];
```

```
54          rb[02] <= ra[5] + ra[4];
55          rb[01] <= ra[3] + ra[2];
56          rb[00] <= ra[1] + ra[0];
57          /* Row 3 */
58          rc[02] <= rb[5] + rb[4];
59          rc[01] <= rb[3] + rb[2];
60          rc[00] <= rb[1] + rb[0];
61          /* Row 4 */
62          rd[01] <= rc[2];
63          rd[00] <= rc[1] + rc[0];
64          /* Row 6 */
65          distance <= rd[1] + rd[0];
66      end
67
68  endmodule
```

A12 Delay Buffer

```
`timescale 1ns / 1ps

/* Delays a stream of values by the specified delay */
module delay_buffer
    #(
    parameter WIDTH = 8,    // Width of values held in buffer
    parameter DELAY = 10    // Depth of the buffer (MIN = 2)
    )
    (
    input clk,
    input reg [WIDTH-1:0] data_in,
    output wire [WIDTH-1:0] data_out
    );

    reg [DELAY-2:0][WIDTH-1:0] buffer;

    /* Assign input and output to start and end of buffer */
    assign data_out = buffer[DELAY-2];

    /* Clears the buffer */
    initial begin
        for(integer i = 0; i < DELAY-1; i=i+1) begin
            buffer[i][WIDTH-1:0] = {WIDTH{1'b0}};
        end
    end

    /* Shifts the data in the delay buffer */
    always @(posedge clk) begin
        buffer[0] <= data_in;
        buffer[DELAY-2:1] <= buffer[DELAY-3:0];
    end

endmodule
```

A13 Feature Tracker

```
`timescale 1ns / 1ps

/* Provided a stream of features, this module is able to track their movement */
module feature_matcher #(
        parameter DW = 128,                 // Width of the descriptors
        parameter PW = 10,                  // Width of the position values
        parameter LE = 04,                  // Number of library entries
        parameter STRW = 8,                 // Width of the strength value
        parameter MATCH_THRESHOLD = 15,     // Max matching distance
        parameter SIMILAR_THRESHOLD = 20,   // Max dist between features concidered similar
        parameter MMF = 10,                 // Maximum missed frames
        parameter MFW = 08,                 // Missed frames counter width
        parameter FCW = 10                  // Frame counter width
    )(
        input clk,
        input new_frame,

        /* New features */
        input wire feature_flag,
        input wire [DW-1:0] feature_d,
        input wire [PW-1:0] feature_x,
        input wire [PW-1:0] feature_y,
        input wire [STRW-1:0] feature_s,

        /* Feature Matches */
        output reg match_flag,
        output reg [PW-1:0] match_xs,
        output reg [PW-1:0] match_ys,
        output reg [PW-1:0] match_xe,
        output reg [PW-1:0] match_ye,
        output reg [FCW-1:0] match_span
    );

    localparam DCD = 6;                 // Descriptor comparator delay
    localparam RE = LE + DCD + 1;       // Number of entries in the shift register
    localparam SW = 8;                  // Width of the comparator summation (log2(DW) +1)
    localparam OW = 10;                 // Width of value that holds library index

    /* States of the library entry state machine */
    typedef enum{
        L_EMPTY,      // No features in this entry
        L_NEW,        // New feature added this frame
        L_OCCUPIED    // Occupied by feature
    } libstate;

    /* States of the register entry state machine */
    typedef enum{
        R_EMPTY,      // No feature
        R_DEFAULT,    // Standard comparisons
```

```
50          R_SIMILAR,    // Similar to another feature
51          R_SEARCHING   // Searching to be placed
52      } regstate;
53
54      /* Data shared by lib and reg */
55      typedef struct packed{
56          reg [PW-1:0] x;      // X position of feature
57          reg [PW-1:0] y;      // y position of feature
58          reg [STRW-1:0] s;    // feature strength
59          reg [DW-1:0] d;      // feature descriptor
60      } feature_data;
61
62      /* Library Entry */
63      typedef struct packed{
64          reg[2:0] s;          // State
65          reg [FCW-1:0] cf;    // Consecutive frames
66          reg [MFW-1:0] mf;    // Missed frames
67          feature_data data;   // Feature data
68      } lib_feature;
69
70      /* Shift Register Feature */
71      typedef struct packed{
72          reg[2:0] s;          // State
73          feature_data data;   // Feature data
74      } reg_feature;
75
76      /* Rester component of matches stored */
77      typedef struct packed{
78          reg occupied;            // Flags if this struct is occupied
79          reg [PW-1:0] xe;         // X position of end feature
80          reg [PW-1:0] ye;         // y position of end feature
81      } feature_match;
82
83      feature_match [LE-1:0] match_list;
84      /* List of the library features */
85      lib_feature [LE-1:0] lib_list;
86      reg [LE-1:0] [SW-1:0] hamming_dist;
87
88      /* List if the features in the shift register */
89      reg_feature [RE-1:0] reg_list;
90
91      /* Variables used for buffering shift register input */
92      reg temp_used = 0;
93      reg_feature temp_reg;
94      reg_feature clean_reg;
95      reg_feature checker_reg;
96
97      /* Used to loop through and output them one at a time */
98      reg [OW-1:0] out_k = 0;
99
100     generate
101         /* Feed the library/register descriptors into a comparator */
102         for(genvar i = 0; i < LE; i = i+1) begin: comparator_gen
103             descriptor_comparator #()
104                 descriptor_comparator_1(.clk(clk), .a(lib_list[i].data.d), .b(
                        reg_list[i].data.d), .distance(hamming_dist[i]));
```

```
105         end
106     endgenerate
107 
108     /* Clear the library and registers */
109     initial begin
110         /* Library entires */
111         for(integer i = 0; i < LE; i = i+1) begin
112             lib_list[i].s = L_EMPTY;
113             lib_list[i].cf = 0;
114             lib_list[i].mf = 0;
115             lib_list[i].data = {{PW{1'b0}}, {PW{1'b0}}, {STRW{1'b0}}, {DW{1'b0
                }}};
116         end
117         /* Register entries */
118         for(integer i = 0; i < RE; i = i+1) begin
119             reg_list[i].s = R_EMPTY;
120             reg_list[i].data = {{PW{1'b0}}, {PW{1'b0}}, {STRW{1'b0}}, {DW{1'b0
                }}};
121         end
122         /* Match entires */
123         for(integer i = 0; i < LE; i = i+1) begin
124             match_list[i].occupied = 0;
125             match_list[i].xe = 0;
126             match_list[i].ye = 0;
127         end
128         /* Temp reg used whe shift reg in use */
129         temp_reg.s = R_EMPTY;
130         temp_reg.data = {{PW{1'b0}}, {PW{1'b0}}, {STRW{1'b0}}, {DW{1'b0}}};
131         /* Clean reg to use when no data */
132         clean_reg.s = R_EMPTY;
133         clean_reg.data = {{PW{1'b0}}, {PW{1'b0}}, {STRW{1'b0}}, {DW{1'b0}}};
134     end
135 
136     /* Clear the match output */
137     initial begin
138         match_flag = 0;
139         match_xs = 0;
140         match_ys = 0;
141         match_xe = 0;
142         match_ye = 0;
143         match_span = 0;
144     end
145 
146     /* Responsible for placing data into the shift register */
147     always @(posedge clk) begin
148         /* Placing an input feature into the shift register */
149         if(feature_flag) begin
150             reg_list[0].s <= R_DEFAULT;
151             reg_list[0].data <= {feature_x, feature_y, feature_s, feature_d};
152         end
153 
154         /* Place shift reg outputs into checker register for analysis */
155         checker_reg <= reg_list[RE-1];
156 
157         /* If the feature is in a state that permits library placement */
158         if(checker_reg.s == R_DEFAULT) begin
```

```
159             /* Wait to place data becuse reg currently in use */
160             if(feature_flag) begin
161                 temp_used <= 1;
162                 temp_reg.s <= R_SEARCHING;
163                 temp_reg.data <= checker_reg.data;
164             /* Put data straight into the shift register */
165             end else begin
166                 reg_list[0].s <= R_SEARCHING;
167                 reg_list[0].data <= checker_reg.data;
168             end
169         end
170 
171         /* Place the backlogged feature into the shift register */
172         if(temp_used && !feature_flag) begin
173             reg_list[0] <= temp_reg;
174             temp_used <= 0;
175         end
176 
177         /* Pass through clean and empty data if the shift reg is not in use */
178         if(
179         !(feature_flag ||
180         (!feature_flag && checker_reg.s == R_DEFAULT) ||
181         (temp_used && !feature_flag))) begin
182             reg_list[0] <= clean_reg;
183         end
184     end
185 
186     /* Shift the registers not shifted by the library adder/swapper */
187     always @(posedge clk) begin
188         reg_list[DCD:1] <= reg_list[DCD-1:0];
189     end
190 
191     /* Loops through the found matches and passes them out sequentially */
192     always @(posedge clk) begin
193         /* If data is ready to be passed out */
194         if(match_list[out_k].occupied) begin
195             match_flag <= 1;
196             match_list[out_k].occupied <= 0;
197             match_xs <= lib_list[out_k].data.x;
198             match_ys <= lib_list[out_k].data.y;
199             match_xe <= match_list[out_k].xe;
200             match_ye <= match_list[out_k].ye;
201             match_span <= lib_list[out_k].cf;
202         end else begin
203             /* Release control to the library (USED FOR IMPLEMENTATION) */
204             // match_list[out_k].occupied <= 1'bZ;
205             match_flag <= 0;
206         end
207         /* Increment the counter and wrap back to 0 */
208         out_k <= (out_k == LE-1) ? 0 : out_k + 1;
209     end
210 
211     /* Adding/Swapping/Matching features in library */
212     always @(posedge clk) begin
213         for(integer i = 0; i < LE; i = i+1) begin
214             /* Perform library upkeeping operations when a new frame is
```

```
                detected */
215             if(new_frame) begin
216                 if(lib_list[i].s == L_OCCUPIED || lib_list[i].s == L_NEW)
                        begin
217                     /* Increment frame counters when a new frame starts */
218                     lib_list[i].cf <= lib_list[i].cf + 1;
219                     lib_list[i].mf <= lib_list[i].mf + 1;
220                     /* Confirm the features state within the library at the
                            end of the frame */
221                     /* Remove frames from library if they have expired */
222                     lib_list[i].s <= (lib_list[i].mf == MMF-1) ? L_EMPTY :
                            L_OCCUPIED;
223                 end
224                 /* Shifts the shift register elements that are touched by the
                        processor*/
225                 reg_list[i+DCD+1] <= reg_list[i+DCD]; // Shift reg values
226             end
227             /* If a feature needs to be added to or swapped into library */
228             else if(
229             /* If the register feature is looking to be placed */
230             reg_list[i+DCD].s == R_SEARCHING &&
231             /* If there is a free spot in the library or a better feature was
                    found in the same frame*/
232             (lib_list[i].s == L_EMPTY || (lib_list[i].s == L_NEW && lib_list[i
                    ].data.s < reg_list[i+DCD].data.s))) begin
233                 /* Set up the library entry */
234                 lib_list[i].s <= L_NEW;
235                 lib_list[i].cf <= 0;
236                 lib_list[i].mf <= 0;
237                 lib_list[i].data <= reg_list[i+DCD].data;
238                 /* Swap out the data (lib data garbage if L_EMPTY) */
239                 reg_list[i+DCD+1].data <= lib_list[i].data; // Shift reg
                        values
240                 /* Keep register searching if it was a swap out (L_NEW) and
                        not just a placement (L_EMPTY) */
241                 reg_list[i+DCD+1].s <= (lib_list[i].s == L_EMPTY) ? R_EMPTY :
                        R_SEARCHING; // Shift reg values
242             /* If a match has been found */
243             end else if(
244             /* If the register feature is looking for a match */
245             (reg_list[i+DCD].s == R_DEFAULT || reg_list[i+DCD].s == R_SIMILAR)
                     &&
246             /* If the library has a feature to match */
247             lib_list[i].s == L_OCCUPIED) begin
248                 /* If the descriptors are a match */
249                 if(hamming_dist[i] < MATCH_THRESHOLD) begin
250                     /* Deactivate the register */
251                     reg_list[i+DCD+1].s <= R_EMPTY; // Shift reg values
252                     /* Number of missed frames is reset */
253                     lib_list[i].mf <= 0;
254                     /* Output the match */
255                     if(!match_list[i].occupied) begin
256                         match_list[i].occupied <= 1;
257                         match_list[i].xe <= reg_list[i+DCD].data.x;
258                         match_list[i].ye <= reg_list[i+DCD].data.y;
259                     end else begin
```

```
260                         /* Release control to the feature output system (USED
                                FOR IMPLEMENTATION) */
261                         // match_list[out_k].occupied <= 1'bZ;
262                     end
263                 /* Flags if detected features are too similar to the ones in
                        the library */
264                 end else if(hamming_dist[i] < SIMILAR_THRESHOLD) begin
265                     reg_list[i+DCD+1].s <= R_SIMILAR;
266                     reg_list[i+DCD+1].data <= reg_list[i+DCD].data;
267                 /* Shift the register values as normal */
268                 end else begin
269                     reg_list[i+DCD+1] <= reg_list[i+DCD];
270                 end
271             /* Shift the register values as normal */
272             end else begin
273                 reg_list[i+DCD+1] <= reg_list[i+DCD]; // Shift reg values
274             end
275         end
276     end
277
278 endmodule
```