POLITECNICO DI TORINO

Department of Electronics and Telecommunications
Electronics Engineering

Master's Thesis

A reconfigurable architecture for event-based optical flow in FPGA

Supervisors:
Prof. Maurizio Martina
Dr. Paolo Motto Ros

Candidate:
Ibrahim Shour

September 2018

Summary

The aim of this thesis is to present a reconfigurable hardware implementation for event-based optical flow on incoming events that arrive from an event based image sensor. The implementation is characterized by its reconfigurability and flexibility in which different parameters can be modified at the synthesis stage, and the target device is a Zedboard Zynq-7000 featuring an XC7Z020-CLG484-1 FPGA. The work of this thesis is developed in collaboration with the Italian Institute of Technology (IIT), where the target application of this study would be the iCub humanoid robot.

The thesis first examines the concept of neuromorphic architectures, computers, and vision sensors. Neuromorphic event-based image sensors are inspired by the human retina and are characterized by having a low latency, low redundancy, high dynamic range, and lower energy consumption, which make them ideal for robotics and unmanned vehicles. In a second stage, the concept of optical flow is highlighted, and some related problems and difficulties are posed. The optical flow corresponds to the detection of moving physical objects in a certain sequence of images. Traditional frame-based optical flow algorithms are based on long timing intervals which make them inadequate for being employed on event-based image sensors. Therefore, a new family of optical flow computation algorithms for event-based image sensors is illustrated, including an algorithm which is the fundamental part of this thesis. The framework allows the estimation of the optical flow by exploiting the local characteristics of the events' spatio-temporal space. The algorithm is applied to a YARP module which is a C++ based software package used for interconnecting the different sensors and processors in robots, integrated as an essential part of the iCub robot. The different code building blocks are analysed and the hardware architecture is defined. After that, a detailed analysis on the working principle of the different hardware blocks is performed, including details on the VHDL modules, and the parallel computation of the amplitude and the orientation of the visual flow. Later on, a simulation is provided as a demonstration of the functioning of the design, Finally, the synthesis results in terms of latency and hardware resources consumption are exposed for the different possible configurations and potential future improvements and steps are discussed, including possible hardware modifications that may allow optimizing the design.

Contents

Chapter 1

Research on Neuromorphic Architectures

1.1 Introduction

Artificial Intelligence (AI) is becoming an essential part of our daily life. From our homes, to our cars, office, etc. So as the complexity of applications required by humans grows, programming manually becomes much more complex, and as a result, the need of machines that could behave in a smart way, and be self-programmed as humans, increases. AI is all about machines that are able to learn and make decisions based on the learned behaviours, it is based on a multidisciplinary approach including neuroscience, biology, electronics, computer science, and many others. Its applications vary from gaming, to robotics, vision systems, speech recognition and countless other applications.

1.1.1 Neural Networks

Artificial Neural Networks (ANN) are one research area in AI. They appear as the capability of designing computing architectures that could emulate the working principle of the brain. They are simply a computational model including a certain number of processing elements (*neurons*) interconnected with each other through a particular type of connection (*synapses*), each neuron and synapse can have a certain weight that could change through learning. Neural networks are so many to count, they differ mainly in the number of levels they support and in the way in which the processing elements are connected. One important example used mostly in image processing are the *Convolutional Neural Networks* (CNN), they are feed forward neural networks where information in the input layer is fed only in one direction to the output layer (no cycles). They are called convolutional since some of the hidden layers in the network are convolutional layers, which emulate the behaviour of a neuron as a response of a visual stimuli, and where

the corresponding network weights are shared among the convolutional layers. Neural networks software demonstrated high efficiency when running on regular computers, but as the need of larger neural networks grows, the required computational power increases, and so, different computation approaches must be adopted.

Trying to emulate more closely the brain's working principle, *Spiking Neural Networks* (SNNs) [1] arrive as the third generation of artificial neural networks. The neurons in these networks generate spikes as a tool of communication, in an energy-efficient event-based manner. The development of such networks raised the hopes of achieving a more realistic brain imitation within a combination of the new generation of neural networks with a suitable computational hardware.

1.1.2 From Neural Networks to Neuromorphic Computers

Neuromorphic computers (NC) are thought to be the solution for the limitations related to using traditional computers. NCs are notable for being highly connected, supporting better parallel processing through collocating both memory and processing through distributed elements, and for requiring much lower power with respect to normal computers. All these characteristics are guaranteed by an asynchronous spike-based system, that imitates the brain more efficiently than ever.

The neuromorphic approach is being adopted more than before for several reasons: [2]

- Von Neumann bottleneck represents the main reason which inspired researchers to move in the neuromorphic direction, it's a major problem in traditional computers for which memory and processing are done in different parts of the system, creating a throughput limitation, that causes many difficulties when there is a need of a real-time system that must act with no delays
- the increasing power demands of different computational model (i.e. neural networks)
- Scalability: neuromorphic computers are known for their ability to be scaled based on the application that has to be implemented on, scaling means increasing the number of neurons in simple way by connecting different chips
- the need of machine learning that is presented by an ability of the system to adapt to dynamic changes at the input
- the near end of Moore's law in scaling devices, so it would be difficult to reach further power reduction and higher performance while keep using the traditional device approach

Neural Network Models

In a neuromorphic computer, memory and processing are distributed among an enormous number of neurons, each neuron communicates with a countless others through synapses, each synapse has a certain weight. Neuron accumulates charge by receiving spikes from the connected neurons, and when it reaches a certain charge threshold, it fires a spike forcing a change on the connected neurons, this change depends on the synaptic weight. All these operations are imposed by a neural network model that controls how the different elements interact with each other. Such a model includes:

- a *network model* that specifies how the elements are related, for instance it defines if the elements are connected in a feed-forward fashion, or in a recurrent fashion in which cycles exist
- a *neuron model* represented by a mathematical equation that describes the behaviour of a neuron, namely it tells how the charge on a neuron varies and when does it fire.
- a *synapse model* which describes the physical aspect of a neuron and whether a synapse supports the *plasticity* mechanism - a mechanism in which the strength (weight) of a synapse changes over time, it varies accordingly with the activity over a synapse.

Learning Algorithms

One key aspect of neural networks is its learning capabilities. Once the network model is chosen with all its corresponding choices, a learning algorithm could be decided. Whenever a learning algorithm must be implemented, some choices must be taken into consideration: [2]

- whether training is done *on-chip* or *off-chip*. Off-chip means that the training phase is done using an external computer, then the trained data is transferred on-chip.
- whether learning is implemented *on-line* or *off-line*
  - on-line learning is illustrated by the ability of the system to adapt to dynamic changes in the environment (inputs), so each time the environment changes the neural network model is adjusted to fit in the new conditions
  - off-line learning includes two phases - a training phase and a testing phase. In this specific implementation, no further training is permitted during the operation. Such an algorithm is implemented only in static environment, in which any change in the settings could cause an unknown behaviour.
- whether learning is *supervised*, *unsupervised* or *semi-supervised*.

- supervised learning where input data is labelled along with the correct answers, the learning algorithm will associate all these information and try to predict the correct answer
- unsupervised learning means that the correct answers are not given to the algorithm (only input data are provided), so that the algorithm must learn if there is a certain pattern in the data and decide how important are the features based on that pattern
- semi-supervised where both labelled and unlabelled data are provided in the training stage. Their are two main advantages of this type of learning. The first is that performing a lot of labelling on the input data is time consuming, and shall be avoided when possible. The second reason is that using unlabelled data improves the accuracy.

Two of the most used learning algorithms are *back propagation* and *spike-timing dependent plasticity* (STDP). Back propagation is represented by the fact that after the training phase - during the testing phase - if the system's decision is incorrect, the error is calculated at the output and is propagated backwards to correct some weights in the network. It is mostly considered as a supervised algorithm, however it has some unsupervised implementations.

On the other hand, STDP is characterized by an on-line change in the synaptic weights as follows: if a pre-synaptic neuron (spike transmitter) fires before (after) a post-synaptic neuron, the synaptic weight between the two neurons will increase (decrease). Briefer is the time difference, higher is the magnitude of change in the weight.

Traditional CMOS synapse implementations have been recently used, however they suffer from two main problems. The first is related to the fact that an efficient and precise system requires analog synaptic weights, either by digitalizing these values and saving them in a memory - this implies that enormous memory is required, or in an analog manner utilizing capacitors which suffer from reliability and leakage problems. The second problem is associated to the increasing size and complexity of neural networks, which also increases the power consumption. Hence, researches were carried on to find alternative implementation possibilities. One of the technologies that could support efficiently the STDP mechanism is the memristor technology. A memristor is a device with a variable resistance, its value changes based on the voltage applied in the past, it has a sort of memory behaviour which makes it ideal for learning mechanisms. It was realized in HP laboratories in 2008 [3], although it firstly appeared in the early 70's (1971) in a work published by Professor Leon Chua. What is interesting about memristors are not only their learning capabilities, but also their energy-efficiency and scalability (scalable to 10 nm), offering a high density structure. Moreover, various memristors are CMOS-compatible [4], making

them a good candidate for a hybrid CMOS/memristor architecture, with CMOS neurons and memristive synapses, in a possible high-density crossbar structure. Although memristors have been mainly used in synapse implementations, fewer neuron implementations do exist.

1.1.3 Hardware Implementations and Applications

There are several possible hardware implementations for neuromorphic computers, these implementations range from analog to digital architectures, with the possibility of having a hybrid analog/digital architecture. Since biology is mostly analog, an analog implementation could simulate the behaviour of the brain more efficiently. One drawback of analog implementations is that analog components are more susceptible to noise that makes them suffer from robustness and reliability problems. However, in a system that is able to learn by itself, noise will not be a problem any more in which the system would be capable of adapting to changes in the environment, making it more robust. Some researchers think that the most effective way would be using analog components for computation and digital approach for the communication between the processing elements [2]. Both analog and digital implementations could be divided into two broad categories, custom and programmable.

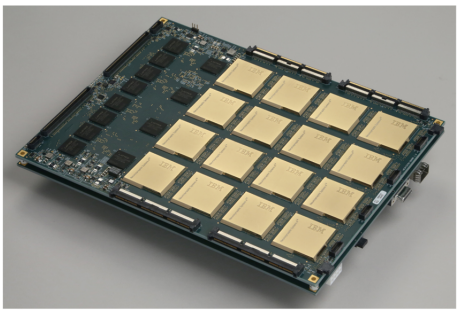
Digital Implementations

For digital systems, both ASIC and FPGA-based approaches showed interesting performance when running different types of applications.

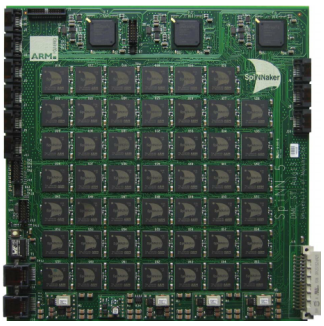
Custom-chip or ASIC-based approach is characterized by having lower power consumption than the FPGA-based approach. Two of the most notable custom-chip neuromorphic implementations are the TrueNorth [5] chip and the SpiNNaker [6] chip.

IBM's TrueNorth chip, a fully-digital one, was a part of the Defence Advanced Research Projects Agency (DARPA) SyNAPSE project. It consists of 4096 neurosynaptic cores organized on a 2D-array, each core contains 256 neurons, reaching more than one million neurons for chip, with 256 million synapse. It was implemented using 5.4 billion transistors in CMOS technology, consuming merely 65 mW. One important characteristic of the TrueNorth chip is its scalability: larger neurosynaptic networks could be achieved simply by connecting different chips (fig. 1.1a). Researchers created a Corelet [7] object-oriented programming language for composing the neural network on the different cores.

Different types of applications from different sectors were tested on the TrueNorth chip/multi-chip architectures. Two interesting on-field implementations were object detection and recognition [8], and an autonomous driving robot [9]. The first consisted of testing the performance of the chip on the Neovision2 Tower dataset (fig. 1.2a) - a set



(a) The NS16e circuit board incorporating 16 IBM TrueNorth chips. (Photo: IBM Research)

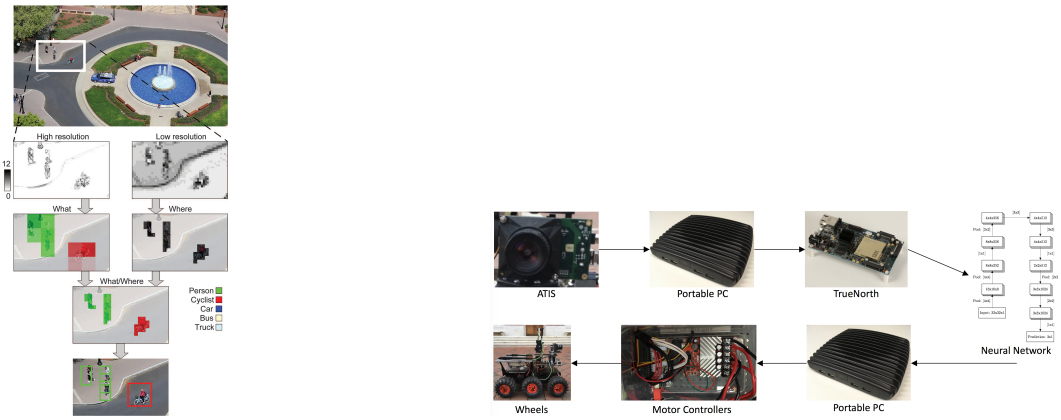


(b) 48-node SpiNNaker Processor. (Photo: University of Manchester)

Figure 1.1: TrueNorth and SpiNNaker: two ASIC neuromorphic chips

that contains images taken at 30 frames-per-second of distinct object categories: people, cyclists, trucks, cars and buses. The chip showed nearly an 80% classification accuracy consuming only 63 mW of power.
The second application instead, consisted of a CNN running on a TrueNorth chip to estimate the steering direction of a self-driving robot, the estimation was based on the output of an *Asynchronous Time-Based Image Sensor* (ATIS). The training was achieved by associating the users' remote commands provided manually, with the output of the ATIS to generate the corresponding neural network (fig. 1.2b), training took place in three different environments. The application demonstrated correct decisions in 82% of time. However, these results are expected to improve through extended training. Despite all the satisfying results, there are still some limitations of the TrueNorth architecture. These limitations are due to the fact that the chip was designed before the major developments in the of neural networks (e.g. CNNs and DNNs). For instance, when researchers tried implementing a certain application that needs 30 thousand neurons [10] - a number that could be provided by a single chip - they were forced to use 8 chips containing 8 million neurons instead.

On the other hand, the spiking neural network architecture (SpiNNaker) project - developed at the University of Manchester as a part of the *Human Brain Project* (HBP) in 2014 - is a massively parallel digital architecture composing billions of computing units with a spike-based communication infrastructure. The architecture is based on parallel processing through ARM cores, each having two local memories: instruction memory and data memory. The chip contains 18 ARM968 cores featuring 80 thousand neurons and 20 millions synapse. When all cores are fully-loaded, the chip reaches a maximum power dissipation of 1 W. Likewise the TrueNorth, the SpiNNaker holds the characteristic



(a) Real-time multi-object recognition on TrueNorth (Photo: [8])

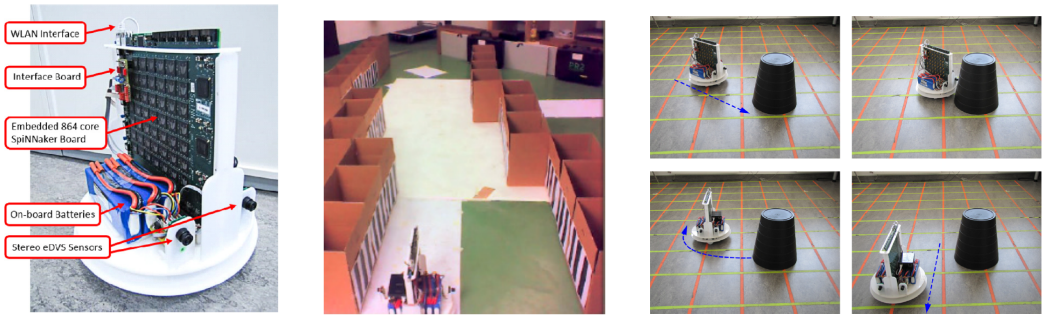
(b) Data flow in the neuromorphic robot system (Photo: [9])

Figure 1.2: TrueNorth: Applications

of being scaled through connecting different chips. One example is the 48-node structure (see fig. 1.1b), a board which implements 768 thousand neurons with 768 million synapse. In contrast to the TrueNorth, the SpiNNaker provides the user with the option to program the neuron and the synapse models. It also supports the plasticity mechanism (e.g. STDP) with some limitations. The applications could be written in high-level neural description languages such as PyNN [11]; a Python-based simulator-independent language; and Nengo [12]; a graphical and scripting software.

The SpiNNaker has proven promising performance in different application categories. In robotics field, it was used as a processor on an autonomous mobile platform [13, 14] along with dynamic vision sensors (DVS) providing the visual input. One application [13] consisted of a trajectory stabilization using the optical flow, in which the robot tried to remain in a central position when traversing a hand-made corridor with vertical stripes on the walls, such a behaviour was achieved by balancing the two optical flows arriving form the two lateral DVS (fig. 1.3b). Another example applied on the same robot platform was a learning-by-example application [14], in which the robot was controlled manually and it was forced to approach and collide with an obstacle. Once it collides with that obstacle, it was ordered to step back and move away from it (fig. 1.3c). After that, a new neural model was generated based on the sensory data and the related manual commands. The entire training phase was carried out in less than a minute.

In addition to robotics-related applications, the SpiNNaker processor was tested in image processing [15]. Specifically, three different image processing algorithms were tested on the board. It showed comparable performance to the GPU in terms of processing speed



(a) The SpiNNaker robotic platform (Photo: [13])

(b) Trajectory stabilization using optical flow: experiment arena (Photo: [13])

(c) Robot stimulus response after learning (Photo: [14])

Figure 1.3: SpiNNaker: Applications

(frames-per-second), with a much lower power budget than both GPU-based and CPU-based approaches.

On the other hand, the FPGA-based approach is characterized by being programmable and much more flexible. Some of the different implementations that have been realized are the following:

- *Minitaur* [16] - it is an event-driven neural network accelerator. It was implemented on a Xilinx Spartan-6 FPGA platform. It featured 65k neurons and 16.78 millions synapse, with a peak power consumption of 1.5 W. It was tested on a MNIST handwritten digits dataset, the recognition accuracy reached 92%. In addition, in a similar application of a newsgroup data set classification, the accuracy was about 71%.

- *NeuroFlow* [17] - a scalable SNN simulation platform. An implementation of 600k neurons with a 6-FPGA system was realized. To evaluate the performance different neural models that differ in size and complexity were tested. It demonstrated to be 33.6 times faster than an 8-core processor, and 2.38 times faster than a GPU, for some specific applications.

- *Darwin* [18] - it was prototyped on an FPGA before being fabricated in 180 nm CMOS process. It featured 2048 virtual neurons starting from 8 physical neurons using the time-multiplexing technique. The power consumption was of 0.84 mW/MHz. When the MNIST handwritten digits recognition was tested - with the same DBN applied to the Minitaur - the accuracy reached 93.8%.

Mixed Analog/Digital Implementations

The mixed analog/digital neuromorphic processors are basically inspired by the idea of having analog computation and digital communication. Three of the most popular examples in this category of processors are *Neurogrid* [19], *BrainscaleS* [20] and ROLLS [21] (*Reconfigurable On-line Learning Spiking Neuromorphic Processor*). Neurogrid is a neuromorphic computer, developed at Stanford University, it provides the real-time simulation of large-scale neural models. It is composed of 16 "NeuroCore" chips, each supporting 65536 neurons working in sub-threshold, incorporating a total of 1 million neurons and 4 billion synapses, with a peak power consumption of 3 W.

On the other hand, the ROLLS processor was founded at the Institute of Neuroinformatics, ETH and the University of Zurich. It comprised 256 neurons and 128k synapses. One interesting characteristic is its learning capabilities. In fact, on the contrary to other chips, ROLLS processor supports the plasticity mechanism (STDP learning). It has a power consumption of 4 mW for typical applications. In robotics, ROLLS was used to interpret the input spikes arriving from a DVS mounted to a "PushBot" robot. In specific, the neural population was used to estimate the speed and the steering direction of the robot in order to avoid obstacles in different environmental situations [22].

1.2 Event-based Neuromorphic Vision Sensors

Conventional video cameras have always been in development in the last few decades, they are based on taking snapshots at a certain rate followed by some processing techniques. Although they are characterized by a high quality image reproduction - by using simple small pixels - problems related to their high data redundancy, restricted dynamic range, and high power consumption limited their utilization in some fields. This is related to the fact that sampling is done even when no dynamic changes took place in the scene.

Inspired by the human retina, event-based image sensors come out as an alternative for conventional video cameras. They are represented by a low-latency event-based data stream with a low-power and fast processing algorithms. In the recent years, various ideas appeared as an alternative for conventional video cameras. However, the most interesting implementations have been the *Dynamic Vision Sensor* (DVS) [23] and the *Asynchronous Time-based Image Sensor* [24]. In the following some of the adopted ideas will be briefly highlighted.

1.2.1 Contrast-based Asynchronous Binary Vision Sensor

A spatial contrast-based asynchronous image sensor was discussed in [25]. The array is composed of a 64×128 elements, and the working principle is based on the 1-bit quantization of a local spatial contrast between a kernel of three pixels, in which only the addresses of pixels presenting a certain contrast magnitude, has to be dispatched. For a typical indoor application, only 5% of the pixels were active consuming only 100 μW of power (55 μW for the imager and 45 μW for the decoding and digital control part).

1.2.2 Colour Differentiating AER Image Sensors

In [26] a dichromatic asynchronous event pixel was presented, it was produced for being exploited by an address event representation (AER) vision sensor. It is based on the stacked photo diodes concept, with PN junctions with a different depth below the surface, which allows achieving a sensitivity of two different colour spectra. The work was extended in [27] a sensitivity of three different spectra was achieved based on stacked photo diodes concept as before. However, a quantification of the three spectra was achieved in pulse density modulated signals which are then transmitted off-chip in AER protocol.

1.2.3 Asynchronous Time-based Image Sensor (ATIS)

At the Austrian Institute of Technology, Posch and his colleagues designed a QVGA array of 304×240 pixels which are fully autonomous. The ATIS pixel including two sub-pixels [24]. The first corresponds to the temporal contrast pixel from the DVS, which is responsible of the change detection. The time-based intensity readings are triggered in the second pixel by the events of the DVS pixel, this part is responsible of the exposure measurement. The intensity depends on the time taken by a photodiode to integrate between the two levels. The exposure-measurement part is a time-based pulse-width modulation PWM imaging circuitry. The benefit of the ATIS pixel is the readout intensity that is characterized by a wide dynamic range the reaches $143\ dB$ in static conditions, and a $125\ dB$ at a temporal resolution which is equivalent to 30 fps. However, the expense is having a large pixel size and low fill factor (the ATIS area is nearly twice that of the DVS pixel). Moreover, another disadvantage is the capture time at low intensities which could reach hundred ms.

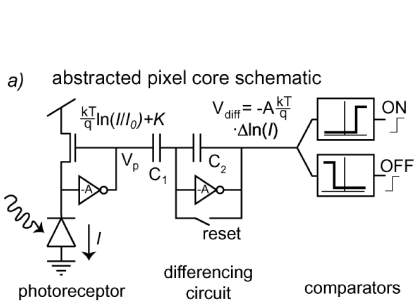
1.2.4 Dynamic Vision Sensor (DVS)

The *Dynamic Vision Sensor* [23] was developed by the group of Professor Tobi Delbruck at ETH Zurich. It was based on the temporal contrast between frames, in which only the pixel having a change in its illumination level by a certain threshold will get its address dispatched. The sensor included the following characteristics:

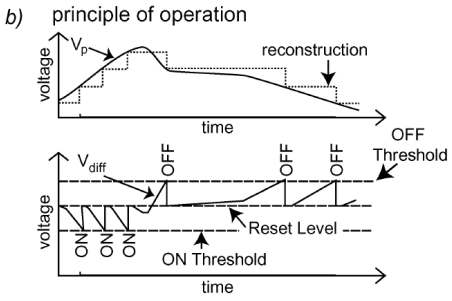
- event-based data communication
- requires low data storage
- 120-dB wide dynamic range making it efficient in uncontrolled light conditions
- 1 μs temporal resolution
- 1 ms maximum latency
- 20 mW power consumption

It is based on the quantization of the changes in the relative intensity, that leads to spiking events at the output. The output is then a stream of address-events (AEs), indicating the coordinates (x, y) of the spiking pixel, the polarization (ON/OFF), and a corresponding time stamp, generated in an asynchronous manner with a precise timing, in what is called *Address Event Representation* (AER) output. The polarization signifies that the illumination level has increased or decreased for a certain pixel, usually caused by a moving object in the scene. Having a latency of 1 μs, it is comparable to thousands of frames per second (fps) related to a conventional frame-based camera. The circuit is mainly composed of logarithmic photoreceptor with a switched-capacitor differencing

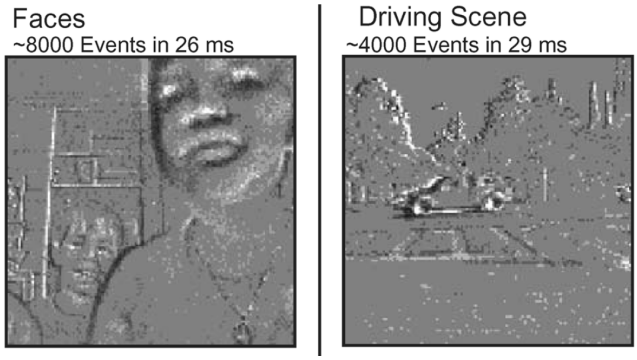
circuit that amplifies changes, and two-transistor comparator in the final stage (1.4a). The pixels are of $40 \times 40\ \mu m^2$ large.



(a) Pixel core schematic (Image: [23])



(b) Principle of operation (Image: [23])



(c) Example of DVS output when either object or camera motion takes place (Image: [23])

Figure 1.4: DVS

Generating spike-events related to changes in the light intensity, a DVS could be interfaced to neuromorphic chips for data processing, composing a complete event-based bio-inspired system, from sensing to processing.

The DVS has shown high efficiency in robotics, data acquisition and targeting. Specifically, it was used to build a fast-calibrating robotic goalie [28], composed of a DVS that provides an input to perform multiple balls targeting on a host PC.The movement of the goalie arm was based on the positions and velocities of the tracked balls. As a result, the system was able to block nearly 70 % of the balls.
In another application, event-based vision sensors were used to realize traffic data acquisition and object tracking systems [29], implementing on one side a system that is able to monitor 4 high-way lanes to estimate the speed and the number of vehicles that pass on those lanes, the vehicles were classified in as cars or trucks based on the information provided by the sensor.

1.2.5 Dynamic and Active-pixel Vision Sensor (DAVIS)

Another family of vision sensors are the *Dyanmic and Active-pixel Vision Sensors* (DAVIS) [30, 31], they combine the functionality of a *dynamic vision sensor* (DVS) with a frame-based *active pixel sensor* (APS) intensity readout. In this way, the objects' recognition and classification, and the scene content analysis could be handled by the frame-based readout, while the tracking of fast moving objects is managed through the output of the DVS. Since tracking is obtained through the DVS, a reasonable low APS frame-rate could be used in order to reduce the computational power, thus obtaining a low-latency and low-power architecture. The sensor has the following characteristics:

- 240×180 array size
- 12 μs minimum latency at 1 klux
- up to 130 dB dynamic range for the DVS, and 55 dB for the APS
- power consumption: 5 μW at low activity, and 14 μW for high activity

Chapter 2

Optical Flow

2.1 Definition

Motion of physical objects is an essential phenomenon in everyday life. This motion, when being observed by a spectator (a camera for instance), induces what is called the Optical Flow (OF). Optical flow [32] became a field of research in the early 80's, and it is defined as the motion of the brightness pattern in a certain sequence of images (Figure 2.1). Such a motion could be caused by a moving observer, a moving scene, or a mutual motion (relative motion between observer and scene objects). A motion may cause a displacement in the brightness values in the related image sequence. However, not every brightness variation corresponds to a motion, and not every motion will result in a brightness variation as well. Optical flow is affected by the illumination conditions that are present, and by different noise sources impacting the image sensor. The optical flow is strongly related to how close the observer and the objects in the scene are, at their corresponding travelling velocities, and to the angle of the scene with respect to the observer's travelling direction. One key issue discussed when mentioning the term "Optical Flow" is the ability of providing a low power architecture, with a high computational speed that is able to guarantee a real-time operation which is crucial in robotics and other autonomous systems, as in driver assistance systems, where performance becomes critical.

Different computation algorithms have been analysed and discussed in the last few decades, these algorithms are evaluated in terms of performance by a set of standards that could vary from one application to another, and they could be classified in various macro categories. Two of the most common categories are:

- Block matching algorithms [33] (Correlation-based) - they are based on assigning a flow vector to group of pixels (macro block) rather than assigning it to an individual pixel. This is achieved by searching the best matching block in the current frame to a corresponding block in a previous frame, in order to determine its displacement. Smart strategies must be used then in order to reduce memory requirements and

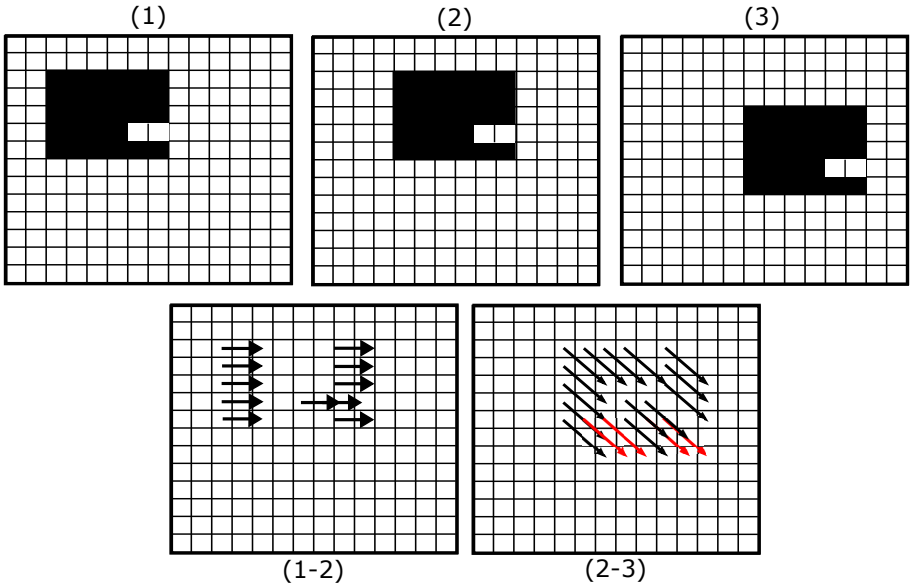


Figure 2.1: Simple Example of Optical Flow

computation time. This type of algorithms usually suffers from accuracy problems since they are pixel-discrete, and from their high computational cost requiring a very high number of comparisons.

- Differential algorithms (Gradient-based)- are the most widely used algorithms for optical flow computation. They are based on the calculation of spatial and temporal derivatives on a certain image. Are usually divided into two categories, local methods and global methods. One famous example of local methods is the *Lucas & Kanade* [34], while a notable global method is the *Horn & Schunck* method [35].

2.1.1 Mathematical Approach

A function I(x, y, t) of the grey values defines the considered image sequence. Where (x, y) represent the pixel position in the array, and t is the time. When calculating the optical flow, it is assumed that the brightness (grey value) of a standing still pixel in two consecutive frames is constant, and a moving pixel has a constant grey value over time, which is know as the *brightness constancy constraint*. The latter assumption is exposed to several limitations, either observer-related limitations corresponding to the noise affecting the image sensor, or scene-related ones associated the illumination variations and conditions in which the number of photons acquired by one pixel can vary over time. Optical flow computation also suffers from what is called the aperture problem, in which different moving patterns at different orientations and different speeds could generate the same optical flow output and identical response to a visual system (observing through a

small aperture), a simple example could be seen in figure (2.2).

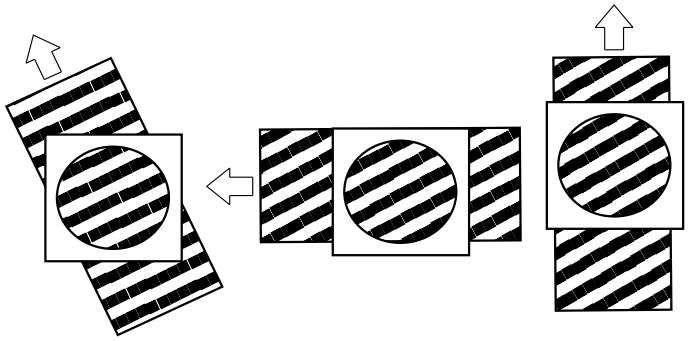


Figure 2.2: The Aperture Problem

The brightness constancy equation exposes the following equation:

$$I(x,y,t) = I(x+u,y+v,t+1) \tag{2.1}$$

where (u, v) is the optical flow vector in Δt = 1 (u.t.). For small movements, the equation (2.1) can be developed in Taylor series (of first order):

$$I(x,y,t) \approx I(x,y,t+1) + \nabla I(x,y,t+1)^t \begin{pmatrix} u \\ v \end{pmatrix} \tag{2.2}$$

$$0 = I(x,y,t+1) - I(x,y,t) + \nabla I(x,y,t+1)^t \begin{pmatrix} u \\ v \end{pmatrix} \tag{2.3}$$

Hence, I_t, I_x, and I_y will be used to indicate the partial derivatives. Where

$$I_t(x,y,t+1) = I(x,y,t+1) - I(x,y,t)$$

That follows:

$$0 = I_t + I_x u + I_y v \tag{2.4}$$

Equation (2.4) is an under-determined equation that features two unknowns in one equation yielding to un infinite number of solutions. Here, the aperture problem becomes clearly visible.

As a result, the problem has to be re-formulated using additional assumptions, such as regularization (smoothness of the optical flow), with a further set of equations provided by new constraints.

Lucas & Kanade Method

The *Lucas & Kanade* [34] method is a local differential method that makes use of the spatial intensity gradient. This method was adopted as an acceptable solution to the ambiguity in the optical flow constraint equation. It assumes that the optical flow field is constant in a neighbourhood of the pixel x, adding more constraints then. The considered neighbourhood is a small n×n pixels region. Such a method allows shifting the system of equations from an under-determined one to an over-determined system by applying the optical flow constraint equation to all the pixels in the considered region, by using the least square criterion, yielding to more equations than unknowns.

Horn & Schunck Method

The *Horn & Schunck* [35] method assumes a smooth variation in the velocity of the brightness pattern almost over the entire image, thus adding more constraints that help in resolving the optical flow constraint problem. This method suffers from the fact of being a slow iterative method, moreover it is more noise-sensitive compared to other local methods.

2.2 Event-based Visual Flow

Recently, the exploitation of unmanned vehicles and robots has increased exponentially in different civilian and military applications including remote sensing, aerial surveillance, search and rescue (SAR) operations, bomb disposal, etc. However, many of the existing machines lack the necessary intelligence which requires a high speed and efficient interaction in some particular fast changing and clustered environments. Frame-based optical flow algorithms are characterized by their high computational cost accompanied by a limited speed of computation, making them inadequate for real-time operation, an essential requirement in robot navigation and autonomous vehicles. One question could be: Could the frame-based algorithms be employed for processing the output of an asynchronous event-based retina?

Frame-based optical flow computation is normally based on long timing intervals (the famous 50/60 Hz frequency for example) that makes them inadequate to be interfaced with a DVS or other event-based image sensors having a temporal resolution of 1 μs and a maximum latency of 1 ms. Therefore, some changes must be performed on the traditional algorithms, and other methods should come to light as well. The new family of algorithms could be employed directly on the output of an event-based image sensor. Some of these algorithms were inspired by previous conventional region-matching algorithms [36], others come out as variants of the traditional *Lucas & Kanade* [34] algorithm [37], and some others used local plane fits as an alternative [38].

One example of robots that make use of event-based sensors is the iCub. The iCub [39] is a humanoid robot developed at the *Italian Institute of Technology (IIT)*. It is a robot which is able to hear and see, and it is disposed with motors that move the hands, arms, head and legs, allowing the robot to interact with the surrounding environment and other humans. In [40] a biologically inspired visual attention system was developed for the iCub. The system is based on input from two event-based dynamic vision sensors (DVS), obtaining a low-latency and efficient computation. The robot exploits a C++ open-source software package that allows interconnecting the different processors, sensors and actuators of the robot, the library is called *YARP* [41] which stands for *Yet Another Robot Platform*.

The framework that was discussed in [38] presented a new method in computing visual flow. The method makes use of the precisely timed output with a microsecond temporal resolution of an event-based retina providing a redundancy-free information. The mathematical approach is a time-oriented one based on a differential computation on the spatio-temporal space of the incoming events, with the corresponding grey levels having no impact on the adopted method, and where the need of solving dynamic equations becomes irrelevant. It is based on fitting a plane to the incoming events' neighbourhood, without any need of calculating spatial or temporal gradients. It also provided a solution for large inter-frame displacements when having fast motions.

The events are generated by a 128×128 *Address-Event Representation* (AER) silicon retina [23], characterized by a very low latency. These events, when transmitted off-chip, are time-stamped before being finally communicated, in packets ($\sim$ one packet/millisecond), to a computer where the computation takes place. The algorithm has been implemented in software and different result parameters have been discussed.

2.2.1 Flow Definition

In this section, the algorithm that was adopted to calculate the optical flow will be described as in [38]:

From the output of the DVS, information relative to the pixel's position and the time stamp are considered. So, the events could be defined as $e(\mathbf{p}, t) = (\mathbf{p}, t)^t$, where $\mathbf{p} = (x,y)^t$ is the position of the pixel, and t is its relative time. A function $\boldsymbol{\Sigma_e}$ maps the time t to each $\mathbf{p}$, providing a surface of active events that are considered when the orientation and the amplitude of the motion are estimated:

$$\begin{aligned} \Sigma_e : \mathsf{N}^2 &\longrightarrow \mathsf{R} \\ \mathbf{p} &\longmapsto \boldsymbol{\Sigma_e}(p) = t \end{aligned} \tag{2.5}$$

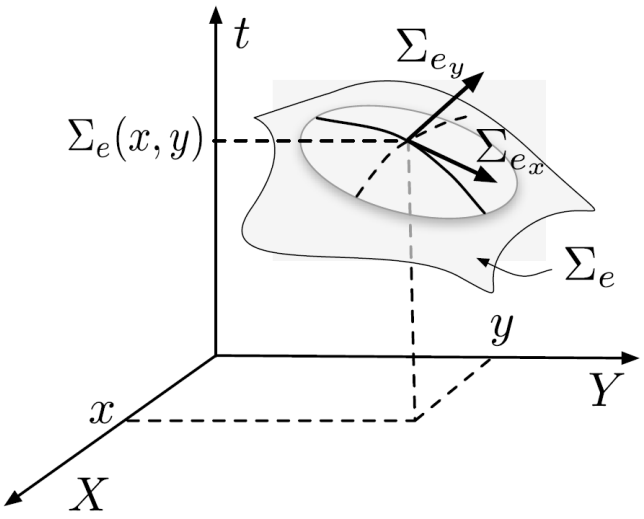


Figure 2.3: Optical flow computation [38]

Since timing is an increasing contribute, $\boldsymbol{\Sigma_e}$ is an increasing function in space with partial derivatives that could be expressed as:

$$\boldsymbol{\Sigma_{e_x}} = \frac{\partial \Sigma_e}{\partial x}$$

$$\boldsymbol{\Sigma_{e_y}} = \frac{\partial \Sigma_e}{\partial y}$$

from which it could be written:

$$\Sigma_e(p + \Delta p) = \Sigma_e(p) + \nabla \Sigma_e \Delta p + o(||\Delta p||) \tag{2.6}$$

where $\nabla \Sigma_e$ = ($\frac{\partial \Sigma_e}{\partial x}$,$\frac{\partial \Sigma_e}{\partial y}$).

Σ_e being a strictly increasing function, it has a non zero derivative at any point. Applying the inverse function theorem at p = (x,y)^t one gets:

$$\begin{aligned} \frac{\partial \Sigma_e}{\partial x}(x,y_0) &= \frac{d\Sigma_e|_{y_0}}{dx}(x) = \frac{1}{v_x(x,y_0)} \\ \frac{\partial \Sigma_e}{\partial y}(x_0,y) &= \frac{d\Sigma_e|_{x_0}}{dy}(y) = \frac{1}{v_y(x_0,y)} \end{aligned} \tag{2.7}$$

Rewriting the gradient as:

$$\nabla \Sigma_e = (\frac{1}{v_x}, \frac{1}{v_y}) \tag{2.8}$$

and having components that are the inverse of the velocity vector, it represents a measurement of the rate of change of time with respect to space, with its corresponding direction. The problem of the definition provided in (2.8) is its sensitivity to the noise, imposed by calculating pointwise partial derivatives. For this reason, the authors of [38] added a regularization process assuming a local velocity constancy. Therefore, $\boldsymbol{\Sigma}_e$ is locally planar, and a fitted plane is calculated based on the incoming events, the slope of that plane with respect to time axis is proportional to the motion velocity. In [38] a robust plane fitting was applied in a spatiotemporal region of L×L×2Δt centred on each incoming event.
At this point any event e(**p**,t) belongs to a plane Π = (a, b, c, d)^t if equation (2.9) is satisfied:

$$\Pi^t \begin{pmatrix} p \\ t \\ 1 \end{pmatrix} = 0 \tag{2.9}$$

The whole optical flow computation algorithm is shown in figure (2.4).

2.3 YARP Module

In this section the YARP module (written in C++) will be analysed and explained. The module includes the algorithm which has been adopted as a starting point for obtaining the hardware architecture the will be presented later. The following work, which includes the hardware realization of the algorithm, is in collaboration with the *Italian Institute of Technology IIT*.

The algorithm starts its computation by receiving events from the event queue, each received event must be assigned to an appropriate surface (128 $\times$ 128 pixels each). The choice is firstly related to the channel from which an event arrives. In fact, there are two channels present in this case, representing the left and right *dynamic vision sensors* (DVSs) of the robot. Secondly, for each DVS channel, two surfaces are available, here the choice is related to the polarity of the arriving event, whether it is a positive or a negative one.

After choosing the appropriate surface, nine regions centred at each neighbouring pixel around the arriving pixel are searched, controlling each time stamp of the events that are present to decide which subregion is most temporally nearby. The searching window is of 3×3 pixels in this case.

Algorithm 1 Local planes fitting algorithm on incoming events.

1: **for all** event $e(\mathbf{p}, t)$ **do**
2: Define a spatiotemporal neighborhood Ω_e, centered on e of spatial dimensions $L \times L$ and duration $[t - \Delta t, t + \Delta t]$.
3: Initialization:
- apply a least square minimization to estimate the plane $\Pi = \begin{pmatrix} a & b & c & d \end{pmatrix}^T$ fitting all events $\tilde{e}_i(\mathbf{p}_i, t_i) \in \Omega_e$:

$$\tilde{\Pi}_0 = \underset{\Pi \in \mathbb{R}^4}{\operatorname{argmin}} \sum_i \left| \Pi^T \begin{pmatrix} \mathbf{p}_i \\ t_i \\ 1 \end{pmatrix} \right|^2 \tag{6}$$

- set ϵ to some arbitrarly high value ($\sim 10e6$).

4: **while** $\epsilon > th_1$ **do**
5: Reject the $\tilde{e}_i \in \Omega_e$ if $|\tilde{\Pi}_0^T \begin{pmatrix} \mathbf{p}_i \\ t_i \\ 1 \end{pmatrix}| > th_2$ (i.e. the event is too far from the plane) and apply Eq. 6 to estimate $\tilde{\Pi}$ with the non rejected $\tilde{e}_i$ in Ω_e.
6: Set $\epsilon = ||\tilde{\Pi} - \tilde{\Pi}_0||$, then set $\tilde{\Pi}_0 = \tilde{\Pi}$.
7: **end while**
8: Attribute to e the velocity defined by the fitted plane.
9: **end for**
10: **return** $v_x(e)$, $v_y(e)$.

Figure 2.4: The OF computation algorithm as provided in [38]

Once the temporally nearby surface is chosen, the plane coefficients could be calculated mathematically, using the following approach: consider that the events belonging the subsurface are organized as discussed in what follows.

A is a 9×3 matrix containing all the spatial coordinates of the valid events on the subsurface (assuming that all events on the 3×3 surface are valid in this case):

$$A = \begin{pmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \\ x_4 & y_4 & 1 \\ x_5 & y_5 & 1 \\ x_6 & y_6 & 1 \\ x_7 & y_7 & 1 \\ x_8 & y_8 & 1 \end{pmatrix}$$

On the other hand, Y is a 9×1 matrix holding the times stamps of the corresponding events:

$$Y = \begin{pmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \end{pmatrix}$$

A plane is normally described by:

$$ax + by + ct + d = 0$$

normalizing the equation, one gets:

$$ax + by + d = -t$$

In the matrix form, the problem is solved in the form:

$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \\ x_4 & y_4 & 1 \\ x_5 & y_5 & 1 \\ x_6 & y_6 & 1 \\ x_7 & y_7 & 1 \\ x_8 & y_8 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ d \end{bmatrix} = - \begin{bmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \end{bmatrix}$$

multiplying both sides by A^t:

$$A^t A \begin{bmatrix} a \\ b \\ d \end{bmatrix} = A^t Y$$

Thus obtaining:

$$\begin{bmatrix} \sum x_i x_i & \sum x_i y_i & \sum x_i \\ \sum y_i x_i & \sum y_i y_i & \sum y_i \\ \sum x_i & \sum y_i & N \end{bmatrix} \begin{bmatrix} a \\ b \\ d \end{bmatrix} = - \begin{bmatrix} \sum x_i t_i \\ \sum y_i t_i \\ \sum t_i \end{bmatrix}$$

from which the matrix [a b d]^t is calculated as

$$\begin{bmatrix} a \\ b \\ d \end{bmatrix} = (A^t A)^{-1} A^t Y$$

where the inverse can be calculated as:

$$(A^t A)^{-1} = \frac{1}{det(A^t A)} adj(A^t A)$$

Finally a control is done on all the events to find out whether they are close enough to the plane, in this way the outliers are eliminated, and the coefficients a and b are recalculated for the remaining events. This operation could be repeated different times, but the result normally converges in 1-2 iterations. Only after that, the gradient is calculated:

$$\frac{dt}{dx} = speed \times cos(angle)$$

$$\frac{dt}{dy} = speed \times sin(angle)$$

where

$$speed = \frac{1}{\sqrt{a^2 + b^2}}$$

$$angle = arctangent(\frac{a}{b})$$

Chapter 3

HW Architecture

In this chapter the architectural choices will be discussed, as further analysis on each building block will be provided later, including details on the corresponding VHDL module. For simplicity, in what follows the dimension of the events-matrix will be assumed as 3×3 matrix, later on more details will be examined on how the dimensions of the matrix could change. The software described earlier referred to four different surfaces based on two distinct image sensors (left and right), and two opposite polarities (positive and negative). Here the design deals with events that are supposed to belong to the same surface. The design could be simply generalized and applied for processing the other surfaces when required.

The top-level entity is the one shown in figure 3.1. The architecture has essentially in input an incoming event including the coordinates X and Y, and a relative time stamp, along with a source ready signal (sready) indicating the arrival of that event . In output the architecture provides both components of the gradient, which are $\frac{dt}{dx}$ and $\frac{dt}{dy}$.

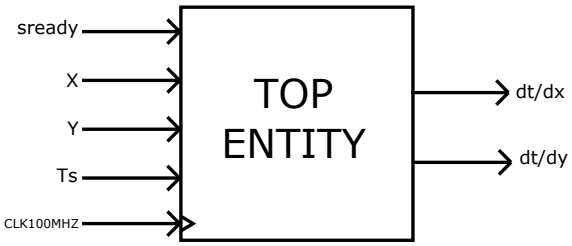


Figure 3.1: Top Entity

A more detailed block diagram is demonstrated in figure 3.2. The block diagram shows the different stages of the output computation, including:

- Memory management: concerning all the performed actions whenever a new event is received, the purpose of this stage is to store the incoming event in its correct

position on the 304×240 pixels-surface (related to the coordinate X and Y), the surface stores all the most recent events for each position for a defined interval of time. If a valid event is already present in that position it is always replaced by the most recent one.

- Matrix operations: including all the executed computations on the matrix of valid elements. Starting from the computation of the product A^tA, to its inverse, to all the required operations that lead to $(A^tA)^{-1}A^tY$ and to the coefficients a and b as a result.

- Iterations: as mentioned before, if the previous result is not satisfying, and starting from the initially provided events - given a and b - some arithmetic operations are completed to detect the outliers from the initial set of events, and then to calculate the new coefficients from the remaining events. For the matrix in consideration (3 $\times$ 3), 1-2 iterations are usually sufficient.

- Gradient computation: the final stage of the algorithm from which $\frac{dt}{dx}$ and $\frac{dt}{dy}$ are calculated using the coefficients a and b.

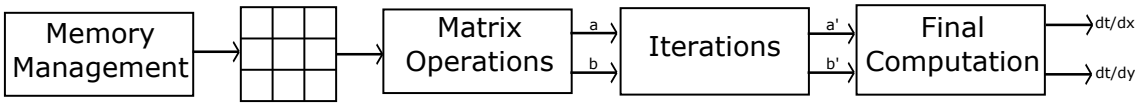


Figure 3.2: Architecture Block Diagram

3.1 Working Principle

In this section, a detailed set of information will be given on the working principle of the architecture, different architectural choices will be highlighted, and various characteristics will be exposed.

In what follows, the *Xilinx Artix-7 35T FPGA* (XC7A35TICSG324-1L) on the *Arty* board is used as a target device for the architecture, and all the initial architectural choices were based on that choice. Later on, and due to some resource restrictions, the target device choice will be replaced by a *ZedBoard Zynq-7000 board* featuring an XC7Z020-CLG484-1 FPGA to make the architecture more flexible.

The Artix-7 35T FPGA has a 100 MHz oscillator which is set to be the reference clock of the entire architecture. As a specification of the project, the timing of the incoming events is related to the characteristics of the events-source, and it reaches in the worst case an input rate of $1\ event/80ns$ (could be seen as a 12.5 MHz input source). Starting from these specifications, it could be concluded that dividing the system's reference clock

of 100 MHz and the input data clock which is of 12.5 MHz, 8 clock cycles are available to exploit some design methodologies such as data pipeline and resource sharing. The dimension of the pixels' array is of 304×240 (related to the *ATIS* [24] resolution) holding the most recent events for a well-defined interval of time, this time frame is a reconfigurable input parameter that is decided at the synthesis level as will be explained in the following sections. As for the input event, it is considered a 32-bit input with:

- 9 bits representing the X coordinate
- 8 bits representing the Y coordinate
- 14 bits representing the time stamp T_s
- one polarity bit

The size of the matrix surrounding the incoming event in this project is also a reconfigurable parameter, it has 3×3 as a lower bound, and 15×15 as an upper bound. It should also be highlighted the fact that the number of rows and the number of columns of this matrix are independent from each other, leading to a wider set of dimensions, where the desired choice depends on the target application, as it varies between different fields of utilization (indoor, outdoor, automotive, etc...).

3.1.1 Memory Addressing

In this part the adopted memory addressing mode will be exploited and justified, the choice is restricted by the memory resources which are available on the Artix-7 35T FPGA. The FPGA provides 50 block RAMs, each of 36 kbit, leading to a total of 1800 kbit of accessible memory. It supports either the use of a single 36K BRAM, or two independent 18K BRAM blocks. It also allows the usage of two independent ports when there is a need of dealing with multiple reading/writing operations. The array that has to be stored in memory is a 304×240 array, each represented by 15 bits: 14 bits for the corresponding times stamp and one additional bit that indicates whether the event is a valid one or not. Selecting the correct mode of addressing, there will be no need of storing the coordinates X and Y in the memory, but only the corresponding time stamps. Considering that the time stamps are of 14 bits, and the total available memory per block is of 36 kbit, the number of needed blocks will be an upper bound of:

$$n_{blocks} = \frac{304 \times 240 \times 16}{36000} \approx 33 \quad blocks$$

This number of blocks, approximated to the nearest even integer which will be 34, could be a suitable choice of block RAM. However, and in order to make the addressing simpler and less consuming in terms of HW resources, 38 memory blocks are used. The

choice is related to the fact that the number of rows of the array (304) is a multiple of 19 which makes the column addressing easier as will be noticed later. As a result, the pixels are distributed among 38 memory blocks, each holding the most recent event of 1920 different pixels.

Returning to the worst case in which the subsurface(matrix) surrounding the active event is of 15×15, and exploiting the availability of 8 clock cycles to read and process all the elements of this matrix (which allows a remarkable reduction in terms of the needed HW resources), reading 30 elements corresponding to two consecutive columns is the solution that has been adopted through the project. Reading 2 columns at a time means that the 38 blocks will be distributed between these two columns (19 blocks per column), and the final array distribution is the one reported in table 3.1, in which the numbers express the memory number in which each cell is stored. The distribution of 38×10 pixels is generalized to the 304×240 array.

Having this type of addressing, the pixels are then organized in the memory in 16×120 mini blocks, each composed of 19×2 pixels. And the address of each pixel is calculated as:

$$address = num_{block} + Y(7 \quad downto \quad 1)$$

where Y is the y-coordinate of the incoming event, and num_{block} varies with the coordinate x. For instance if $0 \leq x \leq 18$ then $num_{block} = 0$, if $19 \leq x \leq 37$ then $num_{block} = 1$ and so on. It can be concluded therefore that num_{block} varies between 0 and 15, depending on x.

3.1.2 Memory Accessing and Data Preparation

When a new event arrives, the 15×15 matrix must be provided at the output 2 columns at a time, in 8 consecutive clock cycles. The addresses in each cycle are calculated for y, where y is:

clock cycle	1	2	3	4	5	6	7	8
y	$y_i - 7$	$y_i - 5$	$y_i - 3$	$y_i - 1$	$y_i + 1$	$y_i + 3$	$y_i + 5$	$y_i + 7$

As it could be observed, the address for which the data must be written in memory before reading is provided either in cycle 4 or in cycle 5 with the corresponding write enable signal set active only for one specific memory. The correct is choice based on the least significant bit (LSB) of Y, if $Y(0) =$ '1' then the writing address is that of cycle 4, otherwise it is the one provided in cycle 5. For what concerns the adjacent pixels, the pixels having a coordinate $x < x_{event_i}$ will have either the same address as calculated for y shown above, or an address that is 120 locations lower. On the other side all the pixels having an $x > x_{event_i}$ will have either the same address as the central event, or a one which is 120 positions higher, according to the adopted memory addressing mode discussed before.

Table 3.1: The distribution of the first 38×10 cells among the 38 available memories.

0	19	0	19	0	19	0	19	0	19
1	20	1	20	1	20	1	20	1	20
2	21	2	21	2	21	2	21	2	21
3	22	3	22	3	22	3	22	3	22
4	23	4	23	4	23	4	23	4	23
5	24	5	24	5	24	5	24	5	24
6	25	6	25	6	25	6	25	6	25
7	26	7	26	7	26	7	26	7	26
8	27	8	27	8	27	8	27	8	27
9	28	9	28	9	28	9	28	9	28
10	29	10	29	10	29	10	29	10	29
11	30	11	30	11	30	11	30	11	30
12	31	12	31	12	31	12	31	12	31
13	32	13	32	13	32	13	32	13	32
14	33	14	33	14	33	14	33	14	33
15	34	15	34	15	34	15	34	15	34
16	35	16	35	16	35	16	35	16	35
17	36	17	36	17	36	17	36	17	3
18	37	18	37	18	37	18	37	18	37
0	19	0	19	0	19	0	19	0	19
1	20	1	20	1	20	1	20	1	20
2	21	2	21	2	21	2	21	2	21
3	22	3	22	3	22	3	22	3	22
4	23	4	23	4	23	4	23	4	23
5	24	5	24	5	24	5	24	5	24
6	25	6	25	6	25	6	25	6	25
7	26	7	26	7	26	7	26	7	26
8	27	8	27	8	27	8	27	8	27
9	28	9	28	9	28	9	28	9	28
10	29	10	29	10	29	10	29	10	29
11	30	11	30	11	30	11	30	11	30
12	31	12	31	12	31	12	31	12	31
13	32	13	32	13	32	13	32	13	32
14	33	14	33	14	33	14	33	14	33
15	34	15	34	15	34	15	34	15	34
16	35	16	35	16	35	16	35	16	35
17	36	17	36	17	36	17	36	17	3
18	37	18	37	18	37	18	37	18	37

The output of this stage is:

- an array of 30 elements of 15 bits, corresponding to the 2 columns of 15 pixels each that are being analysed
- a signal that starts the next stage which is the stage of processing

The first 15 elements of the output array are multiplexed on the outputs of the first 19 memory blocks (with index between 0 and 18), while the remaining 15 elements of the array are multiplexed on the other 19 blocks (with index between 19 and 37). In this case, the multiplexing is based on the memory number of middle row. Moreover, sending 2 columns every cycle for 8 clock cycles means that the total number of columns will reach 16 instead of 15. For this reason either the first column sent in the first clock cycle or the second column sent in the eighth clock cycle will be set to zero in order to be ignored later on in the processing stage, also in this case the choice is related to the LSB of the coordinate Y.

Validation Memory Since all the events must be kept in memory for a precise interval of time, additional memory has to be added to the architecture in order to keep track of the events. The idea is to introduce a distributed memory, working as a first-in, first-out (FIFO), that stores every single event received in the time frame taken into consideration. The time stamp is stored in the FIFO along with its coordinates X_v and Y_v, and an additional bit that indicates whether the content of the cell corresponds to a valid event or not. The FIFO has two internal counters, $read_{count}$ and $write_{count}$, each is incremented once every 8 clock cycles (related to the 12.5 MHz input data rate), keeping a constant time interval between them, which is equal to the time frame for which the events have to be stored. Therefore - and with a frequency of 12.5 MHz - if a new event has arrived, the event is written and the validation bit is set to 1, otherwise the validation bit is set to 0. In both cases the $write_{count}$ is incremented. The outputs of the validation memory are:

- X_v and Y_v: the coordinates of the pixel that has to be validated
- Ts_v: the time stamp of the pixel that has to be validated
- $check$: a signal that indicates if any control on validation has to be done. It is just a signal which is equal to the validation bit mentioned before

Once the *check* signal is asserted, the content of the pixel having $X = X_v$ and $Y = Y_v$ is compared to the time stamp Ts_v provided by the validation memory. If the contents are equal this means that the event is expired and it has to be eliminated by setting all bit - including the validation bit - to zero. Otherwise, if the contents are different, then the event in that position is more recent and no actions have to be taken. The memory address, for which the contents have to be compared, is based on X_v and Y_v and is calculated as explained previously.

3.1.3 Computation of A^tA and A^tY

At this point of the project, A^tA and A^tY have to be calculated properly as

$$\begin{bmatrix} \sum x_i x_i & \sum x_i y_i & \sum x_i \\ \sum y_i x_i & \sum y_i y_i & \sum y_i \\ \sum x_i & \sum y_i & N \end{bmatrix} and \begin{bmatrix} \sum x_i t_i \\ \sum y_i t_i \\ \sum t_i \end{bmatrix} respectively.$$

As it can be noticed, the 3×3 matrix to the left is a symmetric one for which it would be enough calculating 6 elements out of the 9. As previously specified, this block receives an array of 30 elements every clock cycle (two consecutive columns). Since the array contains only time stamps for now, each of these events receives a unique pair of coordinates X and Y ($0 \leq (x,y) \leq 14$ in the 15×15 case) to start composing the matrices A and Y. For each element, if the validation bit is set to '1' then the element will have its coordinates x and y along with its time stamp added to the final result, otherwise they will be ignored and will have no effect on the final result. As a result, each of the 30 elements with x_i, y_i and a time stamp ts_i - when it is valid - will have the following contributes calculated in parallel: x_i^2, y_i^2, $x_i y_i$, $x_i ts_i$ and $y_i ts_i$. After that, all the contributes will be added, and the result will be stored in order to be summed with the related events that will arrive in the consecutive clock cycles. Besides calculating A^tA and A^tY, this block, when receiving the different columns of the subsurface, stores all the elements in a 15×15 array. This operation is necessary for filtering the outliers in following iterations, as will be described subsequently.

3.1.4 Computation of the Adjoint of A^tA, a_{det}, b_{det}, and det

Once the matrix $\begin{bmatrix} \sum x_i x_i & \sum x_i y_i & \sum x_i \\ \sum y_i x_i & \sum y_i y_i & \sum y_i \\ \sum x_i & \sum y_i & N \end{bmatrix}$ is ready, the next step is to calculate the adjoint matrix $adj(A^tA) = \begin{bmatrix} adj_{11} & adj_{12} & adj_{13} \\ adj_{21} & adj_{22} & adj_{23} \\ adj_{31} & adj_{32} & adj_{33} \end{bmatrix}$ which is a symmetric matrix as well, this means that only the following elements - which are relevant for the computation of a and b - have to be calculated as:

- $adj_{11} = N \sum y_i^2 - \sum y_i \sum y_i$
- $adj_{12} = \sum x_i \sum y_i - N \sum x_i y_i$
- $adj_{13} = \sum x_i y_i \sum y_i - \sum x_i \sum y_i^2$
- $adj_{22} = N \sum x_i^2 - \sum x_i \sum x_i$
- $adj_{23} = \sum x_i y_i \sum x_i - \sum x_i^2 \sum y_i$

After that, the architecture progresses by multiplying the first two rows of $adj(A^tA)$ by A^tY, obtaining the two elements a_{det} and b_{det} that will be divided later by the determinant of A^tA (det) to get the coefficients a and b:

- $a_{det} = adj_{11} \sum x_i t_i + adj_{12} \sum y_i t_i + adj_{13} \sum t_i$
- $b_{det} = adj_{12} \sum x_i t_i + adj_{22} \sum y_i t_i + adj_{23} \sum t_i$
- $det = adj_{11} \sum x_i^2 + adj_{12} \sum x_i y_i + adj_{13} \sum x_i$

3.1.5 Iterations and Recomputation of a and b

Starting from the coefficients a and b previously calculated, the events that have been stored in an early stage are verified in a defined number of iterations, which is introduced as a parameter at the synthesis stage. Consider a 15×15 matrix, a pair of columns is taken into consideration at a time in 8 different clock cycles according to the following comparison:

$$a(x - x_c) + b(y - y_c) - (t_s - t_{sc}) < th$$

where x and y are the coordinates of the event that has to be validated and t_s is its corresponding time stamp, whereas x_c, y_c and t_{sc} are the parameters related to the central event of the matrix in consideration. On the right of the inequation, the parameter th is a threshold set at synthesis time and can be modified based on the application in operation. The events that verify the inequation passes through to the next stage, while the other events will have their validation bit set to zero. After that, the computation of A^tA, A^tY, and of $adj(A^tA)$ after them, and finally of a_{det}, b_{det} and det, is executed exactly as before.

3.1.6 Final Computation

The coefficients a and b are calculated using a *Xilinx IP Divider Generator*, which characteristics will be specified in the next section. They are simply calculated as:

$$a = \frac{a_{det}}{det} \quad and \quad b = \frac{b_{det}}{det}$$

The remaining steps are executed using the *Xilinx IP* core that implements the *coordinate rotational digital computer* (*CORDIC*) that allows the exploitation of the following functions that are required by the algorithm:

- $\theta = arctan(\frac{a}{b})$
- $\sqrt{a^2 + b^2}$
- $sin(\theta)$ and $cos(\theta)$

while the term $\frac{1}{\sqrt{a^2+b^2}}$ is computed using *Xilinx IP Divider Generator*.

Finally the following products could be evaluated:

$$\frac{dt}{dx} = speed \times cos(\theta)$$

$$\frac{dt}{dy} = speed \times sin(\theta)$$

3.2 Detailed Hardware and VHDL Description

In this section more details will be provided on the hardware and the VHDL source code. The VHDL code of each block will be investigated, and the specifications of each block in terms of working principle, *data pipeline*, *resource sharing* and *latency* will be highlighted. Through out this section, the worst case, in which processing will be done on a 15×15 matrix, is taken into consideration. When the latter is satisfied, in the other cases in which smaller matrices are needed, the architecture guarantees the correct behaviour, even though some code optimizations would be possible for each specific configuration. It must be highlighted that *Vivado Design Suite 2016.1* is used for the synthesis and analysis of the architecture.

The architecture has these parameters as reconfigurable ones that have to be set at synthesis stage.

Parameter	Description
ROW	the number of rows of the desired output matrix
COL	the number of columns of the desired output matrix
min_events	the minimum number of valid events on a matrix for which it is considered a valid one
n_iter	the required number of iterations. It can be set to 0, 1, 2 or 3 depending on the matrix dimension and the required accuracy
$time\_int$	the time interval for which the events are considered valid in the memory

RD_MTX

A macro block that consists of three different parts.

1. MEM_READ
2. 38 block RAM
3. VLD_MEM

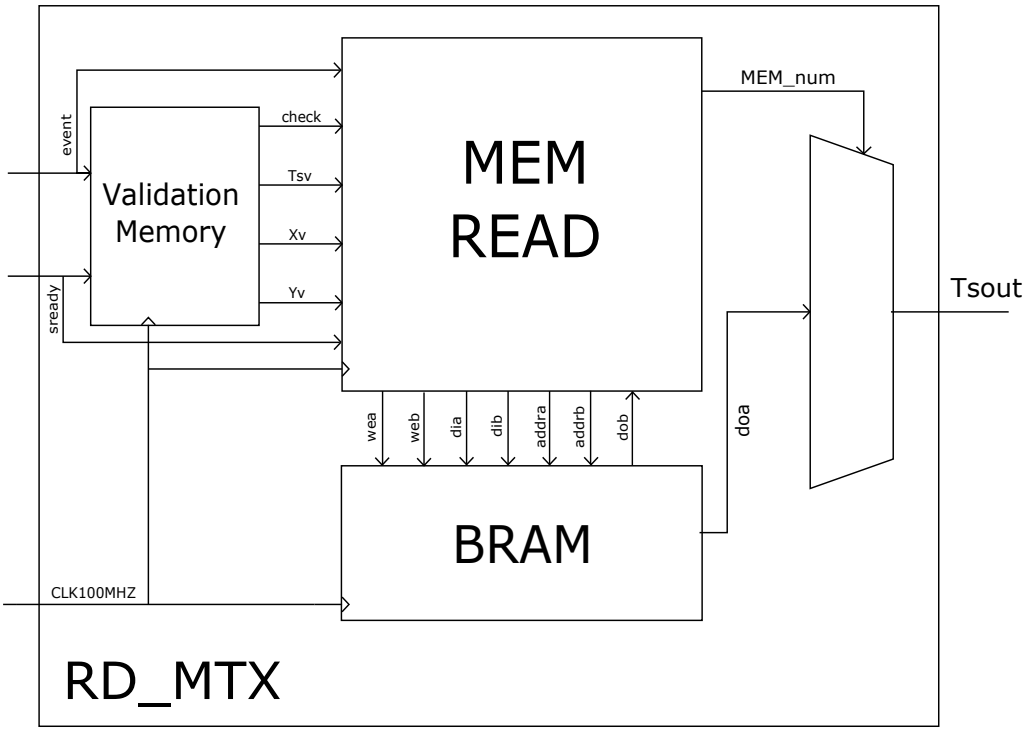


Figure 3.3: RD_MTX Schematic

The schematic of the MEM_READ block is shown in figure 3.3.
The ports of this blocks are:

- $sready$ (input) - active high for one clock cycle when a new events arrives
- X, Y and Ts (input)
- $Tsout$ (output) - a 30 elements array corresponding to all the items of two consecutive columns. Each elements is of 15 bits, 14 corresponds to the time stamp, and one validation bit that indicates whether the event is valid or not
- $start$ (output) - signal that goes active (high) when the first two columns of Ts_{out} are ready for the next blocks (MULT). Once activated, this signal remains active for 8 consecutive clock cycles in the case of an output matrix of 15 columns has to be provided. If the output matrix is of 3 columns for example, this signal remains active only for 2 clock cycles, since 2 cycles are enough to read 3 columns.

There are two main reasons regarding the choice of reading two columns at a time instead of reading all the elements of the matrix at once. Mainly for the 15×15 case:

1. It would not be possible reading all the elements of the matrix due to memory access limitations related to the adopted addressing methodology.

2. Reading all the elements of the matrix means that all the read events must be processed in parallel, which is a hardware expensive solution in terms of LUT and DSP slices. The problem is more evident for larger matrix dimensions.

MEM_READ The block has in input:

- 9 bits X_i, 8 bits Y_i and 14 bits Ts_i: X_i and Y_i are the coordinates of the input event, and Ts_i is its time stamp
- $sready$ is a signal that indicates the arrival of new event to the architecture
- 9 bits X_v, 8 bits Y_v and 14 bits Ts_v: in this case X_v and Y_v are the coordinates of the event that has to be verified if expired, Ts_v is the time stamp that has to be compared with the current one present in that position
- $check$: active high signal: signals the expiry of an event, for which verification has to be performed
- dob: the outputs of port B for all the 38 memories, port B is the BRAM port on which the verification phase takes place. It allows reading the memory content that has to be verified. Each elements of the 38 is of 15 bits.

The outputs of the block are:

- $MEMnum$ - number that varies between 0 and 18 since the memories are organized in 19 rows, this number is used to multiplex the outputs of the memories in order to get the correct output.
- $addra$ and $addrb$ - 11 bit addresses that are provided by this block to the memories. As stated previously, port A of the BRAM memories is used to write the new data in the correct location and to read the output matrix, while port B is used in the expiry validation stage in which an event is firstly read and secondly written (cancelled) in case of expiry.
- $dina$ and $dinb$ - 15 bit inputs of the different memories for both ports
- wea and web - active high write enable signals for both memory ports, activated for a specific memory when an event has to be written.

The table below shows how y changes with respect to y_i among the clock cycles. For each y, an address is calculated according to the formula previously stated.

clock cycle	1	2	3	4	5	6	7	8
y	$y_i - 7$	$y_i - 5$	$y_i - 3$	$y_i - 1$	$y_i + 1$	$y_i + 3$	$y_i + 5$	$y_i + 7$

After that, the different addresses are generated for the memories depending on MEM_num. Suppose that the address that has been generated for a certain y is $address$, then the one that will be assigned to each memory will be one of the following: $address$, $address + 120$, or $address - 120$ according to the position of each memory with respect to the central one. A simple example for a 5×5 is shown in the figure below (fig. 3.4), for a selected pair of columns.

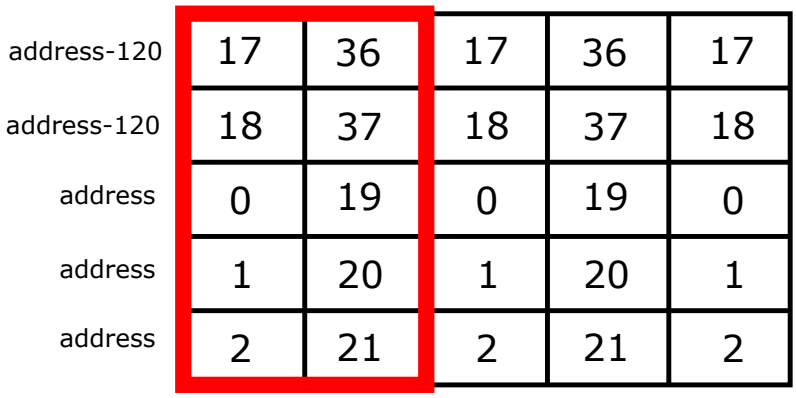


Figure 3.4: Addresses generation for a 5×5 matrix

Regarding the write enable signal wea, one of the 38 write enables (one for each memory) will be activated exclusively for one clock cycle whenever $y(7 \quad downto \quad 1) = y_i(7 \quad downto \quad 1)$, and the choice between the left and the write columns of memories depends on the LSB of y_i, since both cells corresponding to a certain row have the same MEM_num value as can be deduced from 3.1.

VLD_MEM Related to the validation memory which working principle has been described earlier. The inputs of this block are:

- $sready$ - active high that indicates that a new event is ready to be written
- evt - 32 bit input event

And the outputs:

- X_v and Y_v - the coordinates of the event to be validate
- Ts_v - time stamp of the event to be validated
- $check$ - indicator for MEM_READ when a validation is required, it is an active high pin

At the HDL level, the validation in managed in the following steps:

1. When the $check$ signal is asserted, the coordinates of the cell that has to be validated X_v and Y_v are provided to the $MEM\_READblock$
2. The address for those coordinates is calculated and provided to the all the memories
3. Only the output of the specific memory corresponding to that cell is taken into consideration and is compared to the time stamp Ts_v
4. Only if the time stamps match, the event is eliminated, otherwise no actions are taken

3.2.1 MULT

The $MULT$ block has the job of calculating A^tA and A^tY. Figure 3.5 shows how the blocks RD_MTX, $MULT$, and $Adjoint$ are connected. In input it has the following signals:

- $start$ - a signal that arrives from RD_MTX in order to start the computation
- sum - when $start$ is active (for 8 clock cycles), this signal is set to 0 in the first clock cycle, and then it's activated for the remaining 7 clock cycles. It simply indicates whether to take or not the result of a previous stage, since the addition is performed in a loop over 8 clock cycles.
- $y0$ - the least significant bit of the Y of the event for which the computation is being executed, it implies whether to take into consideration or ignore, the first column in the first cycle and the second column in the 8th clock cycle when saving the matrix in $sur15$ and $sur15\_2$, or when performing the computation

The outputs of this blocks are:

- $\sum x^2$, $\sum y^2$, $\sum xy$, $\sum x$, $\sum y$, $\sum xt$, $\sum yt$, $\sum t$, and N. They represent the components of matrices A^tA and A^tY, and their corresponding VHDL notation is: sum_x2, sum_y2, sum_xy, sum_x, sum_y, sum_xt, sum_yt, sum_t and N. The dimensions of these signals are shown in the table 3.2.
- go - active high - signals to the subsequent block to start its computation, based on the outputs of this stage
- $sur15$ and $sur15\_2$ - two dimensional arrays where the input matrices are stored for the next iteration. The reason of the presence of two matrices is that when two consecutive computations - for two distinct events (with an inter-event time interval of 8 clock cycles) - have to be executed, the new arriving matrix could not be stored in $sur15$ since the matrix that is already present their has not been used yet by the iteration block. For this reason, the new matrix will be stored in $sur15\_2$ instead

Another parameter that could be set at synthesis time is min_events, it simply indicates the minimum number of valid events on a plane in which the matrix is considered valid, and therefore the next processing stages are called. The worst case in which all the events in the 15×15 matrix are valid has been considered in this project. However, the number of bits of each element varies with the dimension of the matrix as demonstrated in the table below (table 3.2). The estimation of number of bits - in the case 15×15 - is done starting from X and Y that are represented in 4 bits each, where they vary between 0 and 14, in which the absolute value of X and Y is not relevant in the final computation, but the relative difference between the matrix elements is the term that counts when dealing with derivatives. This choice reduces drastically the consumption of the architecture in terms of hardware resources, since it implies dealing with internal signals with a narrower bit-widths.

Table 3.2: Variation of the number of bits (worst case) for the elements of A^tA with the matrix dimension

matrix dimension	$\sum x^2$	$\sum y^2$	$\sum xy$	$\sum x$	$\sum y$	$\sum xt$	$\sum yt$	$\sum t$	N
15×15	14	14	14	11	11	25	25	22	8
13×13	14	14	14	10	10	24	24	22	8
11×11	13	13	13	10	10	24	24	21	7
9×9	11	11	11	9	9	23	23	21	7
7×7	10	10	10	8	8	22	22	20	6
5×5	8	8	8	6	6	20	20	19	5
3×3	4	4	4	4	4	18	18	18	4

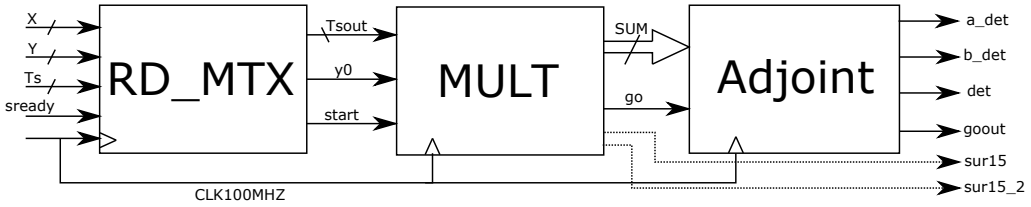


Figure 3.5: A schematic showing how the first set of RD_MTX, $MULT$ and $Adjoint$ blocks are connected

3.2.2 adjoint

The adjoint block completes the calculation of a_det, b_det, and the determinant det starting from the adjoint matrix of A^tA. The computation is performed in 3 consecutive clock cycles, in order to meet the timing requirements of the different adders and multipliers that are included in the computation :

1. computation of the adjoint matrix of A^tA
2. computation of partial products a_{det1}, a_{det2} , a_{det3} , b_{det1} , b_{det2} , b_{det3}, det_1, det_2, and det_3
3. summation of the different terms:
   - a_{det} = a_{det1} + a_{det2} + a_{det3}
   - b_{det} = b_{det1} + b_{det2} + b_{det3}
   - det = det_1 + det_2 + det_3

This block has in input and output the following signals:

- sum_x2, sum_y2, sum_xy, sum_x, sum_y, sum_xt, sum_yt, sum_t and N in input
- go - active high input that arrives from the $MULT$ block to start the computation
- a_det, b_det, and det in output
- $gout$ - a signal that is asserted whenever the computation ot the outputs is terminated, and is sent to the successive block

As it can be noticed, the product $(adj(A^tA))(A^tY)$ is calculated instead of dividing $(adj(A^tA))$ by the determinant of the matrix, which results in getting the inverse matrix of A^tA, as specified in the algorithm. This choice is related to the hardware limitations in which calculating the inverse of A^tA would require using at least 6 dividers, thus consuming more resources. While multiplying $(adj(A^tA))$ by (A^tY), then dividing after that requires only two divisions.

3.2.3 Iteration Blocks

The figure 3.6 and 3.7 show two different implementations of the iteration phase for two cases of n_iter.

The iteration block stores one of the two matrices that are provided by the $MULT$ block, and starts the iteration in order to exclude the outliers from the final computation. The validation of the events is performed in three clock cycles (due to some timing requirements of the different blocks: comparators, adders, multipliers, etc...) as follows:

1. the terms $a(x - x_c)$, $b(y - y_c)$, and $(t - t_c)$ are calculated for each valid events in the matrix. Since x and x_c, as well as y and y_c, are well defined through the entire matrix, then there is no need to introduce any adders when subtracting $(x - x_c)$ and $(y - y_c)$, where these subtractions depend only on the relative position of each event with respect to the central one

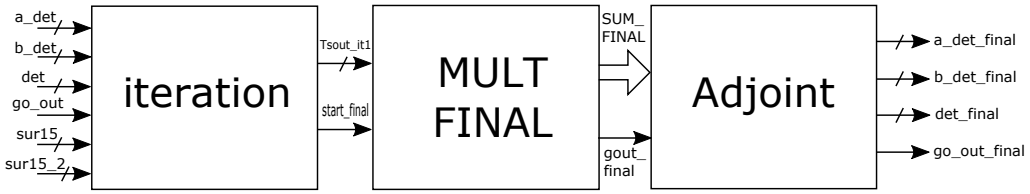


Figure 3.6: A schematic demonstrating the components of the iteration block when the parameter $n\_iter = 1$

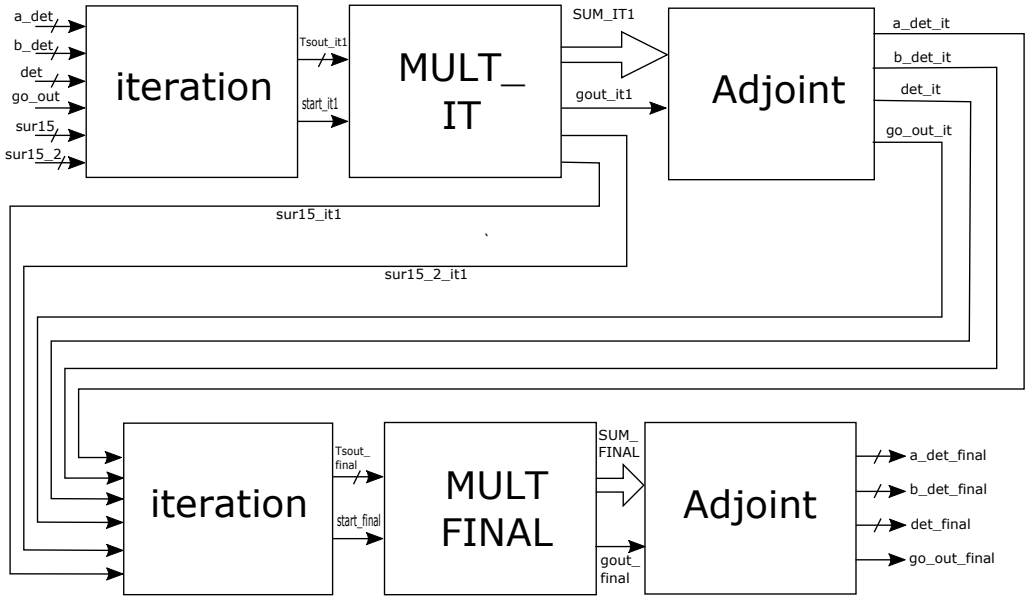


Figure 3.7: A schematic demonstrating the components of the iteration block when the parameter $n\_iter = 2$

2. the absolute value of $a(x - x_c)$ + $b(y - y_c)$ - $(t - t_c)$ is computed for each valid event

3. the values computed in cycle 2 are compared to a certain threshold, and only those satisfying the inequation are approved for the next processing stage

At the end of each iteration the new values of a_det_final, b_det_final, and det_final, are computed exactly as in the first case, by passing the new matrices to the $MULT$ and $adjoint$ blocks, respectively. A distinction is made between $MULT\_it$ and $MULT\_FINAL$ blocks only because the block $MULT\_it$ - in the case where $n\_iter = 2$ - has to store the

input matrix for further processing in the second iteration. Hence, all the data processing is identical between the two blocks.

3.2.4 Divider

The next step is to calculate the coefficients $a = \frac{a_{det}}{det}$ and $b = \frac{b_{det}}{det}$ using the *Xilinx IP Divider Generator*. The objective was to decide between the different possibilities offered by the divider generator, and to choose the most suitable solution namely between these two:

1. *Radix-2* solution - it employs the FPGA logic (registers and LUTs) to obtain different throughput options, reaching a single cycle if required. It is based on integer division and is suggested either when a high throughput is required or when the operands width is around 16-bits. Moreover, since the proposed solution does not require block RAM or DSP primitives, this solution might be useful when those resources are required elsewhere.

2. *High Radix* solution - it exploits block RAMs and DSP slices, it is the recommended when the widths of the operand are greater than 16. This solution is based on a prescaling algorithm in an iteration loop, this means that once this block starts the computation, any new input must wait until the previous calculation has terminated, getting a throughput which is lower than the previous solution

Both solutions were tested for the normalized 56-bit dividend and a 40-bit divisor, with a 16-bit fraction at the output. In order to satisfy the timing requirements (positive slack), either a 40 clock cycles latency with a single cycle throughput Radix-2 divider, or a High Radix divider with a latency of 16 clock cycles (which is equal to the throughput in this case), must be introduced. Since the inputs of this block arrive once every 8 clock cycles, adopting the *High Radix* division will require the use of 4 dividers instead of a single one, the first two are used in order to divide a_{det} and b_{det} in parallel, and the other couple of dividers is used in the worst case in which 8 clock cycles later, another pair of inputs arrives, and this pair is divided in a correct way.

On the other hand, the *Radix-2* solution, which has been adopted in this project, allows to get the maximum throughput with a latency of 40 clock cycles. The inputs a_{det}, b_{det}, and det arrive at the same time. As soon as the inputs arrive, in the first clock cycle a_{det} is fed to the divider as a dividend, and det is fed as the divisor, while in the second clock cycle the dividend is switched to b_{det}. Therefore, the coefficients a and b are obtained in consecutive clock cycles. The divider block has the following interface signals:

- $dividend\_valid$ and $divisor\_valid$ active high signals in input to tell when a new dividend or divisor is valid, respectively

- 56-bit $dividend$ and a 40-bit $divisor$ in input

- $dout\_data$ an output of 32-bit, including a 16-bit fractional part and a 16-bit integer part
- $dout\_valid$ active high signal indicating when a new output is valid

After obtaining the coefficients a and b, two parallel computations are executed as shown in figure 3.8. The first regarding the $\theta = arctan(\frac{a}{b})$ and $sin(\theta)$ along $cos(\theta)$ after that. The other concerning the calculation of the term $\frac{1}{\sqrt{a^2+b^2}}$.

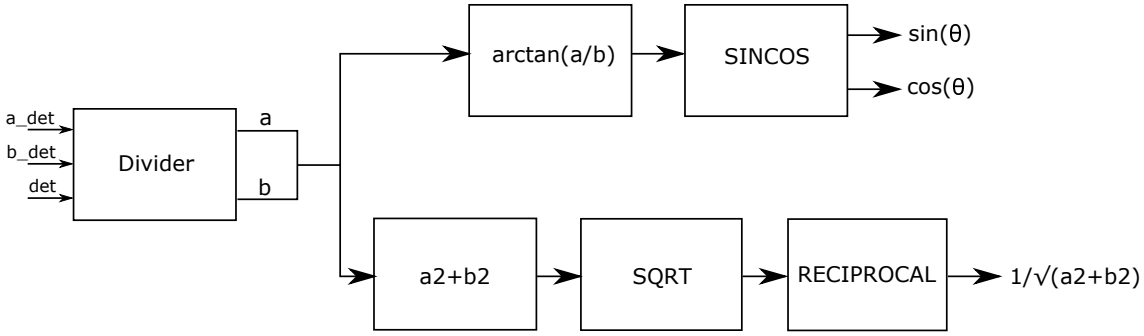


Figure 3.8: Data propagation in two different processing directions

3.2.5 Arctangent

The arctangent of the ratio $\frac{a}{b}$ is computed using a Xilinx IP implementing a *coordinate rotation digital computer* (CORDIC) algorithm. The architectural configuration of the block is set to parallel in order to obtain a high throughput with a maximum pipelining. The block has a 20 clock cycles latency, and the output is expressed in 16 bits fixed-point twos complements number with 13 bit fractional part and the remaining 3 bits for the integer part. It has a 64 bits input, where the input coefficient a is placed in the most significant part *(63 downto 32)* and b on the least significant part *(31 downto 0)*. At the output, an $atan\_v$ signal is asserted whenever an output is ready.

3.2.6 Sine and Cosine

Successively, once obtaining the arctangent result of the previous stage, the function of the $SINCOS$ block is to compute the sine and the cosine of the angle θ. The arctangent is passed to the input through a 16-bits vector and the output is obtained on a 32-bits vector, where the most significant part belong to the cosine of the angle and the least significant part features the sine of the angle, both expressed in radians in the 1QN format (2-bits integer, 14-bits fractional). This CORDIC block has a latency of 20 clock cycles, with a maximum throughput guaranteed through a parallel architectural configuration.

3.2.7 Square Root

The $SQRT$ block, implemented through the *Xilinx IP CORDIC* algorithm, starts the computation of $(\sqrt{a^2+b^2})$ in parallel with the arctangent block, after obtaining the term a^2+b^2 in two consecutive clock cycles, in which in the first clock cycle the terms a^2 and b^2 are calculated, and in the second clock cycle the two terms are added. The input a^2+b^2 is expressed in 32 bits, and the output in 17 bits representing the integer part, and the latency of this block is of 17 clock cycles

3.2.8 Reciprocal

The final step that remains before obtaining the final result is to calculate the reciprocal of the term $(\sqrt{a^2+b^2})$. It was achieved by introducing another divider, adopting the Radix-2 solution as before, and by setting the the dividend width to 2 bits (since it is constant and equal to 1), and the divisor to 17-bits width. The output is expressed in 19 bits with 2 bits for the quotient part and 17 bits for the fractional part. The latency of this block is of 10 clock cycles.

3.2.9 Final Computation

The total delays in the trigonometric part (calculating sine and cosine), and in the arithmetic part ($\frac{1}{\sqrt{a^2+b^2}}$) are:

$$t_{trig} = t_{arctan} + t_{sincos} = 20 + 20 = 40 \quad clock \quad cycles$$

$$t_{arith} = t_{a^2,b^2} + t_{(a^2+b^2)} + t_{SQRT} + t_{RECIPR} = 1 + 1 + 17 + 10 = 29 \quad clock \quad cycles$$

This implies that the result of the arithmetic part arrives 11 clock cycles earlier than that of the trigonometric part. Hence, pipelining the output of the arithmetic part for 11 clock cycles is required in order to compute the products $\frac{1}{\sqrt{a^2+b^2}}sin(\theta)$ and $\frac{1}{\sqrt{a^2+b^2}}cos(\theta)$ correctly.

Chapter 4

Synthesis and Simulation Results

4.1 Simulation Results

In the following section the results of the simulation will be demonstrated, illustrating the main steps of processing for each block. The architecture was fed by the inputs provided in the table below (table 4.1), in which for each event the coordinates X and Y are stated along with the corresponding time stamp. The final set of events where the processing example takes place is shown in figure 4.1. The inputs - as can be observed - are provided to a limited region in order to have enough events in the neighbourhood which can be considered during the processing stage, giving more reasonable results. The main parameters of the architecture have been set as:

- matrix dimension: 5×5 ($row = 5$, $col = 5$)
- min_events set to 0 in order to start the processing stage for all the inputs that are provided, and be able to discuss the obtained results.
- the threshold for which the events pass the iteration stage is set appropriately, allowing all the events to pass this stage. An example in which this threshold is lowered, filtering some events, will be provided later.

4.1.1 Memory Accessing

Figure 4.2 shows some aspects of the initial stage of memory accessing and some data processing. The $sready$ signal is provided with the inputs X, Y and T_s. MEM_num is the memory number that varies between 0 and 18 depending on the input coordinate X. For example, for the first input which has an $X = 93$, the signal simply exploits the modulo function as: $MEM\_num = 93 \mod 19 = 17$, while the LSB of Y indicates whether the memory belongs to the first set of memories (indexed from 0 to

Table 4.1: Table showing the first 18 inputs that have been provided to the architecture

Input	X	Y	T_s
1	93	189	6100
2	92	191	5983
3	96	189	7196
4	95	191	7200
5	94	187	7240
6	95	188	7307
7	97	190	7400
8	94	191	7430
9	94	190	7470
10	94	189	7502
11	94	188	7540
12	95	191	7590
13	97	190	7607
14	97	189	7700
15	97	188	112
16	96	191	107
17	95	190	104
18	95	189	7800

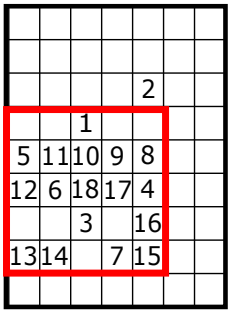


Figure 4.1: Simulation: first 18 inputs with the output window around the last received event

18), or the other set (19 to 37). Mem_num decides where to take the different outputs from. $addr_vector$ are the addresses that are sent for the different memories. As it can be observed, different addresses are provided in the three consecutive clock cycles (required in the case of 5×5 matrix), and the wea signal is only activated in the second cycle and only for one specific memory, where the new event must be written, and read after that.

The addresses that are calculated for the first two inputs are shown in figure 4.3 (shown

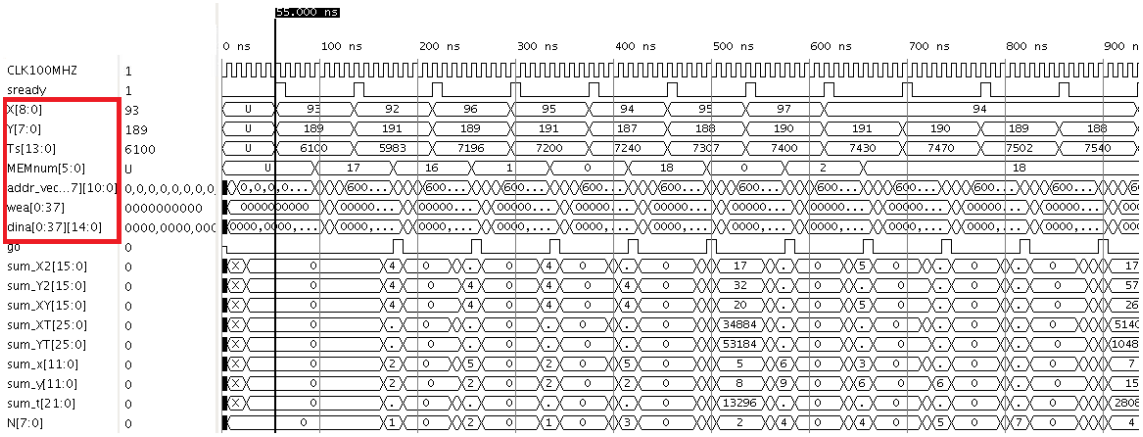


Figure 4.2: Simulation results: input

for the first 19 memories since address(MEM[i+19]) = address(MEM[i])). The addresses are calculated corresponding to the equation that has been reported previously, where $num_{block} = 4$ and $Y(7 \quad downto \quad 1) = 94$, therefore $address = 574$ for the central event. Since the last row of the matrix must be read from the subsequent block of memories, the address corresponding to that row would be $address + 120$ as explained previously, and as can be observed in the waveform.

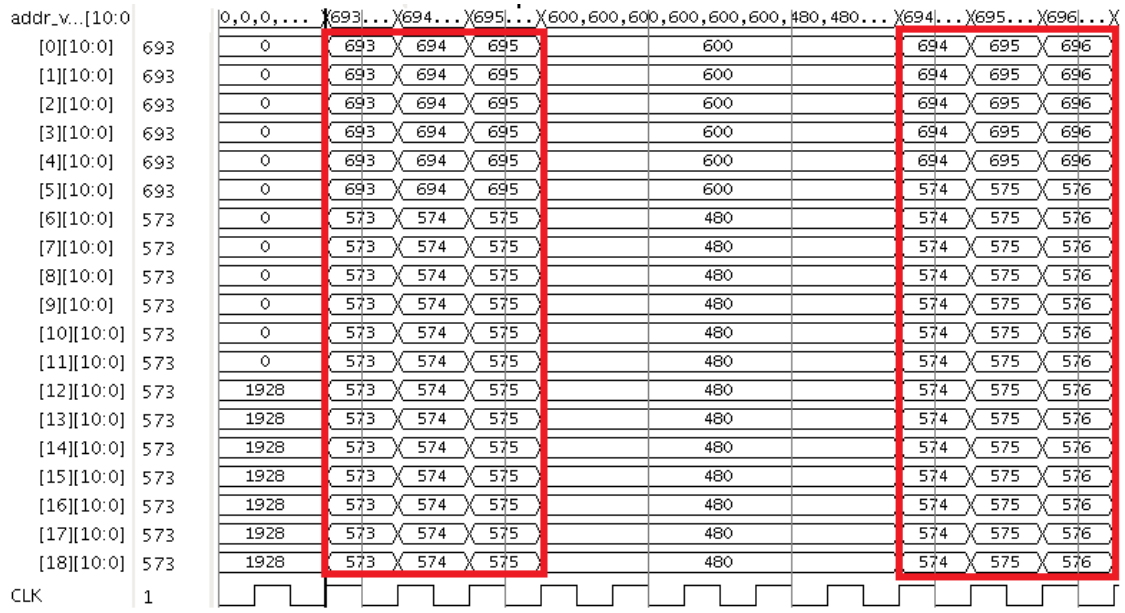


Figure 4.3: Simulation results: address generation

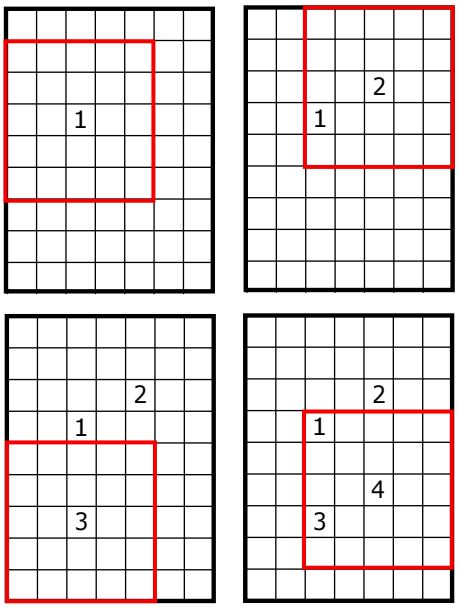


Figure 4.4: Simulation: first 4 inputs and their corresponding output window

4.1.2 Processing Stage

The figure below (fig. 4.5)shows how are the first 4 matrices - related to the first 4 inputs - saved in the $sur15$ and $sur15\_2$ for the iteration stage. The inputs are saved alternatively between the two matrices as can be noticed.

The output matrix corresponding to the 18th input is the one shown in figure 4.1. As can be noticed, the matrix includes 17 out of 18 events that are provided, where the second event is not taken into consideration since it does not belong to the 5×5 region. The outputs of the $MULT$ block are highlighted, they represent both the A^tA and the A^tY matrices, defined as follows:

$$A^tA = \begin{bmatrix} \sum x_i x_i & \sum x_i y_i & \sum x_i \\ \sum y_i x_i & \sum y_i y_i & \sum y_i \\ \sum x_i & \sum y_i & N \end{bmatrix} = \begin{bmatrix} 107 & 80 & 37 \\ 80 & 110 & 36 \\ 37 & 36 & 17 \end{bmatrix}$$

$$A^tY = \begin{bmatrix} \sum x_i t_i \\ \sum y_i t_i \\ \sum t_i \end{bmatrix} = \begin{bmatrix} 210369 \\ 184061 \\ 103405 \end{bmatrix}$$

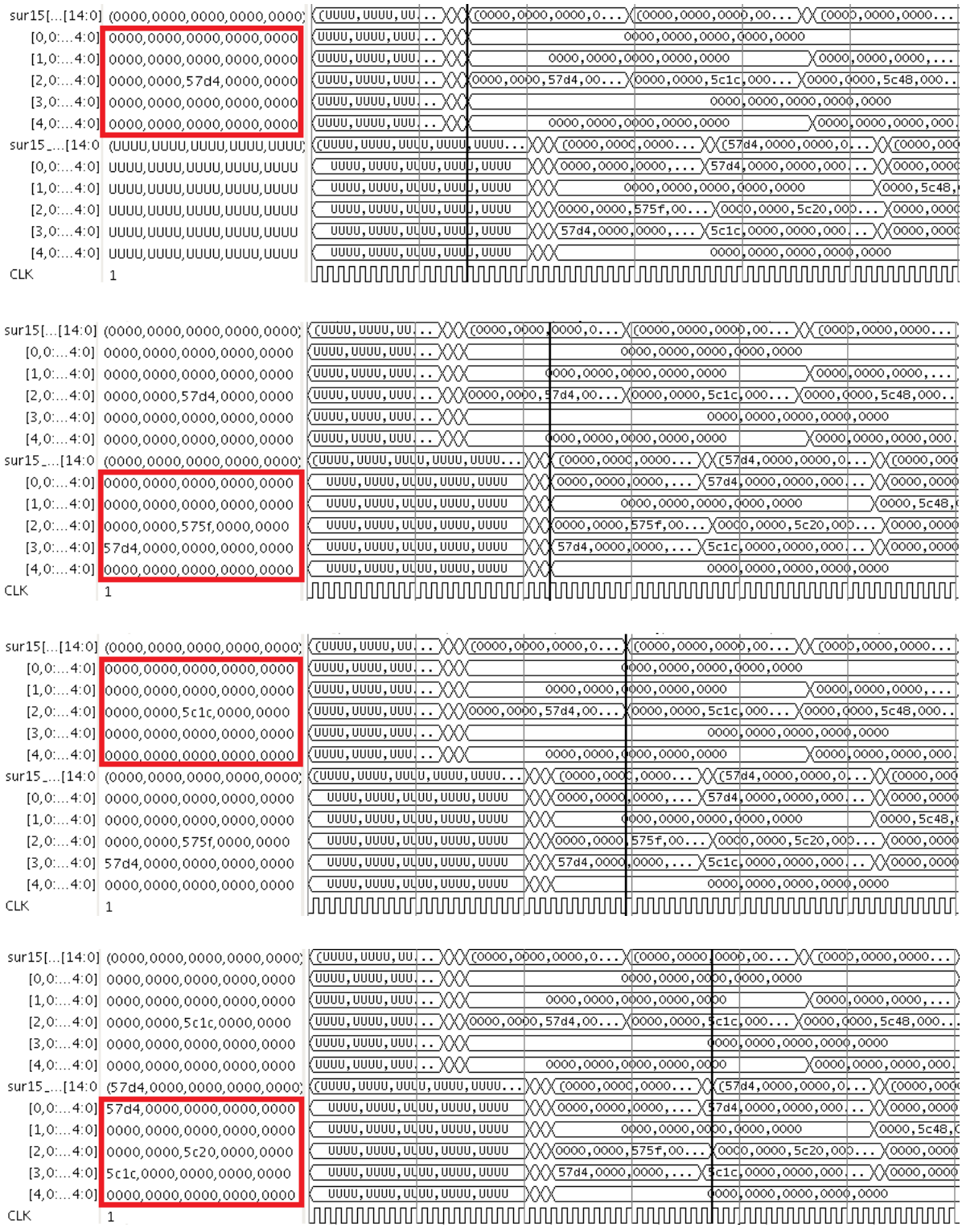


Figure 4.5: Simulation results: $sur15$ and $sur15\_2$ in 4 consecutive cycles

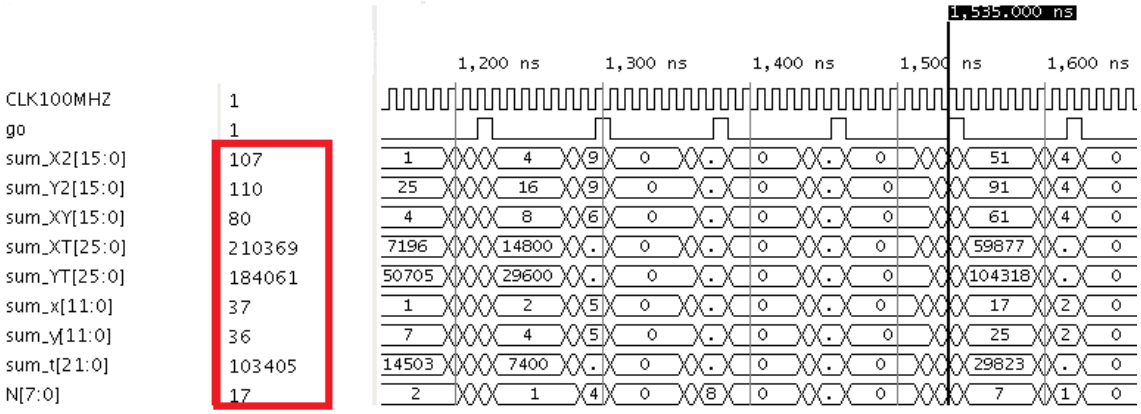


Figure 4.6: Simulation results: output of $mult$ block

The signal go indicates when the output of the $MULT$ block is ready for the successive processing. Once, the outputs are ready they are fed to the adj block responsible of calculating a_det, b_det and det as shown in figure 4.7. The outputs for the input matrices A^tA and A^tY expressed before are:

- $a\_det = -7453852$
- $b\_det = -15300142$
- $det = 15148$

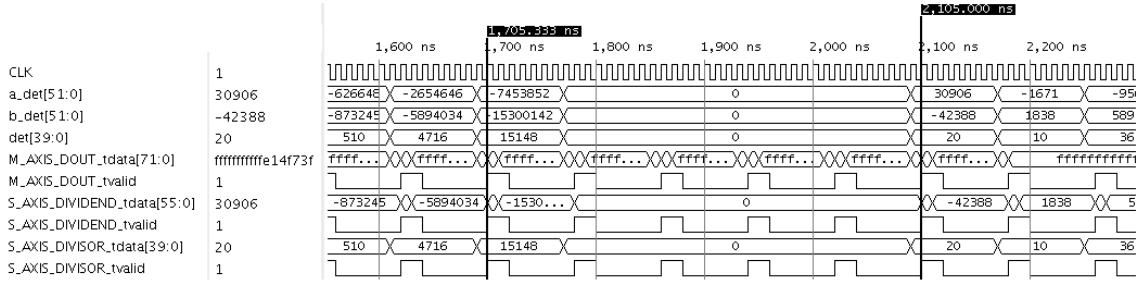


Figure 4.7: Simulation results: output of adj block

The next step would be to divide a_det and b_det by the determinant det, the results of this operation are shown in hexadecimal in figure 4.8, the useful bits are 32 bits, 16 representing the integer part, and the other 16 express the fractional part. The results are provided 40 clock cycles after, and they are as expected, where $a \approx -492$ and $b \approx -1010$.

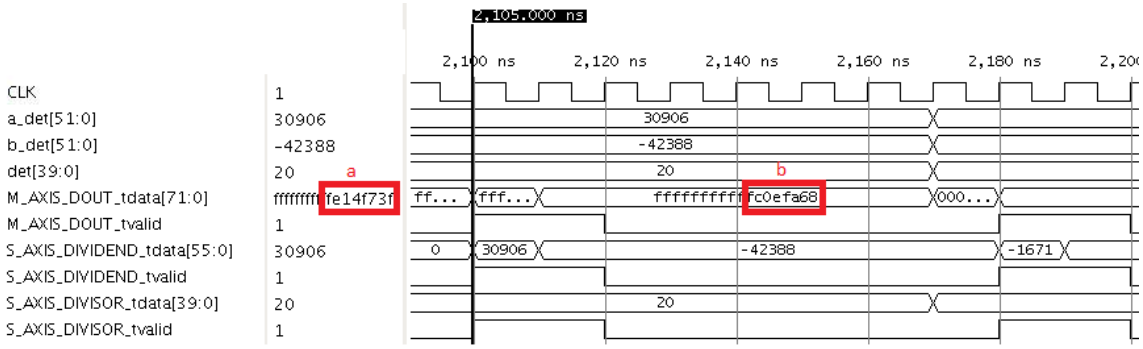


Figure 4.8: Simulation results: output of *divider* block

Once a and b are ready, two parallel computations must take place as explained before. The first regarding the computation θ as the arctangent of $\frac{a}{b}$, then calculating $sin(\theta)$ and $cos(\theta)$. a and b are provided in input of the arctangent block, a on the MSB 32 bits and b on the LSB part. The output is available 20 clock cycles later, it is expressed in hexadecimal as $A9F6$ which corresponds to ≈ -2.689 radians ($-154°$). The result is a negative one related to having both a and b negative, which corresponds to having an angle in the fourth quadrant .

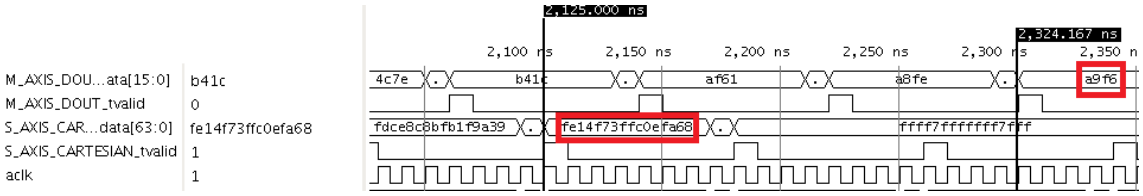


Figure 4.9: Simulation results: output of *arctangent* block

After having calculated the angle θ the next step is to calculate the sine and the cosine of the angle through the *sincos* block. The output is ready 20 clock cycles later, and is composed of 32 bits, (16 bits for each):

- $sin(\theta) = E3FF$ (expressed in hexadecimal) which corresponds to ≈ -0.4376
- $cos(\theta) = C673$ corresponding to ≈ -0.899

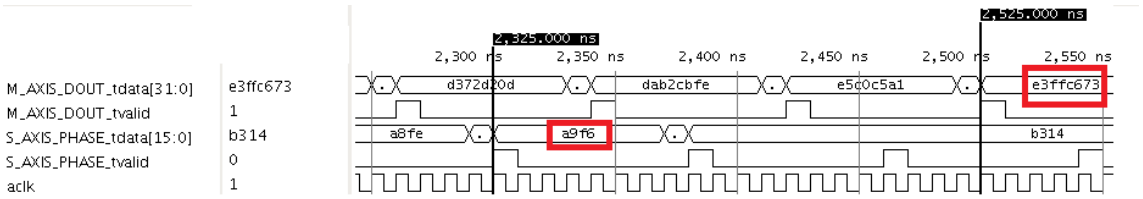


Figure 4.10: Simulation results: output of *sincos* block

On the other hand, and after calculating the term $a^2 + b^2 \approx 1259239$, the output would be 1122, and is provided with an input to output delay of 17 clock cycles.

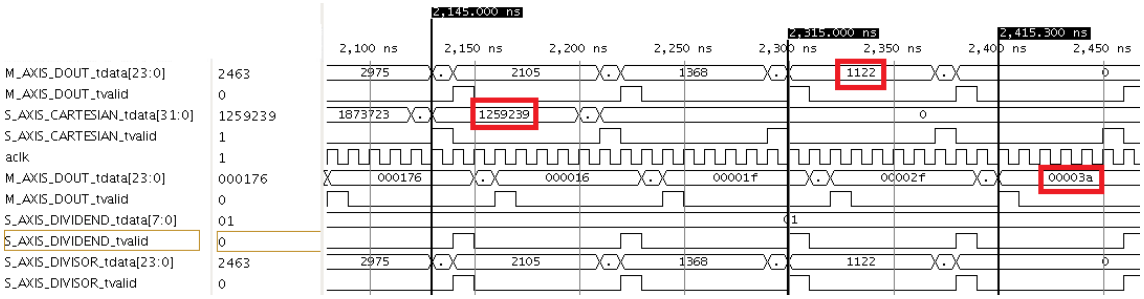


Figure 4.11: Simulation results: outputs of *squareroot* and *reciprocal* blocks respectively

The remaining step after that would be to calculate the reciprocal of the previously calculated square root. The result is $0003A$ (corresponding to $8.85 \cdot 10^{-4}$) with 16 bits expressing the fractional part.

4.1.3 Iteration Stage

The figure below illustrates a possible case in which not all the events pass the first stage. In this specific case, and for a certain threshold, only 6 events of the 9 present in the first stage, pass to the second stage of processing. Only those who satisfy the equation that has been discussed earlier make it through the final processing.

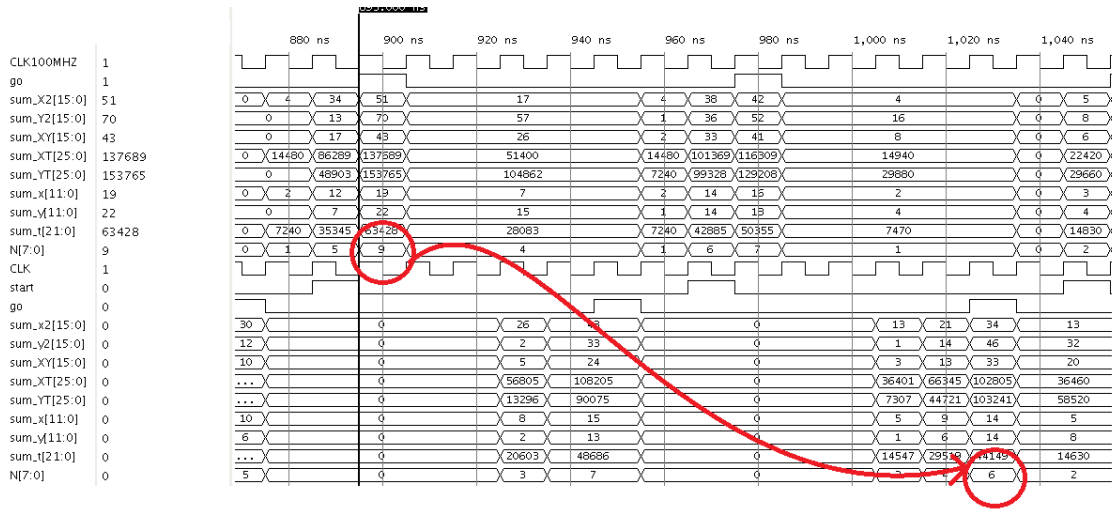


Figure 4.12: Simulation results: output of *iteration mult* block

4.1.4 Validation Memory Simulation

In this part the input data has been changed, the time interval for which the events remain in memory has been modified to 800 ns, and the matrix dimension has been set to 3×3. These modification allows a simple demonstration of the correct behaviour of the validation memory stage. The inputs that have been provided consecutively are presented in table 4.2, with a time interval of 80 ns between each pair of events. As can be observed, the first and ninth events have the same coordinates (7,9) in order to show how is the most recent event is kept for the correct time inside the main memory.

Table 4.2: Table showing the inputs that have been provided to the architecture in the validation memory simulation

Input	X	Y	T_s
1	7	7	256
2	7	8	300
3	7	9	350
4	8	7	400
5	8	9	450
6	9	7	500
7	9	8	550
8	9	9	600
9	7	9	625
10	8	8	650
11	8	8	700
12	8	8	750
13	8	8	800
14	8	8	850
15	8	8	900
16	8	8	950
17	8	8	1000

The image below (4.13) shows how in the third $check$ the event is not cancelled from the main memory, where the web signal remains deactivated. The reason is that a new event has been written on the same position, and that event is not expired yet. While in the eighth $check$ where $X\_v = 7$ and $Y\_v = 9$ again the web is asserted
To the left of figure 4.13, the outputs of the $MULT$ block have been highlighted to show that the last two remaining events are the inputs indexed as 9 and 17 (see table 4.2), whereas the other events in the neighbourhood has been cancelled from the main memory. To the right of the figure the web is activated later to cancel the event with coordinates (8, 8), the reason is that different events have been written in this same position, and only

when the most recent event has expired, the position is deleted.

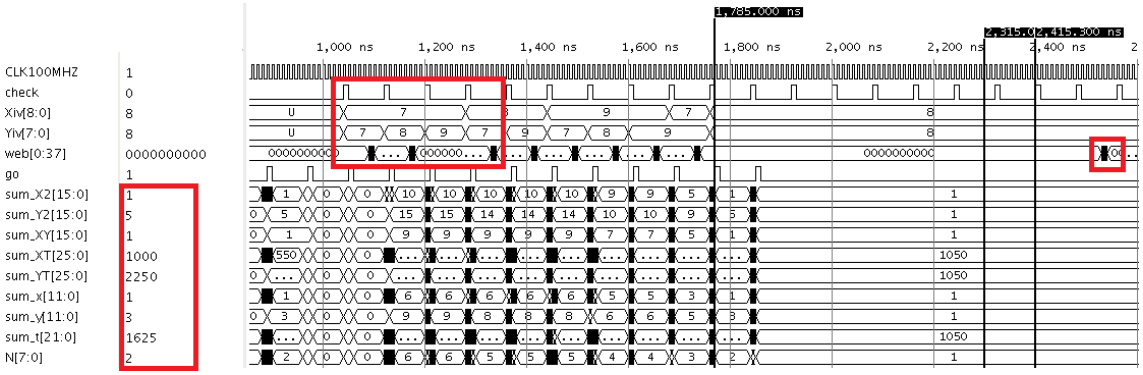


Figure 4.13: Simulation results: events validation

4.2 Synthesis Results

In this section, the results in terms of hardware resource consumption and latency for the different matrix configurations will be highlighted and discussed. The reported results are obtained after the synthesis and implementation stages, in which in the first stage the synthesizer generates a gate-level netlist starting from the written VHDL code, and using a UNISIM component library, which is a Xilinx library that contains the basic primitives. On the other hand, in the implementation stage, the netlist is translated into a placed and routed FPGA design. The place and route stage defines how are the resources located and interconnected on the FPGA device. The target device that has been used is a *ZedBoard Zynq-7000* board.

4.2.1 Resource Utilization and Latency

The design is constrained to an internal clock of 100 MHz, and some necessary pipeline stages have been introduced in order to satisfy the timing requirements of the different blocks. For the two dividers ($divider$ and $reciprocal$ blocks), different implementations have been carried out in order to find the minimum possible latency depending on the input bit-width, and the operation has been carried out for the different blocks. Some hardware resources could have been saved at the cost of a higher block latency, but in this project the aim was to obtain the minimum possible latency, and it is the strategy that has been adopted even at a higher resource utilization. The implementation results in terms of latency and resource utilization have been reported in the tables below. The results are generated for different configurations, these configurations are related to the different output matrix dimensions, which varies between 3×3 and 15×15 with the possibility of setting a number of rows independently from the number of columns.
Another parameter that must be taken into consideration is the dimension of the validation memory responsible of keeping track of all the events that are present in the main memory, and to delete the expired ones. Since the validation memory has been designed as a distributed FIFO, at the implementation stage the some LUT have been exploited for this block instead of using block RAM. The time interval, which depends on the dimension of this memory, has been set to $2\ \mu s$.
One other parameter that has to be noticed is the number of iterations. The results have been analysed for a number of iterations that varies between 0 and 2. Zero iterations indicates that no iterations take place, and the results are obtained through a unique processing stage. This could lead to realistic results for high matrix dimensions (e.g. 15×15), where the outliers weigh less on the final result.

Hardware Resource Consumption The tables below illustrate the hardware consumption for different matrix dimensions and for the three different iteration configurations. As

it can be noticed, the number of BRAMs is constant in the different cases since BRAMs are only used for storing the events of the array, which is independent from the different configuration parameters.
A detailed study of how are the hardware resources distributed among the different blocks, shows that the DSP primitives are mainly exploited by the $adjoint$ block, since the operations that are performed in this block are done on signals with higher bit-width, therefore, at the implementation stage, multipliers and adders exploit DSP instead of LUTs. Keeping a high internal bit-width results in more precise results at the cost of exploiting more DSP blocks.
A constant number of DSP blocks (37) is used for the different matrix dimensions in the iteration-free configuration. The DSP primitives are used in the $adjoint$ block, and in the final computation stages. Both cases are independent of the matrix dimension, leading to a constant number of exploited DSP blocks. The amount of DSP blocks increases with the number of iterations as can be noticed, in which the iteration process requires different multiplications, additions and comparisons. In these cases, the number of DSP blocks is dependent on the matrix dimension.

Two particular cases are reported where the matrix dimensions are of 3×15 and 15×3. The results show how the consumption of the LUT used as logic are similar in the 15×3 and 15×15 cases, and so are they in the 3×15 and 3×3 cases. The reason is that the LUT part as logic is exploited by arithmetic operations, which number depends on the number of rows and not on the number of columns, leading to these comparable results.

Generating the Netlist After resolving the final timing problems and introducing the necessary pipeline stages in various blocks, a Verilog timing simulation was performed using Vivado simulator to verify the correct behaviour of the circuit. The same test bench was used to make sure that both the timing and behavioural simulation provide the same outputs. The post implementation simulation has been performed in two steps:

1. The post implementation netlist corresponding to the entire design was generated
2. Generate an SDF delay file while annotating all the timing delays with the related netlist

(a) 3×3

Iterations	0	1	2
Slices	3815	4598	5983
SliceL	2621	3146	4116
SlcieM	1194	1452	1867
LUTs	9098	12136	15286
LUT as logic	9016	12052	15200
LUT as memory	82	84	86
DSPs	37	74	111
Flip Flops	12709	15112	17777
BRAMs	38	38	38

(b) 5×5

Iterations	0	1	2
Slices	3780	5589	7180
SliceL	2688	3938	5020
SlcieM	1092	1651	2160
LUTs	9840	14660	19484
LUT as logic	9758	14576	19398
LUT as memory	82	84	86
DSPs	37	82	127
Flip Flops	13669	17251	21593
BRAMs	38	38	38

(c) 7×7

Iterations	0	1	2
Slices	4163	6362	8948
SliceL	2943	4381	5960
SlcieM	1220	1981	2988
LUTs	10734	17609	24475
LUT as logic	10652	17525	24389
LUT as memory	82	84	86
DSPs	37	90	143
Flip Flops	14901	20078	26739
BRAMs	38	38	38

(d) 9×9

Iterations	0	1	2
Slices	4687	7554	10852
SliceL	3295	5178	7346
SlcieM	1392	2376	3506
LUTs	11380	20781	30087
LUT as logic	11297	20696	30001
LUT as memory	83	85	86
DSPs	37	98	159
Flip Flops	16263	23241	32665
BRAMs	38	38	38

(e) 11×11

Iterations	0	1	2
Slices	4564	7938	12860
SliceL	3047	6019	9046
SlcieM	1517	1919	3814
LUTs	12268	24162	36183
LUT as logic	12187	24079	36098
LUT as memory	83	83	85
DSPs		106	175
Flip Flops		27020	
BRAMs	38	38	38

(f) 13×13

Iterations	0	1	2
Slices	5598	9852	12948
SliceL	3866	6718	8772
SlcieM	1732	3134	4176
LUTs	12659	27280	41919
LUT as logic	12576	27195	41832
LUT as memory	83	85	87
DSPs	37	114	191
Flip Flops	19916	31297	47767
BRAMs	38	38	38

(g) 15×15

Iterations	0	1	2
Slices	5783	11475	13300
SliceL	4066	7736	8950
SlcieM	1717	3739	4350
LUTs	13418	30920	48326
LUT as logic	13336	30836	48240
LUT as memory	82	84	86
DSPs	37	122	207
Flip Flops	19916	35958	56609
BRAMs	38	38	38

(h) 15×3

Iterations	0	1	2
Slices	4912	8616	12104
SliceL	3311	5787	8199
SlcieM	1601	2829	3905
LUTs	13234	25444	37700
LUT as logic	13152	25306	37614
LUT as memory	82	84	86
DSPs	37	122	207
Flip Flops	16682	25077	34840
BRAMs	38	38	38

(i) 3×15

Iterations	0	1	2
Slices	3752	5284	5983
SliceL	2638	3499	4116
SlcieM	1114	1785	1876
LUTs	9213	13746	15286
LUT as logic	9131	13662	15200
LUT as memory	82	84	86
DSPs	37	74	111
Flip Flops	13789	17304	17777
BRAMs	38	38	38

Latency The table below illustrates how does the latency of the architecture vary with the different parameters. For the iteration-free configuration, the latency increases of one clock cycle for each additional pair of columns (e.g. between 3×3 and 5×5, the latency increases from 98 to 99 clock cycles). The maximum latency is reached for the 15×15, 2-iterations configuration, and it is of 140 clock cycles ($1.4\ \mu s$). The results for the 3×15 and 15×3 configurations are reported in order to demonstrate how the latency depends on the number of columns only and not on the number of rows.

Table 4.3: The variation of the latency with the matrix dimensions and iteration number

Matrix \ Iterations	0	1	2
3×3	98	110	122
5×5	99	112	125
7×7	100	114	128
9×9	101	116	131
11×11	102	118	134
13×13	103	120	137
15×15	104	122	140
3×15	104	122	140
15×3	98	110	122

4.3 Future Perspectives and Conclusion

4.3.1 Future Perspectives

In this section, the next possible steps will be highlighted along with some achievable future improvements, which could result in a more performing architecture, reducing both hardware resource consumption and latency.

Future Testing and Precision Evaluation The architecture that has been presented in this thesis is tested with a simple test bench that verifies the correct behaviour of the various blocks, and the data propagation through the architecture, where no ground truth dataset is available. The next step could be the evaluation of the precision of this architecture and the comparing it with the accuracy of the corresponding software.

Architecture Improvements The intention was to get a reconfigurable architecture, with a reusable VHDL code with a set of parameters that could be decided at synthesis stage. The parameter that has the strongest impact on the entire architecture is the output matrix dimension. The choice of assuming that the matrix dimension is a reconfigurable forced the architecture to be designed in a way that guarantees the correct behaviour in the worst case (15×15), therefore, for lower matrix dimension the code is sub-optimal. The VHDL code could be heavily optimized by studying the correct architectural choices for each case.

Memory Addressing The memory addressing that has been adopted throughout the project is related to a choice in which a trade-off between the number of the exploited BRAM blocks and the simplicity of the memory addressing was chosen. A simple example, where 48 RAM blocks are exploited instead of 38, is shown in the table below (table 4.4), in which each memory exploits 1.5 BRAMs (one 36K block and one 18K block). As it can be noticed, having memory blocks organized in blocks of 16 in the vertical direction, and blocks of 2 in the horizontal direction leads to a simpler address calculation in which:

$$address = Y(7 \quad downto \quad 1) + 120 \cdot X(8 \quad downto \quad 4)$$

the term num_{block} used in this project, which requires a more complex computation vanishes in this equation. A specific addressing technique could be adopted based on the desired output matrix dimension. For instance, in the 5×5 case, a certain addressing methodology could allow reading the 25 matrix elements in parallel, leading to a reduction in the final latency, and introducing the possibility of obtaining a fully pipelined architecture.

Operands bit-width As mentioned before, the bit-width of the internal signals is strongly dependent of the matrix dimension, and has an important impact on the dedicated hardware of the different blocks (e.g. $mult$, $adjoint$, $divider$, etc...). In this design the choice was to keep a high precision event at the expense of exploiting more DSP. The FPGA exploits a single hardware multiplier that accepts two operands, one of 25 bits and the other of 18 bits maximum. If higher bit-widths are used, which is the case, the implementation exploits more than one multiplier per multiplication. Therefore, one methodology that would reduce the number of the exploited DSP slices is reducing the internal bit-width and thus the accuracy. Moreover, a divider with lower bit-width at the operands will certainly induce lower latency, and lower hardware resource consumption as well.

Divider As explained previously, modifying the divider's type would have a remarkable reduction in the latency of the architecture. The adopted Radix-2 divider has a latency of 40 clock cycles, using 4 dividers each of 16 clock cycles latency will lead to a latency which is 24 clock cycles lower, at the cost of using more DSP blocks. One other possibility would be to use some external approximated dividers at the expense of accuracy loss, with the exploitation of less hardware resources.

Reciprocal Having the two coefficients a and b represented on lower bit-width will mean that the term $\sqrt{a^2 + b^2}$ can be expressed on lower bit-width, leading to a reduction of an operand of the $reciprocal$ block, which lead to lower hardware consumption, and possible lower latency.

Iterations Resource sharing between the different iterations could have a great impact on the hardware resource consumption. For instance, if the architecture is designed appropriately to support reading a whole 3×3 or 5×5 matrix elements in parallel, it means that the dedicated hardware, that is being used for the different arithmetic operations, can be shared between two consecutive iterations for example.

Table 4.4: The distribution of 32×10 cells among the 32 available memories.

0	16	0	16	0	16	0	16	0	16
1	17	1	17	1	17	1	17	1	17
2	18	2	18	2	18	2	18	2	18
3	19	3	19	3	19	3	19	3	19
4	20	4	20	4	20	4	20	4	20
5	21	5	21	5	21	5	21	5	21
6	22	6	22	6	22	6	22	6	22
7	23	7	23	7	23	7	23	7	23
8	24	8	24	8	24	8	24	8	24
9	25	9	25	9	25	9	25	9	25
10	26	10	26	10	26	10	26	10	26
11	27	11	27	11	27	11	27	11	27
12	28	12	28	12	28	12	28	12	28
13	29	13	29	13	29	13	29	13	29
14	30	14	30	14	30	14	30	14	30
15	31	15	31	15	31	15	31	15	31
0	16	0	16	0	16	0	16	0	16
1	17	1	17	1	17	1	17	1	17
2	18	2	18	2	18	2	18	2	18
3	19	3	19	3	19	3	19	3	19
4	20	4	20	4	20	4	20	4	20
5	21	5	21	5	21	5	21	5	21
6	22	6	22	6	22	6	22	6	22
7	23	7	23	7	23	7	23	7	23
8	24	8	24	8	24	8	24	8	24
9	25	9	25	9	25	9	25	9	25
10	26	10	26	10	26	10	26	10	26
11	27	11	27	11	27	11	27	11	27
12	28	12	28	12	28	12	28	12	28
13	29	13	29	13	29	13	29	13	29
14	30	14	30	14	30	14	30	14	30
15	31	15	31	15	31	15	31	15	31

4.3.2 Conclusion

The scope of this thesis was to design a hardware architecture responsible of computing the optical flow on events that are produced by a particular family of image sensors which is the event-based image sensors (DVS or ATIS for example). Event-based image sensors are retina-inspired event-based image sensors the came out as an alternative to conventional vision sensors. They are characterized by:

- Low latency and high speed
- Low data volume and memory storage
- High dynamic range

The architecture receives new event at a worst case rate of 1 event/80 ns and stores each in its corresponding position on a 304×240 array that keeps only the most recent event for each location. The processing is performed on a matrix centred at the incoming event. Although the architecture may not be optimal in terms of hardware consumption, it is characterized by:

- flexibility and reconfigurability, in which different parameters are reconfigurable and can be modified based on the target application. These parameters include:
  - the $time\_int$ which is the time interval in which the elements are kept in the main memory
  - the min_events - the minimum number of valid events on a matrix for which the processing is performed
  - the matrix dimension using two independent parameters which are ROW and COL, representing the rows and columns of the desired matrix
  - n_iter representing the desired number of iterations
- low latency
- high precision in which no approximations are performed in the initial processing stage

The future step would be to test the performance of the hardware implementation on a ground truth dataset and integrate the design on the iCub robot.

Chapter 5

Test Bench with a SPAER Interface

The test bench discussed in this section is a part which provides the processing architecture with the event-based data derived from the neuromorphic vision sensor. This part is designed for testing a more complete system, where different subsystems are interfaced toward each other on the FPGA, where some signals provide status information regarding the block that transmits information and the block that is receiving. Therefore, the test bench presented here has not been used to test the architecture described previously.

This test bench consists of:

- a common-clock dual-port RAM which holds the AER events - such events are organized in a sequential manner, each consist in two parts: a time stamp which is related to the timing difference between two successive events - rather than the absolute one - in this way fewer bits are required to represent wide timing ranges; and an address event which represents the address of the active event.

- a programmable pre-scaler that has as an input a global clock and as an output an enable signal which is activated only once each N clock cycles. That is necessary in order to have an application dependent architecture in which the time stamp scale could vary between one application and another (ns, ms, μs, etc...)

- an N-bit counter used to count the time stamp cycles. Once the counter's output coincide with the corresponding time stamp, an event is ready to be triggered.

- an FSM responsible of extracting AER data from the RAM and providing it at the output in a precise timely manner.

The related architecture is shown in figure 5.1.

"Policy" is an application-dependent input that decides which policy must be adopted whenever an event time-out occurs. Event time-out means that the source has a new

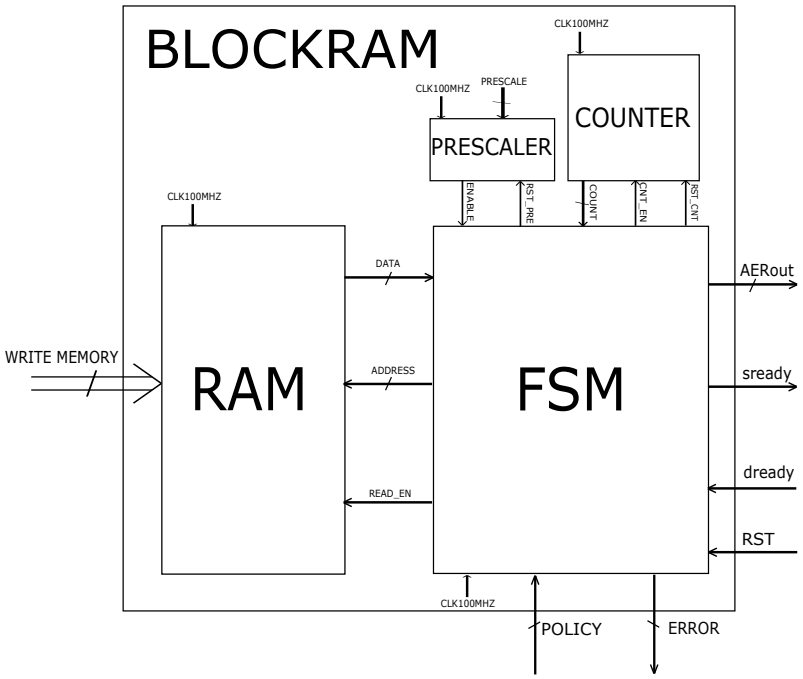


Figure 5.1: Test bench Architecture

event to provide for the destination, while the destination has not captured the previously provided one. Specifically, three different options are available:

- Stall - means no more events shall be provided unless the destination extracts the old one. This option guarantees that all the events are captured at the output.
- Provide new - in which the old event is excluded and the new one is provided at the output. Such an option ensures that the timing of the events is precise, despite the fact that some events could have been ignored.
- Keep old - implies that the new event is ignored and the old one is kept at the output.

The error signal is used to indicate the time-out problem when it happens.

5.1 Interface Protocol

The adopted interface protocol is a Synchronous Parallel AER (SPAER) interface used in [42], in which the source puts the event on the output, activates the "sready" signal, and waits for the destination to respond with an active "dready". In this case it is not necessary to release the "sready" signal if the destination is ready while a new event must be provided, hence guaranteeing the maximum data rate.

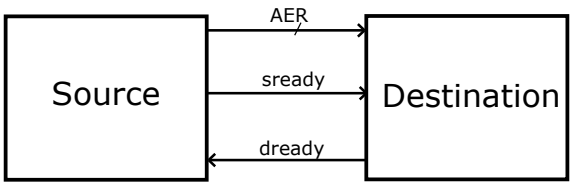


Figure 5.2: Test bench Interface

5.2 Finite State Machine (FSM)

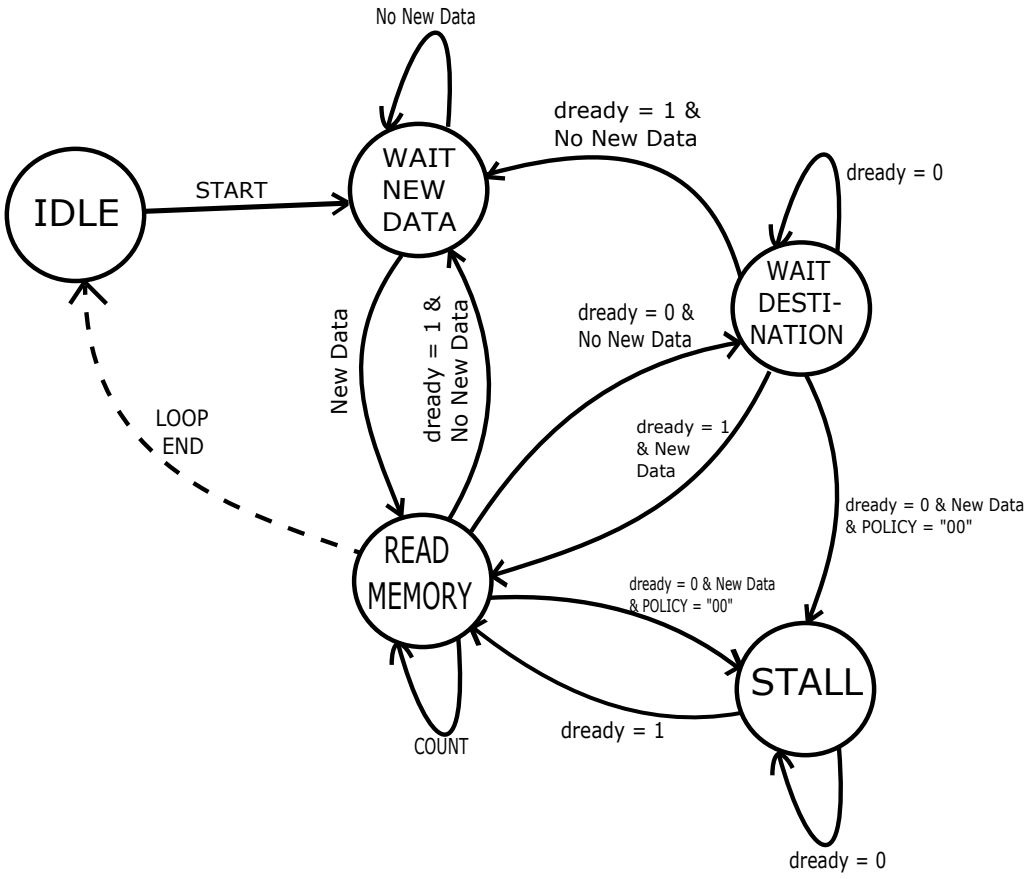


Figure 5.3: Test bench FSM

The implemented FSM is the one shown in figure 5.3. It is a five states FSM organized as follows:

- IDLE - whenever start is released the FSM starts and reads the first event with its related time stamp.

- WAIT NEW DATA - the counter starts counting until it reaches ΔTS. It then moves to "READ MEMORY" to trigger the source ready signal.
- READ MEMORY - "sready" is set high at the output with the associated AER result to tell the destination that an event is ready.
- WAIT DESTINATION - is a state in which the source is still ready, no new event has to be provided, and it is waiting for the destination to respond
- STALL - the state which the system reaches only if "Policy" is set to "00" and a new event has to be provided while the old one was not read by the destination yet.

The correct behaviour of this test bench has been synthesized with *Vivado 2017.1* and tested on the *Xilinx Artix-7 35T FPGA* (XC7A35TICSG324-1L) after loading the bit file on the board. One particularity of this part is the ability of updating the content of the memory without the need of re-synthesizing the design, by simply modifying the old bitstream file. This was achieved by the $updatemem$ command which takes in input:

- a memory mapping information $.mmi$ file
- a $.MEM$ file which contains the new content of the memory
- a $.bit$ file which contains the bitstream of the old design

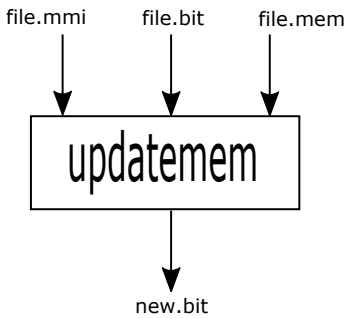


Figure 5.4: $updatemem$ TCL command

The $updatemem$ command provides in output the new $new.bit$ file which contains the same design as before but with the updated memory content. Throughout this part, the file $.mmi$ has been edited manually since $Vivado$ 2017.1 does not support the *Xilinx Artix-7 35T FPGA* for the automatic generation of $.mmi$ file. A $.mmi$ file will be attached with the thesis as an example.

Bibliography

[1] S. R. Kulkarni, A. V. Babu, B. Rajendran, "Spiking Neural Networks–Algorithms, Hardware Implementations and Applications" in *no*, v. 1, pp. 426–431, 2017.

[2] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, J. S. Plank, "A survey of neuromorphic computing and neural networks in hardware" in *arXiv preprint arXiv:1705.06963*, 2017.

[3] D. B. Strukov, G. S. Snider, D. R. Stewart, R. S. Williams, "The missing memristor found" in *nature*, v. 453, n. 7191, p. 80, 2008.

[4] G. Snider, R. Amerson, D. Carter, H. Abdalla, M. S. Qureshi, J. Leveille, M. Versace, H. Ames, S. Patrick, B. Chandler, *et al.*, "From synapses to circuitry: Using memristive memory to explore the electronic brain" in *Computer*, v. 44, n. 2, pp. 21–28, 2011.

[5] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, *et al.*, "Truenorth: Design and tool flow of a 65 mw 1 million neuron programmable neurosynaptic chip" in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, v. 34, n. 10, pp. 1537–1557, 2015.

[6] S. B. Furber, F. Galluppi, S. Temple, L. A. Plana, "The spinnaker project" in *Proceedings of the IEEE*, v. 102, n. 5, pp. 652–665, 2014.

[7] A. Amir, P. Datta, W. P. Risk, A. S. Cassidy, J. A. Kusnitz, S. K. Esser, A. Andreopoulos, T. M. Wong, M. Flickner, R. Alvarez-Icaza, *et al.*, "Cognitive computing programming paradigm: a corelet language for composing networks of neurosynaptic cores" in *Neural Networks (IJCNN), The 2013 International Joint Conference on* IEEE, 2013, pp. 1–10.

[8] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, *et al.*, "A million spiking-neuron integrated circuit with a scalable communication network and interface" in *Science*, v. 345, n. 6197, pp. 668–673, 2014.

[9] K. D. Fischl, G. Tognetti, D. R. Mendat, G. Orchard, J. Rattray, C. Sapsanis, L. F. Campbell, L. Elphage, T. E. Niebur, A. Pasciaroni, *et al.*, "Neuromorphic self-driving robot with retinomorphic vision and spike-based processing/closed-loop control" in *Information Sciences and Systems (CISS), 2017 51st Annual Conference*

on IEEE, 2017, pp. 1–6.

[10] M. Hutson. (2017) Pocket brains: Neuromorphic hardware arrives for our brain-inspired algorithms. [Online]: https://arstechnica.com/science/2017/07/pocket-brains-neuromorphic-hardware-arrives-for-our-brain-inspired-algorithms/3/

[11] D. Brüderle, E. Müller, A. P. Davison, E. Muller, J. Schemmel, K. Meier, "Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system" in *Frontiers in neuroinformatics*, v. 3, p. 17, 2009.

[12] T. C. Stewart, C. Eliasmith, "Large-scale synthesis of functional spiking neural circuits" in *Proceedings of the IEEE*, v. 102, n. 5, pp. 881–898, 2014.

[13] F. Galluppi, C. Denk, M. C. Meiner, T. C. Stewart, L. A. Plana, C. Eliasmith, S. Furber, J. Conradt, "Event-based neural computing on an autonomous mobile platform" in *Robotics and Automation (ICRA), 2014 IEEE International Conference on* IEEE, 2014, pp. 2862–2867.

[14] J. Conradt, F. Galluppi, T. C. Stewart, "Trainable sensorimotor mapping in a neuromorphic robot" in *Robotics and Autonomous Systems*, v. 71, pp. 60–68, 2015.

[15] I. Sugiarto, G. Liu, S. Davidson, L. A. Plana, S. B. Furber, "High performance computing on SpiNNaker neuromorphic platform: a case study for energy efficient image processing" in *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)* IEEE, 2016, pp. 1–8.

[16] D. Neil, S.-C. Liu, "Minitaur, an event-driven FPGA-based spiking network accelerator" in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 22, n. 12, pp. 2621–2628, 2014.

[17] K. Cheung, S. R. Schultz, W. Luk, "NeuroFlow: a general purpose spiking neural network simulation platform using customizable processors" in *Frontiers in neuroscience*, v. 9, p. 516, 2016.

[18] J. Shen, D. Ma, Z. Gu, M. Zhang, X. Zhu, X. Xu, Q. Xu, Y. Shen, G. Pan, "Darwin: a neuromorphic hardware co-processor based on spiking neural networks" in *Science China Information Sciences*, v. 59, n. 2, pp. 1–5, 2016.

[19] B. V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A. R. Chandrasekaran, J.-M. Bussat, R. Alvarez-Icaza, J. V. Arthur, P. A. Merolla, K. Boahen, "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations" in *Proceedings of the IEEE*, v. 102, n. 5, pp. 699–716, 2014.

[20] (2011) The brainscales project. brainscales - brain-inspired multiscale computation in neuromorphic hybrid systems. [Online]: https://brainscales.kip.uniheidelberg.de/

[21] N. Qiao, H. Mostafa, F. Corradi, M. Osswald, F. Stefanini, D. Sumislawska, G. Indiveri, "A reconfigurable on-line learning spiking neuromorphic processor comprising 256 neurons and 128K synapses" in *Frontiers in neuroscience*, v. 9, p. 141, 2015.

[22] M. B. Milde, H. Blum, A. Dietmüller, D. Sumislawska, J. Conradt, G. Indiveri, Y. Sandamirskaya, "Obstacle avoidance and target acquisition for robot navigation using a mixed signal analog/digital neuromorphic processing system" in *Frontiers in neurorobotics*, v. 11, p. 28, 2017.

[23] P. Lichtsteiner, C. Posch, T. Delbruck, "A 128×128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor" in *IEEE journal of solid-state circuits*, v. 43, n. 2, pp. 566–576, 2008.

[24] C. Posch, D. Matolin, R. Wohlgenannt, "A QVGA 143 dB dynamic range frame-free PWM image sensor with lossless pixel-level video compression and time-domain CDS" in *IEEE Journal of Solid-State Circuits*, v. 46, n. 1, pp. 259–275, 2011.

[25] M. Gottardi, N. Massari, S. A. Jawed, "A 100μW 128×64 Pixels Contrast-Based Asynchronous Binary Vision Sensor for Sensor Networks Applications" in *IEEE Journal of Solid-State Circuits*, v. 44, n. 5, pp. 1582–1592, 2009.

[26] J. A. M. Olsson, P. Hafliger, "Two color asynchronous event photo pixel" in *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on* IEEE, 2008, pp. 2146–2149.

[27] Ł. Farian, J. A. Leñero-Bardallo, P. Häfliger, "A bio-inspired aer temporal tricolor differentiator" in *Biomedical Circuits and Systems Conference (BioCAS), 2014 IEEE* IEEE, 2014, pp. 524–527.

[28] T. Delbruck, M. Lang, "Robotic goalie with 3 ms reaction time at 4% CPU load using event-based dynamic vision sensor" in *Frontiers in neuroscience*, v. 7, p. 223, 2013.

[29] M. Litzenberger, A. N. Belbachir, P. Schon, C. Posch, "Embedded smart camera for high speed vision" in *Distributed Smart Cameras, 2007. ICDSC'07. First ACM/IEEE International Conference on* IEEE, 2007, pp. 81–86.

[30] R. Berner, C. Brandli, M. Yang, S.-C. Liu, T. Delbruck, "A 240$\times$ 180 10mW 12us latency sparse-output vision sensor for mobile applications" in *VLSI Circuits (VLSIC), 2013 Symposium on* IEEE, 2013, pp. C186–C187.

[31] C. Brandli, R. Berner, M. Yang, S.-C. Liu, T. Delbruck, "A 240$\times$ 180 130 db 3 μs latency global shutter spatiotemporal vision sensor" in *IEEE Journal of Solid-State Circuits*, v. 49, n. 10, pp. 2333–2341, 2014.

[32] D. Fortun, P. Bouthemy, C. Kervrann, "Optical flow modeling and computation: a survey" in *Computer Vision and Image Understanding*, v. 134, pp. 1–21, 2015.

[33] J. T. Philip, B. Samuvel, K. Pradeesh, N. Nimmi, "A comparative study of block matching and optical flow motion estimation algorithms" in *Emerging Research Areas: Magnetics, Machines and Drives (AICERA/iCMMD), 2014 Annual International Conference on* IEEE, 2014, pp. 1–6.

[34] B. D. Lucas, T. Kanade, *et al.*, in "An iterative image registration technique with an application to stereo vision" 1981.

[35] B. K. Horn, B. G. Schunck, "Determining optical flow" in *Artificial intelligence*, v. 17, n. 1-3, pp. 185–203, 1981.

[36] M. Liu, T. Delbruck, "Block-matching optical flow for dynamic vision sensors: Algorithm and FPGA implementation" in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on* IEEE, 2017, pp. 1–4.

[37] R. Benosman, S.-H. Ieng, C. Clercq, C. Bartolozzi, M. Srinivasan, "Asynchronous frameless event-based optical flow" in *Neural Networks*, v. 27, pp. 32–37, 2012.

[38] R. Benosman, C. Clercq, X. Lagorce, S.-H. Ieng, C. Bartolozzi, "Event-based visual flow." in *IEEE Trans. Neural Netw. Learning Syst.*, v. 25, n. 2, pp. 407–417, 2014.

[39] [Online]: www.icub.org

[40] F. Rea, G. Metta, C. Bartolozzi, "Event-driven visual attention for the humanoid robot iCub" in *Frontiers in neuroscience*, v. 7, p. 234, 2013.

[41] [Online]: www.yarp.it

[42] P. M. Ros, M. Crepaldi, C. Bartolozzi, D. Demarchi, "Asynchronous DC-free serial protocol for event-based AER systems" in *Electronics, Circuits, and Systems (ICECS), 2015 IEEE International Conference on* IEEE, 2015, pp. 248–251.