POLITECNICO DI TORINO

Dipartimento di Elettronica e Telecomunicazioni

Ingegneria Elettronica

Tesi di Laurea Magistrale

FPGA implementation of event-based optical flow for robotic applications

Relatori
Prof. Maurizio Martina
Prof. Guido Masera
Dr. Paolo Motto Ros

Candidato
Gianluca Tortora

Anno accademico 2018-2019

Summary

The speed with which technological evolution has reached increasingly extraordinary goals, in recent years, has now opened the doors to the possibility of develop more precise and refined devices with respect to the recent past. In this way, the research in the Artificial Intelligences field represents a fundamental joint for both possible future implications in the industrial development and in the health world. The purpose of this work is to build up a reconfigurable hardware solution able to interface with modern vision sensors emulating the behaviour of the human retina. This final product, at the end of a close cooperation with the Istituto Italiano di Tecnologia (iit), is significantly useful to have a base-test for the optical flow computation. This could represent, furthermore, an important source of informations from an AI perspective or for feasible biological involvements.

The path taken through this elaborate starts from a close analysis on neural networks, learning systems and the different vision sensors currently in commerce. The frame-based sensors are the oldest solution available, but present critical problems in terms of redundancy, latency, long time intervals and so on. For that reason all the efforts in this direction are focused on the event-based solution, suitable for human-like interfaces. Generally, a moving object generates a variation of brightness in the reference field of a sensor: through a sleek algorithm, which is the case-study of this thesis, it is possible to obtain spatio-temporal informations about the movement itself. In this way, precise visual flow orientation and amplitude can be estimated. Starting from the analysis of the reference paper, the operating algorithm was extracted and reproduced in floating-point version in a parametrized and reconfigurable Matlab script. Subsequently, this function was also adapted in a fixed-point version, offering solutions to some problems coming out from the sensor. By comparing the results coming from a real dataset, kindly provided by the iit, it was possible to make a statistic estimation of the errors in the modelling phase. This represents the starting point for the subsequent RTL design phase of the device. All the solutions adopted are described in detail. The reference SoC module is a Trenz Electronic TE0715-04-30-1C, which integrates a Xilinx Zynq XC7Z030-1SBG485C. At the end, a comparison between the expected results in Matlab and the ones obtained from the hardware simulation, as well as the informations coming from the synthesis of the device in terms of area, power consumption and timing have been realized for different possible configurations.

Contents

1 Neural networks and learning systems **5**
- 1.1 The role of AI in the contemporary world . . . 5
- 1.2 Artificial Neural Networks . . . 6
- 1.3 ANN hardware implementations . . . 8
- 1.4 Neuromorphic Vision Sensors . . . 11

2 State of art **17**
- 2.1 The optical flow . . . 17
- 2.2 Event-based visual flow . . . 20
- 2.3 Performances and limitations . . . 22

3 Modelling the algorithm **25**
- 3.1 Matrix computation flow . . . 25
- 3.2 Matlab description . . . 28
- 3.3 Results and comparisons . . . 42

4 Guidelines for the hardware design **51**
- 4.1 General view . . . 51
- 4.2 Memory Addressing . . . 54
- 4.3 Events extinction . . . 60
- 4.4 The overflow detection . . . 61

5 Hardware description **63**
- 5.1 Memory block and validation FIFO . . . 63
- 5.2 Overflow detector FSM . . . 66
- 5.3 Shifter and valid generator . . . 69
- 5.4 Mult block . . . 71
- 5.5 Adjoint block . . . 73
- 5.6 Iteration block . . . 74
- 5.7 Final computation . . . 76

6 Simulation, synthesis and results **85**
6.1 Single-block testbenches 86
6.2 System simulations 95
6.3 Synthesis outcomes 97
6.4 Future developments 102
6.5 Conclusion 103

Bibliography **105**

Chapter 1

Neural networks and learning systems

1.1 The role of AI in the contemporary world

With the rapid development of technological processes and the research in the artificial intelligence (AI) field, the transition to a completely new society, where human work takes on a marginal presence, seems to be day after day closer. Automation, robotics and the design of intelligent systems represent the driving force behind the contemporary world. The vast majority of the reference sectors in our daily life are dominated by the presence of AI: from economic processes, regulated and controlled by deduction algorithms, to industrial planning, to medicine, to neurosciences such as learning [1].

It is in this last field that the greatest efforts of the academic world are focusing. The attempt to reproduce in a partial or total way the behaviour and the actions of the human brain is one of the main challenges of modern engineering. In this way, in fact, are defined the so-called weak AI and the strong AI [3].

Figure 1.1. iCub (iit)

1.2 Artificial Neural Networks

Artificial neural networks (ANN) are nothing more than computational blocks described in mathematical language [4]. Their main aim is to process informations with a functioning inspired by the human nervous system.

Like a biological system [2], in fact, they are composed by a series of basic processing units variously connected among them. These units receive a set of informations from the surrounding environment, and then return some outputs that flow into the environment itself. The biological units are the neurons, and, due to the behavioural analogies, the bricks of the ANN are called in the same way.

As their physical counterpart ”fire” an electrical signal (if the voltage at the input is above a certain threshold), in the same way the digital units become active if the amount of input signals exceeds a fixed activation threshold. The generated output spike is then transmitted through communication channels, until it reaches other units to which is connected [2][3][4].

The connection points [4] act as filters: they transform an input into an inhibitory or excitatory signal, by increasing or decreasing its intensity depending on their characteristics. Their behaviour is therefore similar to the biological synapses and, therefore, these artificial connections assume the same name.

The function that describes the response wave emitted by a certain node is given by the sum of the products of the input signals for the respective synaptic weights, to which the threshold value of the node itself is eliminated [3][4][5].

However, what determines the conduct of the whole system is not the local behaviour of the single block, but how they are connected to each other. Moreover, some learning parameters are fixed in order to associate a certain synaptic weight to each single node. After a training phase, then, this weight will be shaped according to the preponderance and correctness of the external stimuli [4][5].

ANN Characteristics
The characteristics [4] of a neural network can be divided into three fundamental groups:

- **Flexibility**: the model can be used for different purposes. The network itself, in fact, does not need to know its around environment, but it is able to learn its characteristics only basing on experience.

- **Robustness**: that is the ability to provide an adequate response even if there are problems in the connections, or noise at the inputs.

- **Generalisation**: although training is limited, an ANN must be able to provide correct answers to input patterns never seen before, and similar to those used in training phases.

ANN Architectures
The neural networks organization is generally made on more than one level: from the input to the output side, with in the middle a series of intermediate (or hidden) layers, containing a certain number of neurons.

Typically the connections can be enclosed in two big families [4][5]:

- **Feedforward**: in which the neurons of a level are connected to the others of a next level. Reverse connections or connections in the same level are absolutely forbidden.
- **Recurrent**: they, instead, provide feedback connections, or between neurons belonging to the same level.

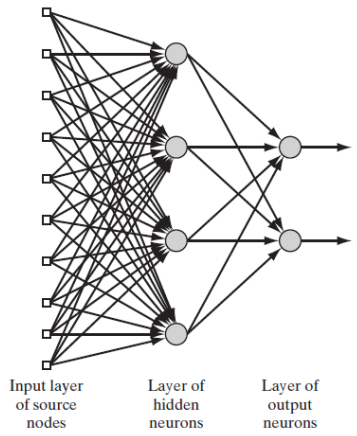


Figure 1.2. ANN Feedforward

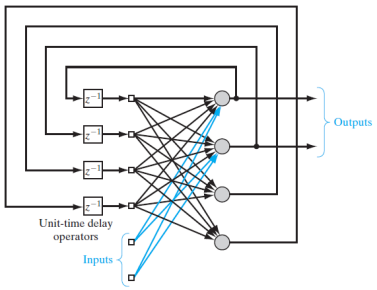


Figure 1.3. ANN Recurrent

In general, the architecture of a neural network is, therefore, completely different from the Von Neumann one typical of modern computers.

This has favoured the birth of neurocomputers (or neuromorphic computer): i.e. devices that process and produce informations, basing on a operation model typical of a neural network [6][7].

Neuromorphic computing
The idea of approaching an information processing mechanism that emulates the behavior of the human brain is the basis of neurocomputers. The goal is to create a complex structure capable of learning and adapting to ever-changing contexts: it is the challenge

that researchers around the world have taken up [6][7].

The term *neuromorph* was born in 1990, and it is referred to a series of high integrated architectures with analog electronic components, capable of imitating the behavior of the nervous system [8].
However, in recent years, they are essentially known as biologically-inspired systems.

The advantages of these structures are the processing speed, the high parallelization degree - that make possible the development of real-time systems able to learn and act instantly - and low energy consumption [7][8][9].

They are able to combine all these fundamental characteristics, thus being suitable for a hardware development that goes beyond the architecture of the traditional computer, limited by the stringent parameters of Moore's law [7][8].

Learning systems
As already mentioned, the building-blocks that characterize the neuromorphic systems are the neurons, the synapses and the connection type. And it is from them that, in fact, it is possible to build the target algorithm of the machine.

The next step is to train the device using different input patterns: they teach it how to perform the specific function for which itself was designed. Learning can take place either on-chip, i.e. directly on the device, or off-chip, i.e. outside the network and then everything is transferred to the structure.

In addition, the training can be either supervised or unsupervised [4][8].

In supervised learning there are input variables (X) applied to a mapping function which maps them on the output (Y):

$$Y = f(X)$$

The idea is to approximate this function so well in order to have a precise prediction mechanism also for other new input data.

In unsupervised learning, instead, there are only input data (X), without generated outputs (Y). In this way, the idea is to model the structure of the datas itself in order to learn more about them.

1.3 ANN hardware implementations

There are three important categories in which it is possible to divide the hardware representations of neural networks: analog, digital and mixed systems [8][9].

Analog systems

In the biological world, all the signals and the systems modes of operation are purely analog. In this way, the human brain expresses its functionality binding itself to physical characteristics such as: total asynchrony, the time-continuous representation of inputs, but also the problems related to the significant presence of a decidedly high amount of noise. Despite this, however, analog devices are best suited to play the role of basic-unit in ANNs.

In general, the analog hardware systems used in this field belong to two different types: programmable and customized [9].

As FPGAs are used as programmable devices in the digital world, also the FPAAs represent their analogue counterparts. The FPAA is a type of integrated circuit that is made up of a certain number of analog blocks (CABs) that can be connected together in various ways. The user can in fact program the type of connection between these different units in order to be able to perform a certain type of function from time to time.

They can operate both in discrete time mode and in continuous time mode: in the first case the devices have a system clock that controls switched capacitors. In the second case, instead, the device behaves as an array of transistors or operational amplifiers.

Some types of FPAA, however, have been designed for special purposes related to neuromorphic development: these are the NeuroFPAA [10]. They have already internally both neurons and synapses. Therefore, the programmability of the structure is essentially aimed at a purely neuronal type of application.

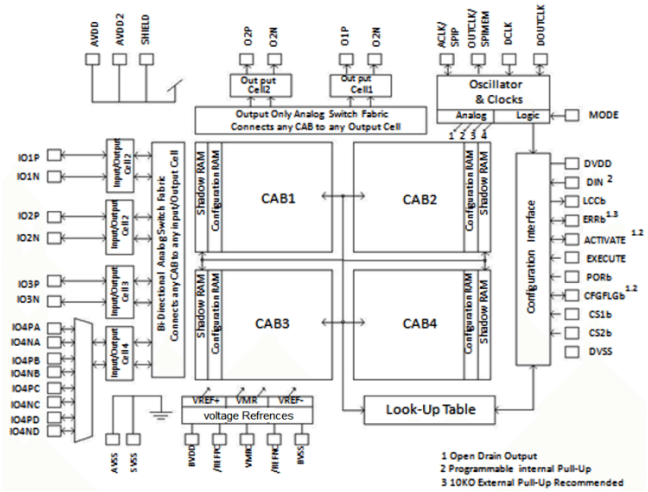


Figure 1.4. FPAA

As already mentioned, the ”analog” characteristic makes these devices the main candidates to imitate the cellular behaviour, but the presence of a high noise component makes them in some ways unreliable. For this reason, there is a tendency to use these devices under threshold, both for the problems related to SNR and low-power consumption.

Digital systems
As the analog systems, digital counterparts can also have the programmable devices or application specifics categories [8][9].

FPGAs allow certainly to manage, design and simulate neuromorphic systems in a faster way than the software approach. Moreover, their re-programmability allows to adopt different architectures from time to time with low implementation costs.

Different is the case, however, for ASICs, which are totally customized by the manufacturer and specific for a single application field. In this way it is possible to push the characteristics of the system towards a low power consumption and reduced complexity.

The most well-known ASIC architectures used in the field of neural networks are the TrueNorth and SpiNNaker chips.

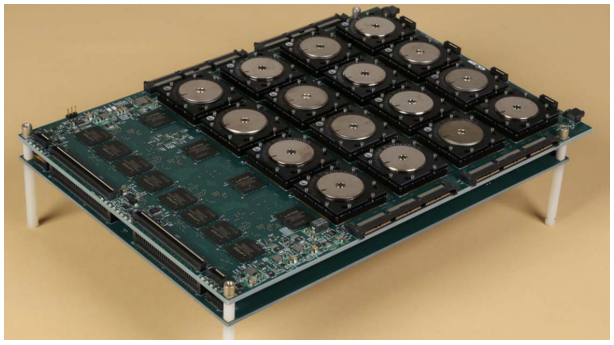


Figure 1.5. TrueNorth chip

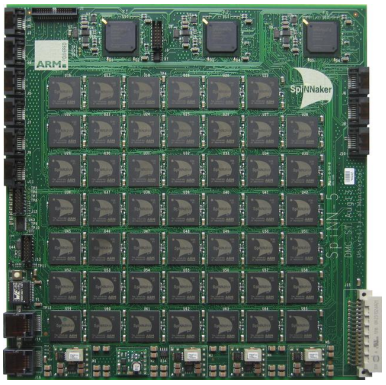


Figure 1.6. SpiNNAker chip

The particularity of the TrueNorth [11] is that it is partially asynchronous. This means that part of the activity takes place asynchronously, while a system clock that regulates the whole. It is organized by 4096 neurosynaptic cores of 256 neurons each, inserted in a 2D grid, everything in CMOS technology. Its high scalability allows it to be suitable for different applications.
An interesting use of this type of device is in the field of object detection and reconnaissance. This permits to obtain a certain amount of informations, coming from the surrounding environment, thanks to the use of vision sensors.

The SpiNNaker [12] is another type of neuromorphic architecture, totally customized

and parallel. The basic cores are connected through an interconnection scheme, configured and optimized for the spike transitions.
Like TrueNorth, it has a high scalability degree, in addition it allows the user to make a push and, from time to time, different modeling of both neurons and synapses.
Its performances are excellent, for example, for the optical flow computation which, starting from event-based vision sensors, is employed by the trajectory management mechanisms in robotic, or in learning-by-example applications.

Mixed Analog/Digital systems
Due to the intrinsically analog nature of the biological systems, digital neural architectures are not always able in this way to correctly emulate their behavior. However, their advantages come from the power consumption and noise immunity [13].
For this reason researchers decided to combine the characteristics of the analog and digital world, creating mixed structures that overcome the problems of both architectures.

The two main devices belonging to this family are Neurogrid and BrainScaleS.

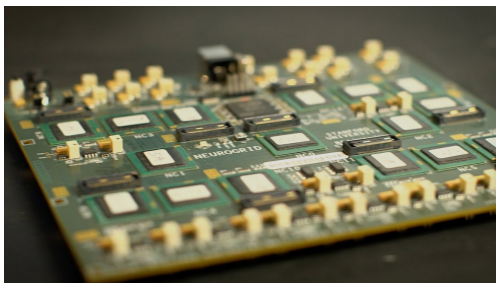


Figure 1.7. Neurogrid

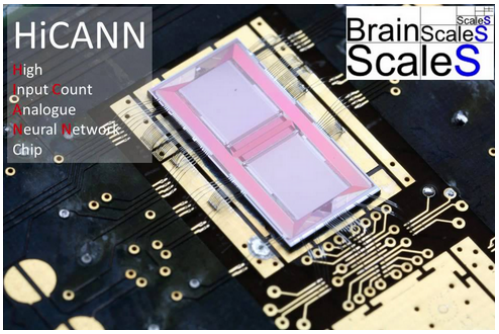


Figure 1.8. BrainScaleS

The first [14] is an analog circuit, with a certain level of digital communication.It is a neuromorphic computer that allows to develop various real-time simulations of neural models, working almost exclusively under threshold.

The BrainScaleS [15], on the other hand, is an implementation with a certain number of analog components, but it is mainly the digital part that composes the entire structure.

1.4 Neuromorphic Vision Sensors

In the last decades, with the concomitant technological evolution, also the devices specific for video capture have known a marked increase both in terms of performance and image quality. Canonical camcorders operates thanks to a frame-based mechanism: the images are acquired with a fixed frequency and, after a sequential processing, they give

life to the visual motion.

They present some good characteristics from the quality point of view, less instead considering the redundancy of information, a limited dynamic range and a high power consumption.

For this reason, event-based devices have been developed in the recent years: the human retina does not generate electrical signals each time for each single part of the captured image, but only for the areas that exhibit a variation in terms of brightness. In the same way, this new type of video-camera allows to generate events (like the spikes in the case of the retina) only in correspondence of the areas which presents modifications in its conformation.

There are many advantages related to this sensors: besides a good image quality there is no redundancy of information but, above all, low power consumption and fast processing.

The two main typologies belonging to this new family of cameras are:

- **ATIS**: Asynchronous Time-based Image Sensor [17];
- **DVS**: Dynamic Vision Sensor [16].

ATIS Camera
This type of sensor is totally asynchronous, and represents the first kind of bio-inspired camera known as ”event-based”.

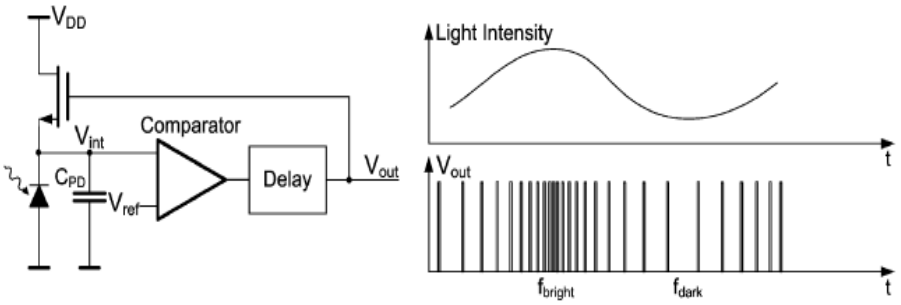


Figure 1.9. The ATIS structure and its behaviour

It is composed of a 340x240 pixels matrix, each of them totally autonomous, and its operation mode is quite simple. Unlike other types of sensors, which measure light intensity by translating it into voltage or current signals, this type instead measure the time taken for the photodiode to reach a certain voltage or charge. This technique is known as PM imaging.

There are two sub-pixels that compose the reference pixel: one receives a change in brightness - and manages it according to its structure - then report it to the second pixel. The other one, instead, generates a PWM waveform directly proportional to the event coming from the first subpixel. In practice, when the brightness exceeds, in all the directions, a certain threshold at the input of the Sense node, the cascade comparator switches its output.

From the outside, afterwards, a reset signal returns the node to its original state. This causes the device to behave like an oscillator, which generates frequency pulses directly related to the photocurrent generated.

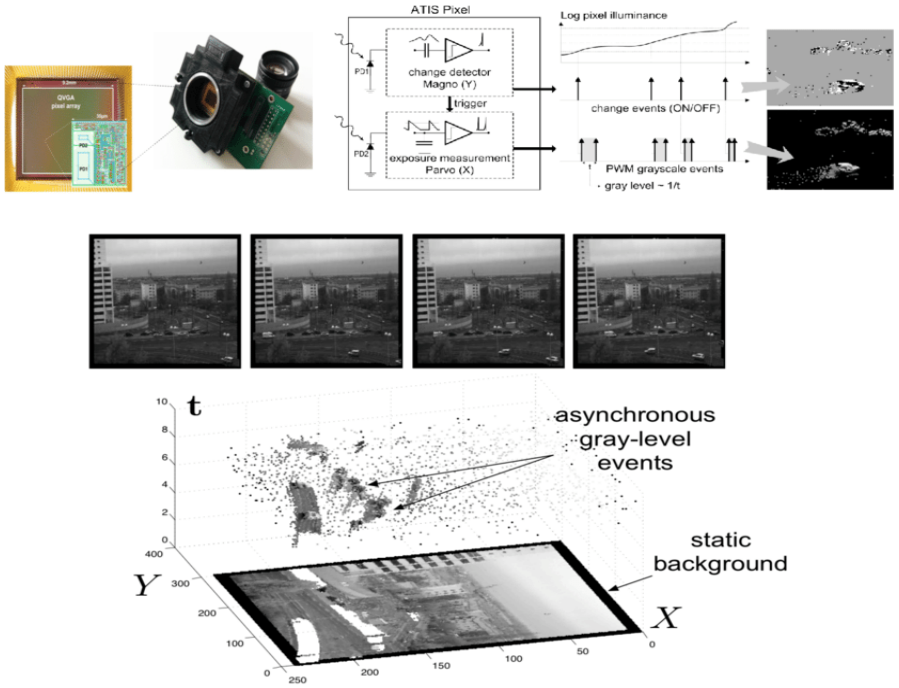


Figure 1.10. ATIS working principle

Some advantages of this architecture are related to a wide dynamic range that reaches 143 dB in static conditions, and 125 dB at a temporal resolution which is equivalent to 30 fps. However, on the other hand, it can also have acquisition times that, at low intensities, can reach up to hundreds of ms.

DVS Camera

The idea for this type of device is to achieve low latency and a good dynamic range, in a small amount of area. The solution, conceived by the researchers of the ETH of Zurich,

was to create a logarithmic photoreception circuit, integrated in a differential configuration, such as to amplify even small changes with high precision.

In this way, the photoreception circuit is able to control the pixel gain with a logarithmic behaviour, responding rather quickly to external stimuli and brightness changes.
The advantages of this topology are mainly related to high temporal resolution (about 1 us), low maximum latency (1 ms), low power consumption (about 20 mW), and wide operating band (120 dB).

As soon as a change in brightness is detected, a spike, which corresponds to a certain event, is generated on the output (Figure 1.12). Several variations in the reference frame lead, therefore, to a stream of output addressed events (EAs). This means that each of them will be associated to a certain reference instant of time (time stamp), to its polarization (if there is an increase or a decrease of the pixel brightness) and to the coordinates of the pixel into the grid where the event occurred.

This type of representation is commonly known as Address Event Representation (AER). The usefulness of this device, increasingly adopted in recent years, has found expression in tracking systems and in robotics.

The dataset on which a number of software simulations were conducted, in this paper, comes directly from a DVS sensor.

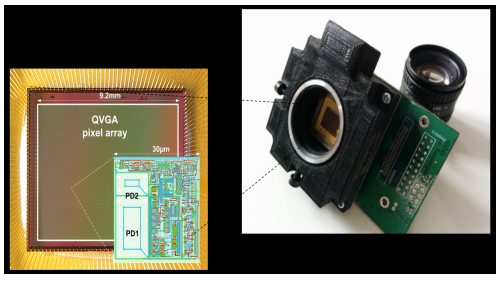


Figure 1.11. The DVS sensor

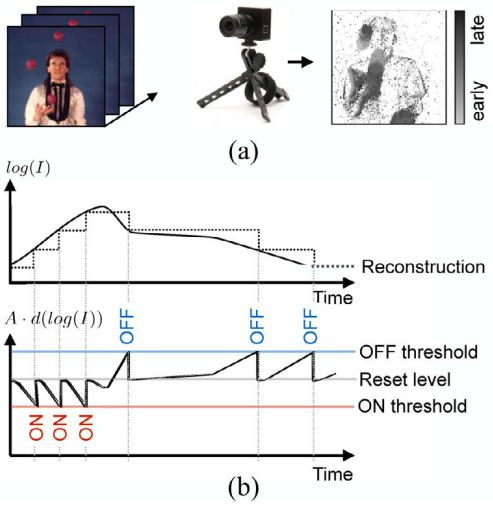


Figure 1.12. The DVS working principle

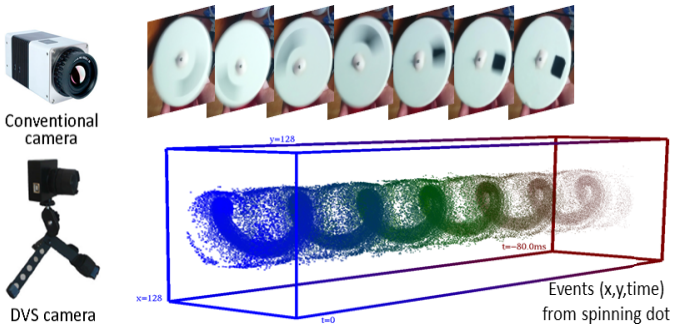


Figure 1.13. DVS flow

Chapter 2

State of art

2.1 The optical flow

Generally, the physiological phenomenon of the visual perception of the surrounding environment in humans arises from the structure of the retina: it, in fact, thanks to the presence of receptor cells that generate an electrical potential from external light stimuli, allows the brain to recreate the structure of the visual field, drawing the information it needs.

When we talk about optical flow, we generally refer to a flow of signals born from a variation in light intensity, caused by the movement of an object in the optical field of the observer. But this is not always the only case: in fact, also the observer can move, and therefore can cause the generation of a flow on the image plane. Moreover, the speed of movement in both cases is an important parameter for the correct generation of the flow.

It is important to underline that, if the entire visual spectrum is composed by a matrix of pixels, their intensity during the movement remains constant. Therefore, each single pixel associated to a specific point of the environment (assuming that the exposure conditions to light are constant) will not change its light intensity during the movement. This is a fundamental hypothesis for all the following treatments that will be made [18].

Application fields
There are wide application domains for the optical flow in the world of computerized vision systems [18].
The increasingly sophisticated techniques in image acquisition, the constant search for visual perfection gave birth, for example, to video compression standards such as MPEG [20], which use motion estimation to predict intermediate frames. Even in robotics, optical flow is proving useful, for example, for the development of control algorithms for assisted driving, or for the faces and objects recognition thanks to machine-learning [21]. In the bio-analytical field, it also allows to estimate informations related to the physical

properties of cells and tissues.

Estimation algorithms
There are various algorithms, centred on the analysis of optical flow, that have been developed in recent years and optimized according to parameters such as performances and application field. However, they can be enclosed in four large groups: correlation-based, gradient-based, energy-based and phase-based [18][19]:

- **Correlation-based (Block-matching)**: in this type of algorithms, generally, each video frame is divided into a number of pixels blocks of a certain size. The goal is to find the best matching-block of the current frame with the previous one, in order to minimize a certain metric. Usually a sum of absolute differences (SAD) is used, but it is extremely heavy from a computational point of view. At the end, a flow vector will be assigned to each block. Problems of resolution, accuracy and high consumption are the negative aspects of this type of process.

- **Gradient-based**: in this case the aim is to calculate the space-time derivatives on the acquired frame. It represents the group of algorithms most adopted in the world of optical flow. Less computational weight and possibility of optimizations in terms of consumption represent the advantages of this typology.

- **Energy-based**: they allow to estimate the optical-flow starting from the output of tunable filters designed in the Fourier domain.

- **Phase-based**: with them it is possible to evaluate the speed in terms of phase behaviour of the band-pass filter outputs.

Modelling principles
From a mathematical point of view [18], the description of the optical flow starts from the definition of a function:

$$I : \Omega \, x \, t \rightarrow R$$

With Ω representing the spatial domain of the frame (x,y), and t corresponding to its acquisition time. A basic hypothesis that is made, during the development of the model, is that each pixel, fixed between two frames, does not change its brightness value spatially. The same is true for a pixel moving between two frames that keeps a constant brightness from the temporal point of view.

Unfortunately, however, as previously discussed, this last hypothesis is not true, because the sensor is affected by noise, moreover it is not easy at all to find fixed lighting conditions.

In any case, this type of assumption is expressed through the brightness constancy constraint equation (BCCE):

$$\frac{dI}{dt}(x(t), t) = 0 \tag{2.1}$$

This means that the brightness of a certain pixel, in spatial and temporal terms, belonging to a real context, in fixed or moving condition, is to be considered constant:

$$I(x + w(x), t + 1) - I(x, t) = 0 \tag{2.2}$$

Assuming to work with small pixel shifts, it is then possible to develop the expression into Taylor's series:

$$\frac{\partial I}{\partial x_1}(x)u(x) + \frac{\partial I}{\partial x_2}(x)v(x) + \frac{\partial I}{\partial t}(x) \tag{2.3}$$

With:

$w(x) = (u(x), v(x))^T$

$w(x) = (x_1, x_2)^T$

In this way the function can be rewritten as:

$$\nabla I(x) \cdot w(x) + I_t(x) = 0 \tag{2.4}$$

Knowing that the spatial gradient is:

$$\nabla \cdot = \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}\right) \tag{2.5}$$

And I_t corresponds to the brightness partial derivative with respect to the time.

Unfortunately this is a sub-determined system, as it presents only one solution to a two-dimensional problem. This implies that the definition of another type of equation will be necessary to obtain the two components of the optical flux vector $w(x)$.

Therefore the imposition of the BCCE alone is not enough. This problem is known as the *aperture problem* [18]: it consists in supposing that the movement of linear structures is ambiguous by nature. It can happen, in fact, that some local structures can be extracted from two completely different frames, which represent two different images, and they can coincide. For this reason, it is essential to take into account the context from which the frame is extracted.
To this scope, in fact, a priori information on w(t) is introduced. This allows to give shape to a spatio-temporal coherence thanks to the imposition of local and global costraints.

2.2 Event-based visual flow

An interesting alternative technique for visual-flow calculation was proposed by researchers Ryad Benosman, Charles Clercq, Xavier Lagorce, Sio-Hoi Ieng, and Chiara Bartolozzi [22]. The aim is to experiment a new methodology in order to obtain dense visual flow from a series of information coming from an event-based asynchronous sensor.

The usefulness of this new approach and all its possible advantages, such as accuracy and low computational weight, make it an ideal candidate to interface, in robotic field, new type of sensors with iCub, the android developed by the Istituto Italiano di Tecnologia (iit).

The goal, from now, will be to describe the functioning of the algorithm under examination to obtain a complete hardware design, necessary for direct tests with an event sensor.

What the researchers have used, as experimental working base in this type of algorithm, is a dataset from a DVS sensor, AER type, with a 128x128 pixel retina. This involves, on the output of the sensor, the generation of an asynchronous signal flow produced in correspondence to a variation of brightness of each pixel inside the matrix. Each of them is independent from the others, and that allows to signal a brightness lowering or increase of the pixel itself.

As already described in paragraph 1.4, this type of sensor has a very low latency and an accuracy, in the temporal domain, of about 1 μs.

Mathematical description On the sensor output, the event-flow generated by a moving object is defined as function of spatial and temporal coordinates. If $e(p,t) = (p,t)^T$ is a generic event, p represents its position in spatial coordinates, and thus $p = (x,y)^T$.
It is possible to describe a function Σ_e that associates to each space position its related instant of time, that is the time when the event was generated. Consequently:

$$\Sigma_e : \mathrm{N}^2 \rightarrow R \tag{2.6}$$

$$p \mapsto \Sigma_e(p) = t$$

Graphically, it assumes the conformation as in Figure 2.1.
Time is an increasing variable, and the function just described is monotonously increasing. Its partial derivatives will be, as a consequence:

$$\Sigma_{e_x} = \frac{\partial \Sigma_e}{\partial x} \tag{2.7}$$

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

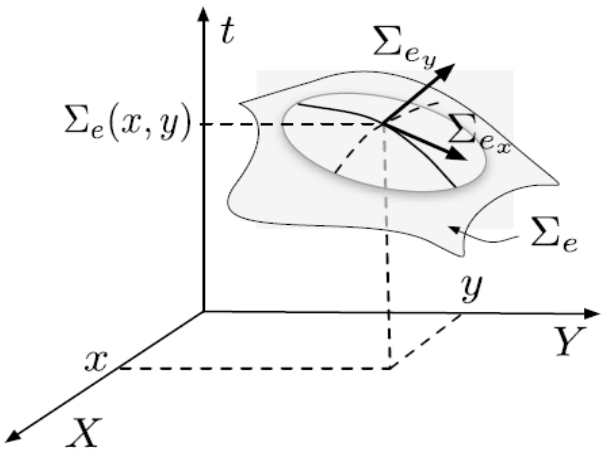


Figure 2.1. Surface of active events for optical flow computation

From which, assuming to have small spatial variations, it is easy to derive:

$$\Sigma_e(p + \Delta p) = \Sigma_e(p) + \nabla \Sigma_e^T \Delta p + o(\|\Delta p\|) \tag{2.8}$$

With:

$$\nabla \Sigma_e = \Big(\frac{\partial \Sigma_e}{\partial x}, \frac{\partial \Sigma_e}{\partial y} \Big)^T \tag{2.9}$$

The partial derivatives of Σ_e depend exclusively on x or y. But also, due to its monotonously growing behaviour, its derivatives will never be null in any point.
This allows to adopt the theorem of the inverse function, in a point $p = (x, y)^T$ from which it is possible to derive:

$$\frac{\partial \Sigma_e}{\partial x}(x, y_0) = \frac{d\,\Sigma_e|_{y_0}}{dx}(x) = \frac{1}{v_x(x, y_0)} \tag{2.10}$$

$$\frac{\partial \Sigma_e}{\partial y}(x_0, y) = \frac{d\,\Sigma_e|_{x_0}}{dy}(y) = \frac{1}{v_y(x_0, y)}$$

The gradient can therefore be written as:

$$\nabla \Sigma_e = \Big(\frac{1}{v_x}, \frac{1}{v_y} \Big)^T \tag{2.11}$$

The reverse components of this vector are nothing more than the components of the velocity vector along x and y, estimated at the point $p = (x, y)^T$.

In Figure 2.3 are shown the optical flow results, obtained from some experimental tests making rotate a disk(Figure 2.2) with a bar drawn on it from the centre to the outer circumference.

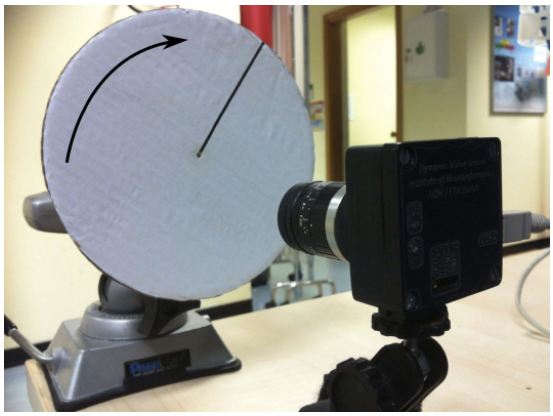


Figure 2.2. The rotating disk

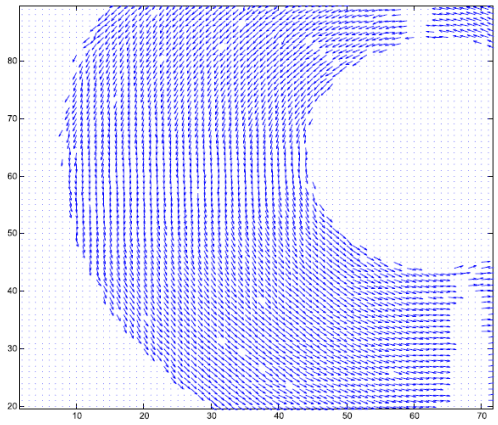


Figure 2.3. Optical flow estimation

This type of function is, however, very susceptible to noise, because the partial derivatives are estimated on time, for each individual event. To avoid this type of problem it is assumed that the local speed is constant. This hypothesis is satisfied for a small number of grouped events, i.e. assuming that the function $\sum e$ is locally planar.

The algorithm in its complexity is described in Figure 2.4.

Starting from each new incoming event, a three-dimensional space-time window centred on the event itself is taken into consideration.
The idea is to fit the plane that best contains the events inside the window. In this way it is possible to find, through various cycles, the parameters of the plane such that the equation (2.12) is satisfied:

$$\Pi^T \begin{pmatrix} p \\ t \\ 1 \end{pmatrix} = 0 \tag{2.12}$$

The th_1 value is set to 10^{-5}, and corresponds to the accuracy searched from the estimation. th_2, instead, is equal to 0.05 and comes directly from the experimental results.
However, a maximum of two algorithm iterations are required to converge the results.

2.3 Performances and limitations

The advantages of the proposed model, compared to the current frame-based processing techniques, are that, first, it considerably reduces computation times and costs (15 % less, compared to traditional calculation methods), but also a much lower expense in terms of power consumption is achieved. This algorithm is extremely versatile for the use in different fields, as well as the optical flow estimation [22].

Algorithm 1 Local planes fitting algorithm on incoming events.

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

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

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

  while $\epsilon > th_1$ **do**
    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.
    Set $\epsilon = ||\tilde{\Pi} - \tilde{\Pi}_0||$, then set $\tilde{\Pi}_0 = \tilde{\Pi}$.
  end while
  Attribute to e the velocity defined by the fitted plane.
end for
return $v_x(e)$, $v_y(e)$.

Figure 2.4. Optical Flow Algorithm

One of the factors that limit the accuracy of the flow calculation is due to the presence of non-idealities in the sensor. The higher the speed of movement, the more difficult it is for the bio-inspired retina to receive and produce signals correctly. In addition, the number of events, in these cases, almost never corresponds to the number of pixels that theoretically should be activated at the motion of the object.

The number of them is in fact less than expected. It follows that also the fitting of the flow is inevitably affected by errors. In order to limit this problem, the use of the device in good lighting conditions can be a recommended solution.

Chapter 3

Modelling the algorithm

3.1 Matrix computation flow

In order to obtain the output differentials of the algorithm, it is necessary to create a flow of algebraic-mathematical operations that will be used as basis for the software, first, and then hardware implementation [23].

As already known, each event coming from the vision sensor (in this case it is supposed to work on DVS) is associated with a triplet of informations: two spatial and one temporal. The first one (x,y) represents the coordinates of the pixel where an event has been recorded, the other one instead is the time stamp, that is the instant when it has happened.

The algorithm aims to extract a space-time window surrounding the new event, which is its centre, and from it make an estimation of the inverse of the velocity vector components associated to it.

Assuming to work, i.e., in a neighbourhood represented by a 3x3 window, the A matrix is defined as:

$$A = \begin{pmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \\ x_4 & y_4 & 1 \\ x_5 & y_5 & 1 \\ x_6 & y_6 & 1 \\ x_7 & y_7 & 1 \\ x_8 & y_8 & 1 \end{pmatrix} \tag{3.1}$$

It includes, in its first two columns, the couple of the event coordinates. The first column of the 3x3 window is put in correspondence of the A rows 0 to 2, the second in the rows 3

to 5, and the third from 6 to 8. The third column of A, instead, represents an indicator of the validity of an event. If there is an event and it is not null, then in the same row there will be a 1, otherwise a 0.

By defining, in the same way, the Y vector of time-stamps:

$$Y = \begin{pmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \end{pmatrix} \tag{3.2}$$

The first three lines of Y belong to the first column of the 3x3 window, the second three lines of Y represent the second column of the window, and the third three lines the third column.

As already explained in paragraph 2.2, the purpose of the algorithm is to model locally the structure of the plane that best fits the course of events contained inside the the space-time neighbourhood.

The equation of a plane is given by:

$$ax + by + ct + d = 0 \tag{3.3}$$

With a, b, c and d which are the coefficients of the plane itself, and that provide its orientation and angle.

Normalizing with respect to c, the 3.3 becomes:

$$ax + by + d = -t \tag{3.4}$$

Its vectorial form is:

$$\begin{pmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \\ x_4 & y_4 & 1 \\ x_5 & y_5 & 1 \\ x_6 & y_6 & 1 \\ x_7 & y_7 & 1 \\ x_8 & y_8 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ d \end{pmatrix} = - \begin{pmatrix} t_0 \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ t_5 \\ t_6 \\ t_7 \\ t_8 \end{pmatrix} \tag{3.5}$$

By multiplying now both sides for A^T, the 3.6 is retrieved:

$$A^T A \begin{pmatrix} a \\ b \\ d \end{pmatrix} = A^T Y \tag{3.6}$$

That is, writing all the components:

$$\begin{pmatrix} \Sigma x_i x_i & \Sigma x_i y_i & \Sigma x_i \\ \Sigma y_i x_i & \Sigma y_i y_i & \Sigma y_i \\ \Sigma x_i & \Sigma y_i & N \end{pmatrix} \begin{pmatrix} a \\ b \\ d \end{pmatrix} = - \begin{pmatrix} \Sigma x_i t_i \\ \Sigma y_i t_i \\ \Sigma t_i \end{pmatrix} \tag{3.7}$$

Applying to the 3.7 the matrix inversion formulas, it is possible to obtain the plane coefficients:

$$\begin{pmatrix} a \\ b \\ d \end{pmatrix} = (A^T A)^{-1} (A^T Y) \tag{3.8}$$

The first step is now to understand how the inverse matrix can be computed, given the product of A and its transposed A^T. Generally, the rule is:

$$(A^T A)^{-1} = \frac{1}{det(A^T A)} adj(A^T A) \tag{3.9}$$

With $adj(A^T A)$ that is:

$$adj(A^T A) = \begin{pmatrix} adj_{11} & adj_{12} & adj_{13} \\ adj_{21} & adj_{22} & adj_{23} \\ adj_{31} & adj_{32} & adj_{33} \end{pmatrix} \tag{3.10}$$

Each element is developed as follows:

$$\begin{aligned} adj_{11} &= N \Sigma y_i^2 - \Sigma y_i \Sigma y_i \\ adj_{12} &= \Sigma x_i \Sigma y_i - N \Sigma x_i y_i \\ adj_{13} &= \Sigma x_i y_i \Sigma y_i - \Sigma x_i \Sigma y_i^2 \\ adj_{22} &= N \Sigma x_i^2 - \Sigma x_i \Sigma x_i \\ adj_{23} &= \Sigma x_i y_i \Sigma x_i - \Sigma x_i^2 \Sigma y_i \end{aligned} \tag{3.11}$$

The matrix product $(A^T A)^{-1} A^T Y$ returns:

$$\begin{aligned} a_{det} &= adj_{11} \Sigma x_i t_i + adj_{12} \Sigma y_i t_i + adj_{13} \Sigma t_i \\ b_{det} &= adj_{12} \Sigma x_i t_i + adj_{22} \Sigma y_i t_i + adj_{23} \Sigma t_i \\ det &= adj_{11} \Sigma x_i^2 + adj_{12} \Sigma x_i y_i + adj_{13} \Sigma x_i \end{aligned} \tag{3.12}$$

From which, if both a_{det} and b_{det} are divided for the determinant det, the values of the coefficients a e b are available.

Now, to have a convergence of the results is necessary to iterate at most twice. The idea is therefore, once obtained the coefficients of the plan, to impose the passage of the events in the plan itself, and to verify if the obtained result is or not inferior to a certain threshold limit.

$$a(x - x_c) + b(y - y_c) + (t - t_c) < thr \tag{3.13}$$

If this inequality is verified, then the event is kept valid inside the space-time window, otherwise it is considered as outlier and deleted.

This process is done on all elements of the matrix and, once completed, the entire algorithm is repeated from the beginning. At the end of the iteration, remains to calculate the inverse components of the speed.

First of all, speed and angle are calculated:

$$\begin{aligned} sp &= \frac{1}{\sqrt{a^2 + b^2}} \\ \theta &= arctan(\tfrac{a}{b}) \end{aligned} \tag{3.14}$$

Starting from them the values of the differentials are:

$$\begin{aligned} \frac{dt}{dx} &= sp \times cos(\theta) \\ \frac{dt}{dy} &= sp \times sin(\theta) \end{aligned} \tag{3.15}$$

3.2 Matlab description

The goal is to have a totally flexible and parametrizable structure in the software modelling, able to emulate the algorithm just explained. In this way, then, it will be possible to adapt the proposed architecture for all types of data, coming from different vision sensors.

The adopted dataset for the experimental tests in exam is produced by the Italian Institute of Technology [par. 2.2]. It is the result of simulations carried out from two 128x128 pixel DVS sensors mounted on iCub. What was done by the robot is a rotary movement of the head in front of a table with some objects above it.

The set of events is grouped, row by row, in a data package that covers six fields expressed in columns:

- **Channel**: i.e. which channel the event comes from (0: left, 1: right);
- **Time stamp**: in clock count;
- **Polarity**: (ON/OFF);
- **Y coordinates**;
- **X coordinates**;
- **N.A.**

The first task of the Matlab list is to separate the components according to the eye from which the events come, in order to avoid mixing data between different sensors.
The second step is to differentiate events with ON polarity, i.e. that have had a positive brightness variation, from those with OFF polarity.

Four different lists of events, separated and organized by camera and polarity are the results of this operation. Starting from now, the parameters used to calculate the visual flow are then defined.

Summarizing and classifying them:

- **dxsx**: allows to select the camera of interest. It is 0 for the right eye, and 1 for the left;
- **pol**: discriminates the polarity. Its value can be 0 for ON, and 1 for OFF;
- **dT_in** : selects the width of the time window (in s);
- **Rf**: represents the number of lines that make up the visual matrix of the sensor (corresponds to the size Y);
- **Cf**: represents the number of columns that make up the visual matrix of the sensor (corresponds to the X dimension);
- **R**: represents the number of rows of the spatio-temporal window;
- **C**: represents the number of columns of the spatio-temporal window
- **min_events**: is the minimum number of events for which the space-time window is considered valid;
- **n_iter**: allows a choice between none, one or two iterations of the algorithm;
- **thr_in**: is the threshold selected to satisfy the iteration inequality, for the elimination of outliers;

- **n_shift** (valid only for the fixed-point version): is the number of bits with which an initial and final right-shift is made, in order to control and manage the overflow problem of the time stamp counter.

An initial dialogue screen allows the configuration of all system variables. However, it is after the conversion of dT_in from seconds to number of clock cycles that the operation of the algorithm starts.

The conversion is done by dividing the reference time value by 80*10-9: a 100 MHz system clock, and a ”fictitious” clock of 12.5 MHz to update the internal register of the sensor (the one related to time stamps) are considered.

Graphically, the software algorithm performs the following operations:

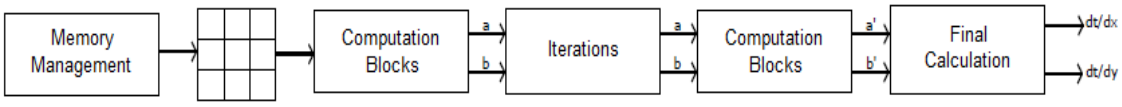


Figure 3.1. Operations in the algorithm

A problem to be taken into account is linked to the extraction of space-time windows that overcome the size of the visual matrix. Let’s suppose, as in Figure 3.2, to receive an event at the pixel coordinates (1,1) (in green) and to take a 3x3 local matrix (in orange).
However, in the local matrix around the pixel just extracted, the events belonging to the first row and the first column do not exist. This is because it goes beyond the detection spectrum of the vision sensor.

In this case, in fact, the neighbourhood will be not a 3x3 but a 2x2 one, which takes only the existing elements boundary to the reference pixel. The dimensions are not sufficient with respect to those required and, for this reason, the space-time window is not processed by the software.

The wrap-around problem
As previously described, the presence of a 24-bit counter inside the DVS allows to record the time when the event occurs. The value of the register is in fact frozen and packaged with its additional informations, according to the handshake of the device. However, it remains to be understood how its reset can be solved and managed in purely mathematical terms, when all the 2^{24} possible combinations have been counted.
An overflow in fact, occurs. This condition is verified when an event at the instant of time t+1 has a time stamp lower than an event recorded in the previous t time. This leads to computational problems that risk to compromise the possible output results of the device.

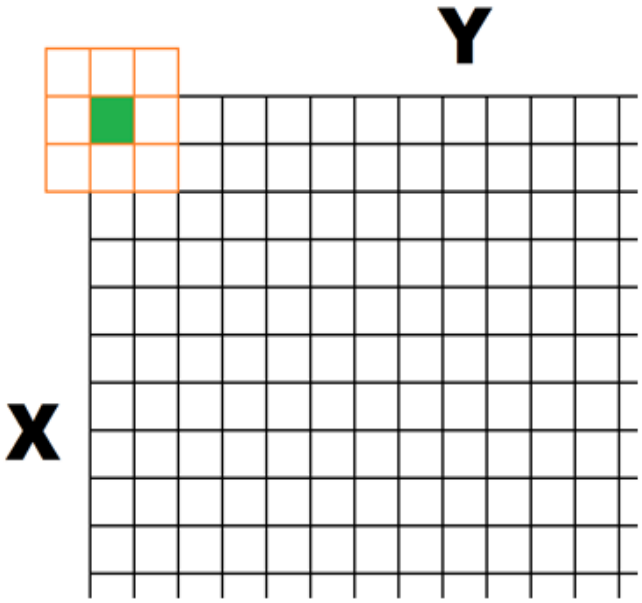


Figure 3.2. Local window that exceeds visual dimensions

To solve this error, a technique that allows to obtain, even in case of multiple overflows, the real distance between two values of the counter is adopted.

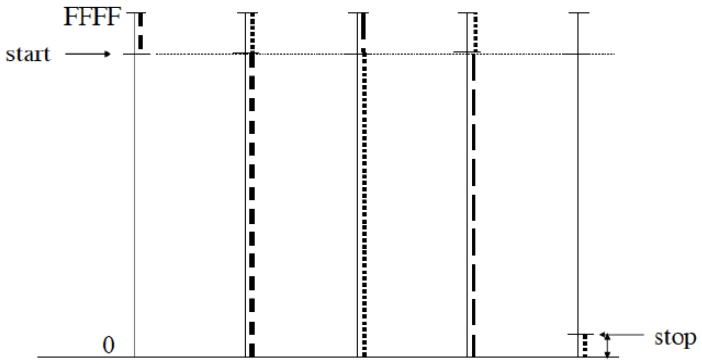


Figure 3.3. Multiple Overflows in a counter

Starting from zero, the first value (start) is recorded at a certain instant of time. In the meantime, the register count grows until it reaches the FFFF value. Here, it resets and starts again from scratch, growing again. And so, for three more cycles, until the value is recorded again instantly (stop).

Looking at the Figure 3.3, then, it is possible to get the expression:

$$\Delta T = (2^N M) + (stop - start)$$

Where M is the number of overflows occurred between the two instants under examination, and N is the number of bits of the counter.

This is a general model however, seen the departure hypothesis, it is not necessary the use of M values of higher than 1. This because the temporal window, considered according to the same specs of the scientific paper explained in the paragraph 2.2, is at the most of 1 ms.
The supposition that between two events has occurred more than one overflow may be interpreted as a wider time-window than the desired one. This is because, in the worst case hypothesis, an event-rate of 1evt/80 ns is considered.

It is important now to understand how this technique can be implemented in software, first, and then in hardware.

There are two models designed in Matlab: floating-point and fixed-point. They represent the same algorithm, but one based on mobile arithmetic, and the other in fixed arithmetic in order to act as a simulation tool before the implementation at RTL level. The solution of the wraparound problem has been realized in two different ways for both.

Floating-point model
For each element of the dataset appropriately differentiated by eye and polarity, the first three columns are considered:

- Time-stamp
- X
- Y

The first step of the algorithm is to compare each Ts of the last event with the value of the previous one. If it is greater, then no overflow has taken place, vice versa yes. In this last case, a quantity equal to 2^{24} is subtracted to all the previous elements saved in the visual matrix (if there has already been one), and will be added instead to the new and the following one.

In example, let's suppose to have two versions of the dataset, with and without overflow [Tab. 3.1].

t	No overflow	Overflow
1	16760784	16760784
2	16776334	16776334
3	16776979	16776979
4	16829919	52703
5	16831704	54488

Table 3.1. No overflow data vs Overflow data

An overflow occurs instantly at t = 4. It turns out, in fact, that the value of the counter is lower than the other of the previous instant.

The differences between the elements at a certain time and the relative antecedents are:

t	$\Delta novf$	Δovf
1	0	0
2-1	15500	15500
3-2	645	645
4-3	52940	-16724276
5-4	1785	1785

Table 3.2. No overflow vs Overflow differences

As can be seen, the Δ remains the same, however, the problem arises precisely at the wrap.

Supposing that this overflow is the first in the incoming set of events, the behaviour of the device will be:

t	Dataset	Managed	Δovf
1	16760784	16760784	0
2	16776334	16776334	15500
3	16776979	16776979	645
4 ($+2^{24}$)	52703	16829919	52940
5 ($+2^{24}$)	54488	16831704	1785

Table 3.3. Ideal data vs Managed data (first overflow)

In the first column there are the values coming from the dataset, as they have been recorded by the vision sensor.

At t = 4 the wrap occurs, and the value of the current Ts is obviously lower than the one in the previous instant of time. From this moment, a quantity equal to 2^{24} is added to all the following values (including the current one) of the events. This mechanism is described in the second column.

The algorithm continues its analysis on the arriving events and, at t = n, a second overflow occurs:

t	Dataset	Saved	Managed
$n-3$ (-2^{24})	33554053	33554053	16776837
$n-2$ (-2^{24})	33554274	33554274	16777058
$n-1$ (-2^{24})	33554427	33554427	16777211
n ($+2^{24}$)	33554531	99	16777315
$n+1$ ($+2^{24}$)	33554789	357	16777573

Table 3.4. Ideal data vs Managed data (second overflow)

The first column (Dataset) shows the real value of the time stamps without any overflow. In the second column (Saved), instead, the data actually saved in the matrix are shown. As soon as the overflow is detected, a quantity equal to 2^{24} is subtracted from the values already saved (previous to the instant 4 when it happened), and from this moment the same quantity is then added to the incoming time stamps. This is described in the third column.

It is interesting to notice how the Δ calculated for the real values of the Ts and those managed by the control mechanism described above are the same:

t	$\Delta Dataset$	$\Delta Managed$
1	0	0
2-1	221	221
3-2	153	153
4-3	104	104
5-4	258	258

Table 3.5. Ideal differences vs Managed differences

Fixed-point model

For the fixed-point model, too, the same mechanism analysed in the previous paragraph has been reproduced.

The idea is the same, however, working on a limited number of bits requires to look for a solution that allows to keep unaltered the parallelism of the data. A simple way can be to

use an extra bit of control that allows to manage the overflow problem.

Let's assume, for example, to work on 2-bit data. A counter follows the cycle indicated:

Control bit	Sequence	Value
0	00	0
0	00	1
0	10	2
0	11	3
1	00	4
1	01	5
1	10	6
1	11	7

Table 3.6. Two bits counter with control bit

However, it is not possible to add one more guard bit for any new overflow that occurs. Mainly because it is not possible to predict a priori the number of resets that the counter will face, and secondly because it would require an unacceptable increase in the complexity of the architecture.

From specs, moreover, the problem is to maintain an amplitude of 24 bits in the field related to the Ts.

Referring then to the second principle of the equations equivalence, according to which if both the members are multiplied or divided by the same quantity the equality is satisfied, it is possible to find a strategy to manage the problem.

Assuming in fact to have two instants t_n and t_{n-1}, and that the difference between them is equal to a quantity p, then:

$$t_{n-1} - t_n = p$$

By dividing the two sides for the same q quantity:

$$\frac{t_{n-1} - t_n}{q} = \frac{p}{q}$$

And then, multiplying everything again by q, the original equation will be recovered.

Therefore, using the two tools described above:

- Control bits

- Second principle of equality

The management of the wrap-around in fixed-point arithmetic is trivial.

The presence of a control bit is necessary to calculate the difference between two time values, even when an overflow occurs. The next step is to make a right-shift of the Ts of a certain amount (by using the control-bit as MSB), to keep unchanged the 24-bit parallelism. At the end of the algorithm, finally, a left-shift of the same amount allows to reach the expected result.

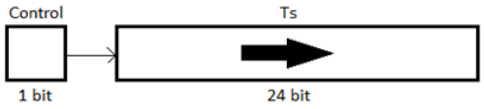


Figure 3.4. Working principle of the control-bit

Repeating, from a numerical point of view, the same series of processes already seen in the previous paragraph, it is possible to have the theoretical values as:

t	No overflow	No overflow (binary)
1	16760784	111111111011111111010000
2	16776334	111111111111110010001110
3	16776979	111111111111111100010011
4	16829919	100000000110011011011111
5	16831704	100000000110101001011000

Table 3.7. No overflow integer vs No overflow binary (first ovf)

The real values are:

t	Overflow	Overflow (binary)
1	16760784	111111111011111111010000
2	16776334	111111111111110010001110
3	16776979	111111111111111100010011
4	52703	000000001100110111011111
5	54488	000000001101010011011000

Table 3.8. Overflow integer vs Overflow binary (first ovf)

As can be seen from the binary representation, it is evident that the presence of an extra bit able to indicate the presence of the overflow is fundamental.

The control mechanism of the guard bit is quite simple: as soon as the algorithm starts, taking the first data from the dataset, Ctrl_bit is set to 0. When the first overflow occurs, Ctrl_bit is put equal to 1 for all subsequent events, including the current one.

t	Overflow	$Ctrl\_bit$	Overflow (binary)
1	16760784	0	111111111011111111010000
2	16776334	0	111111111111110010001110
3	16776979	0	111111111111111100010011
4	52703	1	000000001100110111011111
5	54488	1	000000001101010011011000

Table 3.9. Overflow integer vs Managed binary (first ovf)

Since the number of bits is limited, assuming to perform a shift of one position to right, and then dividing the values by two, results that:

t	Overflow	Right-shift overflow (binary)
1	8380392	011111111101111111101000
2	8388167	011111111111111001000111
3	8388489	011111111111111110001001
4	8414959	100000000110011011101111
5	8415852	100000000110101001101100

Table 3.10. Overflow integer vs Right-shift overflow (first ovf)

Comparing the differences between current and previous instants:

t	$\Delta novf$	$\Delta ovf + rightshift$
1	0	0
2-1	15500	77755
3-2	645	322
4-3	52940	26470
5-4	1785	893

Table 3.11. Ideal differences vs Right-shift differences (first ovf)

It is evident, as expected, that the time distances are halved.

By proceeding with the execution, at a certain time t = n a second overflow occurs: in this case, the control bits previously saved as 1 are reset (all the Ctrl_bit inside the matrix become equal to 0), while those from the same moment on are set to 1.

The expected data are:

t	No overflow	No overflow (binary)
n-3	33554053	11111111111111111010000101
n-2	33554274	11111111111111111101100010
n-1	33554427	11111111111111111111111011
n	33554531	10000000000000000001100011
n+1	33554789	10000000000000000101100101

Table 3.12. No overflow integer vs No overflow binary (second ovf)

The orginal data are:

t	Overflow	$Ctrl\_bit$	Overflow (binary)
n-3	16776837	0	11111111111111010000101
n-2	16777058	0	11111111111111101100010
n-1	16777211	0	11111111111111111111011
n	16777315	1	00000000000000001100011
n+1	16777573	1	00000000000000101100101

Table 3.13. Overflow integer vs Managed binary (second ovf)

By right-shifting the new values the new values are:

t	Overflow	Right-shift overflow (binary)
1	8388418	01111111111111101000010
2	8388529	01111111111111110110001
3	8388605	01111111111111111111101
4	8388657	10000000000000000110001
5	8388786	10000000000000010110010

Table 3.14. Overflow integer vs Right-shift overflow (second ovf)

The differences between current and previous events are now shown in [Tab. 3.15].

Also in this case, as expected, the Δ obtained from the wraparound management are halved (unless an approximation error) with respect to those expected: this, as already pointed out several times, is directly related to the right-shift carried out.

And for this reason the number of shifts parameter has been introduced in the user interface. Thanks to this, the wrap management process allows to: add a control bit as MSB

t	$\Delta novf$	$\Delta ovf + rightshift$
1	0	0
2-1	221	111
3-2	153	76
4-3	104	52
5-4	258	129

Table 3.15. Ideal differences vs Right-shift differences (second ovf)

of the Ts and, only afterwards, make a right-shift of the desired number of bits.

At the end of the computation, the result has to be shifted to the right by the same amount and not, as expected, to the left. This is because, at the end of the square root calculation, the data is inverted.

This implies that, given a generic value x as input, if a n-positions right-shift is done, a division by a n-power of two is performed:

$$x \rightarrow \frac{x}{2^n}$$

The reverse value of it twill be:

$$\frac{1}{x} \rightarrow 2^n \frac{1}{x}$$

This means that the result will be 2n times higher than the expected. To normalize and get the correct outcome, a shift again to the right of n positions has to be done:

$$2^n \frac{\frac{1}{x}}{2^n} \rightarrow \frac{1}{x}$$

Data parallelism and algorithm development
The first thing that is verified inside the visual matrix is the validity of all events: if any of them has a time distance, with respect to the last Ts arrived, greater than the interval defined by dT_in (i.e. the event is expired), then it will be deleted. This means that a 0 will be inserted in its place.

Once the data has been allocated to the matrix and verified its validity, the space-time window is extracted. In the worst case it will be 15x15.

The next step is the processing phase in the Mult.m unit. In this function are defined:

- **Vector X**: on 4 bits, because in the worst case (15x15) the coordinates of the elements from 0 to 14 are expressed;
- **Vector Y**: on 4 bits, the same considerations for X are valid for it;
- **Ts**: on 24 bits.

For both X and Y, all coordinates corresponding to zero Ts values are deleted. This is due to the presence of a null time-stamp, that indicates the absence of an event in that pixel.

Now the computation of the product $A^T A$ has to be performed, as expressed in paragraph 3.1. For this reason, the following vectors are defined for all the elements of the matrix:

- X: on 4 bit;
- Y: on 4 bit;
- T: on 24 bit;
- X2: on 8 bit;
- Y2: on 8 bit;
- XY: on 8 bit;
- XT: on 28 bit
- YT: on 28 bit;
- N: on 8 bit, it represents the number of valid events in the local matrix;

Subsequently, each element of the above vectors is added in order to retrieve the terms of the product matrix. To obtain the number of bits necessary to prevent any overflow in the sum, the following formula is applied:

$$bit\_adder = n + \log k$$

With n = number of input bits, k = number of sums and log is the base-2 logarithm.

Therefore, knowing that in the worst case k = 15*15 = 225, for each element will be derived:

- SX: on 12 bit;
- SY: on 12 bit;
- ST: on 32 bit;

- SX2: on 16 bit;
- SY2: on 16 bit;
- SXY: on 16 bit;
- SXT: on 36 bit
- SYT: on 36 bit;

Once the sums of the matrix have been computed, the calculation of the inverse is carried out. The function called to perform this task is Adjoint.m.

Its components can be calculated according to the formulas inside paragraph 3.1, and have the following parallelism:

- adj11: 24 bit;
- adj12: 24 bit;
- adj13: 28 bit;
- adj22: 24 bit;
- adj23: 28 bit;

Obtained these values, the next step is to have $[a, b, det]^T$, coming from the matrix product $(A^T A)^{-1}(A^T Y)$:

- a_{det}: 62 bit;
- b_{det}: 62 bit;
- det: 42 bit;

The last thing to do is to compute the inverse components of the velocity vector. It is the Cordic.m function that performs this work.

The calculation of a and b values is derived by the ratio between a_{det} and b_{det} with det. Knowing that the number of the quotient bits between two integers is given by their width difference:

- a: 25 bit, 21 of integer part and 4 of fractional part;
- b: 25 bit, 21 of integer part and 4 of fractional part;

To calculate the theta angle, these values are passed to a function that calls up the arctangent in fixed-point (atan2), which returns a 16-bit value (3 integers and 13 fractional) at the output. Then the angle goes to the cordicsincos, that allows to calculate its sine and cosine. On the output, both results will also be on 16 bits (2 integers and 14 fractional).

To obtain, instead, the inverse square root, the first step consists in calculating the squares of both coefficients.

- a2: 50 bit, 42 of integer part and 8 of fractional part;
- b2: 50 bit, 42 of integer part and 8 of fractional part;

Then they are added together, thus obtaining a ”c” value on 51 bits, in order to avoid any overflow (43 integers and 8 fractional). The next module is the square root: Cordicsqrt is the function, and provides out a result on 51 bits (23 integers and 28 fractional). By inverting the output coming from the square radix, the desired value is allocated on 26 bits (6 integers and 20 fractional).
Multiplying both sine and cosine values by the inverse of the root just found, the components of the gradient are ready. Both dt/dx and dt/dy are on 42 bits (34 integers and 8 fractional).

3.3 Results and comparisons

In this section the results coming from the Matlab script simulations are shown. The dataset belongs to two DVS recordings and so, starting from it, the chosen parameters are expressed in the table below.

Parameter	Value
dxsx	0
pol	221
dT_in	153
Rf	104
Cf	258
R	3-5-9-15
C	3-5-9-15
min_events	3-8-15-21
n_iter	0-1-2
thr_in	0.05
n_shift	1-5-8

Table 3.16. Parameters used for the simulations

All the optical flow computations have been done by fixing the dimensions of the visual matrix (128x128), the polarity (ON) and the right-eye as reference.
By varying the size of the local space-time matrix and associating to each of them a minimum number of events, the flow vectors are computed for different numbers of iterations (from 0 to 2).
The information losses, moreover, are caused by the number of shifts done on the Ts: for that reason, their impact on the results has been evaluated in a statistic way.

n_shift = 1

For a right-shift of one position, the mean error, the standard deviation and the distribution of the error for the three types of iterations are computed.

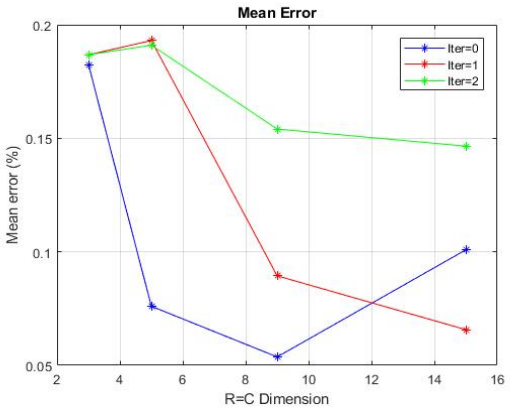


Figure 3.5. Mean Error for r-shift = 1

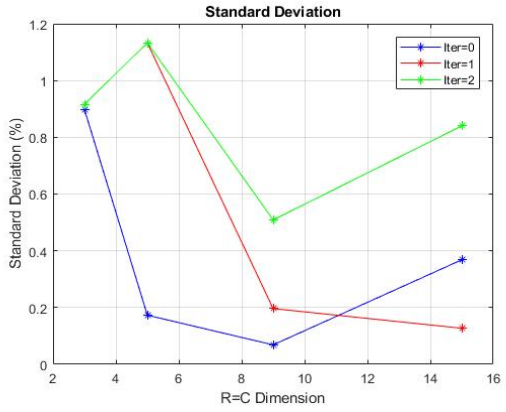


Figure 3.6. Standard Deviation for r-shift = 1

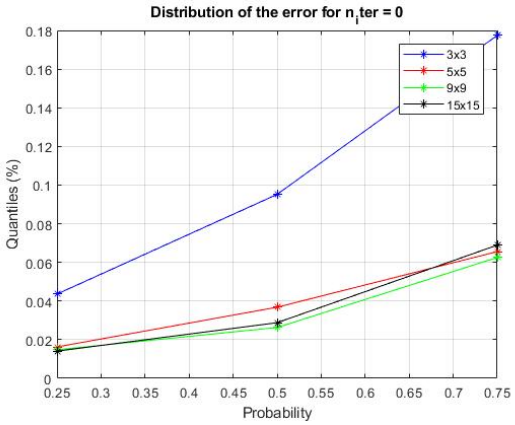


Figure 3.7. Error distribution for iter = 0

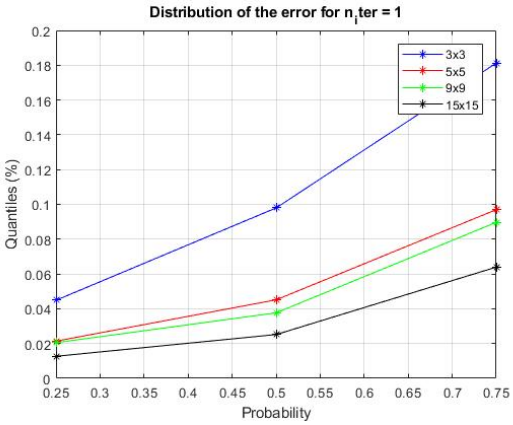


Figure 3.8. Error distribution for iter = 1

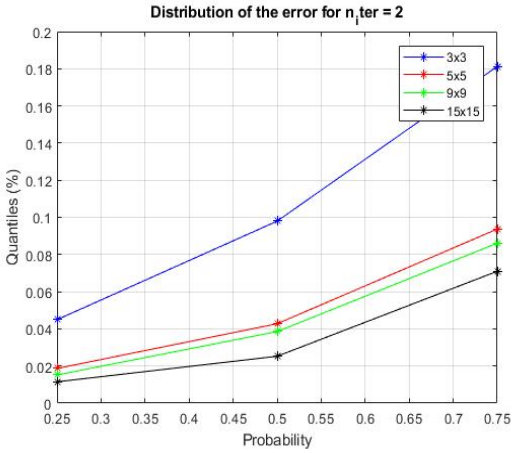


Figure 3.9. Error distribution for iter = 2

As can be seen from the previous graphs, the highest error concentrations are around the 3x3 and 5x5 solutions, for the number of iterations equal to one and two.

Analysing in detail the mean error, the highest percentage is found in correspondence of the smallest matrices. This can be partly explained also because the number of results obtained in output is decidedly higher for small sizes, compared to larger matrices.

For a 3x3 or 5x5 matrix, only 3 or 8 events will be necessary inside them in order to proceed with the processing. By increasing the dimensions of the matrix, the value of the mean error tends to decrease. It is important, however, to observe that the percentages of error are at most contained in a range between the 0.05% and the 0.2%: extremely small values, whose weight can be neglected. In this way all the considerations are superfluous.

However, it is the distribution of the error, for all the possible iterations, that shows the real trend of the same. There are three main informations that can be extrapolated:

- The percentage of error decreases as the size of the local matrix increases: this is due to the lower number of results processed, which also means fewer samples to be compared;
- The error grows as the number of iterations: this is due to the increased number of operations in fixed-point, which accumulate the errors;
- A higher occurrence for the largest errors (around 0.18%), with the same trend described in the first sentence.

It is also important to notice the particular behaviour of the mean error for a 15x15 matrix and $n\_iter = 0$. This is visible also by analysing the distribution of the error for the same number of iterations.

n_shift = 5
The static analysis in this case is done for a five bits right shift.

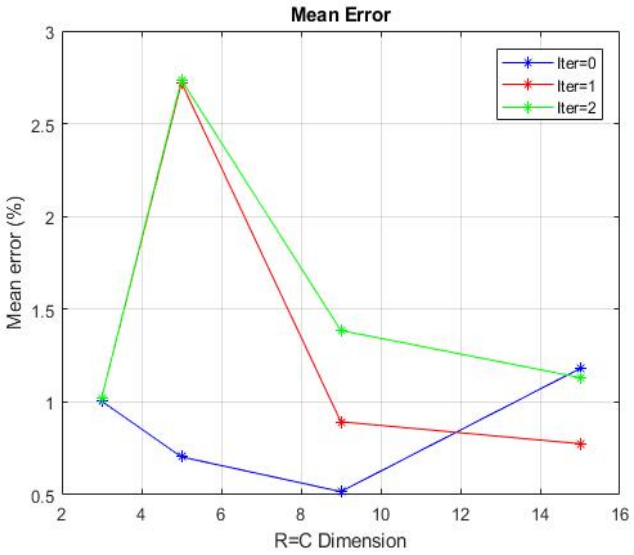


Figure 3.10. Mean Error for r-shift = 5

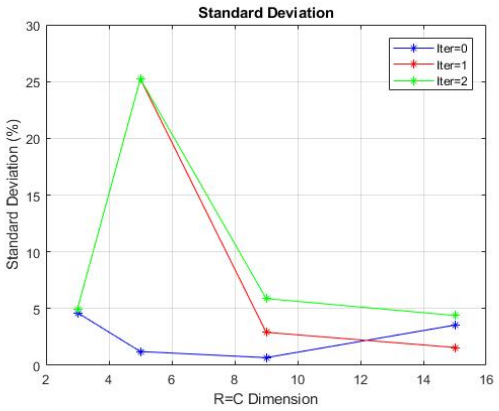


Figure 3.11. Standard Deviation for r-shift = 5

The shift of 5 positions to the right means that, on 24 bits, more or less the 21% of the precision is lost. In this way, the difference between the results can be higher with respect to the previous case, with the possibility to meet nonsense results.

In this case, however, by looking at the mean error, its trend is more or less the same as before. The only difference is related to the increased peak in correspondence of the 5x5 matrix (2.7%), and, the reduced one for the 3x3.

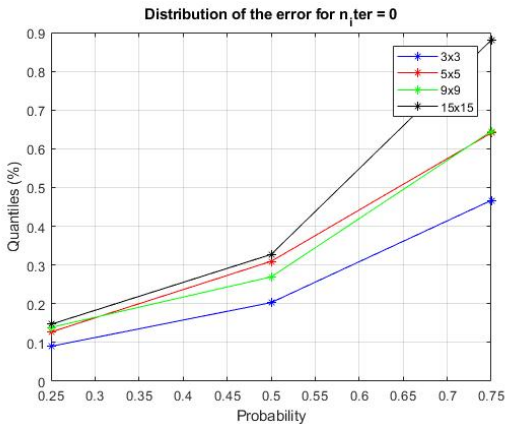


Figure 3.12. Error distribution for iter = 0

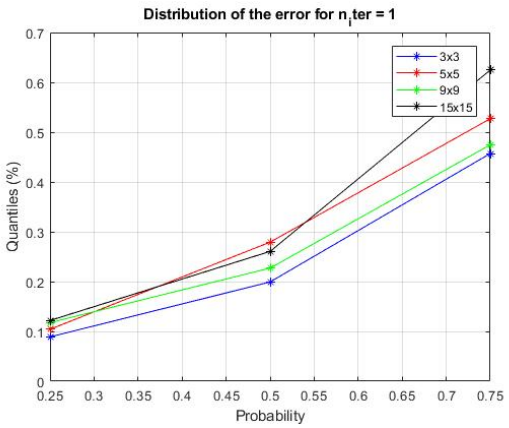


Figure 3.13. Error distribution for iter = 1

But the deep differences emerge focusing on the distribution of the error for all the possible iterations. As just discussed, the error induced by the cut of an important part of the inputs influences in a significant way the result: in fact, the most affected matrix by this issue is the 15x15. In this case, the higher number of elements to be elaborated results in a greater carriage of the error.

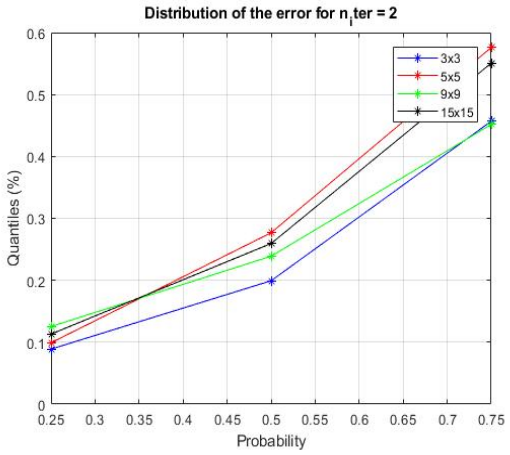


Figure 3.14. Error distribution for iter = 2

This problem is also present, again, in the 5x5 neighbourhood: another explanation is that the way of implementation of the functions used in the Matlab scripts (as cordicsincos, cordicsqrt, divider and so on) for fixed-point version. The same considerations can be done for n_shift = 8.

n_shift = 8

The static analysis in this last case is done for eight bits right shift.

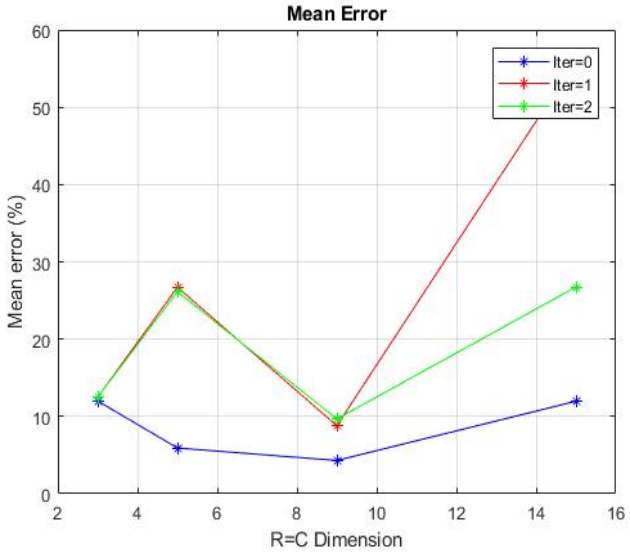


Figure 3.15. Mean Error for r-shift = 8

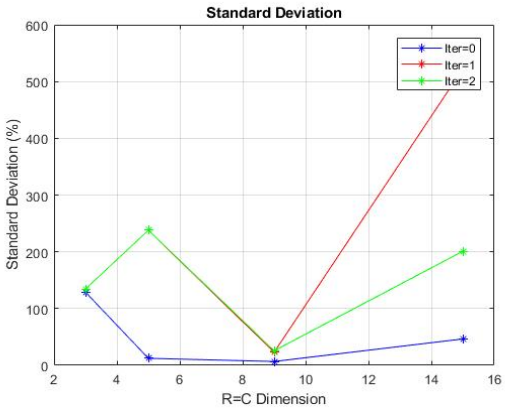


Figure 3.16. Standard Deviation for r-shift = 8

As expressed for n_iter = 5, even more in this case, the results lose all meaning. The main thing that stands out is that, by eliminating about 33% of the time stamp to be processed, the average error goes into a range between 6% and 65%.

The error becomes extremely heavy for a 15x15 matrix, by doing the same considerations as before.

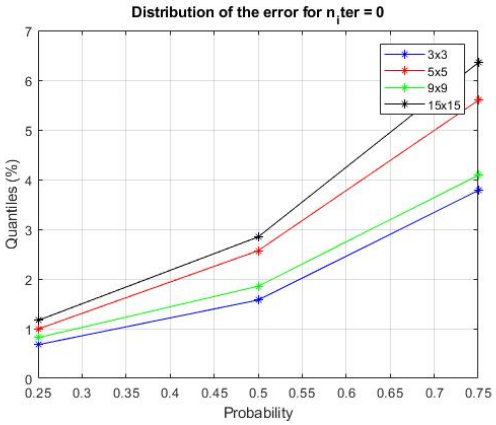


Figure 3.17. Error distribution for iter = 0

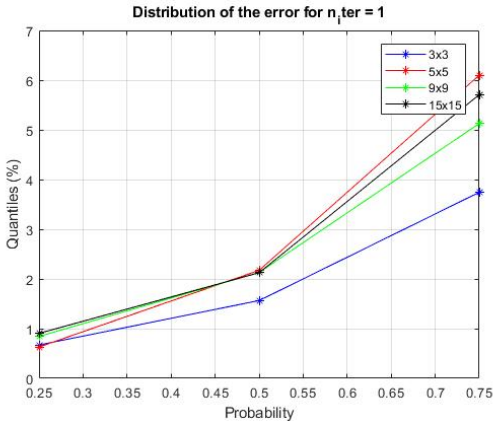


Figure 3.18. Error distribution for iter = 1

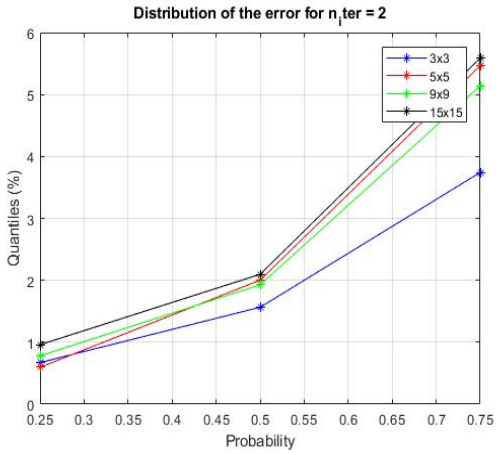


Figure 3.19. Error distribution for iter = 2

As result, the best solution, as expected, is for n_shift = 1. In this way it is possible to take under control the overflow problem, maintaining at the same time the error very low and limited in range.

n_shift	Error range (%)
1	[0.06 – 0.18]
5	[0.5 – 2.6]
8	[6 – 65]

By doing a linear interpolation, the intermediate ranges are obtained:

n_shift	Error range (%)
2	[0.17 – 0.79]
3	[0.28 – 1.39]
4	[0.39 – 1.99]

If the maximum acceptable error has a percentage error lower than 1%, it is possible to perform a right-shift at most of two positions.

Chapter 4

Guidelines for the hardware design

4.1 General view

The purpose of this chapter is to provide some guidelines for the implementation of the hardware structure, basing on the already expressed considerations in the Matlab model. The design flow starts from a definition of the system specifications, passing through the implementation of the building blocks in VHDL language, and finally arriving at the simulation, synthesis and *place&route* of the architecture.
As in the software previously examined, the analyses and the various computations are done separating the data by direction (if right or left eye) and polarity, in the same way the proposed architecture is dedicated to a well-defined type of informations. The proposition of an interface with the data coming from the outside is not the scope of this elaborate. In this case the focus will be only specifically on its processing.
The device therefore presents, in general, as inputs:

- X: vertical coordinate of the event;
- Y: horizontal coordinate of the event;
- Ts: time stamp of the event;
- sready: control signal, asserted at the arrival of a new event from outside;
- clock.

On the outputs, instead:

- dt_dx: inverse component of the velocity vector in the x direction;
- dt_dy: inverse component of the velocity vector in the y-direction;

- go: output signal, asserted at the end of the computation.

The block will therefore have the following conformation:

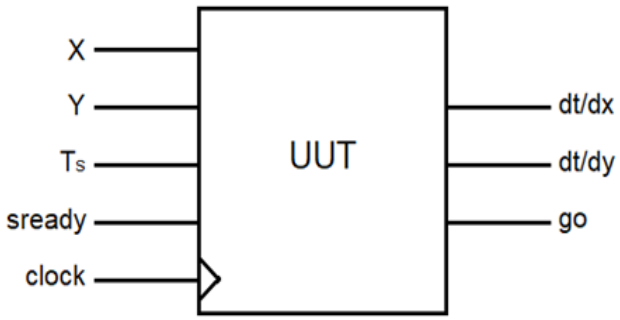


Figure 4.1. The top device

More generally, the pipe queue that composes the entire architecture follows the trend already shown in chapter 3.2.

At the input of the calculation unit there is a memory management block. Its goal is to allocate in the visual matrix the event received at the coordinates (X,Y) provided by the outside and, at the same time, to extract the space-time window around it. In the software algorithm, the allocation of the matrix (R,C) is parametrized: in this way it is easy to provide optical-flow results for different types of sensors and resolutions. In the case of hardware development this solution is not feasible.

However, the size of the memory blocks to be reserved for the allocation of the events has been calibrated for a visual matrix 304x240 (ATIS sensor). This means that have to be allocated:

- 9 bits representing the X coordinates
- 8 bits representing the Y coordinates

Anything prohibits to interface this block with a DVS of size 128x128. The addressing mechanism remains the same, even though fewer bits are needed to represent the coordinates (X,Y). Only the size of the time-stamps changes.

In the first case, in fact, they are expressed on 14 bits, in the second one on 24 bits. The width of a memory line has been sized in the case of 24 bits, just for the worst case.

Once data have been extracted, the next step is to process them through a series of calculation entities set as in the Matlab model: a Mult block for the definition of the $A^T A$

matrix product, followed by an Adjoint block for the computation of $(A^T A)^{-1}(A^T Y)$. The final results will be a_det, b_det, and det.

If necessary, the new window is modelled by an iteration unit that eliminates the outliers, and then repeats the track of the two previous units. At most two cycles are needed to converge.

The last part reserved to the calculation of the differentials includes a divider which returns the a and b coefficients of the plan, followed by a certain number of computational blocks (using the IP cores in the FPGA) which provide at the end the values dt_dx and dt_dy.

Technical characteristics
The SoC used in this work is a Trenz Electronic TE0715-04-30-1C, which mounts a Xilinx Zynq XC7Z030-1SBG485C. The FPGA model belongs to the Kintex-7 series. This target device has been adopted due to the good availability in terms of hardware resources: this allows fewer restrictions on the number of processing units to be used.

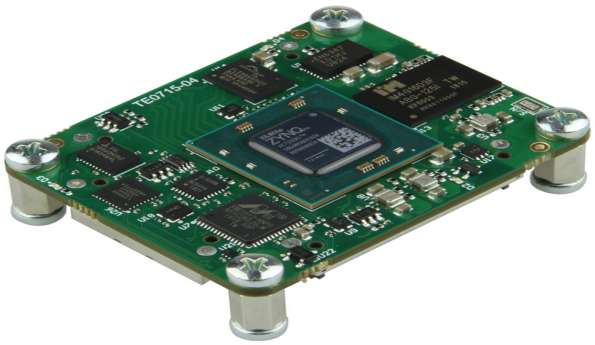


Figure 4.2. The Trenz Electronic SoC

According to the device datasheet, the internal characteristics of the FPGA are:

- Programmable logic cells: 125000;
- LUTs: 78600;
- Flip-Flops: 157200;
- RAM block (36 Kb blocks): 265, true dual-port;
- DSP Slices (18x25 MACCS, 48 bit adder/accumulator): 400;
- f_{max}: 667 MHz;

The oscillator is fixed at 100 MHz: this is the operating frequency of the device. Another fundamental parameter to be taken into consideration is related to the timing of the input events: in the worst case, in fact, the input rate is one event every 80 ns. Is is like to have a new data to be processed every eight clock cycles. This is important to optimize the best resources available, without any waste.

4.2 Memory Addressing

As previously defined, in the worst working case, a new event arrives every 80 ns. The idea, therefore, is to do all the necessary operations to extract from the memory (and then process) the neighbourhood matrix in the eight available clock cycles between the arrival of an event and the next.

Assuming, then, to work in the worst case condition, that is with a space-time window of 15x15 (so that 15 columns have to be extracted in eight cycles) the only solution is to take two columns per cycle out from the memory. Therefore, one of the key points of the design is that, for any size of the window, two columns at a time are elaborated. Each extracted column contains a number of events equal to twice the number of rows of the window itself, and each event, in turn, is represented on 24 bits.
The idea of how to organize the data in the memory is to divide the vertical dimension of the visual matrix (X) into a number of blocks of 15 pixels.

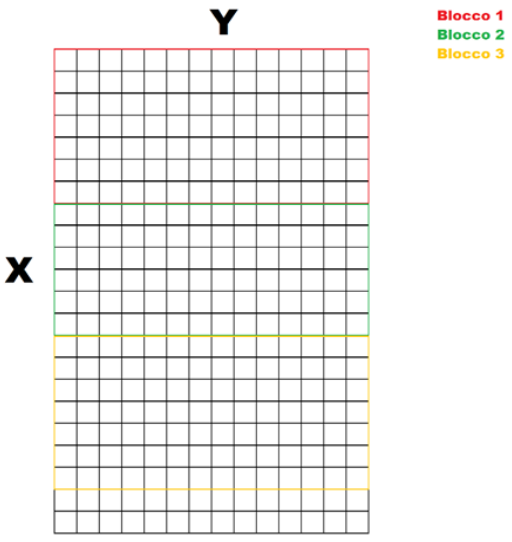


Figure 4.3. Block separation

Let's assume to work with an ATIS sensor. Its matrix dimensions are 304x240.

It will result:

- Along the X direction a subdivision into 21 macro-blocks of 15 lines (pixels) each;
- 240 columns for each macro-block.

It should be noted that 21 macro-blocks of 15 rows correspond to a size (X) of 315 elements, that is 11 more than the actual 304. The latter eleven are the same allocated as memory space, but not used.

The individual columns that make up each macro-block are then allocated in memory with the idea to pull out twice of them at a time. One solution, in this sense, could be to put thirty BRAM blocks in parallel, and associate to each row a pixel belonging to the same column of the visual matrix.

Consider the figure below:

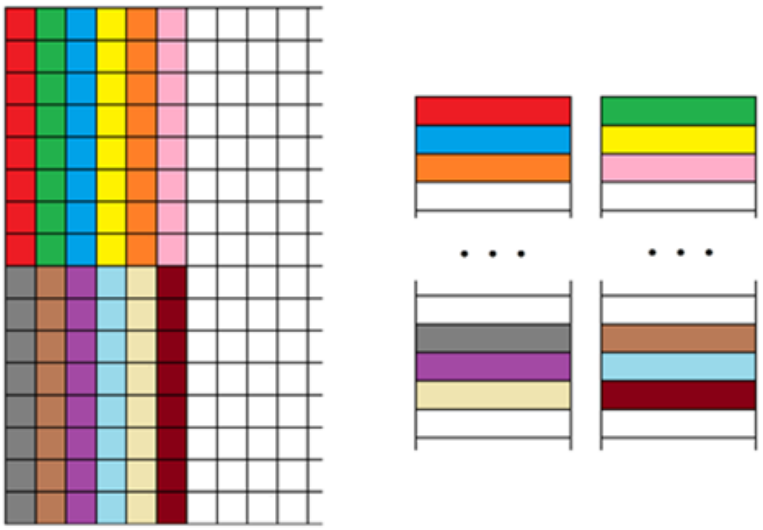


Figure 4.4. Memory allocations

Let's assume, for example, to divide the matrix into two macro-blocks (upper and lower), each of them composed by eight rows and Y columns. The sorting in memory is organized as follows:

- The first eight elements (red) of the first column are allocated in rows 0 (24 bit each) of the BRAM blocks from number 0 to 7.
- The first eight elements (green) of the second column are allocated in rows 0 (24 bit each) of the BRAM blocks from number 8 to 15.

- The first eight elements (blue) of the third column are allocated in rows 1 (24-bit each) of the BRAM blocks from number 0 to 7.
- The first eight elements (yellow) of the fourth column are allocated in rows 1 (24-bit each) of the BRAM blocks from number 8 to 15.

And so on, until the end of the first macro-block of the visual matrix. It will then occupy the first Y/2 lines of each BRAM, from 0 to (Y/2) - 1.

The second macro-block, in turn, will occupy the rows from Y/2 to Y-1. By looking at the following organization there are:

- The second eight elements (grey) of the first column, are allocated in the rows Y/2 (24 bit each) of the blocks BRAM from number 0 to 7.
- The second eight elements (brown) of the second column, are allocated in the rows Y/2 (24 bit each) of the blocks BRAM from number 8 to 15.
- The second eight elements (purple) of the third column are allocated in the rows (Y/2)+1 (24 bit each) of the BRAM blocks from number 0 to 7.
- The second eight elements (blue) of the fourth column are allocated in the rows (Y/2)+1 (24 bit each) of the BRAM blocks from number 8 to 15.

Again, until the end of the columns of the second macro-block.

This allocation scheme allows to optimize the extraction of two columns at the same time. It is clear, therefore, that the addressing mechanism presents two parts that make up the final address of the object:

$$address = n\_block * 120 + Y(7 downto 1) \tag{4.1}$$

With Y that is the coordinate of the incoming event, and n_block that allows to select which of the 21 macro-blocks should be taken into account.

As said, therefore, if the X coordinate is between 0 and 14, then the event belongs to macro-block 0, if it is between 15 and 29 to macro-block 1 and so on. Y, in turn, allows to select the row of the memory block where a certain column of the macro-block is allocated.

Shortly, n_block allows to select a certain group of rows of the visual matrix, and therefore a certain group of columns allocated in the BRAM. Y(7 downto 1) instead points to the desired column.

Working principle
The main hypothesis is to work in the worst case (i.e. a 15x15 window), and so to have 8 clock cycles between one event and the next to pull out 15 columns. As soon as a new input comes, thanks to the state machine that generates the addresses, two columns are extracted at a time, for each cycle. The n_block is kept fixed, what is changed is the line address:

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

Figure 4.5. Y generation for addresses

This means that, at the end of the eight cycles, 16 columns will be available, so one more than expected: either the first or the last one. It remains to be understood which is the one to be neglected.

As the memory structure has been organized, the address obtained by combining the block number and Y(7 downto 1), allows to point simultaneously to the locations of two adjacent columns. It is thanks to this mechanism that it is possible to pull out thirty elements at a time (in the worst case).

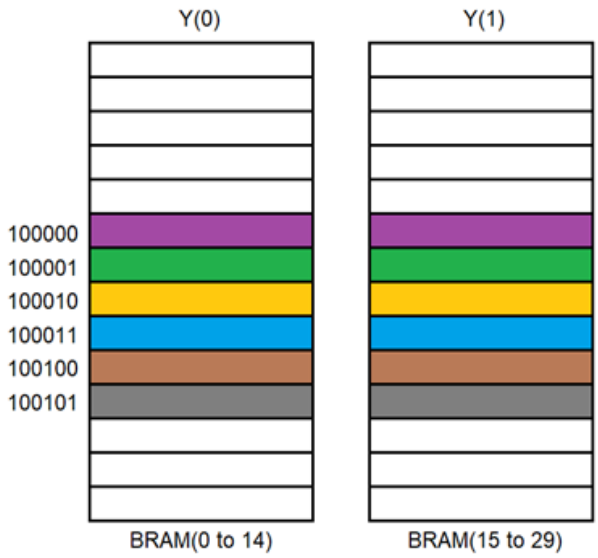


Figure 4.6. Memory separations

In the example above, in fact, let's suppose to extract a 5x5 matrix, and the central event is at the address 1000010 (yellow line). This opens up two possibilities:

- If the column in which the central event is contained is [Y(0) = 0], the events corresponding to the addresses 1000001, 1000010, 1000011 are extracted.

The result will be:

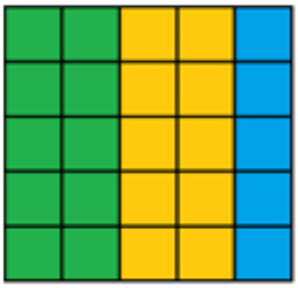


Figure 4.7. Matrix for Y(0) = 0

It is therefore clear that the last column [Y(0) = 1] at address 1000011 should be disregarded.

- If the column in which the central event is contained is [Y(0) = 1], the events corresponding to the addresses 1000001, 1000010, 1000011 are extracted.

The result in this case will be:

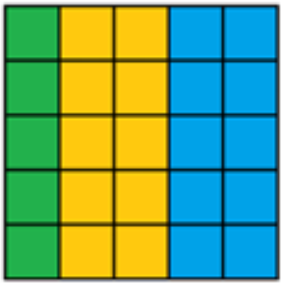


Figure 4.8. Matrix for Y(0) = 1

The first column [Y(0) = 0] at the address 1000001 will be now neglected.

The last case to be analysed is when two adjacent pixels belong to different blocks. Let's suppose to have the usual visual matrix, from which extract the event of interest:

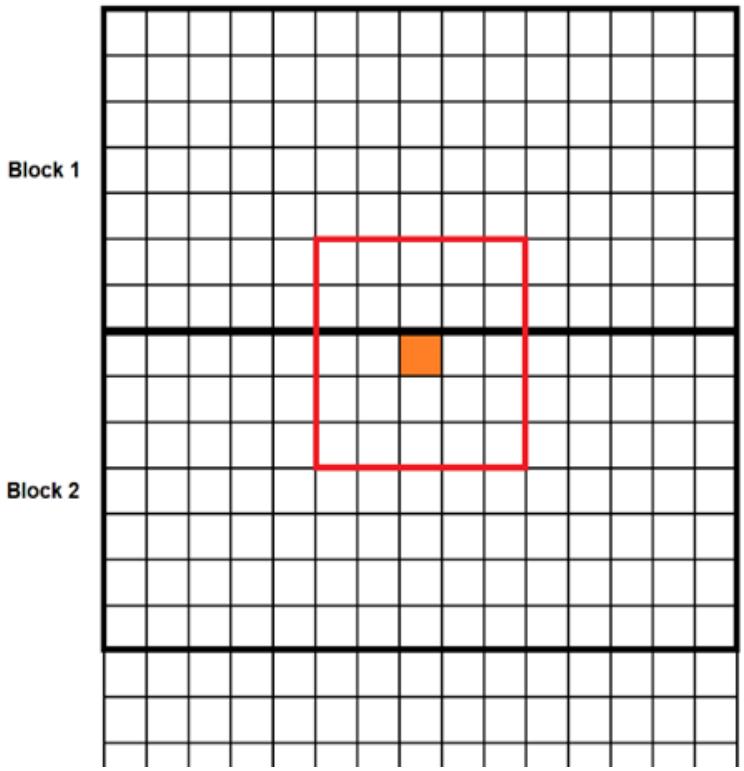


Figure 4.9. Window between two blocks

As highlighted, the event (in orange) belongs to block 2, however the surroundings in part fall into the adjacent block 1. In this case, the Y(7 downto 1) coordinates of the various columns to be extracted are the same, what changes is only n_block.

The address, in this case, for some BRAMs will refer to the same line number, but identified in the previous block. Assuming to work with an ATIS sensor (304x240), the corresponding position of the columns contained in block 1 will be 120 locations higher than the address calculated for block 2.

In summary:

- address - 120: if the matrix exceeds block borders above;
- address: if the matrix is perfectly inside the block;
- address + 120: if the matrix exceeds block borders below.

The output from the memory block will be, for each clock cycle, a 30 events array of 24 bits each (i.e. the two columns of pixels at a time), combined with a start signal for the

next block in cascade, and a Y(0) bit able to identify the column to be ignored.

As previously expressed in the disquisition on the software model, the extraction of a local matrix that goes beyond the dimensions of the visual one does not take place. The event is saved, but the start signal associated with the extraction of the columns is not asserted.

If (X,Y) are the coordinates of the new central incoming event, will be defined:

$$n_{row} = \text{floor}(R/2)$$

$$n_{col} = \text{floor}(C/2)$$

By considering:

$$X_{min} = X - n_{row}$$

$$X_{max} = X + n_{row}$$

$$Y_{min} = Y - n_{col}$$

$$Y_{max} = Y + n_{col}$$

If $X_{min} < 0 \,||\, X_{max} > Rf \,||\, Y_{min} < 0 \,||\, Y_{max} > Cf$ then the start signal is maintained equal to zero.

4.3 Events extinction

Another important element to take under control is the lifetime of each single event. The spatio-temporal window in exam, by considering the description in the paper exposed in 2.2, is dT ms wide: this means that the difference between the oldest and the newest incoming event should not be higher than dT. Also this is a parameter of the reconfigurable structure.

By looking at the Matlab description, the idea is to compare, every time an event is in input, its time stamp with the others stored inside the visual matrix. If their difference is higher than dT then the Ts is substituted with zero, else it's maintained.

In order to perform the same operation in hardware, the idea is to allocate a FIFO, that saves the incoming triplet (X,Y,Ts) in the time frame under observation. Internally, this memory has two counters: $read_{count} and write_{count}$. These increments every 8 cycles, allowing to keep track of the time evolution. The value at which they reset, if the dT = 1 ms, should be 12500. That's because if there is an increment of 1 every 80 ns, in order to

have 1 ms 12500 counts are needed.

So, if a new event comes, it is stored inside the FIFO, a validation bit is set to '1' and the write counter is incremented. A check signal in out indicates when a comparison has to be done. When it is asserted, the triplet coming out from the validation memory (Xv,Yv,Tsv) is compared with the other stored in the event-memory at the same coordinates. If they are equal, the event has to be eliminated.

In this way, the Ts is now set to zero. If, instead, the contents are different, the event stored is more recent. Also for this case, the coordinates (Xv, Yv) are calculated as before.

4.4 The overflow detection

As seen in the paragraph related to the Matlab model [3.3], overflow detection involves comparing the time-stamp related to the new incoming event with the previous one and, based on the result, setting a control bit.

The idea, very simple, is to insert an input registers where the reference MSB of the Ts of the newer event is taken and compared with that of the previous one. Normally, the possible combinations are:

$Ts_n[MSB]$	$Ts_{n-1}[MSB]$	Out
0	0	0
0	1	1
1	0	0
1	1	0

The function that describes this behaviour will return a null value at the output for all the combinations, except for the one where the MSB of the time-stamp at the current instant (0) is lower than that of the previous (1).

The combinatorial circuit thus elaborated will be the one in Figure 4.10.

The current value is saved in the upper register, the older one is passed instead in the lower register. As soon as there is an overflow, it is immediately indicated by the cascading AND.

It remains to be understood how to allocate and modify the MSBs in the memory from time to time. The overflow management mechanism is the same as previously illustrated

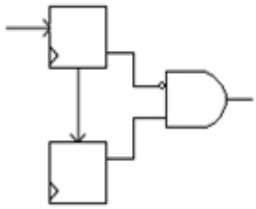


Figure 4.10. Circuit for overflow detection

in the fixed-point model: the idea is to associate a guard-bit to each incoming time-stamp. Every time an overflow occurs, the bits already saved are reset to 0, while the incoming bits are set to 1. In this way is replicated in hardware what has been seen previously in the scripts.

In this sense, an additional memory space has been allocated. Each control bit is saved at the related coordinates of the event. The addressing mode is the same viewed in paragraph 4.3. There is no need to replicate the address generation machine, it is possible to safely use the outputs intended for BRAM also for this additional memory space. Each location reserved for the Ts, hence, will have the same address as the one reserved for the related control bit.

In addition, the memory space for the guard bits can be reset. This allows to delete the content in case of overflow. There will be a finite state machine dedicated to administering the operations and the control process.

Chapter 5

Hardware description

This section describes in detail all the properties of the individual blocks used in the VHDL design phase. The digital project has been developed, as already expressed in the previous chapters, considering the worst operating conditions, in order to stress, as possible, the characteristics of the device. The tool adopted is *Vivado Design Suite 2017.1*. The structure has been totally parametrized. In this way is allowed to modify the working algorithm at synthesis time. The variables adopted are the follows:

Parameter	**Use**
R	Number of rows for the spatio-temporal window (3-5-7-9-11-13-15)
C	Number of columns for the spatio-temporal window (3-5-7-9-11-13-15)
dT	Life-time of an event
thr	Threshold set to eliminate the outliers
min_events	Minimum number of events to process the window
n_iter	Number of iterations of the algorithm (0-1-2)
n_shift	Number of shift to control the overflow problem (0 to 8)
type_sens	Allows to choose between DVS (0) and ATIS (1) architecture

Table 5.1. Parameters used for the hardware design

5.1 Memory block and validation FIFO

The first block in input is the memory management. It is made-up of four main basic units:

- MEM_READ
- BRAM
- VLD_MEM

- mux_out

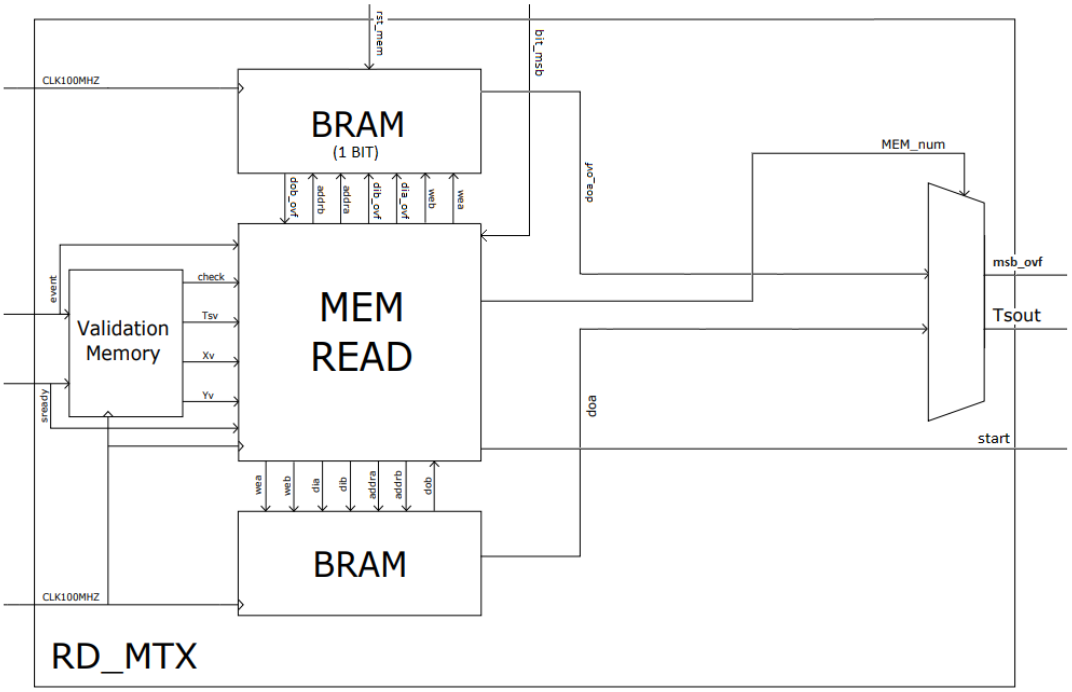


Figure 5.1. Memory block scheme

MEM_READ
This part is composed of a finite state machine that at each clock cycle, as described in paragraph 4.2, provides the memory address from which to extract two columns at a time of the local space-time matrix.

The inputs of this block are:

- Xi: on 9 bits, is the vertical coordinate of the event placed in the visual matrix;
- Yi: on 8 bits, is the horizontal coordinate of the event placed in the visual matrix;
- Tsi: on 24 bits, is the time stamp associated with the event that occurred at the coordinates (Xi, Yi);
- Xv: on 9 bits, is the vertical coordinate of the event to be evaluated for validity in temporal terms;
- Yv: on 8 bits, is the horizontal coordinate of the event to be evaluated for validity in temporal terms;

- Tsv: on 24 bits, is the time stamp associated with the event to be evaluated for validity in temporal terms;
- check: is an active high signal, and indicates when an event is expired;
- dob: it is an input matrix of 30 elements, 24 bit each. It corresponds to the output of the 30 BRAM blocks, from which it's possible to extract two columns per cycle;
- bit_msb: it corresponds to the bit set by the overflow manager (FSM) in order to control the wraparound problem.

The outputs are instead:

- addra, addrb: these are the addresses related to the BRAM ports A and B. The first is used to write the time stamp related to the new event, and read the output columns, the second is used for the validation part;
- wea, web: these are active high signals, for both BRAM ports, which are useful to signal to the memory that a data is ready to be written;
- dina, dinb: these are the 24 bit inputs, reserved for time stamps, of the BRAM ports A and B;
- dia_ovf, dib_ovf: these are the 24 bit inputs, reserved for the overflow control bits of the BRAM ports A and B;
- MEM_num: it allows to multiplex the outputs of the memories according to how the data are organized internally, in order to position them correctly.

BRAM
The input/output signals have been described previously. It is important, in this case, to use two different memory spaces: one reserved to the time stamps, the other instead to the control bit set in order to prevent the wraparound.

VLD_MEM
This block is represented by a FIFO, whose task is to provide a useful tool that controls the state of an event. A generic input e_t arrives at a certain instant t, crosses the memory with a frequency of 12.5 MHz, and after a certain interval of time dT (externally set) it is put on the output. When this condition is verified the event is considered extinct.

The inputs are:

- sready: is an active high signal, which allows the FIFO to store the incoming event;
- evt: is the input event represented on 41 bits, consisting of the triplet (X,Y,Ts).

The outputs are expressed inside the triplet (Xv, Yv, Tsv) accompanied by the check signal, which signals the just expired event.

mux_out
It represents a register that acts as a multiplexer. It allows to position the output events coming from the BRAMs correctly. The first column in positional order will occupy the first R positions (from 0 to R-1), the second column will occupy the following R positions (from 15 to 15+(R - 1)).

The inputs are:

- clock;
- sready: is the active high signal that indicates the validity of a data from the memory management;
- doa: is the output matrix from the BRAM blocks, composed of 30 time stamps (in the worst case) of 24 bits each;
- doa_ovf_t: is an array of 30 control bits related to the output time-stamps on doa;
- MEMnum: multiplexer selection signal. It allows to vary the position of the elements (both Ts and guard-bit) in order to organize them according to the desired matrix structure.

The outputs are:

- srdy_out: is the active high signal associated with the input sready, delayed with its relative time-stamp by the register;
- Tsout: is the matrix associated with doa, after being properly multiplexed and delayed by the register. Its dimensions are always 30 elements of 24 bits;
- doa_ovf: is the array related to doa_ovf_t, also multiplexed and delayed by the register.

5.2 Overflow detector FSM

This unit has the task of detecting the presence of an overflow and set, consequently, a control bit that allows the management. The detection task is entrusted to a purely combinatorial block described in section 4.4, while the guard bit selection is linked to a finite state machine (ovf_detector).

Its schematic is in Figure 5.2.

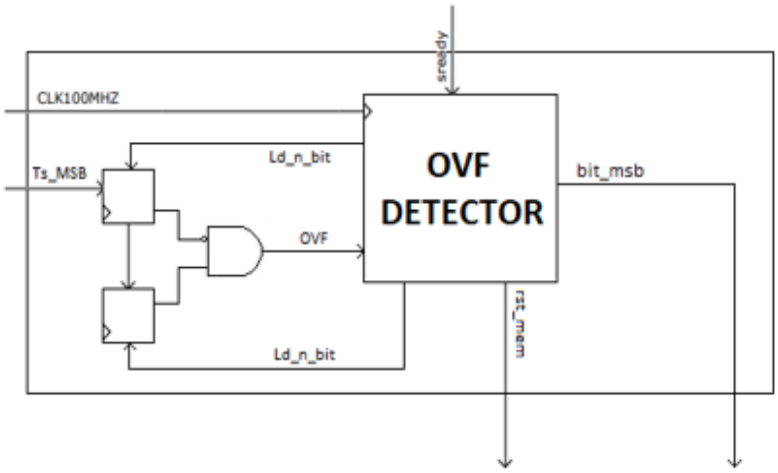


Figure 5.2. Overflow control

The inputs are:

- clock;
- sready: active high signal, when a new event is available;
- rst_in: active high signal, when there is a reset from outside;
- ovf: is the output bit of the combinatorial circuit that signals the presence of an overflow;

The outputs are:

- rst_mem: control signal whose task is to start a reset of the memory space where the guard bits are allocated. Every time an overflow occurs, in fact, the control bits stored in the BRAM are reset;
- ld_n_bit: control signal that, in correspondence of a new input event, activates the register where the MSBs have to be allocated. In addition, it also activates the loading of the register where the MSB of the previous event is saved;
- bit_msb: corresponds to the control-bit of the new event to be stored in memory.

The flow chart of the FSM is illustrated as follows:

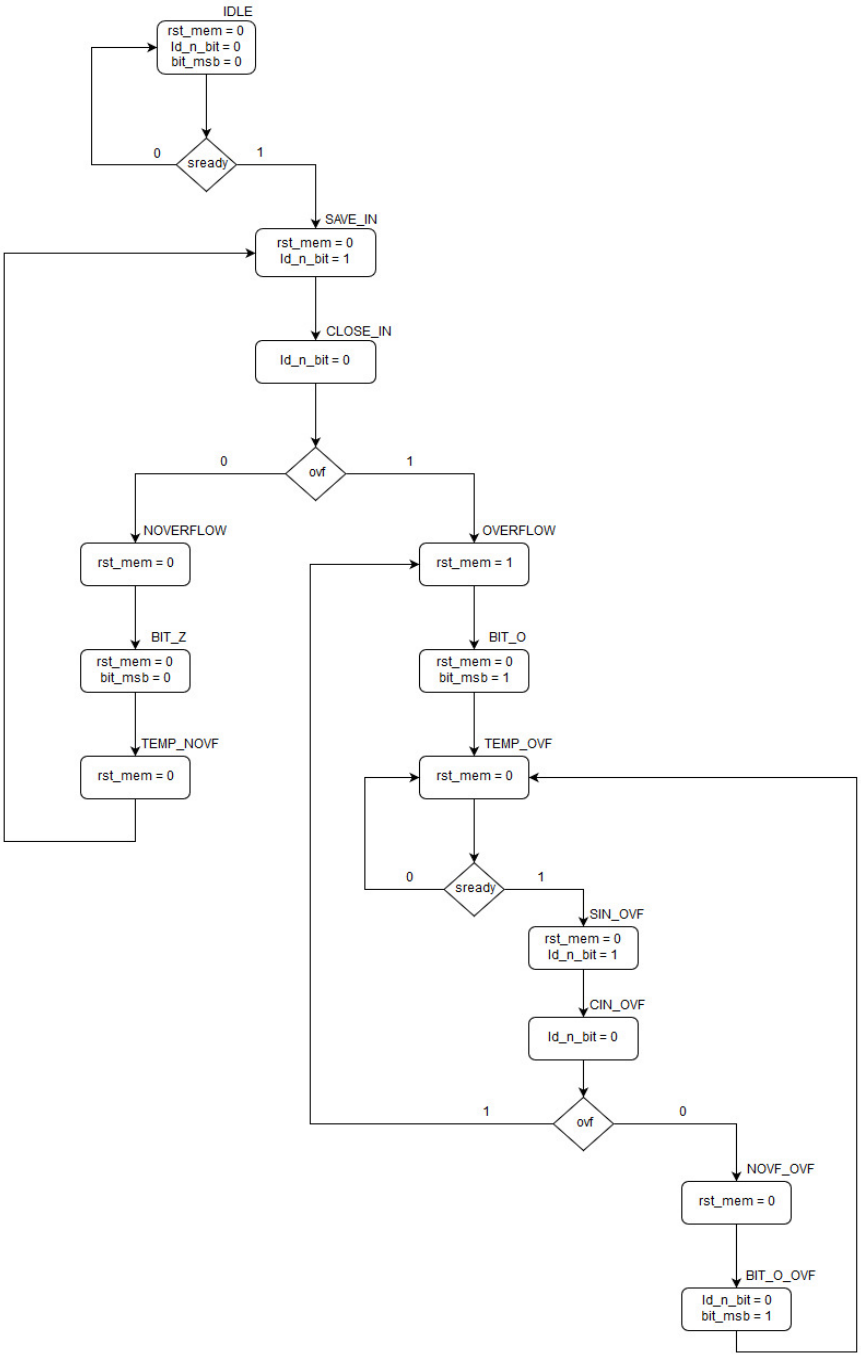


Figure 5.3. Control chart of the state machine

5.3 Shifter and valid generator

In this part of the design the developed elements have the main scope to take the matrix coming out from the memory manager, mix it with the guard-bit, shift it to the right of a certain quantity, and generate the valid vector. The outputs are, at the end, destined to the computation part of the pipe chain.

This block is made up of three important elements:

- Input register
- gen_valid
- pipe_mem_mult

Its global scheme is represented as follows:

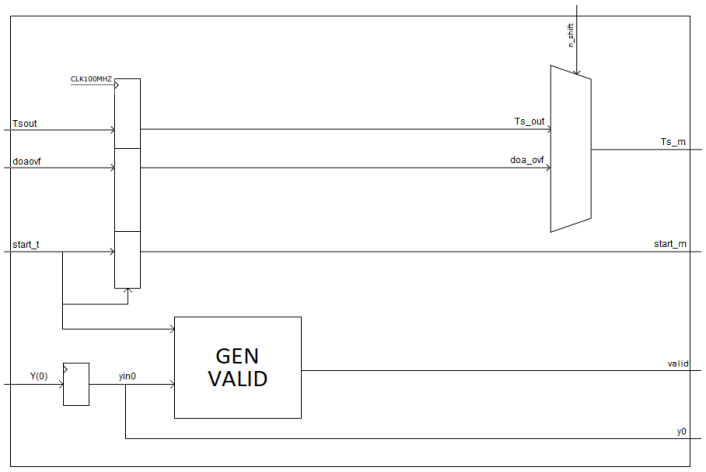


Figure 5.4. Valid generator and shifter blocks

Input register
This is an input pipe register that saves the values of the two Ts columns per cycle taken out from the memory. It also stores their related guard-bits, used to manage the overflow problem, and the start signal, delayed to maintain coherence with the time-stamps. This last is used also as load signal for the register itself, in order to avoid the storage of un-useful elements.

It acts also as a multiplexer that shifts the two input columns to have a position as follows in Figure 5.5:

Figure 5.5. Shifter behaviour

In this way, let's suppose that R is the number of the selected rows (with R < 15) for a single column. In the original implementation, if Tsout is made up of 30 rows, the elements of the first column are allocated from the line 0 to R-1, and the others of the second one are allocated from 15 to 15+(R-1).
With this multiplexer, the two columns are placed adjacent to each other, by occupying the first 2*R lines (from 0 to 2*R-1). This is done for Tsout and msb_ovf, that is the control bit.

gen_valid
It is a block whose purpose is to generate a different valid vector associated to the two columns that, each cycle from the time that a new event is generated, come out from the memory.
It is a 30 rows array of one bit each. In this way, its value can be 0 in correspondence of an un-valid event (like for Ts = 0), or 1 for the others.
The inputs of this block are:

- yin0: it represents the LSB of the Y coordinate, useful to understand if the first columun at the beginning cycle, or the last at the last one, has to be ignored;
- start_t: it is the active high signal that starts the generation of the valid vector.

The output is:

- valid: it is the valid vector generated as described before.

pipe_mem_mult
It represents a pipe stage whose main aim is to combine the time stamps with their relative control-bit and then right-shift them of a n_shift quantity.

The inputs of this block are:

- Ts_out: it is the matrix (30 x 24 bits) representing the two extracted columns;
- doa_ovf: it is the input vector (30 x 1 bit) of control-bit associated to the two columns of Ts;
- n_shift: it is an input parameter, at synthesis level, which allows to choose the number of bits to shift;

The outputs of this block are:

- Ts_m: it is the output matrix (30 x 24 bits) representing the time stamps with the managed control bit associated to them, and shifted to the right of n_shift.

5.4 Mult block

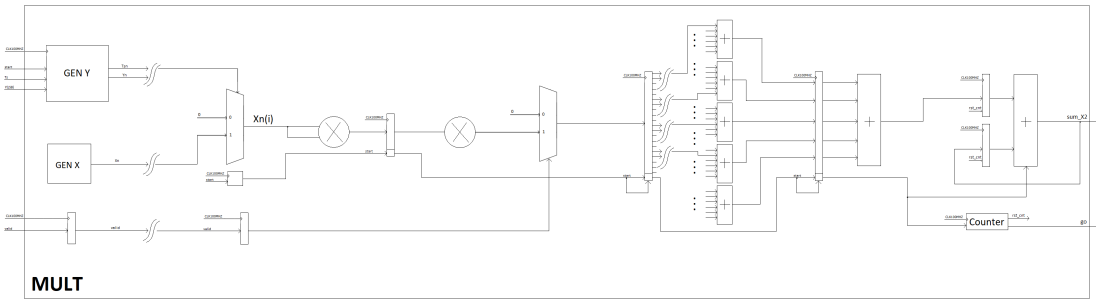


Figure 5.6. Mult block scheme

This block, as described in the section 3.2 (fixed-point model), is designed to calculate the matrix product $A^T A$. To do this, the idea is to take each column of time-stamps that compose the neighbourhood of the new incoming event, and associate to it its spatial coordinates (X_0, Y_0).

Starting from the new event, a couple of adjacent columns is extracted from the memory and processed. Thereby, two blocks (GENY and GENX) have been designed to generate the coordinates for any element of the local window.

As input values, there are:

- X, Y: matrix composed by 2*R rows,of 4 bits each;
- T: matrix composed by 2*R rows of 24 bits each;

The first step is to eliminate the coordinates (by substituting zeros instead of their real values) where the time stamps are null. That is done using a multiplexer.

The results that have to be generated, later, are X, Y, T, X2, Y2, XY, XT, YT. In order to obtain them, for each row a multiplier has to be allocated.

- X, Y, T: no need for a multiplier;
- X2, Y2, XY: 2*R multipliers to allocate, 8 bits each;
- XT, YT: 2*R multipliers to allocate, 28 bits each.

In the figure above, only a little fraction of this block is shown. In particular, the part under analysis shows the calculation of only one of the rows of X2. So, it's easy to understand that the structure has to be replicated for other 29 times, to have the complete X2 vector, and for all the other needed vectors Y2, XY and so on.

The decision to allocate a pipelined architecture for so huge multipliers has been taken to reduce their critical paths. In this way, no risk for timing problems is encountered.

The next part of the pipe queue is made up of adders that have the task of adding-up the results just obtained: all the 2*R raws of X, Y, T, X2, Y2, XY, XT, YT are added, resulting in their corresponding single-row values SX, SY, ST, SX2, SY2, SXY, SXT, SYT. It is important to remember that this partials are related to only two columns extracted per cycle.

At the end, in order to have the total result of the whole columns that compose the neighbourhood, an accumulator is put in cascade.

The inputs of this block are:

- start: it is the active high signal that starts the computation;
- y0: it is the LSB bit of Y that allows to ignore the first or the last column;
- Tsin: it is the input matrix that contains the two columns of events;
- valid: it is the valid vector that allows to validate or not its related time stamp;

The outputs of this block are:

- SX, SY, ST, SX2, SY2, SXY, SXT, SYT, SN: they are the output sums (2*R rows of a certain number of bits each) of the block, that correspond to the components of the product matrix $A^T A$.
- sur15, sur15_2: they are two storage elements whose scope is to save the input columns for the next iterations. Two of these blocks are necessary because, if a new event comes after exactly 8 cycles the previous one, its corresponding input columns cannot be stored (i.e. in sur15). That's because sur15 is already occupied by the previous event neighbourhood. For that reason is necessary to allocate another set of registers ($sur15\_2$).
- go: it is the active high signal which indicates to the next block that a new computation can start.

5.5 Adjoint block

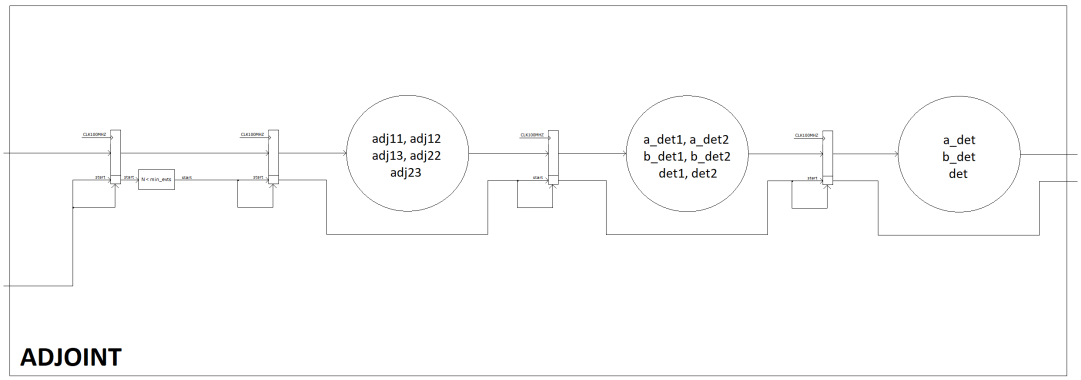


Figure 5.7. Adjoint block scheme

This block performs the matrix product $(A^T A)^{-1}(A^T Y)$: it allows to calculate the partial plane coefficients a_{det},b_{det} and det.
It is composed by three pipe stages, which, first, compute the adjoint matrix components, then, the temporary results for a_{det},b_{det} and det, and at the end their final sum.
The initial part allows to understand if the number of valid events is lower with respect to the set (at synthesis level) min_events. If this condition is not respected, the start signal is not propagated inside the structure, and no computation occurs.

The inputs of this block are:

- start: it is the active high signal that corresponds to the "go" coming out from the MULT block;

- SX, SY, ST, SX2, SY2, SXY, SXT, SYT, SN: they are the components of the product matrix $A^T A$.

The outputs of this block are:

- a_detin, b_detin and detin: they are the partial plane coefficients expressed on 62 bits (the two first) and 42 bits (the last one);

- go_iter: it is the signal which allows the next block to start its process.

5.6 Iteration block

This is the block that eliminates the outliers by verifying the condition:

$$a_{det}(x - x_c) + b_{det}(y - y_c) + (t–t_c) < thr$$

Where (x_c,y_c,t_c) is the triplet of the central event, and a_{det},b_{det} are the coefficients of the local window around it.

As defined in 5.4, the matrices from which remove the events that do not respect the inequality above can be sur15 or sur15_2. Thanks to a switching mechanism, it is possible to process the correct neighbourhood related to the a_{det} and b_{det} plane values calculated in the previous stages.

In the following figure, the structure of the device is shown:

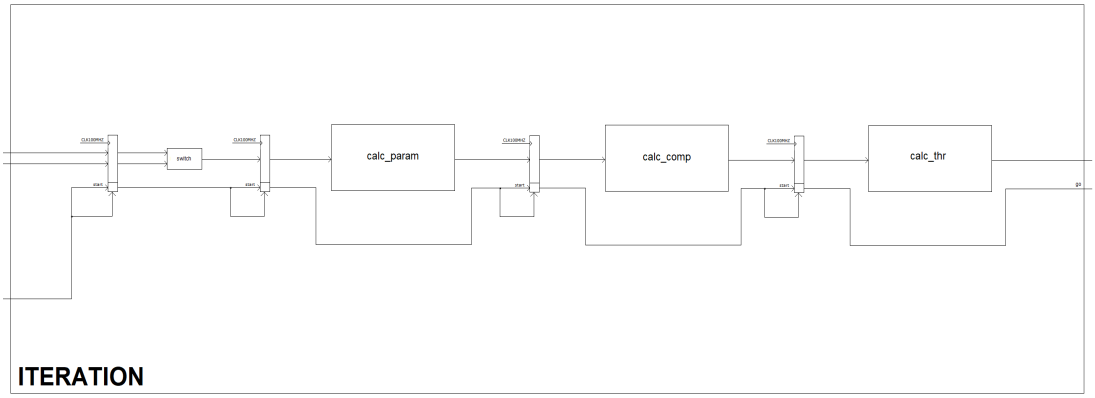


Figure 5.8. Iteration block scheme

For each cycle, two columns are processed. There are four stages that compose the block:

- switch: as described before, it is a little process that, every time a new couple of plane coefficients are available in input, switches between sur15 and sur15_2. This allows to process the correct neighbourhood;

- calc_param: it is the first part of the process that calculates in parallel – for each pixel belonging to the two extracted columns per cycle – their related values $adet(x - x_c)$, $bdet(y - y_c)$ and $(t–t_c)$;

- calc_comp: this stage of the pipe chain adds the results of calc_param, for each element, in order to obtain the sum $adet(x - x_c) + bdet(y - y_c) + (t–t_c)$;

- calc_thr: at the end, the sums of the last part are compared to *thr* in order to remove (by substituting zeros) the outliers.

The choose of process two columns per cycle was made in order to have a correct management of a Mult-block in cascade. That's because Mult itself works with two columns per cycle.

The inputs of this block are:

- sur15_1, sur15_2: they are the two storage elements coming from the Mult-block;

- git: it is active high only at the first cycle, when the computation starts. In this way, when 8 cycles (in worst case) are elapsed, it changes the "sur" input basing on switch;

- git_evt: if git is low, it means that any new computation can start. Though there's the possibility that externally the parameters of the plane and their original neighbourhood are available, but cannot be processed because SN is lower than n_events. In this case is necessary to switch anyway internally the input "sur", in order to maintain the correct behaviour of the device. This signal is useful to understand this eventuality;

- a_detin, b_detin, detin: they are the plane-coefficients coming from Adjoint-block.

The outputs of this block are:

- sub15o: it is the output matrix (2*R rows of 24 bits each) without outliers;

- valid: it is the valid matrix (2*R rows of 1 bit each) related to the new subsurface sub15o;

- go: it is the active high signals to start the next operation.

5.7 Final computation

This is the last stage that performs the final computation of the algorithm. Taking the results a_{det}, b_{det} and det coming from the matrix operations, and the iterations done to eliminate the outliers, the next step is: know the value of the plane-coefficients and elaborate them in order to obtain the inverse components of the velocity vector.

In order to do this, the flow of the operations is shown as follows:

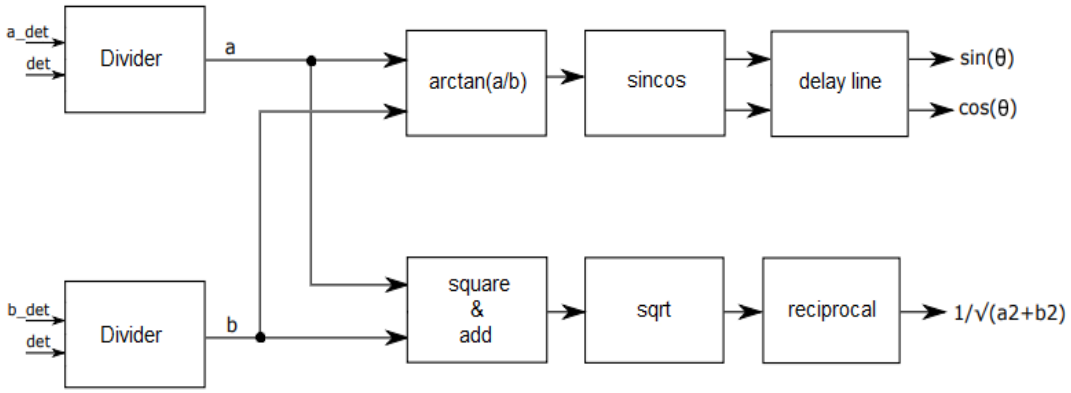


Figure 5.9. Final stage block scheme

Each element of this structure has been included in the project by using the ”IP Core Generator” tool of *Vivado Design Suite 2017.1*. In this way, by modelling the core with a simple wizard of configuration, it is possible to choose the best available solution for this purpose.

Divider
The first core-block is composed by two input dividers that perform in parallel the operation:

$$a = \frac{a_{det}}{det}$$

$$b = \frac{b_{det}}{det}$$

The inputs of this block are:

- s_axis_dividend_tdata: it is the dividend of the division, and it is represented on 64 bits;

- s_axis_dividend_tvalid: it is an active high signal, ready when the dividend is available;
- s_axis_divisor_tdata: it is the divisor of the division, and it is represented on 48 bits;
- s_axis_divisor_tvalid: it is an active high signal, ready when the divisor is available;

The outputs of this block are:

- m_axis_dout_tdata: it is the result of the division, composed by an integer part and a fractional one. It is represented on 72 bits;
- m_axis_dout_tvalid: it is an active high signal, and it is ready when the output is available;

As seen in section 3.2, in fact, the operand widths are not the ones set during the design phase. To remember:

- Dividend: integer on 62 bits;
- Divisor: integer on 42 bits;
- Quotient: fractional number on 66 bits, composed by 62 bits of integer and 4 bits of fractional part.

The presence of some extra-bits is due to the data packing used in the protocol of the device, as follows:

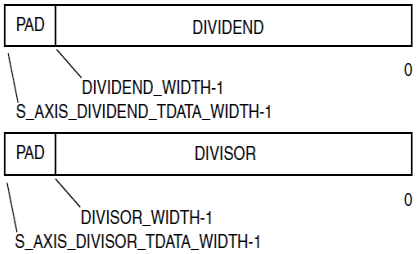


Figure 5.10. Data packing for division inputs

It is necessary, so:

- To allocate the divider in the first 62 bits, leaving zeros in the 2 leading bits;
- To allocate the divisor in the first 42 bits, leaving zeros in the other leading bits.

The output is expressed as follows:

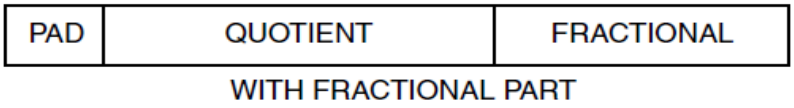


Figure 5.11. Data packing for division outputs

It is important to precise that only the first 25 LSBs are taken out from it: the ones belonging to the range 24-4 represent the integer part, the last one in the range 3-0 are the fractional component.

By looking at the communication protocol, there are two ways of working:

- Non blocking mode: it means that the division is performed only when all the input channels receive a tvalid high. In other case any operation is done.

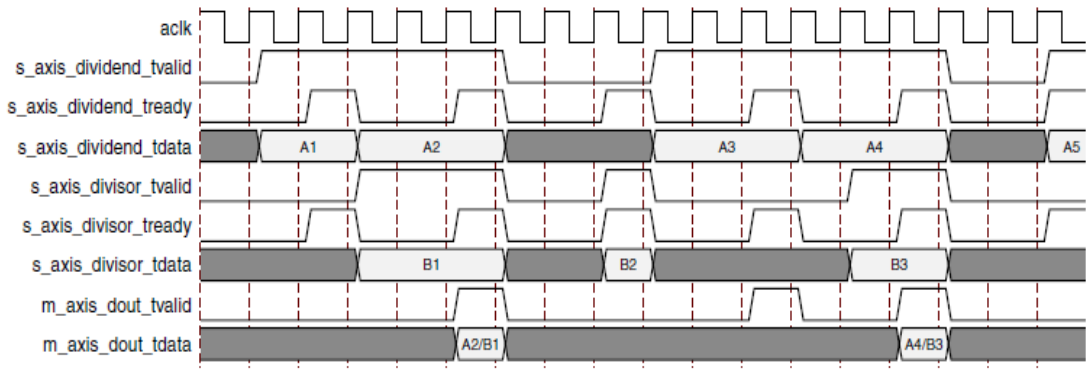


Figure 5.12. Non-blocking mode

- Blocking mode: it means that, even if only one of the channels receives a tvalid, its related input is buffered. This will be available to perform the operation when the other channel will receive its own tvalid (Figure 5.13).

The first way has been chosen during the design phase of the core.

The latency of this block is 70 clock cycle to complete: that's due to the radix-2 architecture adopted in this work. There was another, faster, possibility that is the high-radix solution. In this case, in fact, the latency will be reduced to 40 clock cycles but with the counterpart of the increased throughput. The radix-2 divider is a pipelined structure,

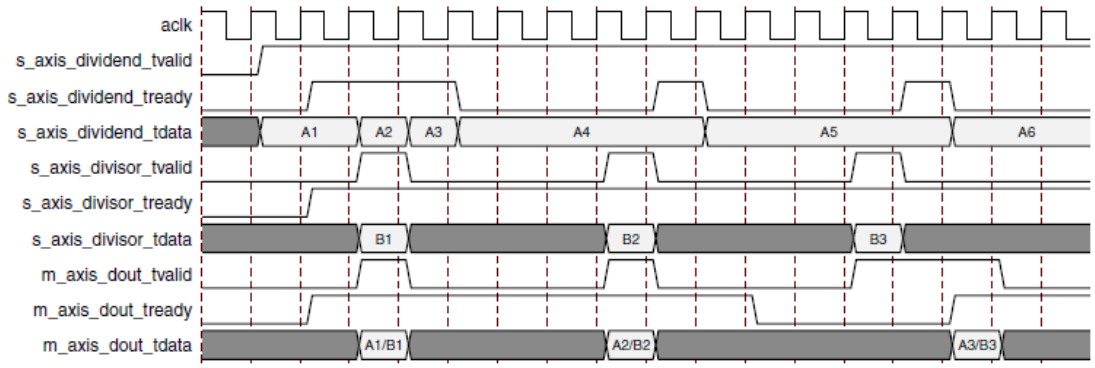


Figure 5.13. Blocking mode

which the highest achievable throughput: it is possible to start a new computation every clock cycle. The high-radix, instead, presents an internal loop which severely limits the performances. Any new operation can start without completing the last. Furthermore, latency is not a critical parameter in this kind of application and the timing, related to the entire structure, is not heavily affected by the presence of 70 clock cycles. Throughput, vice-versa, is a fundamental specific, even more considering the strong flow of the pipe framework.

Arctan

The scope of this core is to obtain:

$$\theta = arctan(\frac{a}{b})$$

The inputs of this block are:

- s_axis_cartesian_tdata: it is the input composed by a and b values in a single line vector. It is represented on 64 bits;
- s_axis_cartesian_tvalid: it is the active high signal which indicates if a data is available in input.

The outputs of this block are:

- m_axis_dout_tdata: it is the output result represented on 16 bits, divided into 3 integer and 13 fractional bits;
- m_axis_dout_tvalid: it is the active high strobe that signals the end of the computation;

The input/output formats are expressed as follows:

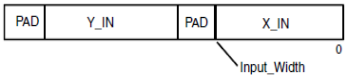


Figure 5.14. Data packing for arctangent inputs

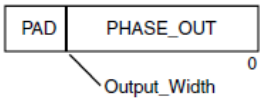


Figure 5.15. Data packing for arctangent outputs

In the Y_in field is put the “a” value, in the X_in one instead the ”b” result. Thereby, from the bit 0 to 24 ”b” is allocated, instead, from the bit 49 to 25.
The latency of this block is 12 clock cycles.

Sincos
The scope of this core is to obtain:

$$sin(\theta)\ and\ cos(\theta)$$

The inputs of this block are:

- s_axis_phase_tdata: it is the angle vector represented on 16 bits, 3 integer and 16 fractional;
- s_axis_phase_tvalid: it is the active high signal which indicates that the angle is available in input.

The outputs of this block are:

- m_axis_dout_tdata: it is the output result represented on 32 bits, indicating the sine and the cosine of the input angle;
- m_axis_dout_tvalid: it is the active high strobe that indicates the availability of the results;

The input/output formats are expressed in Fig. 5.16 and 5.17.

The Y_out field is the sine value, the X_out one represents instead the cosine. The latency of this block is 20 cycles of clock.

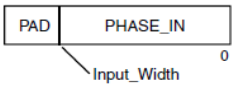


Figure 5.16. Data packing for sine and cosine inputs

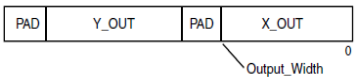


Figure 5.17. Data packing for sine and cosine outputs

Square and add
This part is a pipelined structure which is made up of a multiplier followed by an adder, which performs:

$$c = a^2 + b^2$$

The results ”a” and ”b” are represented both, as described previously, on 25 bits (21 integer and 4 fractional). Two multipliers act in parallel to calculate their relate squares. In this way, their outputs are on 50 bits (42 integers and 8 fractional). After, a pipe stage, separates them from the adder that, in the next cycle, calculates their sum. The two inputs, as known, are on 50 bits, the result in order to avoid overflow, is on 51 bits (43 integers and 8 fractional).

Its latency is two clock cycles.

Square root
The scope of this core is to obtain:

$$\sqrt{c}$$

The inputs of this block are:

- s_axis_cartesian_tdata: it is the “c” vector given by the sum a2 + b2, and it is represented on 48 bits;
- s_axis_cartesian_tvalid: it is the active high signal which indicates if a data is available in input.

The outputs of this block are:

- m_axis_dout_tdata: it is the result of the square root represented on 24 bits;
- m_axis_dout_tvalid: it is the active high signal that indicates the end of the square root computation;

The input/output formats are expressed as follows:

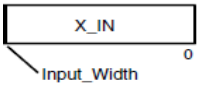


Figure 5.18. Data packing for square root inputs

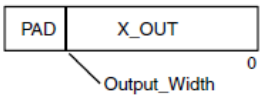


Figure 5.19. Data packing for square root outputs

If the PAD bit is excluded, the output result is represented on 23 bits (from the 0 to 22). The latency of this block is 23 clock cycles.

Reciprocal
This block is a divider that performs the reciprocal operation of an input:

$$\frac{1}{\sqrt{c}}$$

The inputs of this block are:

- s_axis_dividend_tdata: it is the dividend of the division, in this case its value is 1, and it is represented on 24 bits;
- s_axis_dividend_tvalid: it is an active high signal, ready when the dividend is available;
- s_axis_divisor_tdata: it is the divisor of the division, and it is represented on 24 bits;
- s_axis_divisor_tvalid: it is an active high signal, ready when the divisor is available;

The outputs of this block are

- m_axis_dout_tdata: it is the result of the division, represented on 25 bits (5 integer and 20 fractional);
- m_axis_dout_tvalid: it is an active high signal, and it is ready when the output is available;

The latency of the reciprocal block is 46 clock cycles.

Total computation
The difference between the times needed to complete the two branch operations is given by calculating the latency of each of them:

$$\text{First brench: } t_1 = t_{atan} + t_{sincos} = 12 + 20 = 32 \text{ clock cycles}$$

$$\text{Second brench: } t_2 = t_{pipe} + t_{sqrt} + t_{recip} = 2 + 23 + 46 = 71 \text{ clock cycles}$$

The difference in time is of 39, so it means that a 21 clock cycles delay is required for the first branch in order to have both results at the same time. Only at this point is possible to calculate the products:

$$\frac{1}{\sqrt{a^2 + b^2}} sin(\theta) \quad , \quad \frac{1}{\sqrt{a^2 + b^2}} cos(\theta)$$

Chapter 6

Simulation, synthesis and results

In this chapter all the simulation and synthesis results are shown. The analysis is conducted by using three different couples of data: one coming from the same dataset used for the Matlab simulation in chapter 3, the second that starting from the previous one includes also an additional part where an incoming wraparound is artificially inserted, and the other properly prepared in order to highlight some particularities of the hardware algorithm. Part of the simulation has been made on each single block of the entire architecture, the other one, instead, is focused on the system level. All the waveforms have been obtained with the simulation environment *ModelSim - Intel FPGA Starter Edition 10.5c*

The first two sets of events are applied to illustrate the working principle of the memory interface (both in writing and reading phases), how the device behaves in presence of an overflow, and all the computational steps to be performed. The parameters set in the three cases are:

Parameters	**Dataset**	**Modified Dataset**	**Casual events**
R	15	15	3
C	15	15	3
dT	205	205	9
min_evts	10	10	3
n_iter	2	2	2
n_shift	0	1	0

Table 6.1. Parameters used for hardware simulation

The tables 6.2 and 6.3 show the temporal evolution of the received events.

t	X	Y	Dataset	Modified Dataset
1	65	68	9586520	9586520
2	68	58	9593330	9593330
3	68	56	9593705	9593705
4	64	62	9593751	9593751
5	67	57	9593820	9593820
6	74	57	9593869	9593869
7	77	67	9594050	9594050
8	78	67	9596366	9596366
9	69	64	9596483	9596483
10	70	64	9596543	9596543
11	71	63	9596578	9596578
12	70	63		8388606
13	71	64		9586520

Table 6.2. Dataset and modified version

t	X	Y	Time Stamps
1	1	3	2
2	1	2	33
3	0	4	100
4	4	1	256
5	2	3	320
6	4	3	550
7	0	3	675
8	3	1	1024
9	3	0	3240
10	0	2	6449
11	0	0	7024
12	2	4	7470
13	2	2	7590
14	1	1	7600

Table 6.3. Casual events

6.1 Single-block testbenches

Memory addressing

The flow of incoming events used in this phase is a small extract of the dataset given by the Istituto Italiano di Tecnologia (iit). It's interesting to notice how the time stamps are

saved, inside the different allocated block RAMs, by evaluating the generated addresses for the memory locations.

In figure 6.1 a general view of the memory management, basing on the Table 6.2, is proposed.

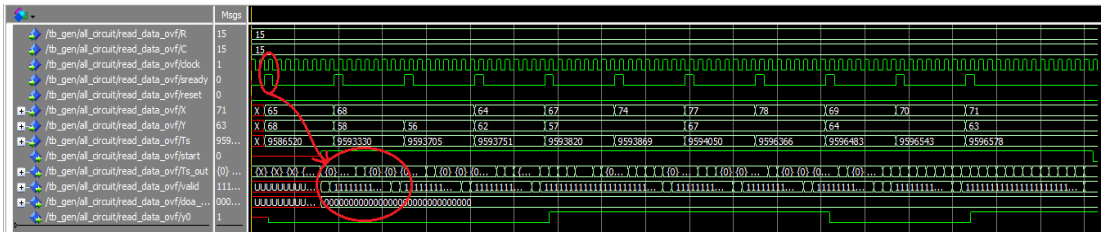


Figure 6.1. Working principle of the memory block

As can be noticed, when a new event is ready, the sready signal is pulled-up. In correspondence of the rising edge of the clock, this signal is acquired by the control machine of the memory interface and contemporary starts the reading phase. After six clock cycles two columns per cycle are extracted from the memory and, at the same time, the new time stamp is allocated at its related coordinates inside the BRAM.

Let's now suppose to extract the 15x15 matrix around the event number 11 in table 6.2. The expected neighbourhood should be:

		Y														
		56	57	58	59	60	61	62	63	64	65	66	67	68	69	70
	64	0	0	0	0	0	0	9593751	0	0	0	0	0	0	0	0
	65	0	0	0	0	0	0	0	0	0	0	0	0	9586520	0	0
	66	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	67	0	9593820	0	0	0	0	0	0	0	0	0	0	0	0	0
	68	9593705	0	9593330	0	0	0	0	0	0	0	0	0	0	0	0
	69	0	0	0	0	0	0	0	0	9596483	0	0	0	0	0	0
	70	0	0	0	0	0	0	0	0	9596543	0	0	0	0	0	0
X	71	0	0	0	0	0	0	0	9596578	0	0	0	0	0	0	0
	72	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	73	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	74	0	9593869	0	0	0	0	0	0	0	0	0	0	0	0	0
	75	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	76	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	77	0	0	0	0	0	0	0	0	0	0	0	9594050	0	0	0
	78	0	0	0	0	0	0	0	0	0	0	0	9596366	0	0	0

Figure 6.2. 15x15 matrix

The address generation block, by remembering the equation 4.1, will return:

- n_block: 4 for the $60 < X \leq 75$, 5 for the $75 < X \leq 90$. The central event is in block 4, thereby will be necessary to add 120 in order to select the other;

- Y(7 downto 1): its values will be 28, 29, 30, 31, 32, 33, 34, 35.

By observing the waveform, the results are the same as expected:

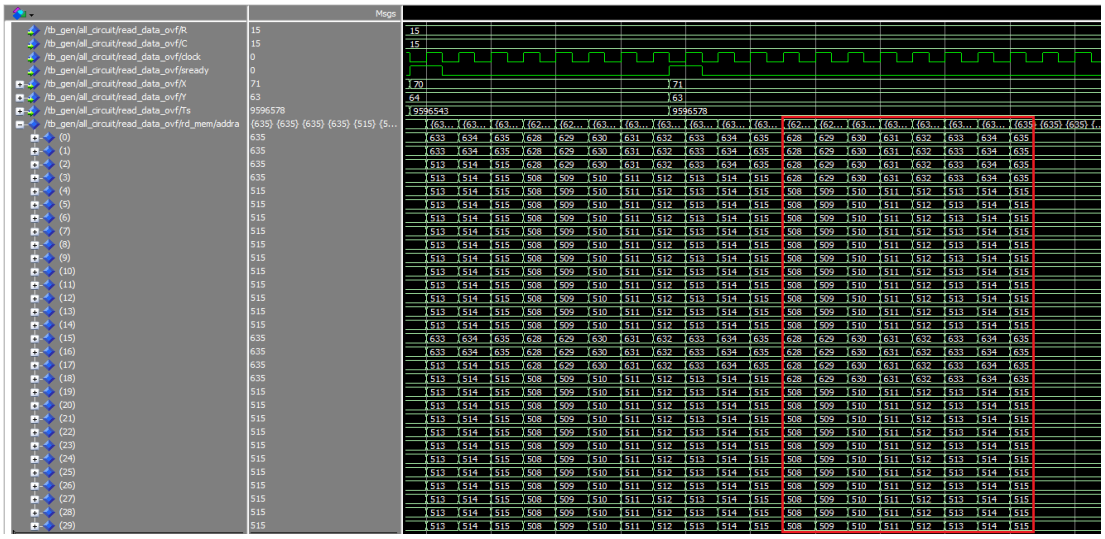


Figure 6.3. Generation of the addresses

The output matrix extraction is directly related to this step: in this way it is possible to have the corresponding 15 columns in 8 clock cycles. Thanks to the Y(0) bit, the column in excess is deleted by associating it to a valid vector with zeros inside. The description step-by-step of this process is shown in the following simulations:

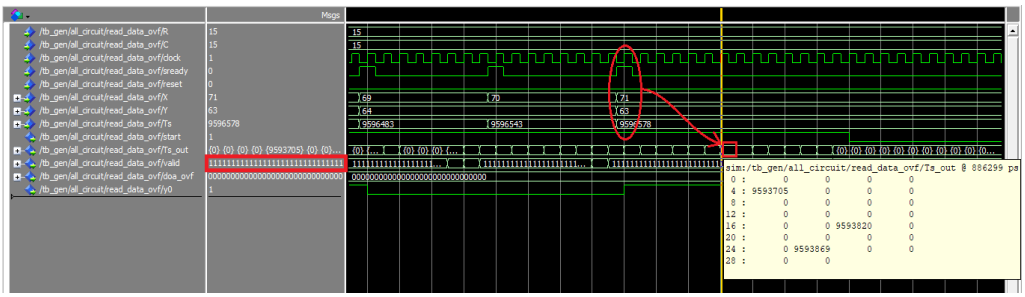


Figure 6.4. First cycle

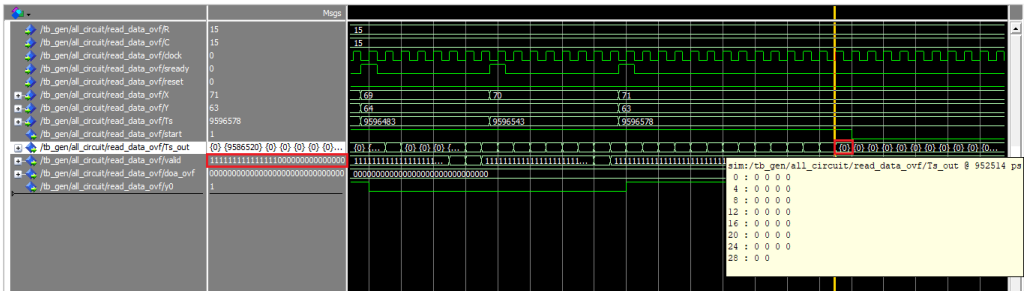


Figure 6.5. Last cycle

The local matrix is the same expected and, in this case, the last column is totally ignored.

Processing blocks

This part of the digital implementation represents the computation stage where the two columns per cycle are processed following the algorithm steps. It is composed by three units: Mult, Adjoint and Iteration blocks.

The math operations involved have been described in section 3.1. The outputs of each unit were compared with the Matlab results to verify their correctness.

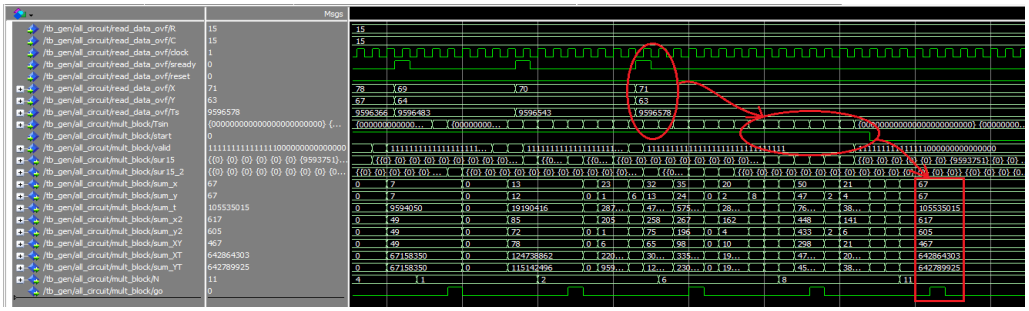


Figure 6.6. The Mult block

The results for Mult, without iterations, are:

- SX = 67
- SY = 67
- ST = 105535015
- SX2 = 617
- SY2 = 605

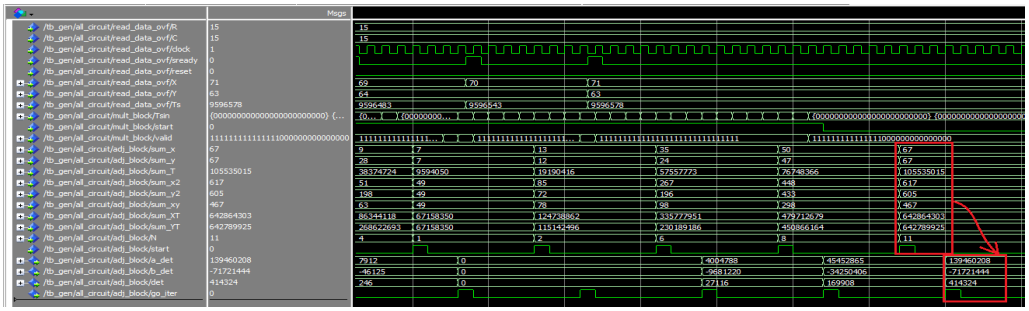


Figure 6.7. The Adjoint block

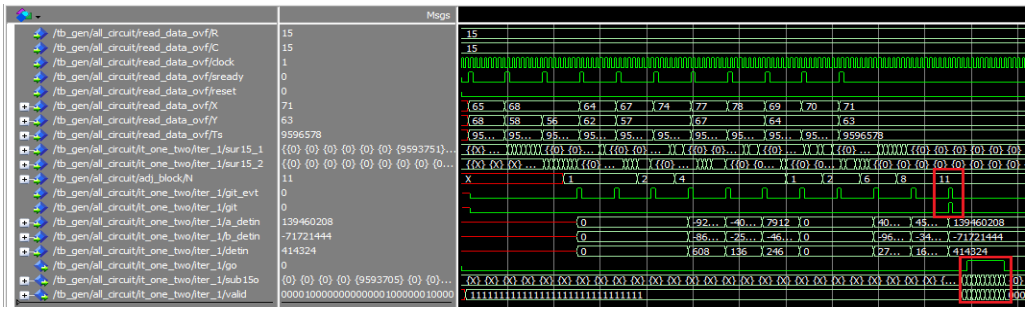


Figure 6.8. The Iteration block

- SXY = 467
- SXT = 642864303
- SYT = 642789925
- N = 11

The results for Adjoint, without iterations, are:

- a_{det} = 139460208
- b_{det} = -71721444
- det = 414324

It is important to remember that the output generation of the Iteration block is similar to the mechanism of the memory addressing: two columns per cycle are taken out, so that is possible to re-use the Mult entity to process again the neighbourhood. Note well that if min_evts condition is not reached, no iteration stage is called. In this way the algorithm stops and waits for the next computation.

Overflow detection
As described in section 3.3, the overflow detector allows to signal the presence of a wraparound and manage a control bit. This last, every time a wrap occurs, is set equal to zero for the past events just saved in memory, and will be one from the overflow event onward. Thanks to this state machine that controls the process, it is possible then to maintain fixed the distance between the events.

The following figure shows how this part works:

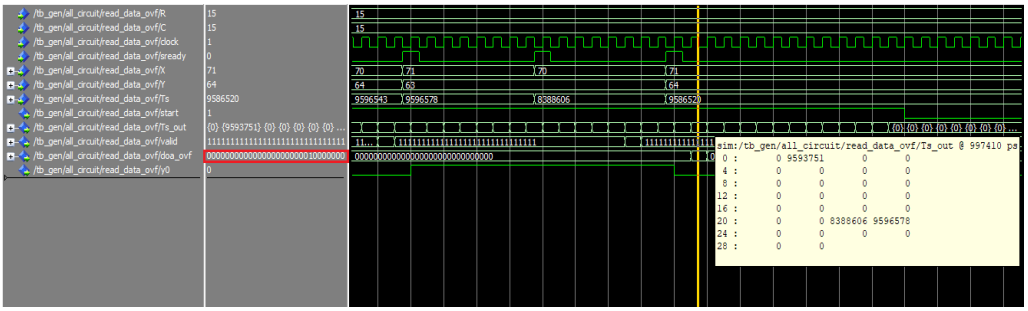


Figure 6.9. The Overflow detection block

From Figure 6.9 it is clear that the presence of a time stamp as 8388606 lower than the previous one (9596578) generates a reset of the doa_ovf bits. It is interesting to notice, though, that this new event will have no zero as control bit any more, but one.

Final stage
The last stage is composed by a certain number of logic cores which perform several steps in order to obtain the values of the differentials.
The first operation is made up by two dividers in parallel which calculate the divisions:

$$\frac{a_{det}}{det} \text{ and } \frac{b_{det}}{det}$$

The simulations for this step are done in *Vivado Design Suite 2017.1*.
The expected results are:

- a = $\frac{a_{det}}{det}$ = 336.59
- b = $\frac{b_{det}}{det}$ = -173.10

The outputs of the block are:

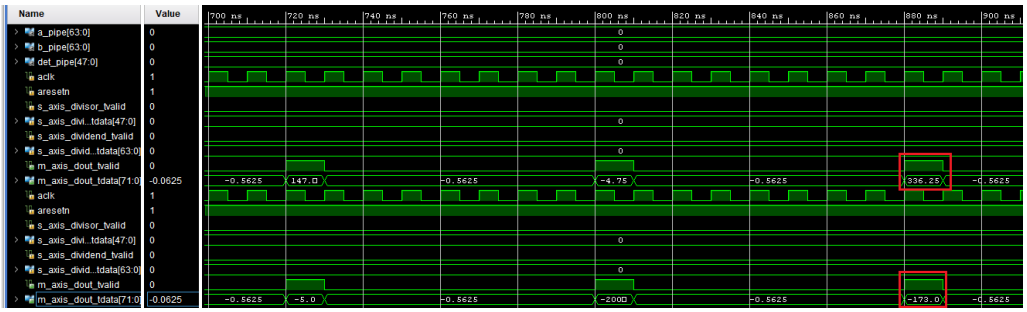


Figure 6.10. The Divisor block

Then the quotients are passed to two different branches. The first is made up of the arctangent followed by the cordic for sine/cosine computation. Let's analyse it.

The math results are:

- θ = atan(a/b) = 2.0458
- sin(θ) = 0.8893
- cos(θ) = -0.4573

The outputs of atan and cordic blocks are:

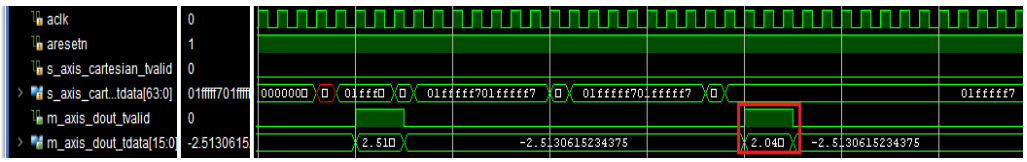


Figure 6.11. The Atan block

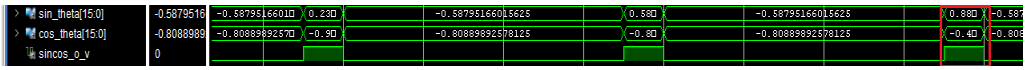


Figure 6.12. The Cordic block

The other branch performs firstly the square, followed by the sum, of the division outputs. Then, the sum is sent to a cordic in vectoring mode (to calculate square root) and, at the end, its output is inverted.

The math results are:

- $c = a^2 + b^2 = 143260$
- $\sqrt{c} = 378.4970$
- $(\sqrt{c})^{-1} = 0.0026$

The outputs of square and add, square root and reciprocal blocks are:

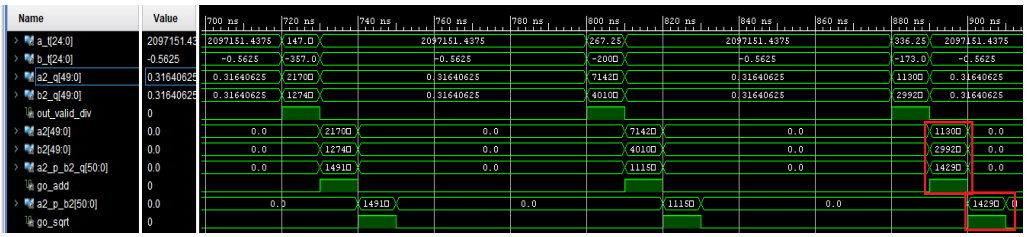


Figure 6.13. The Square and Add block

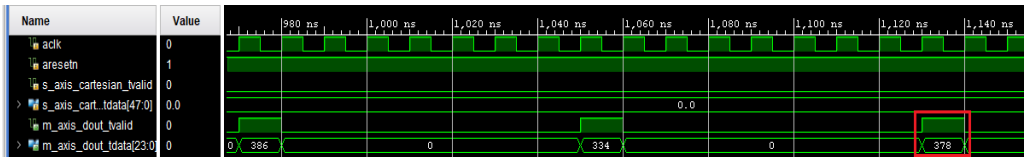


Figure 6.14. The Square Root block

Figure 6.15. The Reciprocal block

Validation memory

In order to verify the correct behaviour of the validation memory, the dataset and some simulation parameters have been changed. As described at the beginning of this chapter, the ”Casual events” set of data [Table 6.3] substitutes the one used until now. Also the local matrix becomes a 3x3 neighbourhood, with a minimum number of events equal to 3, and a dT lifetime of 9.This last condition means that, due to the fact that the FIFO increments its own counter with a frequency of 12.5 MHz (in the worst case of 1 event/80 ns incoming), an event can be considered extinct after 9*80ns = 720 ns.

The visual matrix associated to the Table 6.3 is represented as follows:

				Y		
		0	1	2	3	4
	0	7024	0	6449	675	100
	1	0	7600	33	2	0
X	2	0	0	7590	320	7470
	3	3240	1024	0	0	0
	4	0	256	0	550	0

It is clear that, after 9 events received, the oldest are deleted from the memory. The addressing mechanism is the same. The output of the FIFO returns the time stamp to delete and its related coordinates. These last are elaborated as the memory manager does for the new incoming events: if they point to a memory location where the content is equal to the FIFO-out, the event is expired. Otherwise the event in the BRAM is more recent, and so anything has to be done.

By looking at the figure 6.16, it is shown how the local matrix around the last event (n. 14) should not present the value of the time stamp (33) at the coordinates(1,2). This because between them there are more than nine events. As highlighted in 6.16, in fact, the expected Ts between 6449 and 7590 is not present.

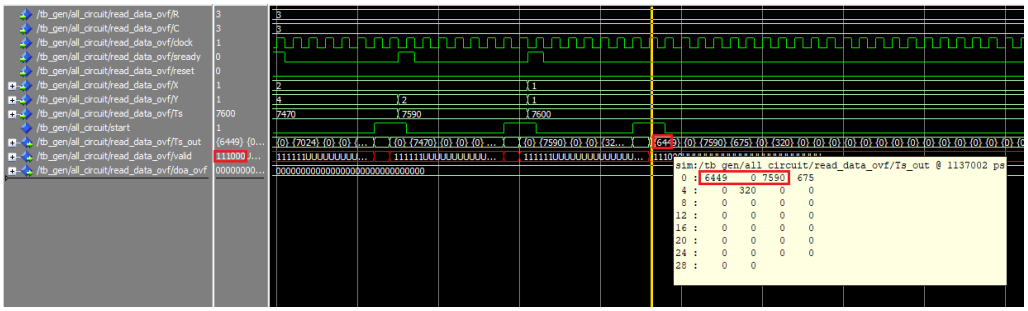


Figure 6.16. Second cycle for 3x3 matrix

Borders detection
Also for this case, the used dataset is ”Casual events”. The idea is to see for which inputs the computation starts. By looking at the table 6.3 and to the final expected matrix shown above, for the elements belonging to the entire column 0 and the row 0 any neighbourhood can be extracted.

The time stamps not considered are 100, 675, 3240, 6449 and 7024. In figure 6.17 these are marked in red:

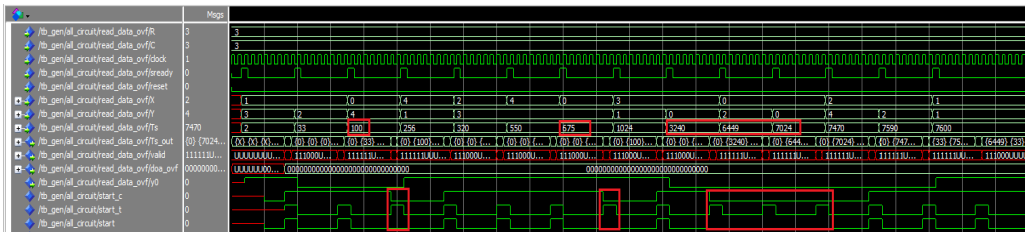


Figure 6.17. Border elements

6.2 System simulations

In order to have a complete view of the architecture behaviour, it results necessary to show the different working conditions for n_iter ∈ [0-1-2], and the latency associated to each of them. The used dataset is, as before, the one furnished by the iit.

The first case is related to a 15x15 local matrix, with no iterations.

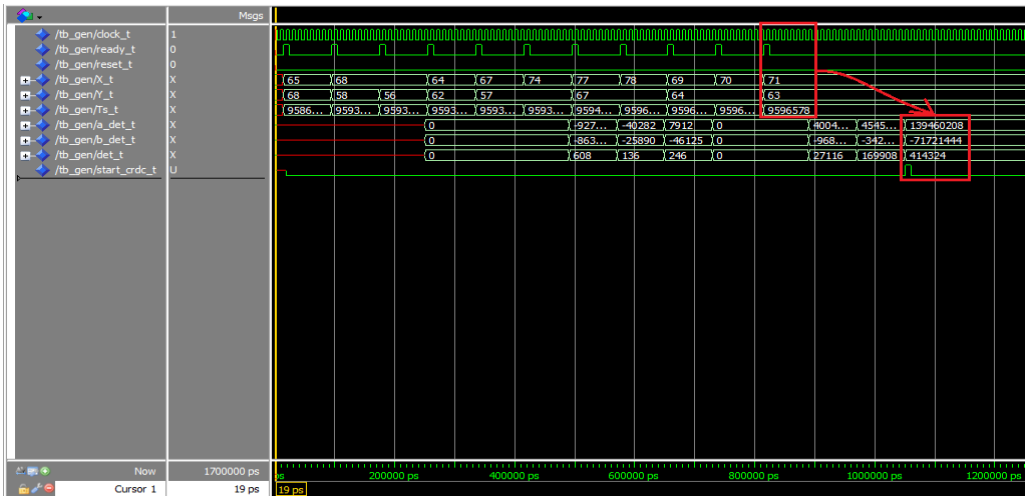


Figure 6.18. Top view with n iter = 0

As can be seen, only the last received event is elaborated and then passed to the other computational block (final stage). That's due to the min_events condition (10) that is not reached. The same happens for the other two simulations (Figures 6.19 and 6.20): in both cases the results are different from the first. This can be explained thanks to the removing of the outliers introduced by the first iteration stage. In the second, instead, no event is deleted and for that reason the outputs are the same of the first.

Figure 6.19. Top view with n iter = 0

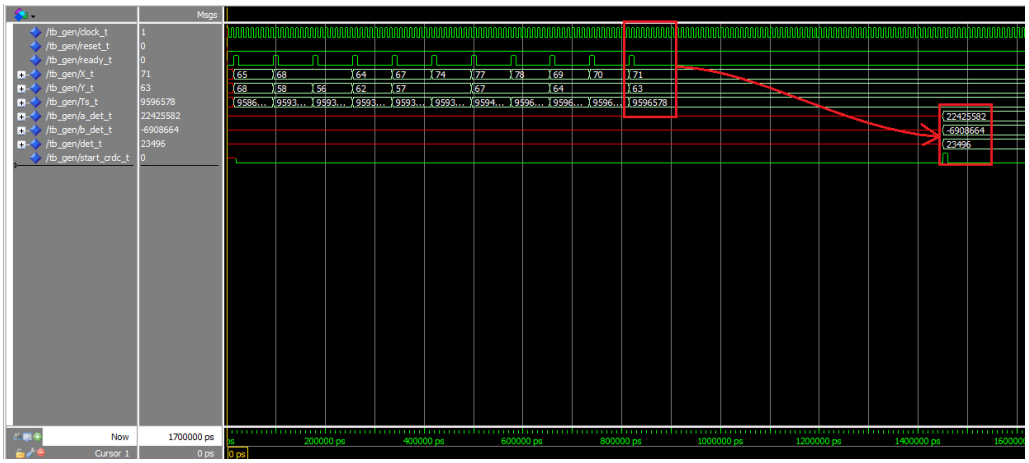


Figure 6.20. Top view with n iter = 0

The latency values, by using a 100 MHz clock, are:

Iteration	Latency [ns]
0	230
1	430
2	630

Table 6.4. Latency for different iterations

6.3 Synthesis outcomes

In this section all the informations related to the hardware implementation are shown for different configurations of the parametrized structure. By using the target device (Zynq XC7Z030-1SBG485C) inside the Vivado environment, all the libraries used for this kind of architecture are called. The estimations on power consumption, delays and area, hence, are explored after the synthesis and the implementation stages. The software tool, starting from the RTL design in VHDL language, generates a gate-level netlist in the first part and, then, moves on to the Place & Route in the second phase.

As just described, the only constraint used for this step is a 10 ns clock period. The structure is fully pipelined in order to satisfy this working condition. The entire digital design has been made to achieve the highest possible throughput, sacrificing at the same time the latency. In order to do this, an important parameter to take under control is the number of

FPGA resources used.
The analysis has been conducted by varying the neighbourhood dimensions (from 3x3 to 15x15) and the number of iterations (from 0 to 2).

The number of active BRAMs inside the structure is constant (16.98%): that's because this is not a parametrized element. It is independent from the variables set at synthesis level. Same considerations have to be done for the IO part (72.67%).

In the following graphs are shown, instead, the percentages of use of the units depending on the configuration parameters.

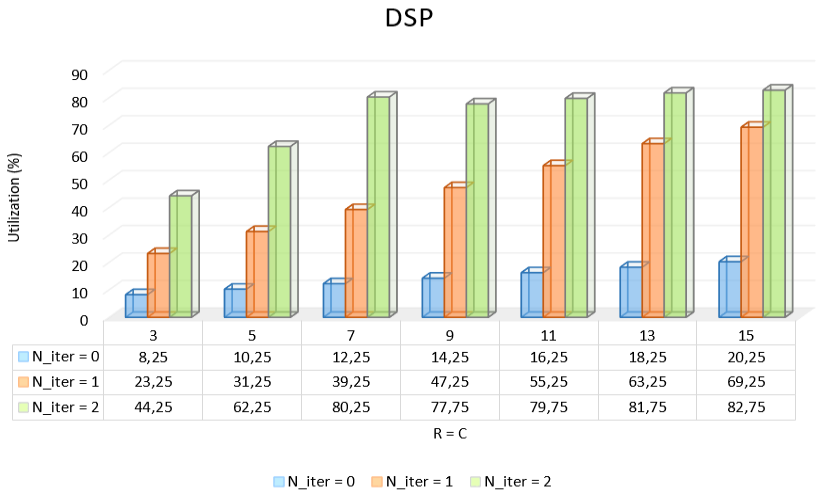


Figure 6.21. Number of used DSPs for different iterations

As expected, the growth associated to the increasing dimensions of the local matrix is linear for every kind of working condition. It is interesting to notice how also the difference between the values for n_iter equal to 0, 1 and 2 becomes relevant with this trend. Another important aspect to consider is related to n_iter = 2: after the R = C = 7 condition, the number of DSP stops growing. This can be explained with the capacity of the synthesizer to optimize better the resources inside the FPGA. This is evident also for other elements like LUTs and Flip Flops.

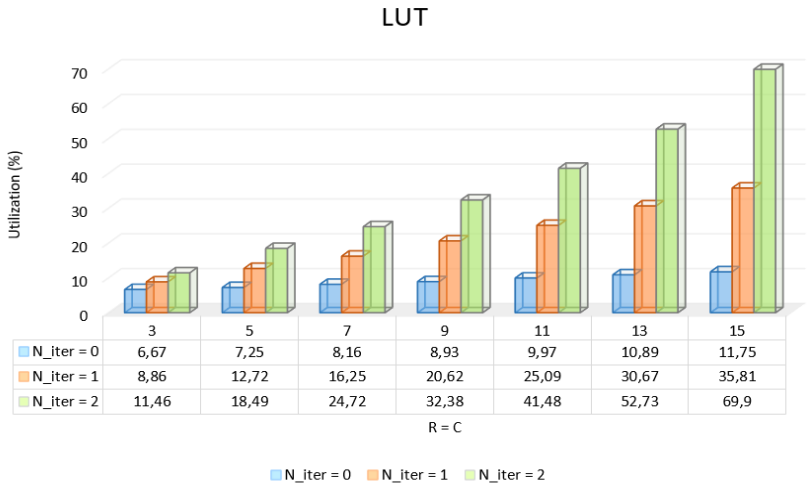


Figure 6.22. Number of used LUTs for different iterations

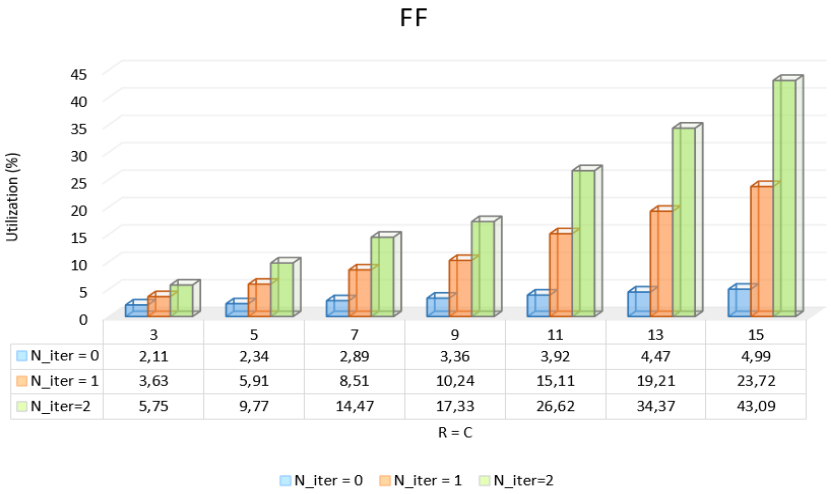


Figure 6.23. Number of used FFs for different iterations

The power consumption is a critical aspect of a system design. The implementation stage returns, in fact, an estimation of the total on-chip power and the fractions related to the dynamic part and the static one, only by using the set constraint on the clock.

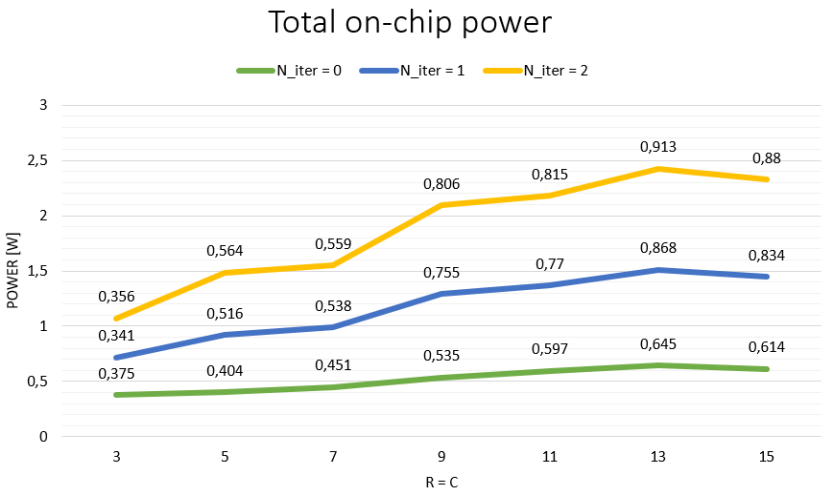


Figure 6.24. Total power consumption for different iterations

Also in this case the behaviour doesn't present any surprise. By increasing the number of iterations, the chip power needing becomes markedly dominant for huge neighbourhood dimensions. For n_iter = 0 any significant difference is appreciated. For n_iter = 2, vice-versa, the growth rate is decidedly higher.

The distribution of the two contributions is shown in Figure 6.25, 6.26 and 6.27.

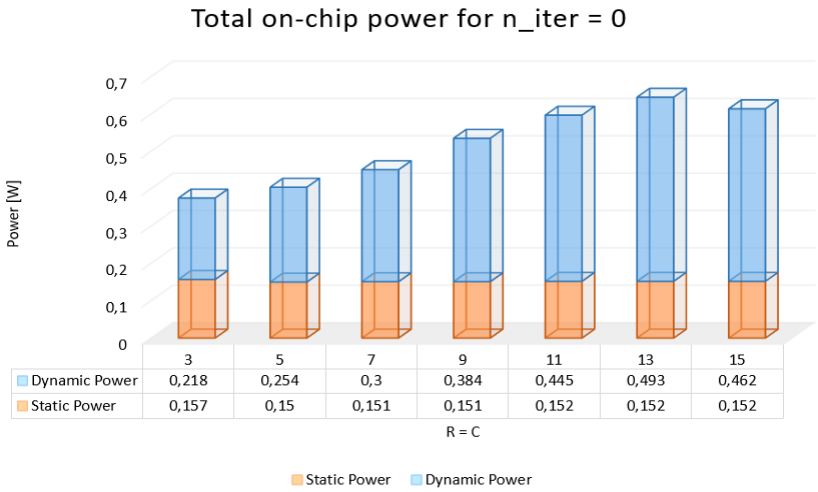


Figure 6.25. Static and Dynamic contributions for n iter = 0

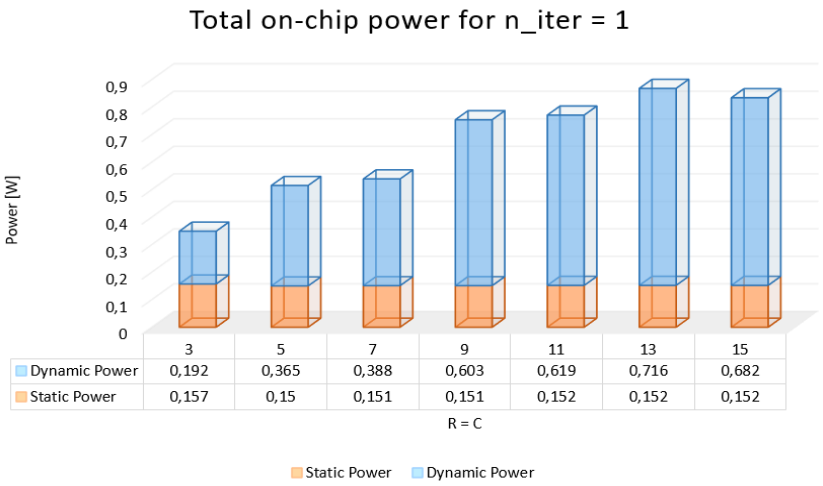


Figure 6.26. Static and Dynamic contributions for n iter = 1

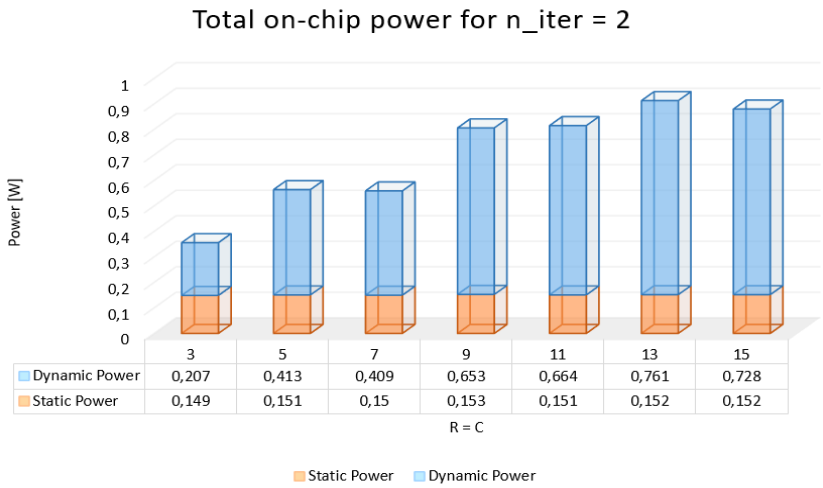
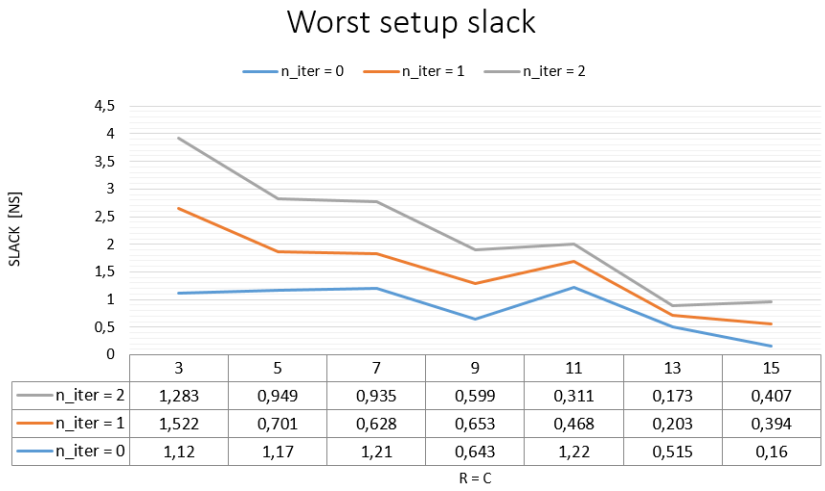


Figure 6.27. Static and Dynamic contributions for n iter = 2

The last element analysed at the end of the design synthesis is the worst setup slack. It is a fundamental value which indicates whether the timing constraints are met. It represents the remaining time that in the path, between two registers, can be taken without losing the system stability. In this case, it is referred to the critical path, before the arriving of a new edge (setup time).



	3	5	7	9	11	13	15
n_iter = 2	1,283	0,949	0,935	0,599	0,311	0,173	0,407
n_iter = 1	1,522	0,701	0,628	0,653	0,468	0,203	0,394
n_iter = 0	1,12	1,17	1,21	0,643	1,22	0,515	0,16

Figure 6.28. Worst setup slack

As can be seen from the Figure 6.28, it is evident that the growth of the system area (and so the number of the digital elements used) will result in a reduction of the available time to complete the path. The synthesizer will have more difficult in finding the best possible placement of the individual blocks that does not affect timing performances.

6.4 Future developments

This work has closely followed the entire development process of a reconfigurable and flexible architecture, which ensures the best performances starting from fixed system specs.
The aim will be, in this paragraph, to propose possible solutions to further improve the device both in terms of used resources and speed.

Test and comparisons
The next step will be to physically test the algorithm on FPGA and to compare the obtained results with their corresponding in the software version. This has already been done, partially, at simulation level with the insertion (inside the various test-benches seen in paragraphs 6.1 and 6.2) of a portion of events coming directly from the real dataset.

Fix parameters
As already seen in paragraph 3.3, the best architectural solution that allows an optimal use of the algorithm is for a number of iterations equal to 2 (in this case the results converge). The cost is to lose something in terms of precision. The error, in these cases, tends to be reduced for large matrices (case n_shift = 1), and this indicates a way to choose the best parameters. The idea of a reconfigurable device at synthesis time is useful for the different

experimentations, because it allows to evaluate from time to time the hardware impact of every combination of parameters. However, with the perspective of an ASIC realization it is advisable to rethink and optimize the system in order to allow savings in terms of area, consumption and performance.

Reduce the data parallelism
Also the choice of the righ-shifts to make on the input time stamps, in order to allow a correct management of the wraparound (2 at most), can be an interesting way to reduce the data parallelism. This can also involve significant reductions in terms of area of each block, always taking into account the possible implications from the point of view of the result accuracy. Moreover, the idea of adopting a dedicated library for each type of computational block (adders, multipliers and so on) can bring considerable advantages.

6.5 Conclusion

The purpose of this paper was to create an architecture, both in software and hardware, able to calculate the optical flow coming from an event-based vision camera. This new type of sensor introduces significant improvements in terms of performances (low latency) and the use of memory resources (no data redundancy), compared to the frame-based version.

The proposed device, totally reconfigurable, can be adapted to both the main types of sensors in commerce (ATIS and DVS). For any new incoming event, it extracts from the memory a matrix of neighbouring events that occurred in the previous moments. From that, it is then possible to obtain informations about the movement perceived by the camera.

The parameters that allow a personalized management at synthesis time are:

- The type of sensor to interface with;
- The dimensions of the local matrix, both rows and columns;
- The extinction time, after which an event is considered too old and, therefore, no longer valid;
- The minimum number of valid events, thanks to which is possible to start the computational step;
- The number of necessary iterations to make the results converge;
- The number of shifts to allow an optimal wrap-around management, without excessively losing in precision.

The statistical analysis carried out at software level, using a dataset provided by the Istituto Italiano di Tecnologia (iit), has highlighted the need to adopt a dedicated structure, optimized both for matrix dimensions and data parallelism, and, at the same time, with a minimum impact in terms of loss of precision.

Once all the advantages and possible disadvantages have been analysed, thanks to accurate tests on the structure, will be possible to integrate it with the iCub robot.

Bibliography

[1] [web] Il futuro dell'AI, [$https://www.wired.it/attualita/tech/2016/12/21/futuro-intelligenza-artificiale/?refresh\_ce=$]

[2] D. J. Amit, *Modeling brain function*, New York, NY, Cambridge University Press, 1989.

[3] Jan Larsen, *Introduction to Artificial Neural Networks*, 1st Edition, 1999.

[4] Simon Haykin, *Neural Networks and Learning Machines, Third Edition*, McMaster University, Canada, 2009, chapter 1.

[5] [web] Reti Neurali [$http://bias.csr.unibo.it/maltoni/ml/DispensePDF/8\_ML\_RetiNeurali.pdf$]

[6] J. Von Neumann and R. Kurzweil, *The computer and the brain*, Yale University Press, 2012

[7] C. Mead, *Neuromorphic electronic systems*, Proceedings of the IEEE, vol. 78, no. 10, pp. 1629–1636, Oct 1990.

[8] 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.

[9] N. Izeboudjen, C. Larbes, and A. Farah, *A new classification approach for neural networks hardware: from standards chips to embedded systems on chip*, Artificial Intelligence Review, 2014.

[10] M. Liu, H. Yu, and W. Wang, *FPAA based on integration of cmos and nanojunction devices for neuromorphic applications*, in Nano-Net. Springer, 2009, pp. 44–48.

[11] 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.

[12] 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.

[13] I.S. Han, *Mixed-signal neuron-synapse implementation for large scale neural network*, Neurocomputing, vol. 69, no. 16, pp. 1860–1867, 2006.

[14] B.V. Benjamin, P. Gao, E. McQuinn, S. Choudhary, A.R. Chandrasekaran, J. Bussat,

R. Alvarez-Icaza, J.V. Arthur, P.A. Merolla, and K. Boahen, *Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulation*, Proceedings of the IEEE, vol. 102, no. 5, pp. 699–716, 2014.

[15] [web] The brainscales project. brainscales - brain-inspired multiscale computation in neuromorphic hybrid systems. [https://brainscales.kip.uniheidelberg.de/]

[16] P. Lichtsteiner, C. Posch, T. Delbruck, *A 128x128 120 dB 15 µs Latency Asynchronous Temporal Contrast Vision Sensor*, in IEEE journal of solid-state circuits,2008.

[17] 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, 2011.

[18] 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.

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

[20] M. Jakubowski and G. Pastuszak. *Block-based motion estimation algorithms, a survey*. Opto-Electronics Review, 2013

[21] 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

[22] R. Benosman, C. Clercq, X. Lagorce, S.-H. Ieng, C. Bartolozzi, *Event-based visual flow* in IEEE Trans. Neural Netw. Learning Syst., 2014.

[23] Ibrahim Shour, A reconfigurable architecture for event-based optical flow in FPGA, Master's Thesis in Electronics Engineering, 2018.