Università di Pisa

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Elettronica

Final work

Design and FPGA implementation of a fast clustering algorithm for satellite lightning imaging applications

Nominee: Andrea Lampredi

……………………………………

Supervisors: Prof. Ing. Luca Fanucci

……………………………………

Ing. Daniele Davalle

……………………………………

Session 2012/2013

Summary

Thanks

Desidero dal profondo del cuore ringraziare tutte le persone che mi hanno aiutato ad arrivare a questo momento così fortemente voluto da sempre ma che all'inizio sembrava solamente una chimera.

Primi tra tutti i miei genitori e mio fratello, che mi hanno sempre sostenuto e sopportato nei momenti difficili e mi hanno donato la forza per non arrendermi mai. Hanno ascoltato le mie sfuriate, hanno passato anni a consolarmi e hanno riso e pianto con me, sempre.

Tutta la mia famiglia che mi ha visto tante volte uscire vittorioso ma a volte anche sconfitto.

La mia splendida fidanzata che ha reso facili e piacevoli tutti i momenti più difficili, che mi ha fatto tornare il sorriso anche dopo tuffi profondi e ha permesso che questa tesi arrivasse alla fine con la sua grandissima intelligenza e disponibilità.

I miei compagni di università che hanno fatto in modo di trasformare questi anni di università in una bellissima esperienza da vivere a pieni polmoni.

I miei compagni di squadra che, forse, hanno notato solo la parte più distruttiva del mio lavoro vedendo spesso, sulla mia faccia, la stanchezza di giornate passate sui libri ma che, nonostante tutto, mi hanno sempre permesso di ricaricare le pile.

I miei amici che mi ha sempre incoraggiato in tutto quello che facevo.

I professori che mi hanno seguito e mi hanno permesso di diventare quello che sono adesso.

Grazie.

Introduction

This thesis deals with the implementation of a new kind of clustering algorithm that will be used by a Lightning Imager (LI) mounted on ESA meteorological satellites of third generation (MTG). The development of this algorithm is commissioned to the Department of Information Engineering by Selex ES in cooperation with ESA (European Space Agency). At first, the algorithm is designed to meet the stringent timing requirements. Then, the new clustering algorithm is implemented on MATlab and VHDL languages.

The thesis is organized as follows:

- Chapter 1, Meteorological satellites, introduces the principal orbits used by every kind of satellite, the data provided by meteorological missions and the history - past, present and future - of meteorology. A particular section is reserved to EUMETSAT and its missions with a large presentation of EUMETSAT three generations of satellites:.
    - Meteosat First Generation series. A short resume of history, position, lifetime and technical data with a particular section dedicated to the principal instrument on board, the MVIRI;
    - Meteosat Second Generation, the active series in orbit around the Earth. It gives details about satellite principal instruments, SEVIRI and GERB, and their characteristics;
    - Meteosat Third Generation. A presentation about the new MTG series that will be soon in orbit. It explains which instrument will be on board and which characteristics they will have. Also, the difference between MTG-I and MTG-s will shown;
- Chapter 2, Lightning Imager, exposes what is an imager and the requirements necessary to implement this device. Then, it presents the problems during the capturing and which methods are used to increase the quality of events detection;
- Chapter 3 is focused on the LI technology and his most important parts: requirements will be analyzed and their feasibility will be evaluated. The next step is the design of the clustering algorithm and the work will be concluded with three output codes: a MATlab High-Level (MHL) model, a MATlab Bit-True (MBT) model and a prototype of VHDL model for hardware implementation.

Chapter 1

Meteorological satellites

In the modern meteorology, the satellites have several and fundamental functions: thanks to their continuous presence in orbit around the Earth, the weather is always monitored to find new meteorological phenomena. Furthermore, these satellites can pick up a lot of data about temperature, wind, NO_x gas, pollution, both on land and sea. Then, this great quantity of data is sent to the ground stations: they provide weather previsions (with model of meteorological forecast), update conditions about Ozone hole, Earth's pollution and many other applications..

1.1 Satellite orbit

Satellites can work on several orbits and the orbit depends on the kind of mission.

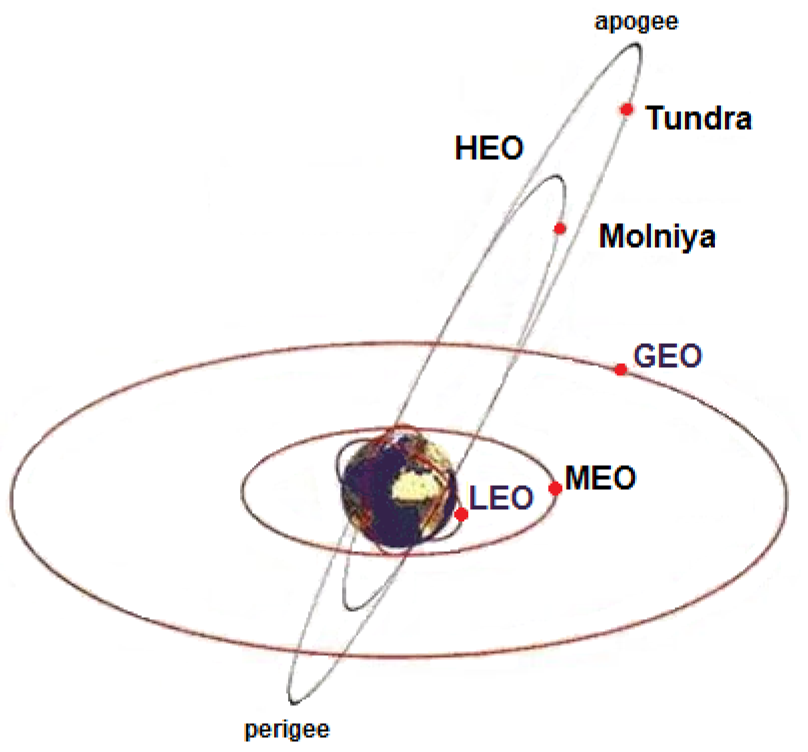


Figure 1.1: Kinds of orbit for meteorological satellites.

The orbits for satellites of commercial telecommunication's missions are usually GEO (Geostationary Earth Orbit) but, recently, there are some new satellites on LEO (Low Earth Orbits) and inclined orbit. On the other hand, meteorological satellites use geostationary and polar orbits. There are also many different possible orbits like MEO (Medium Earth Orbits) or HEO (Highly Elliptical Orbits).

In the following, a classification and a description of the different meteorological satellite orbits are presented.

1.1.1 GEO

The Geostationary Earth Orbit (GEO) was introduced in 1945: Arthur C. Clarke, a lieutenant of Royal Air Force (RAF), published a work ("Extraterrestrial Relays") on the English magazine "Wireless World" in which he talked about artificial satellites in orbit around the Earth [1]. He proposed only 3 satellites, synchronous with Earth rotation (geostationary), which can be used like "radio gate" in the space and have warranted communication for the whole planet.

Today, Geostationary satellites orbit on 35800 km around the Earth: this height is called "Geostationary orbit" because the time of revolution around the Earth is identical to the time of Earth's rotation (23 hours, 56 minutes and 4 seconds); so they are stationary relative to the surface (satellites rotate in the same direction of Earth's rotation). This type of satellites is also called geosynchronous. This is why this kind of satellites is usually used for meteorological forecasts: at 35800 km, a single satellite can scan about a third of all surface and it can continuously follow atmospheric phenomena like cyclones and hurricanes. Furthermore, geostationary satellites are usually used for device which have point-to-point or broadcasting communication because antennas of these devices are simple and fixed to the same direction: they don't need a continuous calibration towards the satellite.

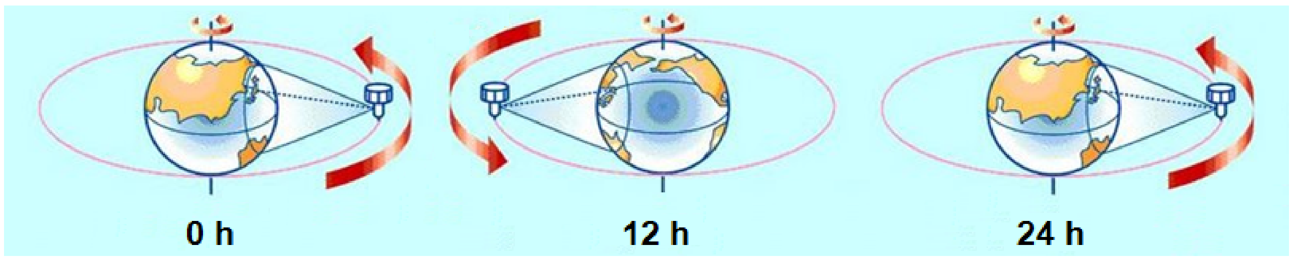


Figure 1.2: Revolution period around the Earth of geostationary satellite.

5 satellites are needed in order to cover all the Earth's surface. At the moment, there are lots of geostationary satellites [2]:

- METEOSAT for Europe;
- GOES for USA;
- MTSAT for Japan;
- Fengyun-2 for China;
- GOMS for Russia;
- KALPANA for India.

A geostationary orbit must stay on the same plane of the Equator; so satellites have a decreasing view towards poles. Polar and sub-polar area are difficultly covered by this kind of satellites because antennas could be pointed under the horizon line.

Typical parameters of GEO:

- Height above equator: 35,786 km;
- Orbit radius: 42,155 km (Orbit circumference: 264,869 km);
- Orbital velocity: 11,066 km/h = 3.07 km/s;
- Latitude coverage: between 75° North to 75° South.

As already mentioned, GEO satellites have significant advantages: just 5 GEO satellites are sufficient to cover all the communication on the planet. In addition, GEO satellites have a fixed position relatively to the Earth surface, which eases the satellite pointing for ground devices.

This orbit has also some cons. First of all, satellites on this orbit need expensive missions for their launch, because of the high distance from the surface. They usually use multistage launcher (often Ariane vehicles were used in the past for these missions, [21]) . Moreover, the propagation of radio waves has high delay, 0.12 seconds, still due to the high satellite distance. This delay is satisfactory for broadcast communication but hardly sufficient in telephone communication.

1.1.2 LEO

When a satellite orbit around the Earth at an altitude up to 2000 km, the orbit is called “Low Earth Orbit” (LEO). Typically, LEO’s satellites work between 300 and 800 km. Under 300 Km, because of high density of gasses in atmosphere, the high friction against them decelerate the satellite resulting in a fast orbital decay. Too much energy (high velocity) is necessary to maintain the satellite in orbit. Above 2000 Km, there is the “Van Allen belt” (an area full of free charges held together by the Earth’s magnetic field) which can provoke malfunction in the electronic circuits on board due to the high exposition to radiations [1].

As defined by Keplero's laws, LEO's satellites orbit near the planet with very high velocity, about 30000 Km/h, and make a full revolution around Earth in 90 minutes. Almost all of space voyagers had place in low orbit and the spatial station stay, even today, in this orbit too (Saljut, MIR and International Space Station, [20]).

Compared to GEO satellites, LEO's satellites orbit near the surface: it allows communication with low delay (20–25 ms, like some terrestrial communication) so they are used for remote sensing and military missions. Furthermore, the cost of LEO's launch is cheaper than GEO's launch. LEO's satellites are also suitable for the communication with the surface: low distance allows low power and simple antenna for communication.

On the other hand, low orbit has many cons. First of all, LEO's satellite are visible by the ground station only for few minutes over the horizon and, during this time, it's very fast. The rest of time, the satellite is useless for that station. Obviously, because of his velocity, the ground station antenna must be oriented every time towards satellite in order to maintain the communication. Moreover, the communication is affected by an high Doppler effect: this effect must be compensated automatically by electronic circuits on board but they increase the complexity of the satellite's system.

In the past, LEO satellites were almost unused because of their short view time, but recently they are largely used thanks to the high number of in-orbit satellites on several orbital planes: today, with about 100 satellites the planet is fully covered and communication is continuously allowed.

LEO satellites are used mainly for mobile telephone with full planet's coverage like Globalstar or Iridium.

If the orbit of these systems is very inclined, it becomes polar orbit.

1.1.3 Polar orbit

Polar satellites orbit around Earth at an altitude of 800 km. A single satellite stays on the same area of surface for only twice a day: more than one satellite and a system of coordination between them are needed for meteorological observation. At the moment, meteorological satellites on polar orbit are [2] :

- METOP by EUMETSAT for Europe;
- NOAA and QuikSCAT for USA;
- Meteor for Russia;
- Fengyun-1 for China.

Polar satellites guarantee a better vision of planet than GEO's satellites: polar satellites can observe even polar areas and that's why they are used for missions of remote sensing and surveillance.

Polar satellites have orbits inclined of about 90 degrees with respect to the Equator plane and, usually, they are sun-synchronous, i.e., (the same area is scanned by the same satellite at the same moment of the day, during every season). Therefore, while the satellite scans from North to South on his orbit, the planet makes his rotation orbit and the result is that satellite scans all the surface, step by step. Because of these characteristics, polar satellites are perfect to scan the evolution of meteorological phenomena in the same condition and compare these for long time periods.

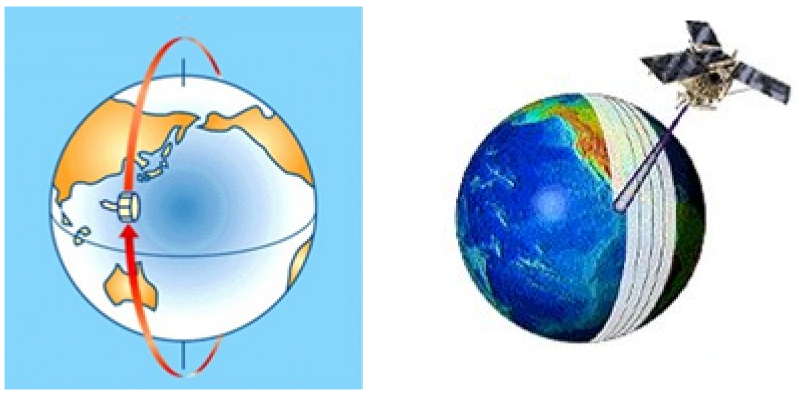


Figure 1.3: Full disc scan by polar orbit.

1.1.4 MEO

The "Medium Earth Orbit" (MEO) is a circular orbit at an altitude of about 10000 Km. Their orbital period is about 6 hours and the maximum period of over-horizon time is about some hours; with 10-12 satellites (on 2-3 orbit planes) the planet is fully covered. The most famous system based on MEO is the Global Positioning System (GPS), a system for accurate positioning on the planet surface.

1.1.5 HEO

The "Highly Elliptical Orbit" (HEO) was used for the first time by Russia to create communication with sub-polar zones which are isolated for GEO's satellites. HEO have three geometrical characteristics:

- Perigee, the lowest altitude point in the orbit, at about 500 km;
- Apogee, the highest altitude point in the orbit, at about 50000 km;
- Elliptic orbit at 63.4 degrees with respect to Equator.

In HEO, Earth stays in one of the two foci of the elliptical path. For this reason, the satellite stays for two third of its period near the Apogee: so, with a right positioning of the Apogee point, the area of interest can be rightly covered. Obviously, when the satellite is on the Perigee, the coverage of the affected zone is not guaranteed: so, more satellites are needed for a sufficient coverage, all together on the same orbit and with an accurate time distance. An example of this orbit is the system Molniya, used to cover Siberia. HEO has the same cons of GEO and LEO: for high distance (Apogee), communication have high delay and need high power (like GEO) and, for the high velocity, there is an high Doppler effect (like LEO).

1.2 History of meteorological satellites

The first satellites in the world, Sputnik, was sent by Russia in 1959 but it didn't have meteorological instrument on board. The first satellite with meteorological instruments on board was Vanguard-2, in 1959. This satellite, by NASA, was destroyed during the first part of mission, so its data was not available. First images of the Earth were transmitted in 1960 by TIROS-1 (Television and Infra Red Observation Satellite), again by NASA: this satellite had two cameras and was used for only 78 days. These kinds of satellites have become important since 1961, when images of the hurricane "Carla" helped to save a lot of people in Gulf of Mexico.

In 1964 the first satellite of Nimbus series, Nimbus-1, introduced some news about technology: it was the first satellite stable on all the 3 axes, so it could point always towards the same direction, and it was the first polar sun-synchronous satellite.

In 1969, Russia started its meteorological satellites program and, in the same year, Nimbus-9 was sent in orbit: this satellite had on board measure's instruments to pick up data about temperature, pressure, wetness..

In 1974 USA sent its first GEO's satellite, SMS-1 (Synchronous Meteorological Satellite), and, in 1977-1978, also Europe and Japan sent their first GEO's satellite (Meteosat for Europe and GMS, Geostationary Satellite Meteorological, for Japan). With Meteosat-1, some wavelengths, typical of water vapor (6.7 mm), were analyzed for the first time. In the 80's, also India sent its first GEO's satellite: with this one, the planet was fully covered and meteorological data were available 24/24 hours.

Now, the research will be focused about forecasting over European continent and about European missions in the past, in present day and in the future. First of all, it is necessary to know which organization is responsible of European meteorological missions.

1.3 EUMETSAT

EUMETSAT [3] (European Organization for the Exploitation of Meteorological Satellites) is an international organization created in 1986 to handle European meteorological satellites. The organization manages launch and control of satellites and their data transmission for meteorological and climate conditions. EUMETSAT is composed by 27 European states: Austria, Belgium, Croatia, Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Ireland, Italy, Leetonia, Luxemburg, Norway, Netherland, Poland, Portugal, UK, Romania, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey. Also EUMETSAT has cooperation accord with Bulgaria, Iceland, Lithuania and Serbia.

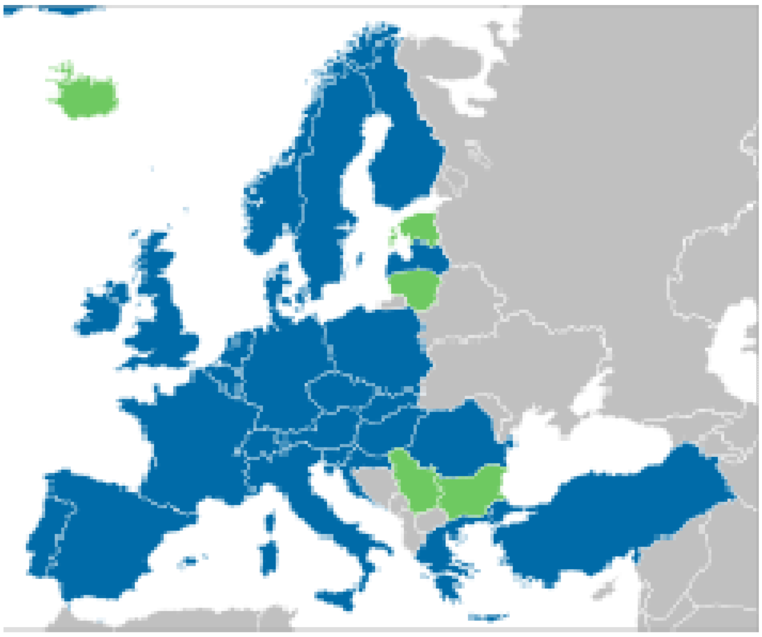


Figure 1.4: EUMETSAT members and country cooperating.

🟦 Member states.

🟩 Cooperation accord states.

As already said in the Sections 1.1.1 and 1.1.3, EUMETSAT's meteorological satellites are of two kinds: geostationary (Meteosat) and polar (Metop). Other similar satellites are handled by NOAA, the USA agency; EUMETSAT works together with other international agencies, (included NOAA), to distribute meteorological information and exchange on board instrument's technologies.

In the past, geostationary satellites of EUMETSAT were of two kinds and a third will be able soon:

- MFG (Meteosat First Generation);
- MSG (Meteosat Second Generation);

- MTG (Meteosat Third Generation).

In order to understand when every satellite must be launched, it is necessary to know that every satellite has a programmed lifetime: for old satellites, lifetime is about 5 years, instead for new satellites it can also be 7-10 years. So, change of satellite's generation overlaps with the end of old satellites' nominal lifetime.

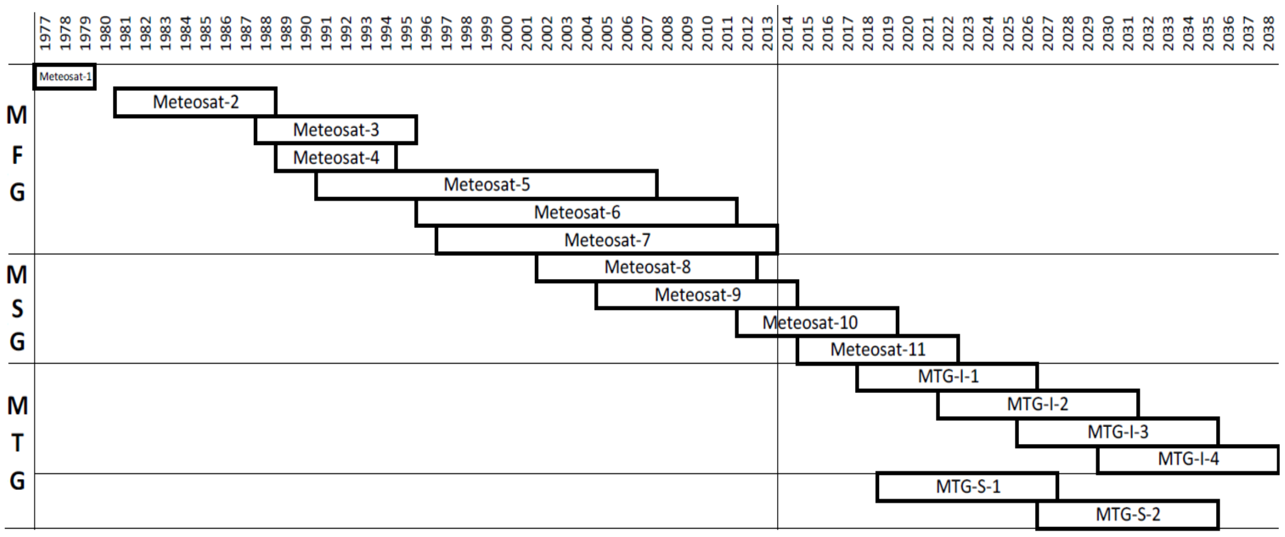


Figure 1.5: Lifetime of EUMETSAT's satellites.

In the image it is possible to see that there are at least two active satellites in orbit: one on position 0°N-0°E and one on position 0°N-9.5°E with backup functions. There were only a data gap of 20 months between the failure of Meteosat-1 and the launch of Meteosat-2.

1.4 Meteosat First Generation (MFG)

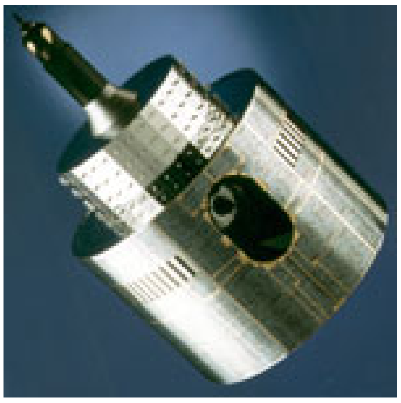


Figure 1.6: MFG satellite.

In 1968, the nations of ESRO (European Space Research Organization), now called ESA (European Space Agency), started studies of satellites' application, including weather satellites. Meteosat introduced a global system of geostationary platforms capable to observe in near real-time atmospheric condition and weather around the equator. In September 1972 ESRO officially adopted the Meteosat program and launched the first prototype of MFG, Meteosat-1, in November 1977, followed, in August 1981, by Meteosat-2, [17],[22] .

Meteosat-1 was the first European satellite to send images of the Earth surface from a geostationary orbit: obviously, they were in black and white but the definition was good. In Figure 1.7, the first image of Meteosat-1, clouds and continent edges are perfectly defined and it is quite easy to distinguish desert, ocean, rainforest and polar territory.

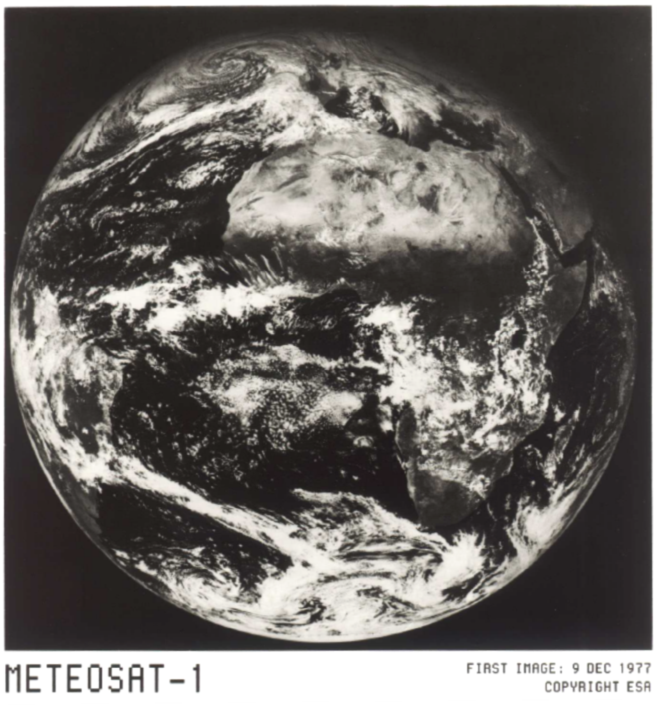


Figure 1.7: First image from an EUMETSAT's satellite (Meteosat-1, 9 December 1977).

The imager of Meteosat-1 failed prematurely in November 1979. Meteosat-3 was an old engineering prototype, similar to Meteosat-2, which was launched in 1988 after refurbishment to successfully fill the gap between Meteosat-2 and Meteosat-4. Between 1991 e 1995, Meteosat-3 was repositioned over

West Atlantic to replace temporarily GOES services[22]. Meteosat-4, Meteosat -5 and Meteosat-6 were launched between 1989 and 1993. These three satellites were part of MOP (Meteosat Operation Program) missions: Meteosat-4 was MOP-1, Meteosat-5 was MOP-2 and Meteosat-6 was MOP-3. Meteosat-5's primary mission was a routine service IODC (Indian Ocean Data Coverage) to provide data over Indian Ocean. Meteosat-6 had the RSS (Rapid Scanning Service) function: it allowed a rapid scan of the full disc of the Earth. [22]

In May 1991, EUMETSAT decided to establish an independent ground segment, to replace the system created by ESA in 1977. This was the start of the Meteosat Transition Program (MTP), which covered the phasing out of the MOP to the begin of the Meteosat Second Generation program. On 15 November 1995, the control of Meteosat satellites in orbit passed to EUMETSAT. Meteosat-7, the last satellite of series, was launched in orbit on 2 September 1997and it is still operative over the Indian Ocean. [5]

LIFETIME OF MFG SATELLITES		
Satellite	**Prime date**	**Retirement date**
Meteosat-1	09/12/1977	25/11/1979
Meteosat-2	16/08/1981	11/08/1988
Meteosat-3	11/08/1988	31/05/1995
Meteosat-4	19/06/1989	04/02/1994
Meteosat-5	02/05/1991	16/04/2007
Meteosat-6	21/10/1996	15/04/2011
Meteosat-7	02/09/1997	2016 (still operating)

Table 1.1: Lifetime of MFG series.

1.4.1 MFG characteristics

MFG satellites were 2.1 meters in diameter and 3.195 meters long; their original mass in orbit were 282 Kg but the propellant (hydrazine) used for orbit-keeping added 40 Kg at the beginning of mission. In orbit, the satellite was spin-stabilized; it spun at 100 rpm around its principal axis, which was almost aligned to the Earth's rotational axis. [4]

Meteosat MFG was composed by a main cylindrical body, with a drum-shaped section (diameter 1.3 m) on the top. Others two cylinders were stacked concentrically. The main body contained most of the satellite systems like the radiometer. Its external surface was covered with 6 solar cells (more than 8000 cells) used to produce electrical supply for a total power of 200 W (average). These panels had also sensors, thrusters and external connectors. The cylindrical surface of the smaller drum-shaped section contained an array of radiating dipole antenna elements; its function was to ensure that transmissions (in S-band) would always directed towards the Earth. The two cylinders on top of the satellite were toroidal pattern antennas for S-band and low UHF respectively.

During the launch, an apogee boost motor with solid propellant was mounted on the bottom of the satellite. This was used to move the satellite from the position post-launch, highly elliptical orbit, into geostationary orbit. When the motor was used, it was jittered to leave a gap and have a better cooling of the radiometer infrared detectors. [22]

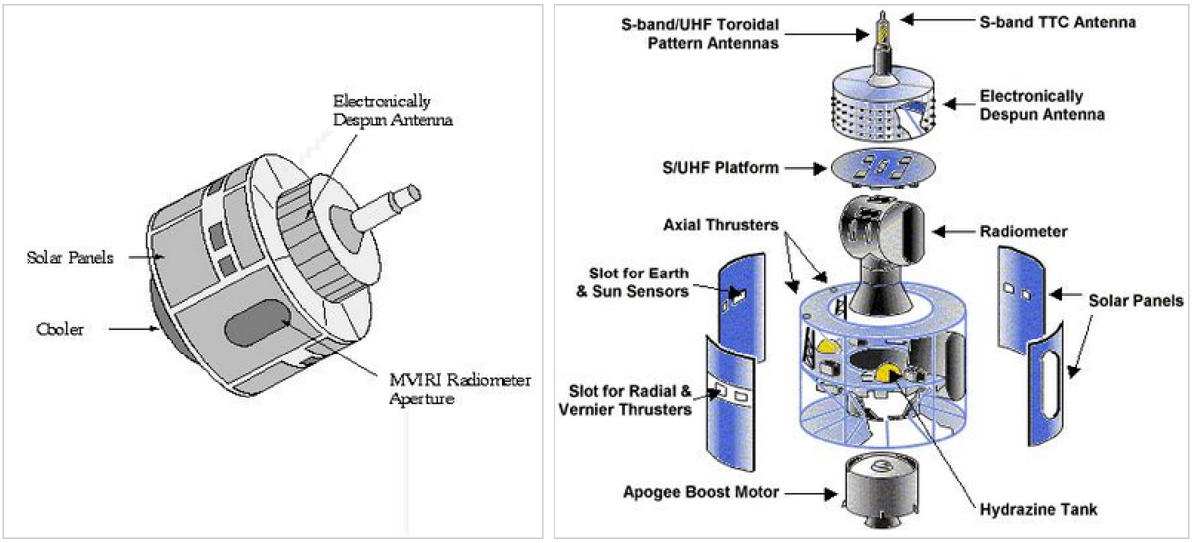

Electronically Despun Antenna
Solar Panels
Cooler
MVIRI Radiometer Aperture
S-band/UHF Toroidal Pattern Antennas
S-band TTC Antenna
Electronically Despun Antenna
S/UHF Platform
Axial Thrusters
Radiometer
Slot for Earth & Sun Sensors
Solar Panels
Slot for Radial & Vernier Thrusters
Apogee Boost Motor
Hydrazine Tank

Figure 1.8: Structure of a MFG satellite.

MFG's primary mission was to capture high resolution images of Indian ocean. The main instrument of MFG was the MVIRI (Meteosat Visible and Infra-Red Imager), a high resolution radiometer with three specific bands. MVIRI had a weight of 63 Kg and a height of 1.35 m; it provided the principal data of Meteosat system, in form of radiances from visible and infrared parts of electromagnetic spectrum. These radiations were gathered by a reflecting telescope, with a primary mirror diameter of 400 mm and a secondary mirror diameter of 140 mm.

MVIRI's procedure of data capture was easy: it acquired images and data from full Earth disc during a period of 25-minutes with a max resolution at Nadir of 5 Km in IR and 2.5 Km on Visible; this period was followed by others 5 minutes necessary to reposition the satellite. So, a complete set of full Earth disc images was available every 30 minutes. A great pros of this kind of satellite was that instruments on board allowed continuous imaging of the Earth. Furthermore, MVIRI provided data for many researches and meteorological applications, as a detailed control of atmosphere's state; these data, with the past ones of the atmosphere, can be used to make a prediction of future conditions.

1.4.2 MVIRI

Radiometer MVIRI operated in three spectral bands, important for images distribution [6] :

- Visible band (VIS). It was positioned between 0.45 and 1.0 µm and it was used for imaging during daylight. This band corresponded to peak of solar irradiance. This channel had a spatial resolution of 2.5 x 2.5 km^2 ;

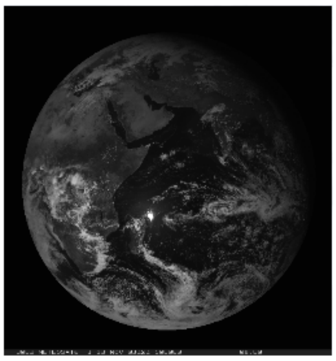


Figure 1.9: MVIRI visible band.

- Water Vapor absorption band (WV). It was positioned between 5.7 and 7.1 µm and it was used to measure quantity of water vapor (wetness) in the upper troposphere. It was easy to measure it because atmosphere is very opaque if water vapor is present but it is transparent if air is very dry. This channel had a spatial resolution of 5 x 5 km^2 ;

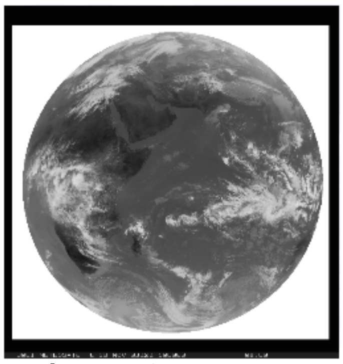


Figure 1.10: MVIRI water vapor absorption band.

- Thermal Infrared band (IR). It was positioned between 10.5 and 12.5 µm and it was used for imaging by day and night to determinate temperature of cloud tops and ocean’s surface. In fact re-emission of atmosphere and surface‘s radiation peak is proportional to their temperature. This channel had a spatial resolution of 5 x 5 km^2.

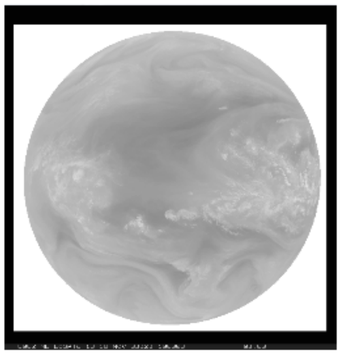


Figure 1.11: MVIRI infra-red band.

Every channel had a FOV of 18°.

1.5 Meteosat Second Generation (MSG)

In 2002, the first MSG satellite was launched. Today, there are 4 active satellites in orbit: Meteosat-8 and Meteosat-9 over Europe, Meteosat-7 over Indian ocean. Meteosat-10, launched in 2012, is the prime operational geostationary satellite (0°N, 0°E). Meteosat-7, launched in 1997, is the last of MFG satellites; it operates on Indian ocean and it is used to fill the data gap over this area. The last MSG satellite, MSG-4 (will became Meteosat-11), is in design phase and it will be launched in 2014. Meteosat-8, launched in 2002, is a backup-data satellite; furthermore, Meteosat-8 has function of Rapid Scan: it sends an image of Europe and North Africa (between 15° lat. and 70°lat. North) every 15 minutes. These images are useful to follow high-impact meteorological phenomena. Meteosat-9, launched in 2005, provides a Rapid Scanning Service, a fast sequence of images, every 5 minutes, of Europe, Africa and adjacent zones [4] .

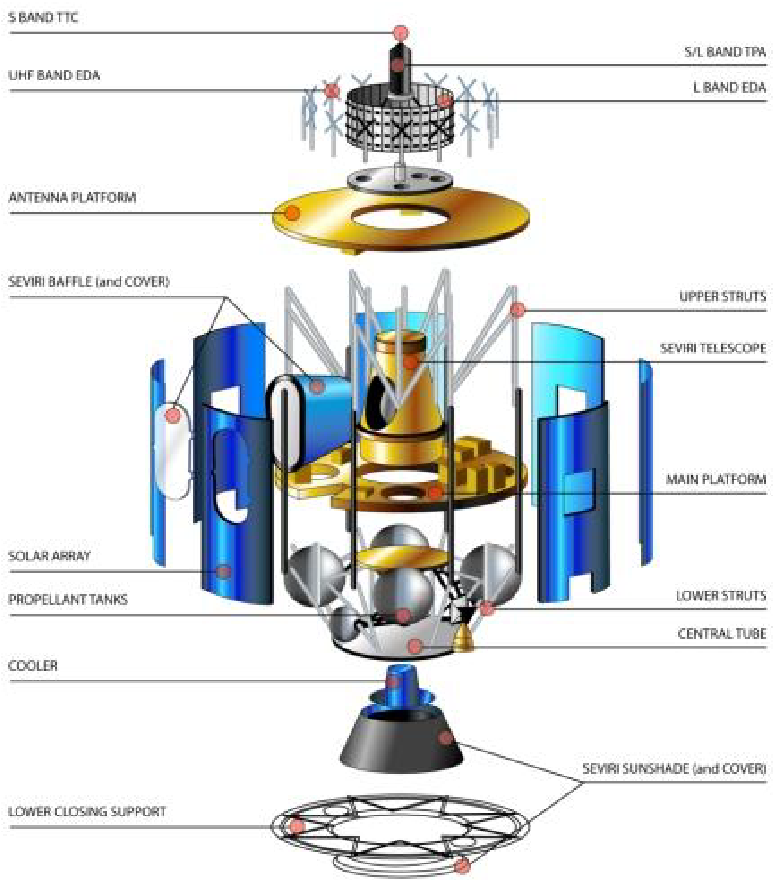


Figure 1.12: parts of MSG satellite.

MSG satellites send an images of Earth in 12 different spectral channels every 15 minutes. These data are used to monitor high impact phenomena to save lives or properties; an early detection of these phenomena has just saved thousands of lives and a lot of damages were just avoided to industries, transports, agriculture and energy.

This kind of satellite has a main cylindrical body, 3.2 meters of diameter and 2.4 meters of height, it has a total weight of 2040 kg and it is spin-stabilized with a rotation speed of 100 rpm. Its energy consumption is 600 w. It is composed by three principal parts: measuring central system, communication system and support-movement platform.

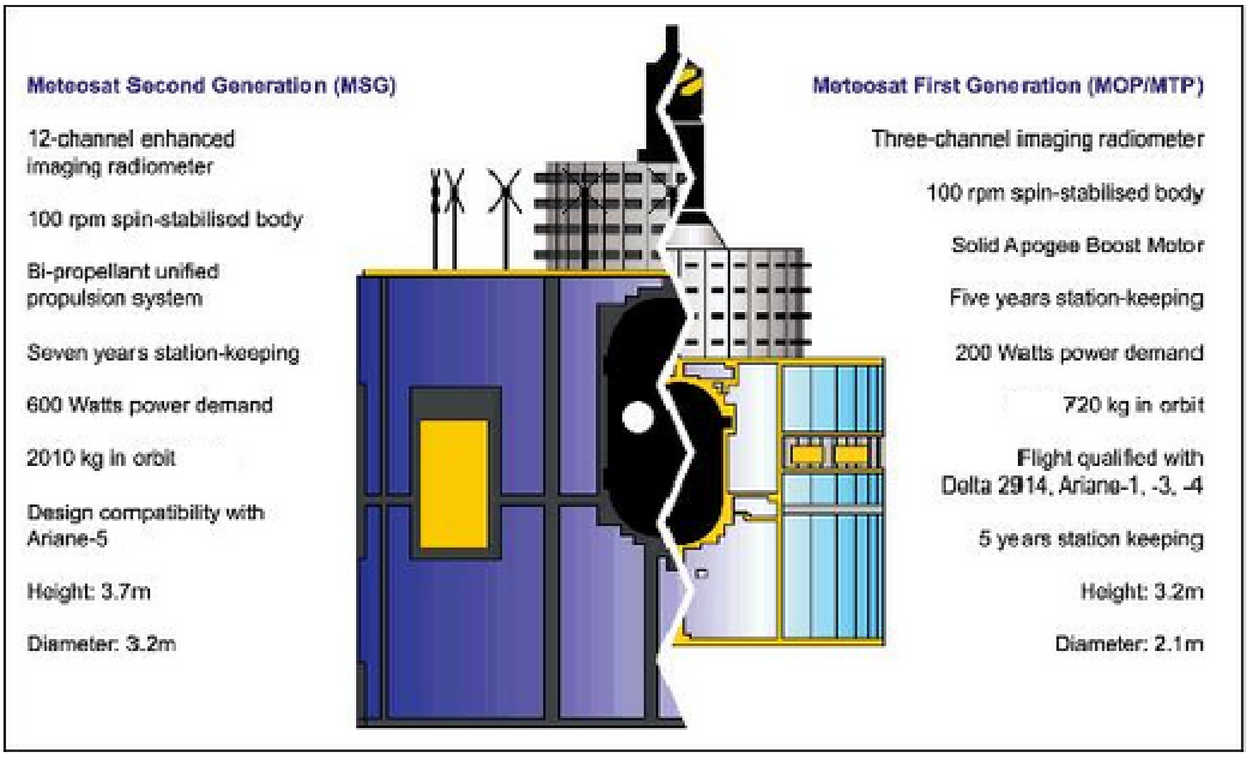

Meteosat Second Generation (MSG)
12-channel enhanced imaging radiometer
100 rpm spin-stabilised body
Bi-propellant unified propulsion system
Seven years station-keeping
600 Watts power demand
2010 kg in orbit
Design compatibility with Ariane-5
Height: 3.7m
Diameter: 3.2m
Meteosat First Generation (MOP/MTP)
Three-channel imaging radiometer
100 rpm spin-stabilised body
Solid Apogee Boost Motor
Five years station-keeping
200 Watts power demand
720 kg in orbit
Flight qualified with Delta 2914, Ariane-1, -3, -4
5 years station keeping
Height: 3.2m
Diameter: 2.1m

Figure 1.13: Comparison between MFG and MSG.

Each MSG satellite has an active lifetime in orbit of about 7 years. The current policy is to keep in orbit two operable satellites and launch a new satellite when the fuel in the eldest one is almost over.

Lifetime of MSG satellites		
Satellite	**Nominal fuel lifetime**	**Position**
Meteosat-8	28/08/2002 – until 2019	3.5° E/36 000 km
Meteosat-9	21/12/2005 – until 2021	9.5° E/36 000 km
Meteosat-10	05/07/2012 – until 2022	0° E/36 000 km
Meteosat-11	2015 – until 2023	-/36 000 km

Table 1.2: Nominal fuel lifetime and position of MSG series.

Since first satellite, instruments on board have become more particular and specific. Today, every meteorological program has its complex instruments, which try to provide more specific data of atmosphere and surface.

Main functions of Meteosat satellites are detecting and predicting high impact meteorological phenomena up to 6 hours. These satellites pick up atmospheric and surface information with an instrument called radiometer. MSG satellites in orbit have two radiometer on board: SEVIRI (Spinning Enhanced Visible and Infra-Red Imager) and GERB (Geostationary Earth Radiation Budget).

1.5.1 Radiometer SEVIRI

The principal radiometer on MSG series is called SEVIRI (Spinning Enhanced Visible and Infra-Red Imager). It is a new generation of geostationary orbit instrument for imaging and sounding. SEVIRI measures a physical variable called Radiance: it is a flux density of electromagnetic radiation for solid angle (it is just an intensity of electromagnetic radiation measured in a specific frequency band); these radiations are kept up by the telescope, channels are separated by mirrors on the telescope's focal plane and then they are focalized on detectors. SEVIRI uses a bi-dimensional scansion that combines satellite and on-board-mirrors rotation: with this movement, it scans Earth's surface every 15 minutes on 12 different spectral channels to provide data about atmosphere, temperature, clouds and surface. At each satellite revolution, three images lines are acquired: it has a scan capability of 22° N-S and 18° E-W [23]. A full Earth's disc image is created in about 12 minutes; others 3 minutes are used to position the mirror on its initial position and to recalibrate it with a black-body on its optical path [16]. In particular, SEVIRI has an HRV channel (High Resolution on Visible) with max resolution of 1 Km, which is used to predict high-impact meteorological phenomena in local and extended area.

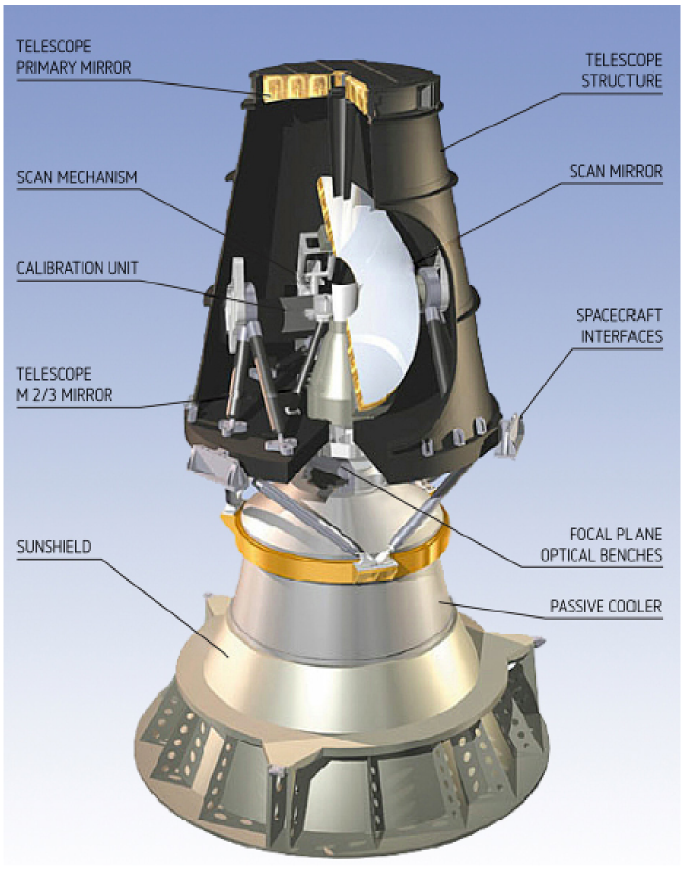


Figure 1.14: Radiometer SEVIRI on MSG satellite.

SEVIRI's primal functions are [18] :

- <u>Monitor convective storms</u>, like thunderstorms; they are usually accompanied by strong winds and heavy rainfalls (or hail) and they can create problems to people and properties on Europe and

Africa. SEVIRI allows to monitor this kind of weather phenomena from the beginning and to follow it with a continue scan of surface: it is fundamental to issue timely warnings. In the image, convective storms are shown like red areas.

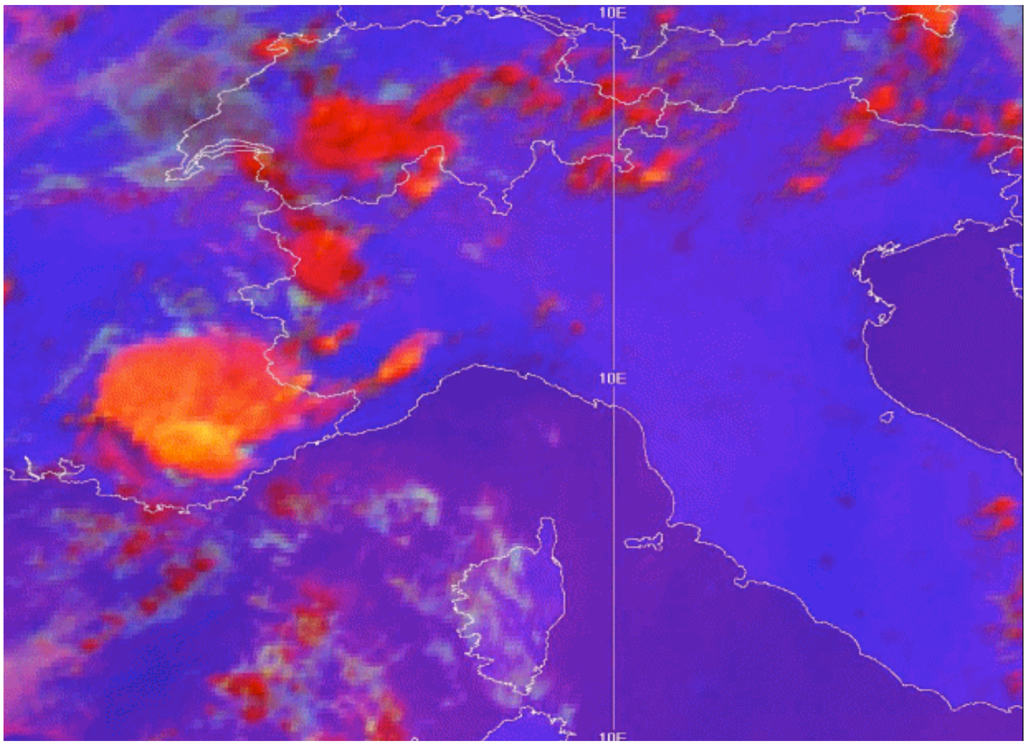


Figure 1.15: satellite's image of convective storm over Italy and France.

- Monitor volcanic ash clouds. This capacity is extremely important to manage air traffics because this kind of clouds are very dangerous for airplane's engines: when a plane fly thought volcanic ash clouds, ash can enter in airplane's engines and can stop them. Data about volcanic ash are sent to London and Toulouse VAAC (Volcanic Ash Advisory Centers) which are responsible to warnings advisor for air traffic. New SEVIRI's algorithms will soon allow to evaluate height, effective radius and others parameter of volcanic ash clouds.

- Monitor fog. With the combination of several techniques, Meteosat allows a continuous monitoring of fog's distribution. This information is still fundamental for air traffic, but for principal road networks and shipping routes too. In the image, the black indicates thick fog; lighter gradients indicate lower intensity of fog. Grey areas are no-fog zones.

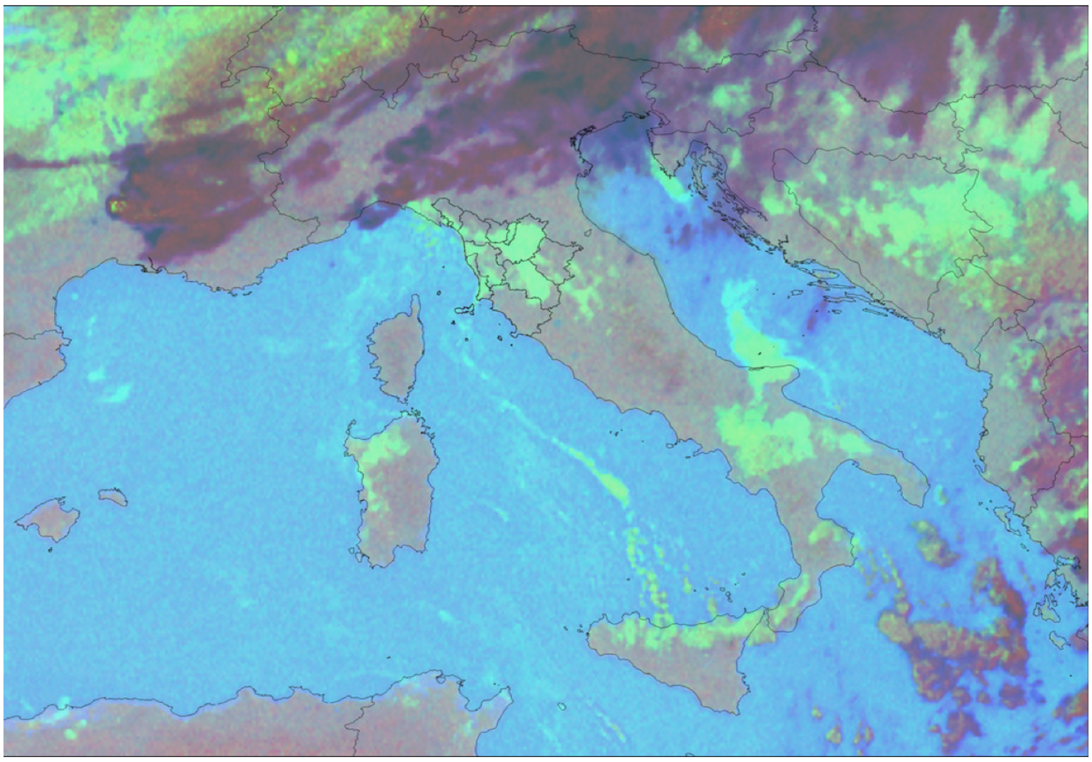


Figure 1.16: Image of fog over Italy.

As stated previously, SEVIRI scans 12 spectral channels on 3 bands [2] [16] :

CHANNEL	**NAME**	**SPECTRAL BAND CHARACTERISTICS (µM)**		
		ΛMIN	**ΛCEN**	**ΛMAX**
1	VIS0.6	0.56	0.635	0.71
2	VIS0.8	0.74	0.81	0.88
3	NIR1.6	1.50	1.64	1.78
4	IR3.9	3.48	3.90	4.36
5	WV6.2	5.35	6.25	7.15
6	WV7.3	6.85	7.35	7.85
7	IR8.7	8.30	8.70	9.1
8	IR9.7	9.38	9.66	9.94
9	IR10.8	9.80	10.80	11.80
10	IR12.0	11.00	12.00	13.00
11	IR13.4	12.40	13.40	14.40
12	HRV	Wide band (between 0.4 and 1.1 µm)		

Table 1.3: 12 SEVIRI's channels with name and wavelengths.

Every channel is accuracy positioned to a specific belt of frequencies; this is because every belt allows to provide information about specific phenomena. Let's analyze every function's band.

Wavelengths of Visible channels are 4, one in high resolution (HRV); they provide information about the quantity of sunlight reflected by Earth and atmosphere. Channels are used to detect clouds, identify their composition and conditions of the surface (snow, water, mountain, flora,....): all this data was used to create images of the planet. These kinds of images are easy to read because human eyes are sensible to the same kind of light (just visible). However, these channels are able to provide data only during the daylight; indeed, during night, these channels are useless. In the images, four visible channels, in order from left to right (channels 1,2,3 and 12); the last image is of the Wide Visible Band:

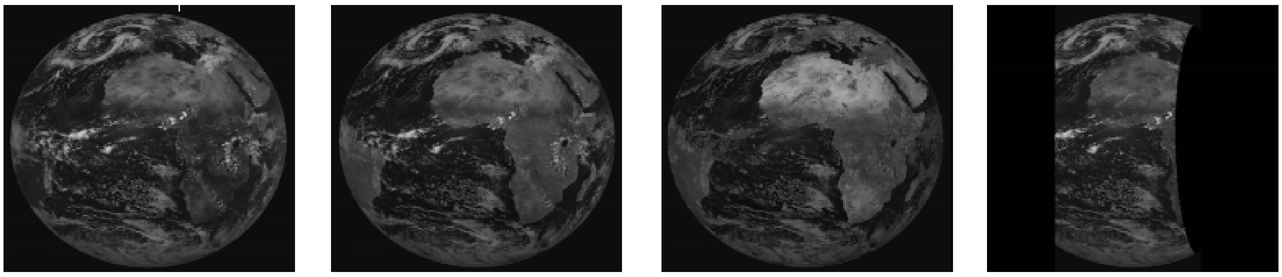


Figure 1.17: Images of the 4 visible channels of SEVIRI.

Infra-Red channel provide information about radiations issued by Earth; data are available 24/24 hours. With these channels, satellites detect temperature of surface and clouds to estimate, for example, the altitude. Some of these IR bands have the same wavelengths of some important gasses in atmosphere, like ozone and carbon dioxide; so, they are used to pick up information about gasses concentration and clouds' composition. All the six channels on IR Band is shown, in order from top left to bottom right (channels 4,7,8,9,10 and 11), in images:

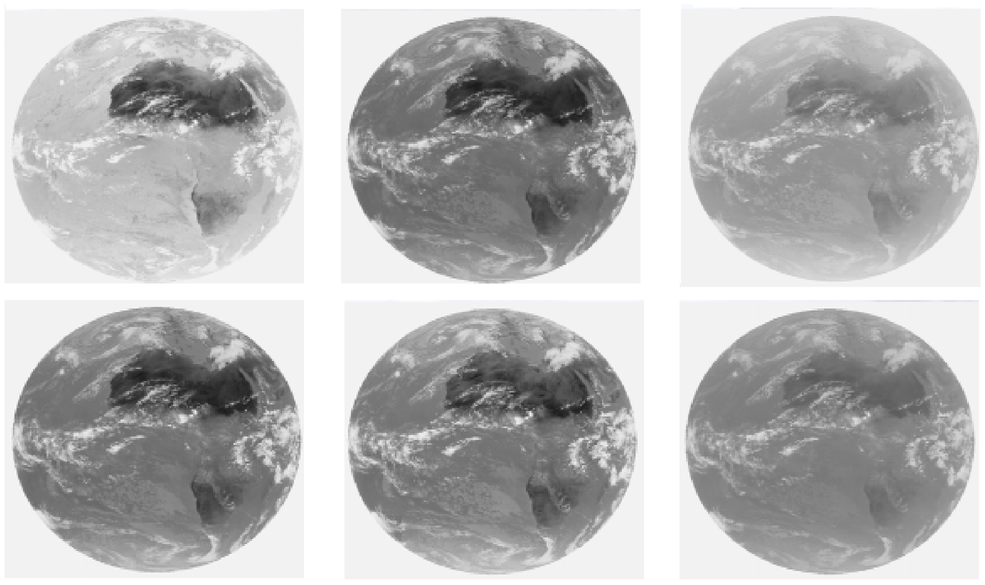


Figure 1.18: Images of the 6 infra-red channels of SEVIRI.

Water vapor channels are positioned in water vapor's principal absorption bands. In this band, SEVIRI gives few information about surface but a lot about distribution of water in atmosphere, so, it can determinate the presence of clouds, the index of wetness and the intensity of winds. In images, two WV channels (channels 5 and 6):

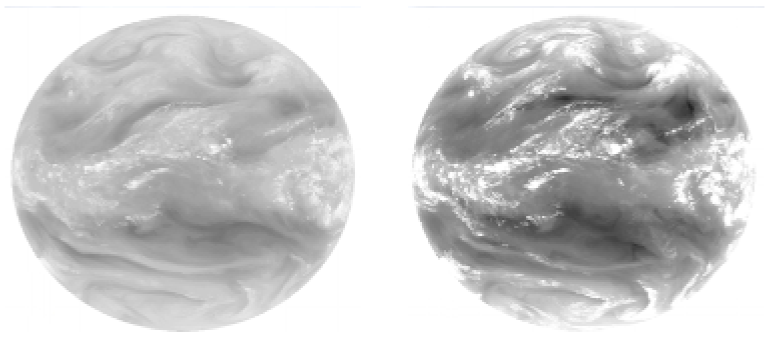


Figure 1.19: Images of the 2 water vapor channels of SEVIRI.

With these data, SEVIRI allows to create Earth images from the space with a great resolution.

In the Visible band, objects in these images have a color proportional to their reflection capacity: light colors for high reflection capacity, dark color for low reflection capacity; so it is easy to indentify clouds (and storms) and characteristics of surface like desert, mountain, ocean,..

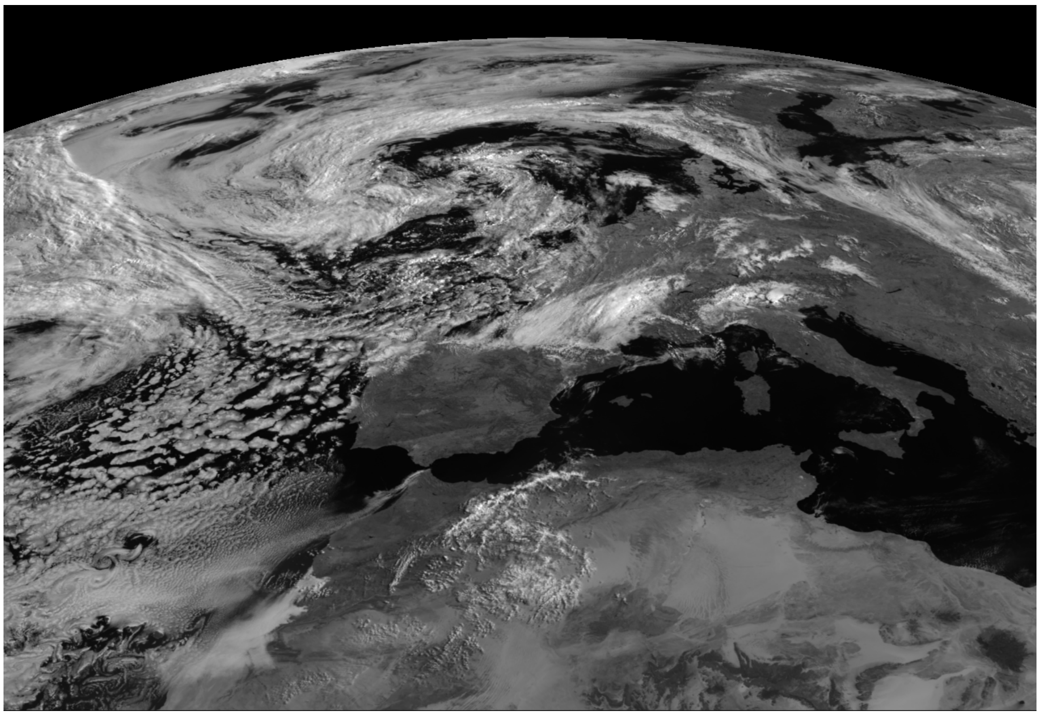


Figure 1.20: Clear satellite image of visible band over Europe.

The images in Infra-Red channels instead give information about temperature: dark colors mean cold objects and light colors mean hot objects. So, clouds should be black and the surface should be white; but often, to have a better reading and a better comparing with visible images, IR images are displayed in inverted colors.

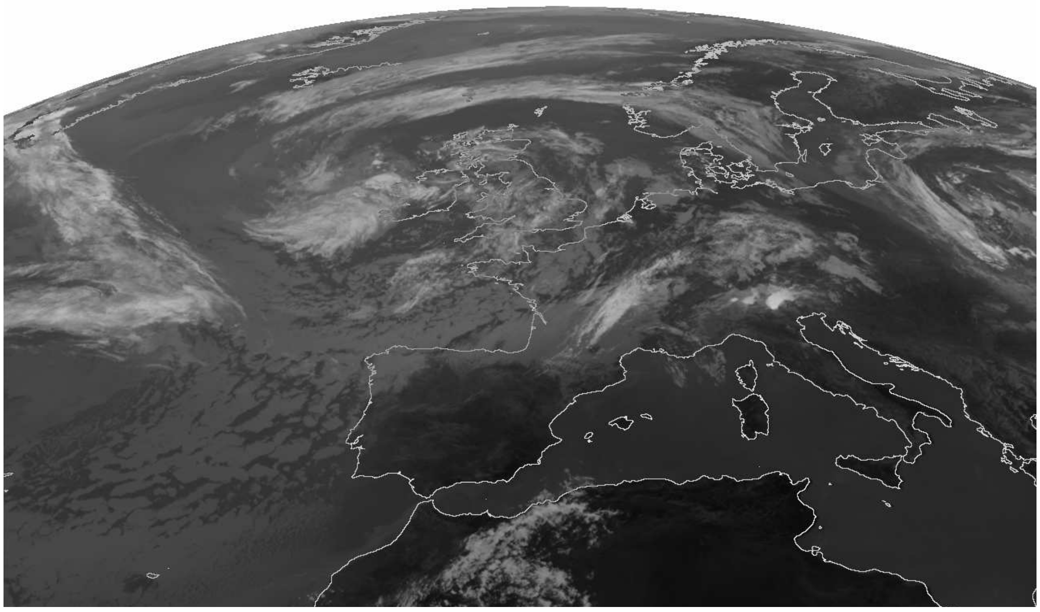


Figure 1.21: Clear satellite image of infra-red band over Europe.

Sometimes, to mix information from several channels, it is possible to create composed images: Red, Green and Blue (RGB) colors are associated to intensity of three different channels. RGB are usually used to underline particular phenomena. In the image, white are used for clouds (cyan's gradients for high ice-clouds and red's gradients for rain-clouds). Surface with flora is green because of high reflection capacity, deserts are ocher and seas are dark because of very low reflection capacity in all channels.

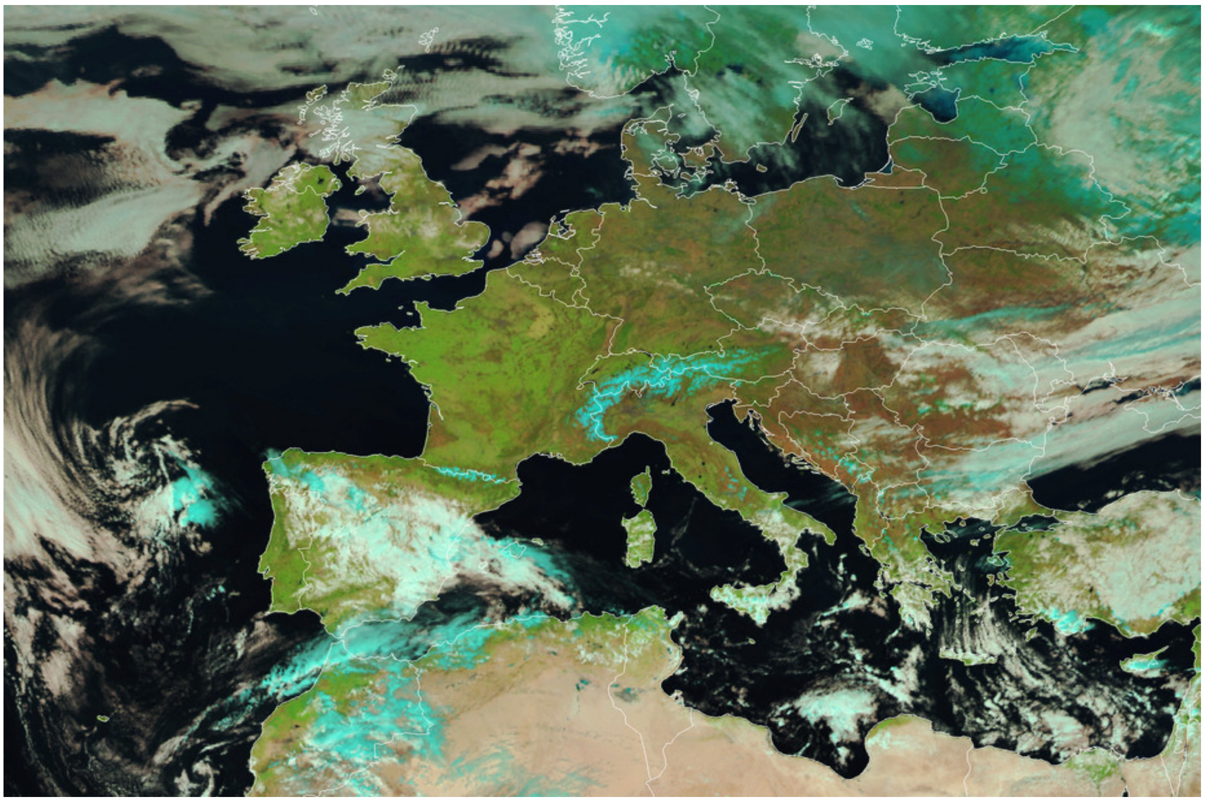


Figure 1.22: RGB image of Europe

1.5.2 Radiometer GERB

GERB [7] (Geostationary Earth Radiation Budget) is a radiometer used to study climate and its evolution; it provides information about radiations on the top of atmosphere within a large IR-band. In this band, the information provided are about clouds and water vapor, forecasting and climate changes. GERB is the first radiometer that gives this kinds of information in a geostationary orbit because previous radiometer was used in LEO. On GEO, it is useful to analyze climatic evolution through clouds and water vapors. Radiometer provides information of full disc Earth every 15 minutes with a spatial sampling of 45x40 km^2 at nadir. It has an height of 25 kg and power consumption of 35 W. Radiations provided by GERB is much variable in function of solar heating. Earth's radiation is detected by a thermo-elastic detector array (Bolometer) of 1x256 pixels, designed to scan full Earth's disc (18° FOV) in North-South direction:

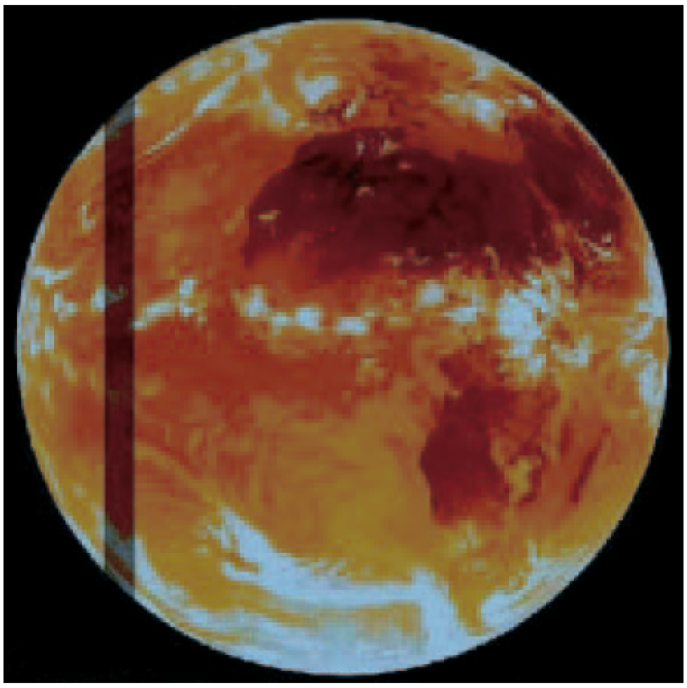


Figure 1.23: Single strip 1x256 pixels detected by GERB.

Every N_S scan is limited to a strip of 40 ms during MSG rotation so, a full coverage of the disc is provided by a continuously FOV from West to East and back again. The full Earth coverage is complete every 15 minutes.

GERB is composed by two meaning parts: the IOU (Instrument Optical Unit) and the IEU (Instrument Electronic Unit).

IOU is very compact 56x35x33 cm^3 and is composed mainly by:

- The telescope: a typical astigmatic system with three mirrors;
- A de-scanning mirror: it is used to hold the image on the target during the rotation of satellites around its axis. It has a continuously rotation of 50 rpm (rotation per minute), in the opposite direction of satellite, during the scan period of 40 ms;
- A wideband detector array. It is a linear thermo-electric array, 1x256 pixels, with its amplifier and processing circuitry, including ASIC and DSP;
- A quartz-filter used for change the wavebands (total and short wave) to provide different kind of data;
- Calibration block, like SEVIRI. It is composed by a black body on optical path, a solar diffuser to monitor the reflectance of the mirrors, a quartz-filter transmittance and a detector adsorption;

- A solid-optical-bench structure.

These principal parts of IOU is shown in the image:

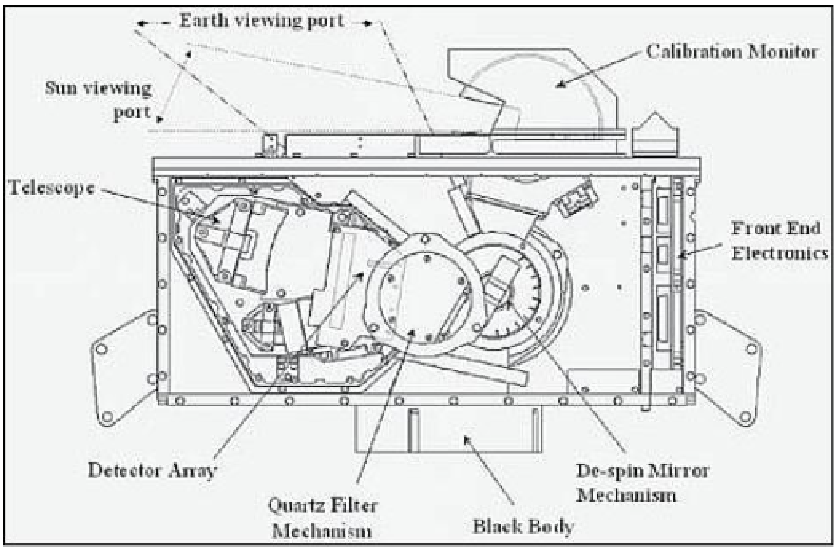


Figure 1.24: Main part of IOU in the radiometer GERB.

Instead, the IEU is a small unit, 22x27x25 cm^3, used to receive detected data and pass them, in a specific format, to the spacecraft's data handling system. Furthermore, IEU gives power to all the instrument's component.

1.6 Meteosat Third Generation (MTG)

MTG (Meteosat Third Generation) is a new series of satellites that will take place MSG series in a few years. MTG satellites are of 3-tonne class.

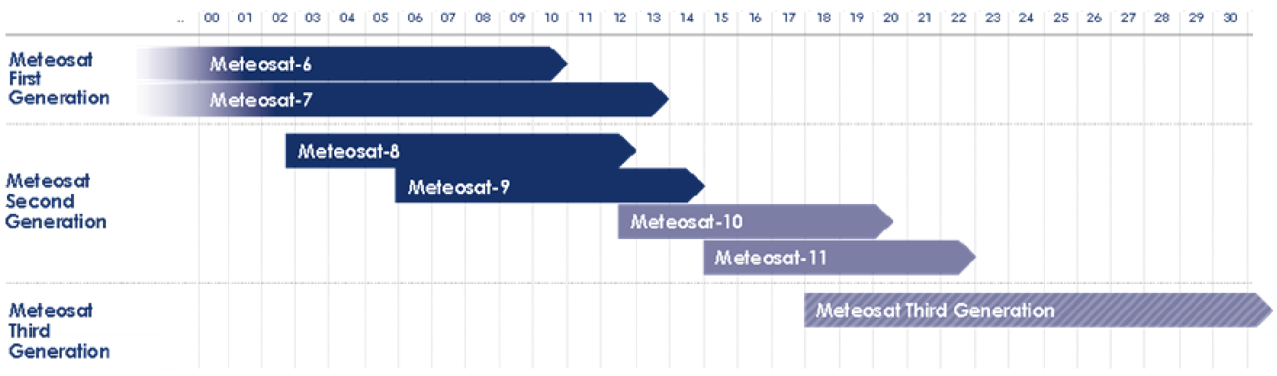


Figure 1.25: timeline of operating satellites and MTG series.

MTG series is composed by 6 satellites with a lifetime of 8.5 years, first one ready for the launch in 2018; this series guarantee information about meteorological phenomena up to 2030. MTG systems start with cooperation between EUMETSAT and ESA; this one has already contributed to the initial research about technologies of these satellites.

In orbit, the six satellites will be divided in parallel positioned couples: all will be stabilized on 3 axis and their instrument will always point towards Earth; first two satellites will be positioned in geostationary orbit, between 10°E and 10°W above the Equator. The scan of full disc will be available every 10 minutes on 16 several channels, 8 in band of solar spectrum with max resolution of 1 Km and 8 in band of IR spectrum with max resolution of 2 Km. Rapid Scan function will analyze Europe every 2.5 minutes with an additional 0.5 Km max resolution channel in visible band. In details, main instruments on MTG satellites are [9] [10] [19] **Errore. L'origine riferimento non è stata trovata.**:

- Full Disk High Spectral resolution Imagery (FDHSI):
  - global scales (Full Disk) over a repeat cycle of 10 minutes;
  - 16 channels at spatial resolution of 1 km (8 solar channels, Visible band) and 2 km (8 thermal channels, IR band);

- High spatial Resolution Fast Imagery (HRFI):
  - local scales (25% of Full Disk, Europe area) over a repeat cycle of 2.5 minutes;
  - 4 channels at high spatial resolution: 0.5 km for 2 solar channels (Visible band), 1.0 km for 2 thermal channels (IR band);
- Infra-Red Sounding (IRS):
  - global scales (Full Disk) over a repeat cycle of 60 minutes;
  - spatial resolution of 4 km, providing hyper-spectral soundings at 0.625 cm^{-1} sampling in two bands:
    - Long-Wave-IR (LWIR: 700 – 1210 cm^{-1} with about 820 spectral samples)
    - Mid-Wave-IR (MWIR: 1600 – 2175 cm^{-1} with about 920 spectral samples)
- Lightning Imagery (LI):
  - global scales (80% of Full Disk) detecting and mapping continuously the optical emission of cloud-to-cloud and cloud-ground discharges;
  - Detection Efficiency (DE) between 90% (night) and 40% (overhead sun);
- UVN Sounding, implemented as GMES Sentinel-4.

This generation of satellites will be divided in 2 principal missions: MTG-I (Imager) and MTG-S (Sounder). Here a timeline of the missions in MTG series:

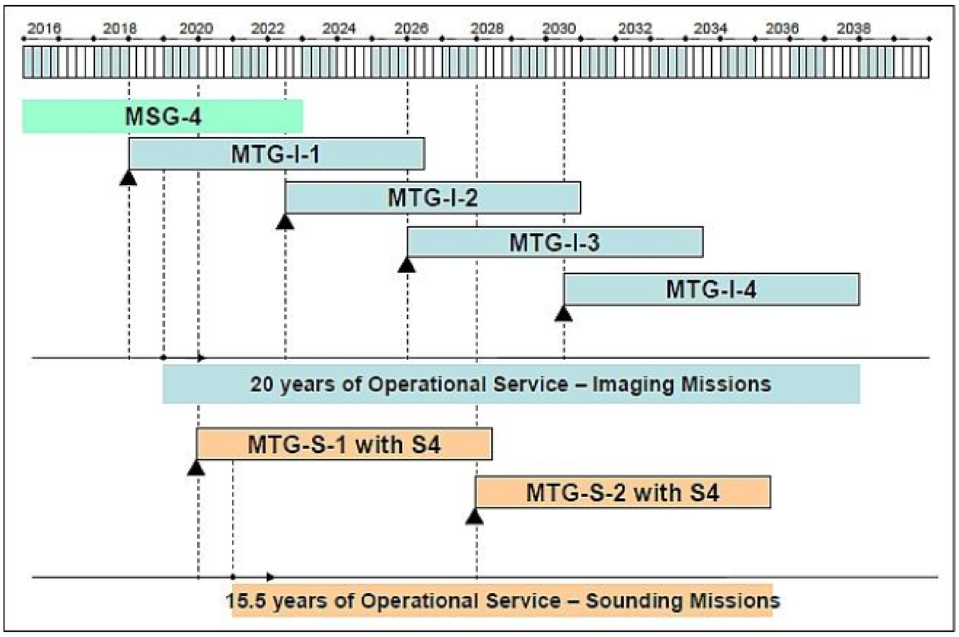


Figure 1.26: Missions of MTG series.

Both of them communicate to the surface with K_U-band antenna; K_U-band is a belt of microwave frequencies, from 12 to 18 GHz, used in space communications (it is an international standard).

1.6.1 MTG-S

MTG-S (Meteosat Third Generation – Sounder) mission is composed by 2 geostationary satellites of MTG series with a sounder on board; they are powered by two deployable solar arrays which store energy in some batteries. The sounder is the main innovation of this new program: for the first time, Meteosat satellites will analyze the atmosphere layer-by-layer, just not with only image weather systems, to provide details about chemical composition.

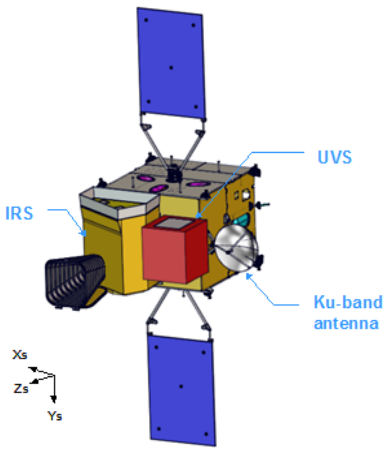


Figure 1.27: MTG-S satellite.

Sounder instrument preliminary parameters are: dimension 1.44x1.30x1.25 km^3, mass of about 438 kg and power consumption of 858 W.

It is composed by more components, most important are shown in the image [16] :

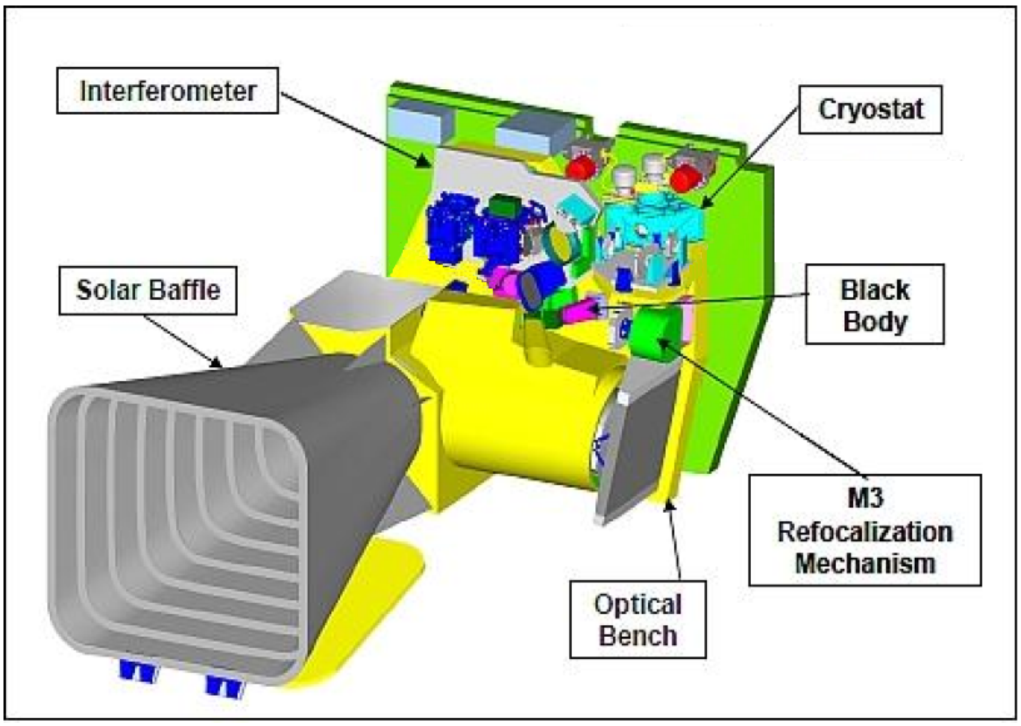


Figure 1.28: Principal parts of the sounder on MTG-S.

Solar baffle deflects the sunlight, the interferometer is on the back side of the sounder to keep images. As SEVIRI, black body and M3 Re-focalization Mechanism are used to take the sounder in initial position. Optical bench is used to adjust lights in optical path; the cryostat is fundamental to hold cold the instrument.

As stated previously, MTG-S will be on board two kinds of interferometers [19] :

- The UVN or UVS (Ultraviolet, Visible and Near-infrared Spectrometer); it is a GMES (Global Monitoring for Environment and Security) Sentinel-4, instrument designed for geostationary chemistry applications. It will take measurements in the ultraviolet band (UV: 305 – 400 nm), the visible band (VIS: 400 – 500 nm) and the near infrared band (NIR: 755 – 775 nm) with a spatial resolution of better than 10 km (~8 km). Its observations are restricted to Earth area coverage, from 30° to 65°N in latitude and from 30°W to 45°E in longitude. The observation repeat cycle period will be shorter than or equal to one hour;

- The IRS (Infra-Red Sounder) with hyper spectral resolution in thermal spectral domain. For the first time, an instrument will be able to provide information on horizontally, vertically and temporally (4-dimensional) structures of the atmosphere. To provide information about structures of humidity (about 2 km resolution with 10% accuracy) and temperature (about 1 km with 0.5° - 1.5° accuracy), it is required to measure water vapor and carbon monoxide absorption bands with extremely high spectral resolution and accuracy. This is the reason why interferometer is based on an imaging Fourier-interferometer with a hyper-spectral resolution of 0.625 cm^{-1} wave-number, taking measurements in two bands, the LWIR band (Long-Wave Infra-Red) and the MWIR band (Mid-Wave Infra-Red) at spatial resolution of 4 Km; the instrument has a global scales over a repeat cycle of 60 minutes.

IRS samples in two bands to provide data about different layer of atmosphere [9]:

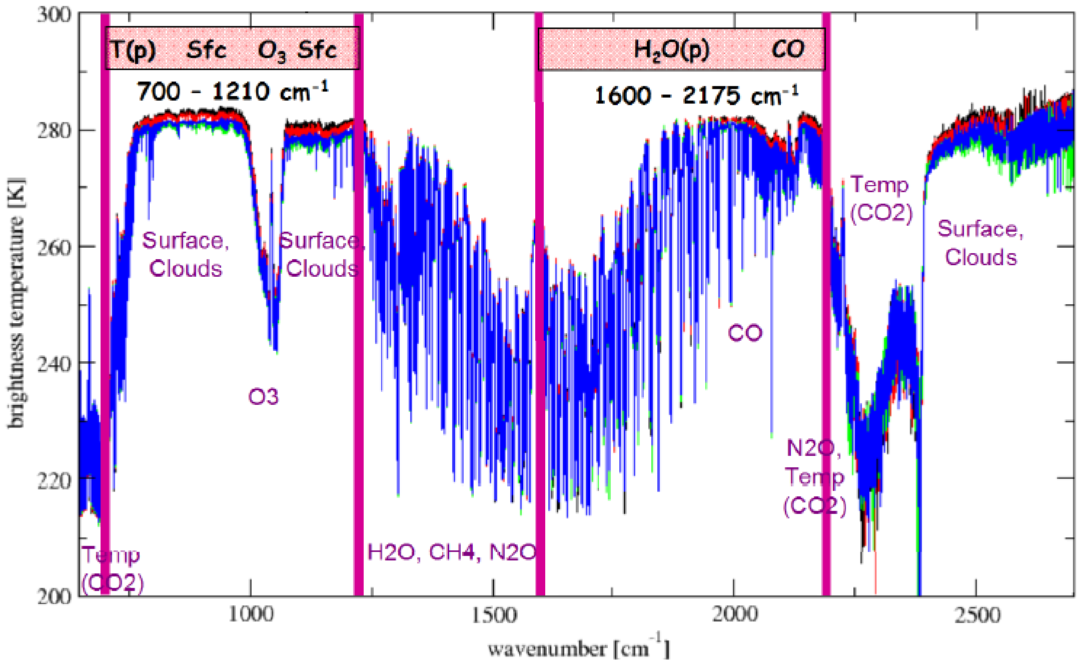


Figure 1.29: Spectrum of the wavelengths detected by IRS.

In the image the two bands are shown. In LWIR band (Long-Wave-IR: 700-1210 cm^{-1}), IRS provides data about ozone (O_3) gas, in MWIR band (Mid-Wave-IR: 1600-2175 cm^{-1}), IRS provides data about water vapor (H_2O) and carbon monoxide (CO) gasses.

1.6.2 MTG-I

MTG-I (Meteosat Third Generation – Imager) mission is composed by the other 4 geostationary satellites of MTG program. These kinds of satellites will be similar to MTG-S satellites but they will have on board an imager instead of a sounder. MTG-I mission is planned to add lightning imagers to geostationary satellites to a specifically measure IC (intra-cloud, cloud to cloud) lightning for better locating areas of intensive convection within extended storm systems. In view of a more unified operational GEO observing system, the MTG LI is intended to provide a real time total lightning detection capability of IC and CG (cloud-to-ground) flashes, with no direct discrimination between the two types. Furthermore, MTG-I satellites provide information for GEOSAR (GEO Search and Rescue) missions and for the DCS (Data Collection System) database [8].

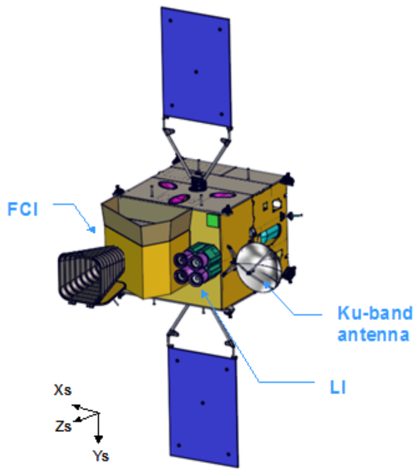


Figure 1.30: MTG-I satellite.

Two principal instruments on board on MTG-I satellites are:

- FCI [16] [19] (Flexible Combined Imager), an instrument that will provide information about high-impact weather such as thunderstorms or fog; it will be the follow-on instrument of SEVIRI. Furthermore, will allow to make an important contribution to air-quality monitoring and, with its high-resolution capability in the thermal-infrared, will provide data for fire detection and climate monitoring. This instrument's mission is divided into another two "easier" missions:
    - the FDHSI (Full Disk High-Spectral-resolution Imagery) mission; this instrument uses 16 channels, with a spatial sampling of 1-2 Km on different band, to scan a full disc over a repeat cycle of 10 minutes;
    - the HRFI (High spatial Resolution Fast-refresh Imagery) mission; it uses instead 4 channels, with a spatial sampling of 0.5-1.0 km on different band, to scan local area over a repeat cycle of 2.5-5 minutes;

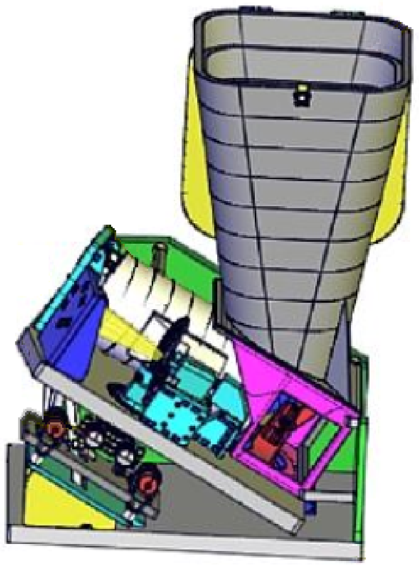
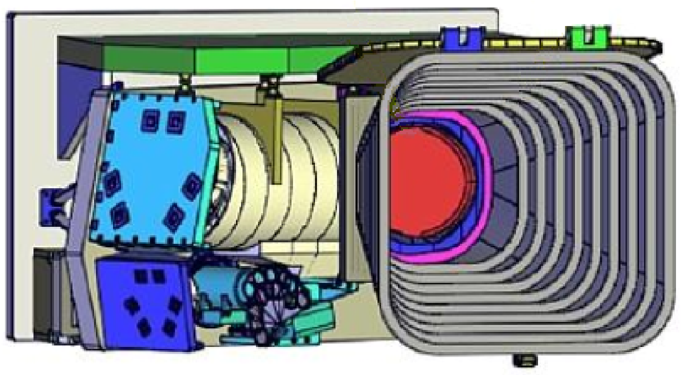



Figure 1.31: views of FCI.

Channels of FCI are [15] :

CHANNEL	CENTRE WAVELENGTH Λ_0 (µM)	SPECTRAL WIDTH $\Delta\Lambda_0$ (µM)	SPATIAL SAMPLING DISTANCE (DSS)
VIS 0.4	0.444	0.060	1.0 km
VIS 0.5	0.510	0.040	1.0 km
VIS 0.6	0.640	0.050	1.0 km; 0.5 km*
VIS 0.8	0.865	0.040	1.0 km
NIR 0.9	0.914	0.020	1.0 km
NIR 1.3	1.380	0.030	1.0 km
NIR 1.6	1.610	0.050	1.0 km
NIR 2.2	2.250	0.050	1.0 km; 0.5 km*
IR 3.8 (TIR)	3.800	0.400	2.0 km; 1.0 km*
WV 6.3	6.300	1.000	2.0 km
WV 7.3	7.350	0.500	2.0 km
IR 8.7 (TIR)	8.700	0.400	2.0 km
IR 9.7 (O_3)	9.660	0.300	2.0 km
IR 10.5 (TIR)	10.500	0.700	2.0 km; 1.0 km*
IR 12.3 (TIR)	12.300	0.500	2.0 km
IR 13.3 (CO_2)	13.300	0.600	2.0 km

Table 1.4: 16 channels of FCI. The red channels are used by FDHSI and HRFI. The second value in DSS column (*) is the value of HRFI instruments. TIR as Thermal Infra Red.

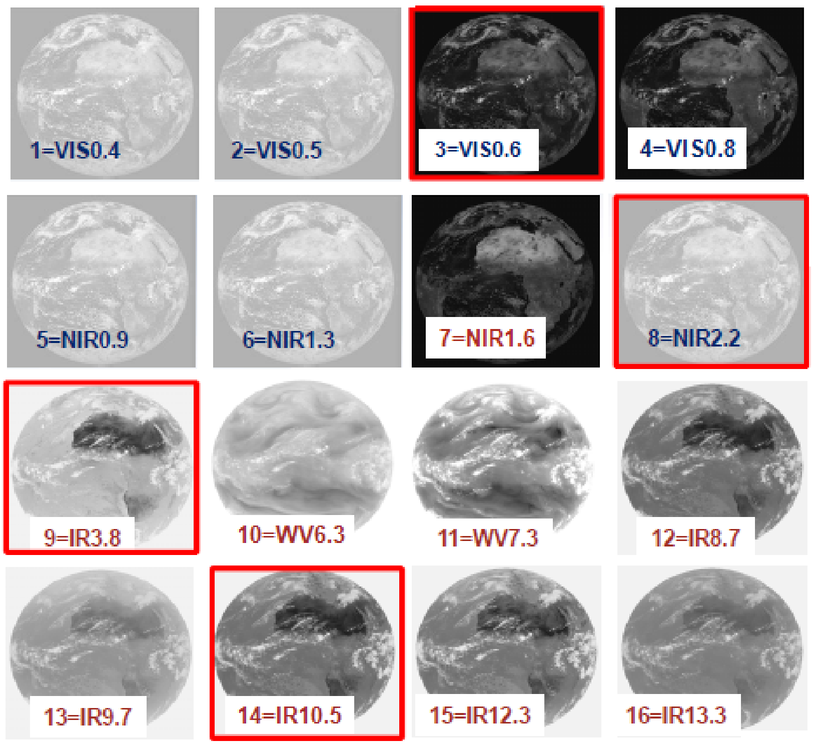


Figure 1.32: Earth image of every FCI channel.

- LI (Lightning Imager), an imaging detection instrument with high resolution. Its most important objective is to add complimentary information to the existing ground lightning detection systems, with the benefit to provide a wider coverage, including poorly populated areas, and a reference to correlate different ground systems and networks. Furthermore, it allows to provide data about atmospheric chemistry and climate monitoring.

Chapter 2

Lightning Imager

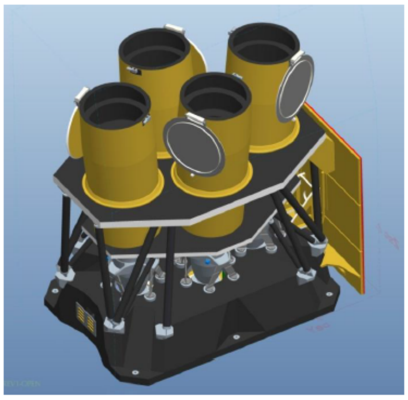


Figure 2.1: Lightning Imager design.

LI [10] [12] (Lightning Imager) for Meteosat Third Generation is an on board instrument used to provide information to the location and detection of cloud-to-ground and cloud-to-cloud lightning over the full Earth disk from geostationary orbit in day and night conditions. LI's data from geostationary orbit are regarded as a complementary source of lightning data provided by the ground-based Lightning Location Systems (LLSs); Global LLS networks limit their detection capability mainly on CG flashes. But local LLS networks are limited on industrialized countries; so lightning data over oceans or Africa are provided with less quality than industrialized countries and data are no homogeneous over all the territory because of network geometry and territorial difference. Instead, LI will provide data over the hemisphere with the same quality: this will be a great vantage because data about climate changes, lightning activities and NO_X gas (important for ozone hole and acid rain) will be provided homogeneously. With this new instrument, in cooperation with the two NOAA GLMs (Geostationary Lightning Mappers on board of GOES-R satellites), the planet will be fully covered [13] .

Primary objectives of LI [24]:

- Provide the ability to detect the very first cloud flashes, thus giving valuable additional lead-time for precise warnings of lightning strikes;
- Provide lightning data continuously for any location in the FOV (field of view), apply the algorithm and send important information about dangerous storm for the air traffic and people security;
- Provide data about convection storms for nowcasting and forecasting;
- Improve rainfall measurements when combined with other satellite measurement;
- Measure NO_X and other gasses to monitor atmospheric chemistry conformation and their effect on global and local climate changes (effects of these gasses are a matter of great uncertainty at this time, and long-term observations of their sources will prove valuable as the subject develops);
- Create a database of lightning events for weather services and security operations
- Can be used for validating NWP models (Numerical Weather Prediction is a model used to short-medium weather forecasting; with a lot of weather's data, powerful computers elaborate nowcasting to make prevision of future possible events);

As stated previously, LI must have strict requirements to observe continuously and simultaneously the full visible disk with high temporal resolution.

Requirements necessary to a precision detect IC and CG lightning are provided by ESA; they are [10][11][19]:

PARAMETER	REQUIREMENT
FOV	84% of visible Earth disk (including all Eumetsat member states, instantaneous view) or 16° shifted northward
Spatial sampling	< 10 Km at 45° Latitude
Data rate	< 30 Mbps
Dynamic range of Earth background (Lbkg)	$0 \div 296.5$ W/(m^2 * µm * sr) (night)
Optical pulse dynamic range (LLp)	$6.7 \div 670$ mW/(m^2 * sr)
Sensitivity pulses	4 µJ/(m^2 * sr) for 100 km^2
Detection Efficiency (DE)	• 70% as average over the FOV • 90% for latitude 45 deg • > 40% over EUMETSAT member states
False Alarm Rate	2.5 (false) flashes every second
LI Optical Head Envelope	718 x 1200 x 1456 mm^3
Optical pulse spectral range	777.4 ± 0.17 nm with 1.0-1.5 nm spectral pass-band filter
Minimum optical pulse duration	0.6 ms
Maximum number of optical pulses (in FOV)	• 25 in 1 ms • 800 in 1 s
Background image's cycle	60 seconds
LI OH overall dimension	715 x 1100 x 1200 mm
LI Main Electronics overall dimension	300 x 240 x 160 mm

Table 2.1: Requirements of LI.

LI is composed by one Optical Head (LI OH) and one Electronic control unit (LI Main Electronics); Mass of OH must not exceed 70 kg, 93 kg for OH and Electronic unit, and LI's max-power consumption is 320 W. OH consists in 4 identical Optical Channel (OC); 4 because all the OH must satisfy technique requirements and every Channel scans a specific cardinal zone. Furthermore, to distinguish true lightning from false one (generated by random noise, sun glints or micro-vibrations), every channel uses a multi-dimensional filter: this filter works,

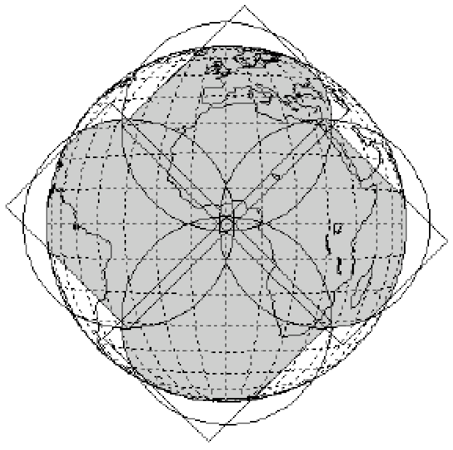


Figure 2.2: 4-channels Earth subdivision.

at the same time, on spectral, spatial and temporal domains.

Spectral filtering uses a very narrowband filter centered on the bright lightning O_2 triple line (777.4 nm ± 0.17 nm).

Spatial filtering is achieved with the valuation of lightning's size: if it is smaller than a typical lightning pulse, it is discarded.

Temporal filtering takes advantage of continues sampling of Earth disk every 1 ms.

A multi-Optical Channel architecture is useful to optimize spectral filtering (narrow band filtering): with a single optical channel, the quantity and the quality of provided data are not sufficient (telescope cannot provide information as required) and the difficulty of on board architecture are considered unacceptable for power consumption, number of operations and elaboration time.

Single optical channel means a very large detector array, about 5 Megapixel, and an high-efficiency electronic system which could have to elaborate about 5 Gigapixel per second in near real time. Furthermore, with a single optical channel, the detector is reduced into a Galileian telescope. In Galileian telescope, an high angle of incidence of blue spectral is not sustained by spectral filter: it will not be uniform.

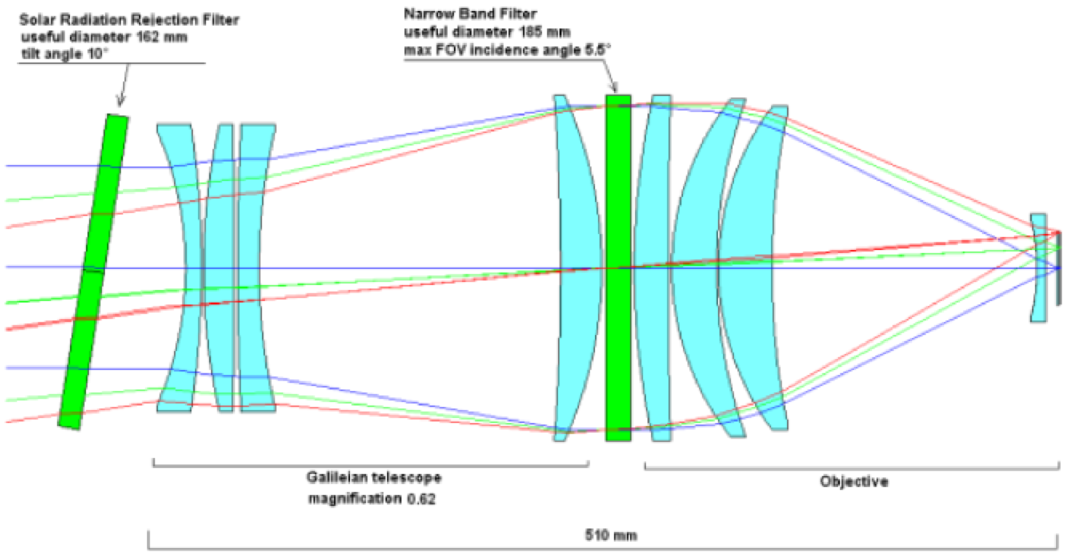


Figure 2.3: Blue angle of incidence in optical path on Galileian telescope.

In a multi-channel LI, Galileian telescope is no more required. Furthermore, more Optical Channels improves (narrowband) filtering performances.

For these reasons, LI has four Optical Channels.

LI has to make more operations to detect flashes; they are divided in 3 principal kinds of processes [10] :

1. In-orbit data acquisition;
2. Level-1 processing;
3. Level-2 processing.

These LI processes are detailed in the following functions, step by step:

- In- orbit acquisition:
  - Earth image acquisition for continuous monitoring of the lightning's presence in the FOV;
  - Calculation of pixel-by-pixel adaptive background to cope with non-uniformities and low terms variations of the image and to reject at the same time noise effects and spurious events;
  - Removal of background level from the overall pixel signal to obtain the illumination's level of every pixel;
  - Use of adaptive thresholds: lower thresholds in low noise dark areas, higher thresholds only for highly illuminated areas;

- Level-1 processing:
  - Pixels for which the difference between the pixel value and the estimated background signal exceeds the threshold are kept as Detected Transients (DT);
  - On-board DTs filtering to reduce number of noise and False Transients (FT)

- Level-2 processing
  - On-board add information for the ground processing with a dedicated processing electronics (Geolocation, time tagging and radiance-energy association);
  - Conversion of the DT video data into digital information (row and column over a digital grid) and conversation in a level compatible with the platform downlink data rate constraints (<30 Mbps).

Main output data of this algorithm are groups of/single lightning events with their location, time and radiance.

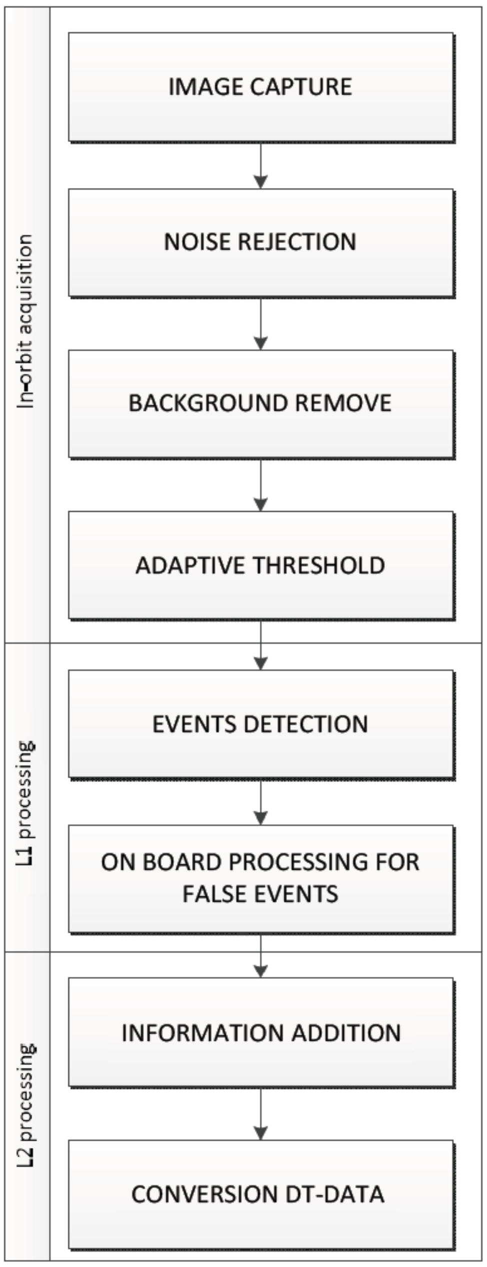


Figure 2.4: Flux of on-board operation for events detection and data tramission.

2.1 Image capture

As stated previously, Lightning Imager [10] is a single telescope with 4 different channels; they have a focal length of around 600 mm and a pupil size of 110 mm. They use an APS (Active Pixel Sensor) in C-MOS technology with a resolution of 1000x1170 pixels and a time refresh of 1 ms. FOV of this instrument must allow to provide lightning information mainly on Africa (a contribute for global lightning) and on Europe and South-America with lower frequency (ocean too); it is because lightning events are more frequent in the tropical areas like Africa and South-America:

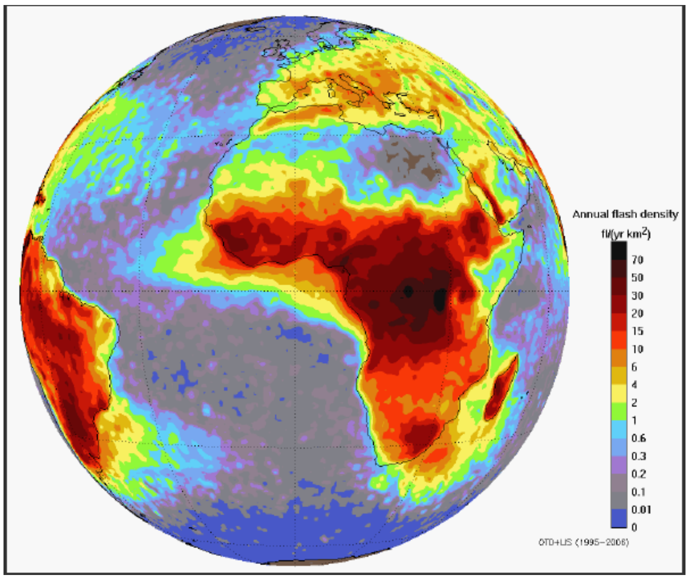


Figure 2.5: Annual flash density in MTG FOV.

LI's FOV is shown in the image:

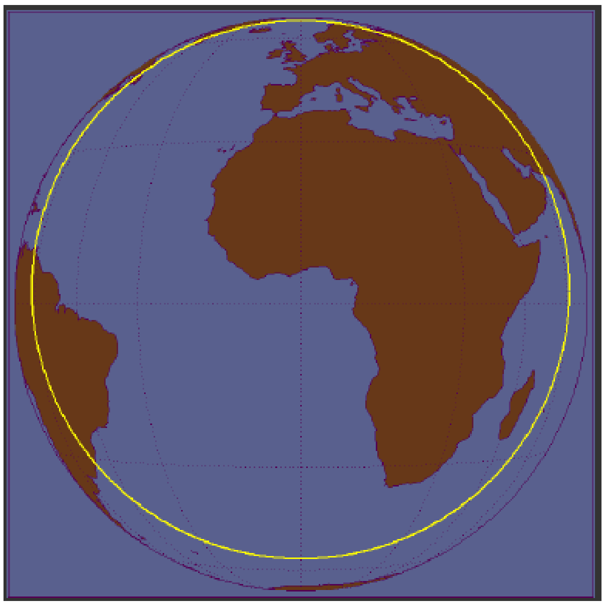


Figure 2.6: FOV of Lightning Imager.

With a single FOV, LI covers all the zones; but more satellites are necessary to cover a greater part of surface. With the collaboration of USA, more than half of the Earth is covered with high overlap between MTG and GOES-R (East and West GOES-R) FOVs:

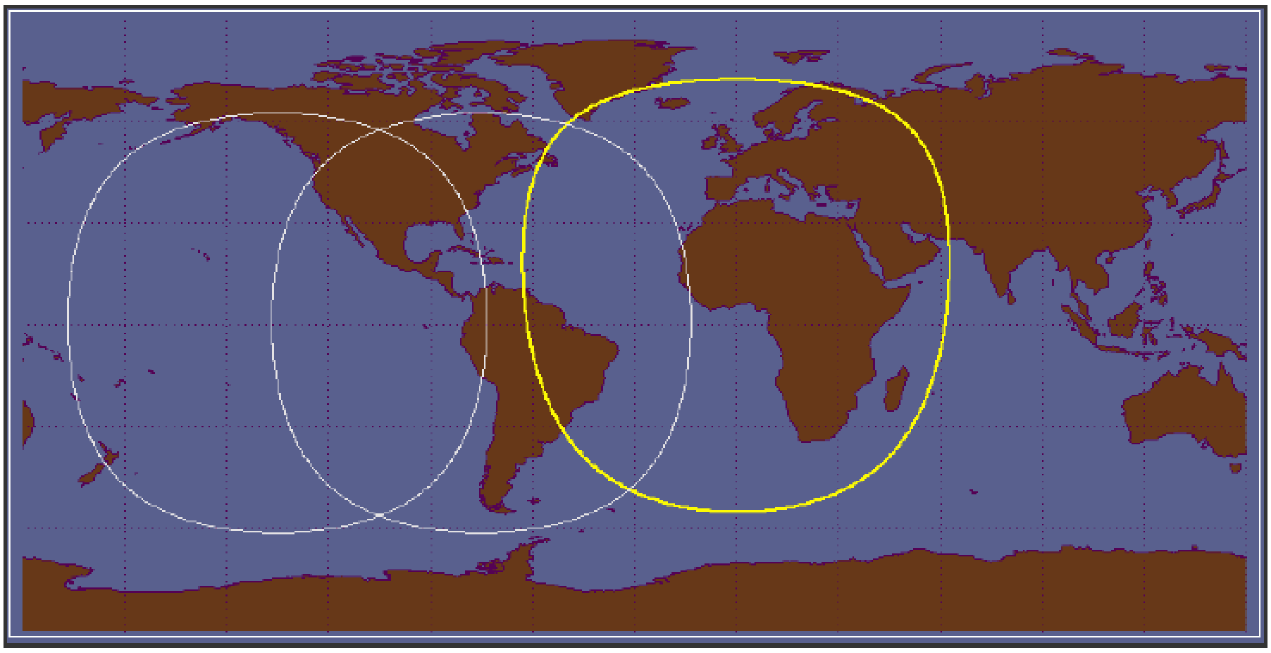


Figure 2.7: Earth coverage by MTG and GOES-R FOVs.

This kind of FOV has a high distortion: on zenith, the view is perfect (over Equator) but, towards Earth's edges, the view is highly distorted and, on large nadir angles, pixels are quiet indistinguishable: this is a big problem because pixels indistinguishable means indistinguishable events [10] :

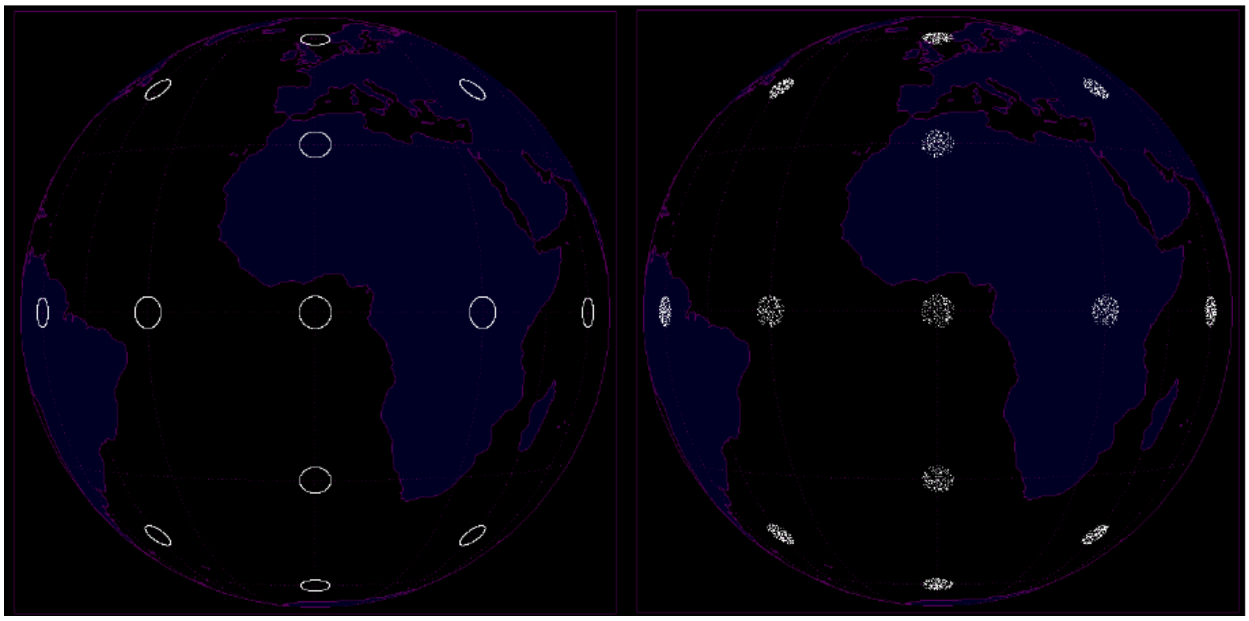


Figure 2.8: Deformations of detected images.

Because of this problem, LI must have some specific algorithms and instruments to correct this problem.

2.2 Noise rejection

Principal problems of image capture is the noise: an high level of noise make unreadable the image provided by the telescope. In the following image, different FER (False Event Rate) are shown for the same scanned zone: in a time of 500 ms, a FER of 40000 (so 20000 false events in 500 ms) does not allow to provide accuracy data and some filter are necessary to detect true events. With a FER of 400000 the image is completely unreadable [11] .

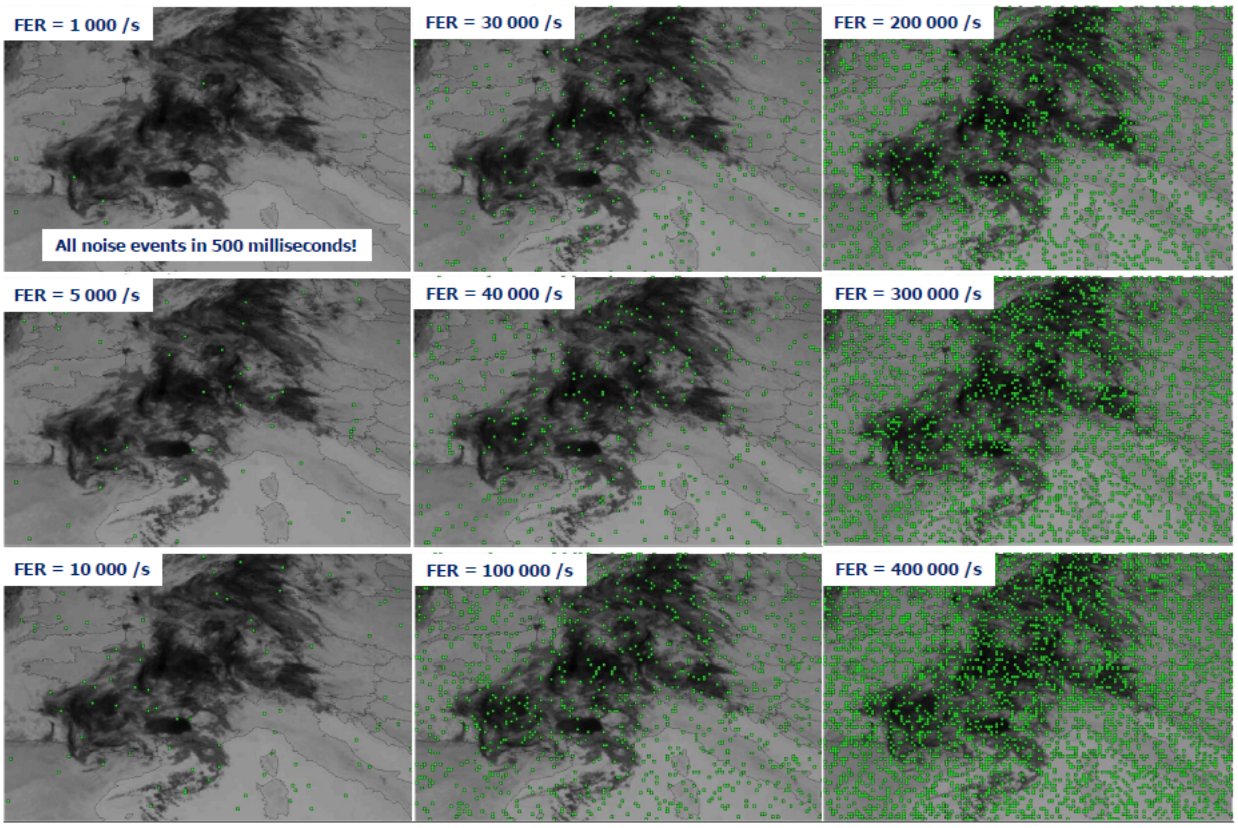


Figure 2.9: Examples of False Error Rate over Europe.

There are two kinds of noise: internal noise and external noise. Internal noise is generated by the component in the satellite's instruments like telescope and electronic components. On the other hand, external noise is due to external conditions of atmosphere and light.

Main causes of internal noise are [15] :

- Electronic noise: this is a noise typical of electronic components on board, specially from power source and amplifier stages.
- Thermo-mechanical noise: this noise comes from movement and rotation of mechanical mobile part and from thermal source in telescope.

- Stray light noise: this noise is made by lights which do not follow the correct path and, for this reason, creates interference with useless light.

On the other hand, the main causes of external noise are:

- Cloud radiation: a part of solar radiation is reflected by clouds. In images, this radiation is interpreted as light and it can be added to the Detected Events (DE).
- Sun glint and solar eclipse: sun glint is an optical phenomenon where the sun light is reflected by oceans with the same angle of satellite's view. Sun glint and solar eclipse are considered noise because both of them change the radiance level of an area: high level of radiance can be exchanged to lightning events.

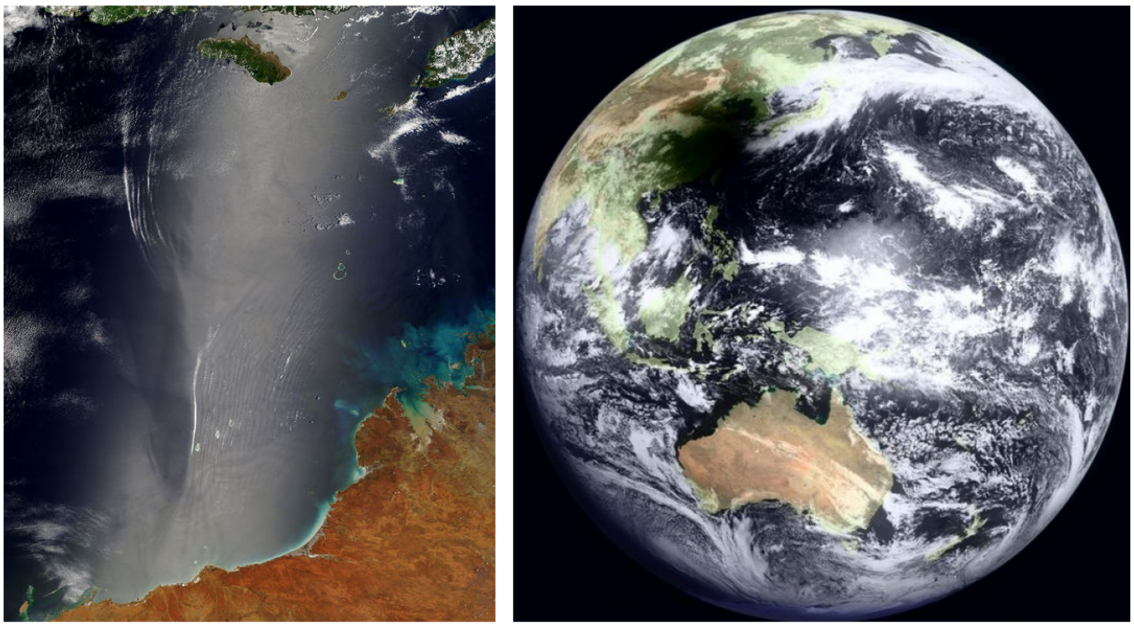


Figure 2.10: Examples of sun glint and solar eclipse detected by satellite.

- Particles flux: especially in Van Allen belt, it can modify the IFOV with the reflection of sunlight in several bands or with a magnetic contribute which deflect light.
- Jitter: it is a typical video phenomenal which occurs when the horizontal lines of two consecutive frames are not synchronized because of movement or micro-vibrations. This noise is particularly problematic in time subtracting of two frame to identify a lightning events: a jitter can create a False Detect [11] .

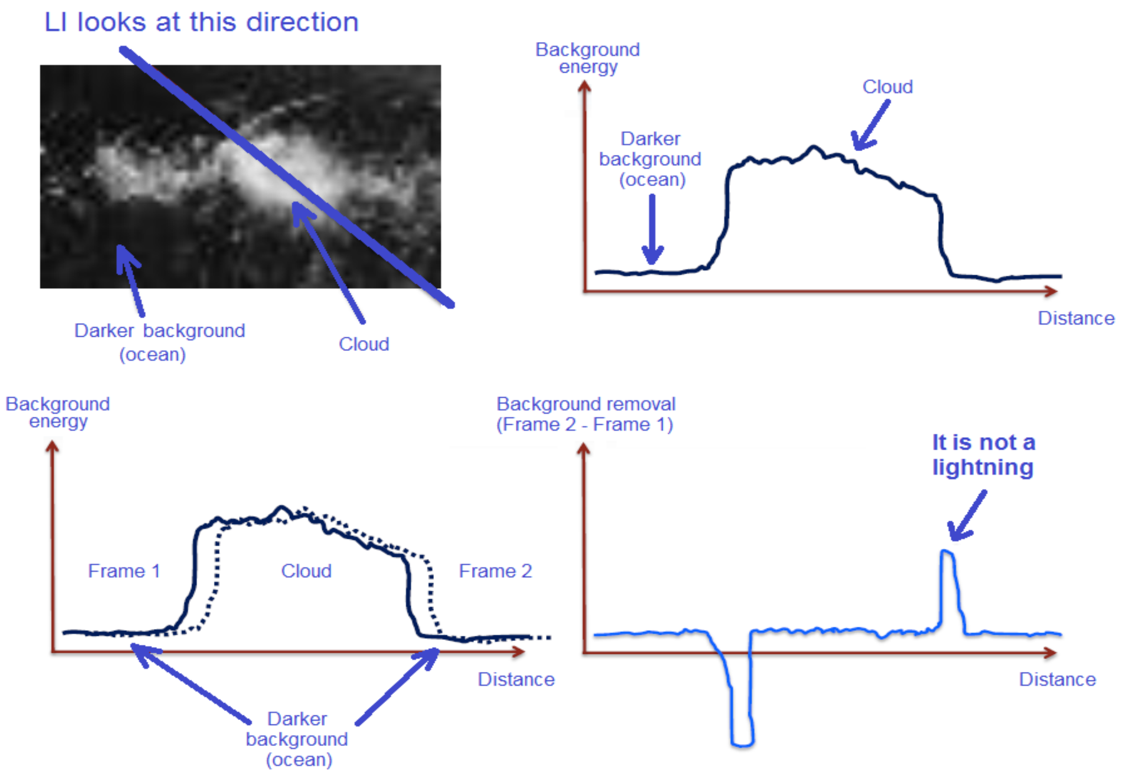

LI looks at this direction
Darker background (ocean)
Cloud
Background energy
Darker background (ocean)
Cloud
Distance
Background energy
Frame 1
Cloud
Frame 2
Distance
Darker background (ocean)
Background removal (Frame 2 - Frame 1)
It is not a lightning
Distance

Figure 2.11: Jitter example by frame subtraction.

2.3 Background removal and adaptive threshold

Background is defined as all image's object considered static in a short time frame. The contrast between background and lightning radiance determinate the capability of detect event: this contrast set SNR (Signal to Noise Ratio) level to detect lightning events. When SNR is high, for example during the night, dark background allows to detect lightning events and clouds.

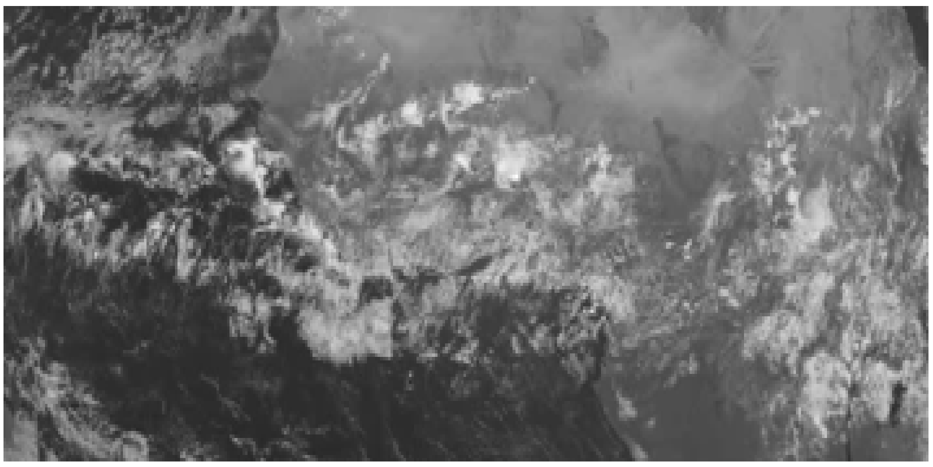


Figure 2.12: Example of FOV with good SNR.

But, for a better readability, background removal is not sufficient. SNR value is also imposed by light condition like day and night or solar eclipse. It is clear in the image: every two hours, the in-orbit view changes and it is not easy to provide any data or image.

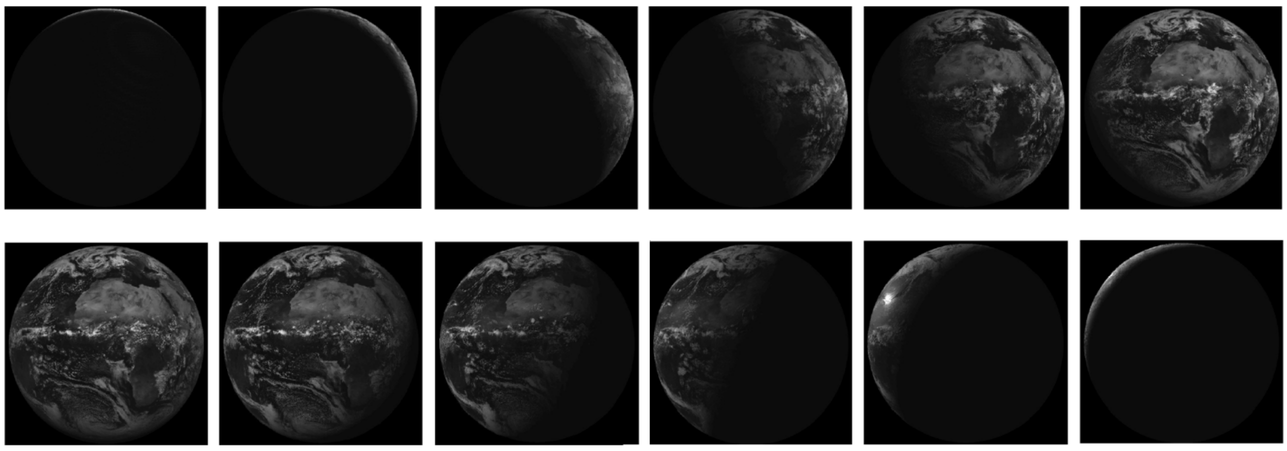


Figure 2.13: Surface bright condition every 2 hours.

During the night, low level of light cannot allow to distinguish anything on the ground. In these critical cases is necessary an adaptive threshold to detect lightning events and provide every kind of information [14] .

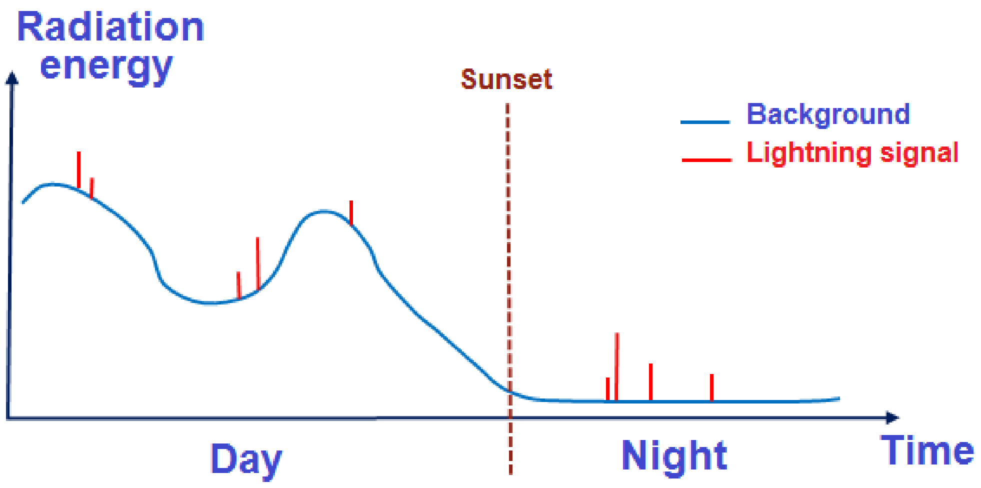


Figure 2.14: Background trend in a day. Red spikes are lightning events.

An adaptive threshold is the best solution of background in-homogeneity. In fact, with just a single low threshold level, there could be a large numbers of lightning and false events and, with a single high threshold level, there could be a few lightning events and some others, with low radiance level, could be discarded. Threshold must not be too high because a lot of lightning events could not be detected [9]. In Figure 2.15, it is possible to see what happen with high threshold: some events is not detected because their radiance level is lower than threshold level.

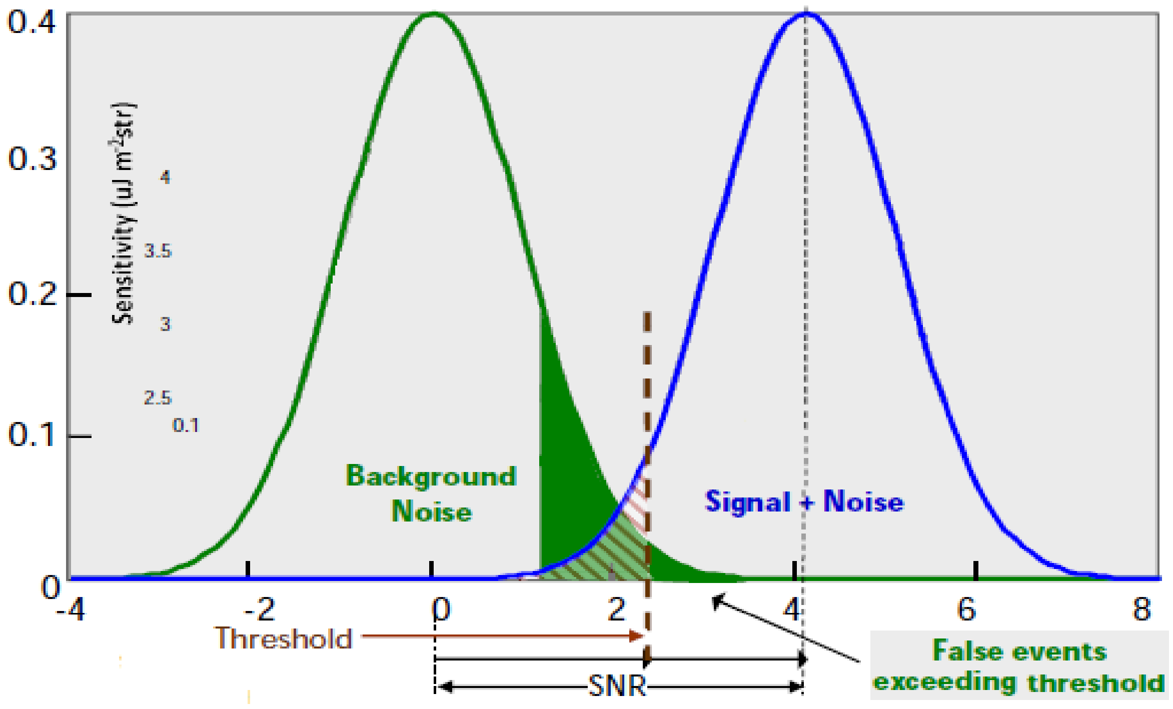


Figure 2.15: Distributions of background and Background + signal in SNR scale. Also threshold used to optimize SNR value.

For this reason, a trade-off between detection sensitivity and false event rates are necessary.

All problems are summarized in the Figure 2.16. In the daylight, a high level of background radiance fixes a high value of threshold to detect lightning events. But only a single level of threshold is not sufficient because of the light during the day is not homogeneous. So, in the daylight, there are more than one threshold (Threshold 1 and Threshold 2); in some cases, lightning on top of bright background is not recognized by its bright radiance but by its transient short pulse character: in these cases a temporal filtering is used. A critical time is on sunset because radiance level of background decrees rapidly and events are hardly detectable: during sunset, threshold must change quickly. During the night instead, just one threshold is necessary to detect lightning events. [14]

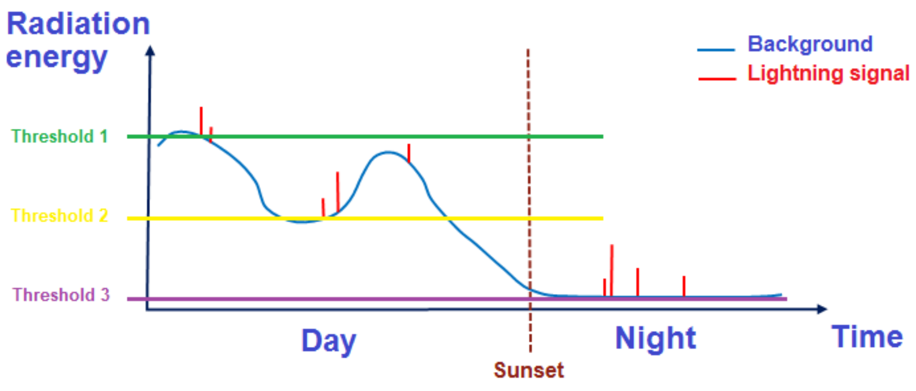


Figure 2.16 : adaptive thresholds.

This specific criteria is not easy to apply because background removal and adaptive threshold are different for every pixels or zones in the images. As shown in previously images, on sunset, a part of surface is in daylight but a part is dark because of night and the LI must differentiate the two zones to subtract different background and apply a different level of threshold.

All these operations allow to detect "easily" an event.

2.4 Events detection and on-board processing for FEs

Without the previous operations, the events detection is quite impossible. This image shows a single lightning event detected over Ethiopia: without a deep zoom is impossible to distinguish the event [10].

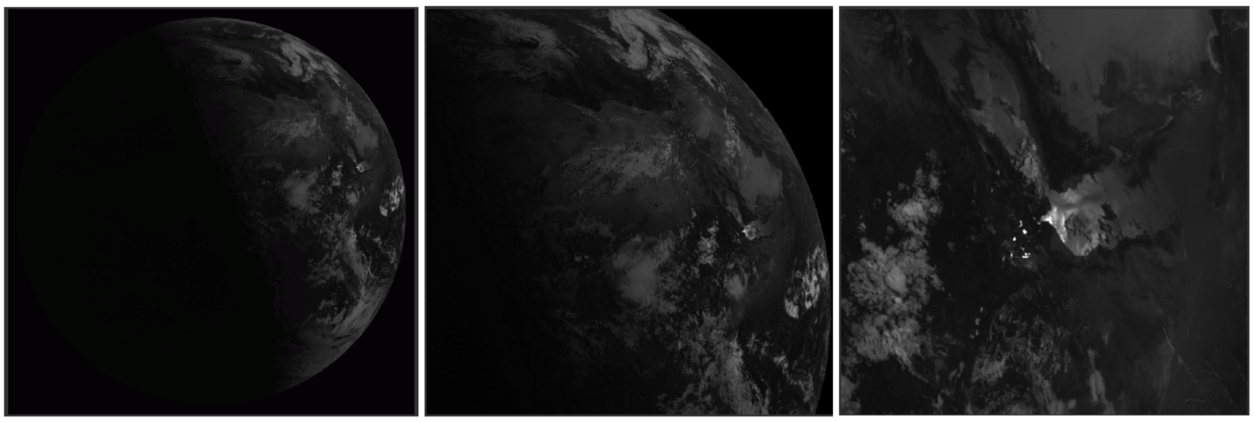


Figure 2.17: Event detected over Ethiopia.

For this reason, every LI's image goes through the algorithm of detection for background removal and adaptive threshold. In this phase, filtering is carried out. There are 3 kinds of filter:

- Spatial filter: compare the radiation of detected event in IFOV (Instant FOV) with the typical lightning event radiation's view;

- Spectral filter: a special narrow band filter centered on a wavelength of 777,4 nm make a frequency filtering. 777,4 nm is the wavelength of the first ionized energy of Oxygen:

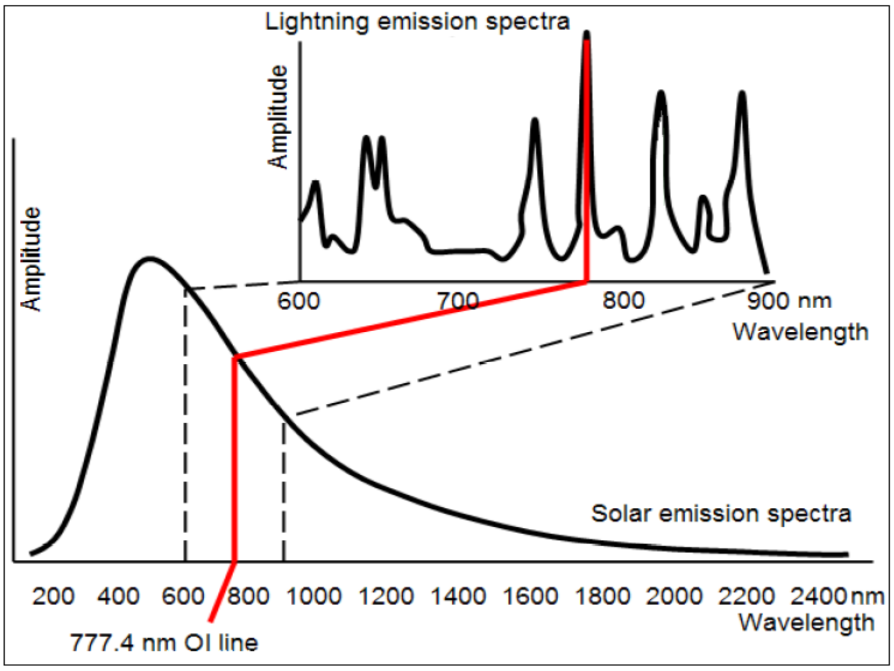


Figure 2.18: Wavelength of first ionized energy of Oxygen

- Temporal filter: temporal filtering is simply a time compare between pulse and background which is considered constant in the time scale of seconds.

If these 3 filtering are not sufficient, a frame-to-frame background subtracting is used. This is a technique used when the radiance ratio between lightning and background is still under the threshold.

With these 3 levels of filtering, almost all the false events are eliminated. False events are not related to a real lightning but are due to noise or distortions or artificial light on the surface: they must be eliminated for a correct now/forecasting and data providing. Figure 2.19 shows the difference between filtered and not-filtered scans [15] :

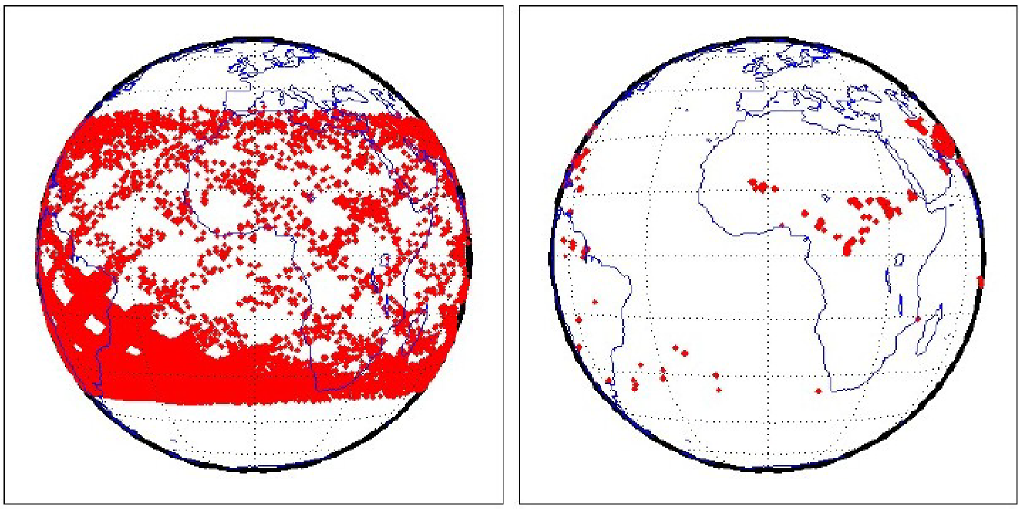


Figure 2.19: Difference between no-filtered and filtered scans.

Obviously, this optimal condition of lightning detection is possible only if all the previous technique were used. Images after background removal, adaptive threshold, filtering and event detection is similar to this one:

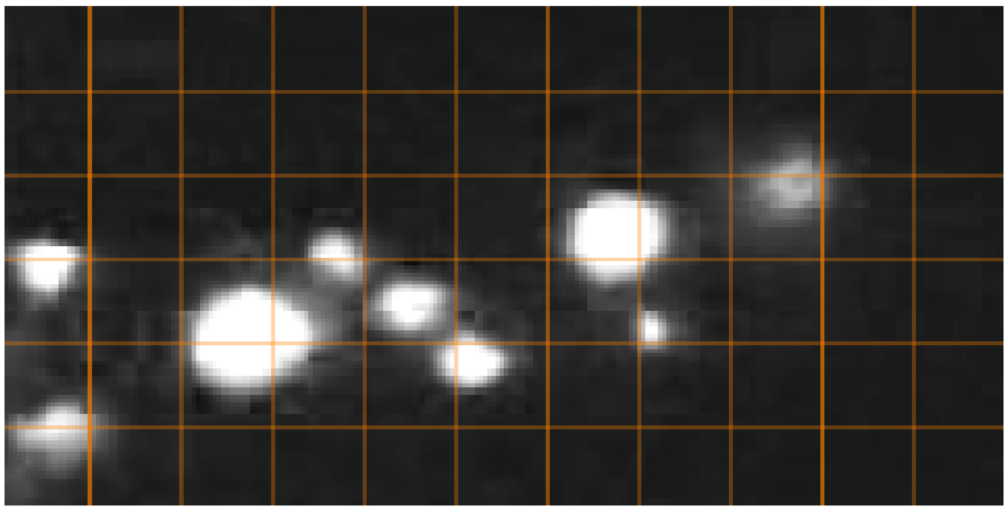


Figure 2.20: Events detected after background remove, adaptive threshold and filtering.

Now, events are more definite and, with a grid in background, it is possible to detect the location and the size of lightning.

2.5 Information addition

In this phase, data provided by LI is associated to every event. This a quite simple step. Main information is radiance, time and geolocation but also information about size and altitude are associated to give as many data as possible. Geolocation is realized by overlapping between IFOV and geographic map of FOV. Radiance is simply the radiation's value provided by LI and associated by an event. Time notice is given in UTC (Universal Time Coordinates measured by Greenwich's meridian).

Instead, size and length are measured with grid specific.

2.6 Conversion DT – data

When an event is detected, an application of a grid on background allows to location the event. Grid is big as FOV and every pixel has a dimension of 10x10 km^2: so, all the events in FOV are detected and location in their grid-position [14]:

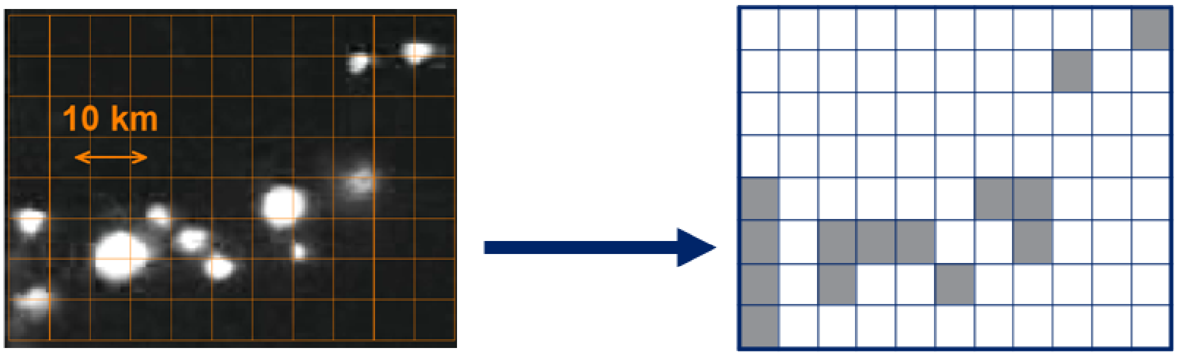


Figure 2.21: Abstraction of detected events.

The result of the localization is a grid with events. Now every kind of operation with the event is possible. First of all, positioning in row-column coordinates is used to allow the transmission to ground stations.

But to allow a better transmission and a few flow of data, LI has to make some operation over the detected events: the most important operation is the grouping of correlated events. This operation is the goal of this thesis and it is obtained by means of a new clustering algorithm. The result of algorithm is shown in Figure 2.22:

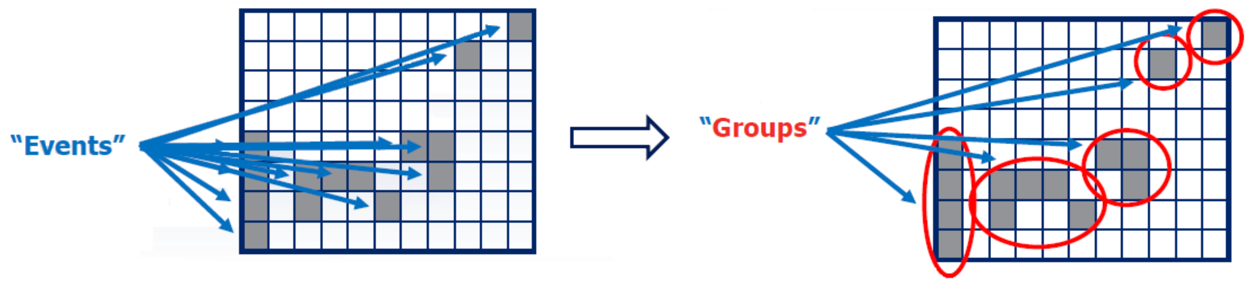


Figure 2.22: Clustering algorithm goals.

Chapter 3

Clustering algorithm

The clustering algorithm is a geometrical algorithm used to regroup some points under the same cluster. In LI, the algorithm is used to regroup every single lightning detected event in a window, storm in this case. To regroup every event, they have to observe some geometrical rules.

However, first of all, it is necessary to describe how event's information arrives to the algorithm and which components are necessary to satisfy timing and functional requirements.

3.1 Hardware and timing requirements

Dedicated hardware is composed by 3 principal parts:

- Detector: acquires events detected from the telescope;
- ASIC: identifies and makes operations over images. Then, it transforms all the events detected to give them a raster order;
- FPGA: contains the clustering algorithm and applies it to all the events.

Connection between all these components is shown in the image:

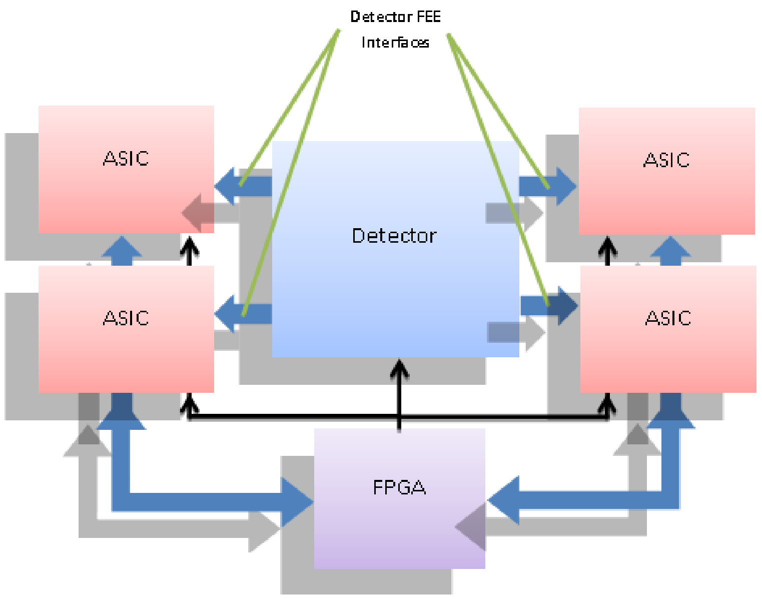


Figure 3.1: hardware in Lightning Imager for detected events.

The detector has a size of 1000x1170 pixel and it is divided in 4 identical 500x585 pixel quadrant. The order of coordinates for every quadrant is shown in the following picture:

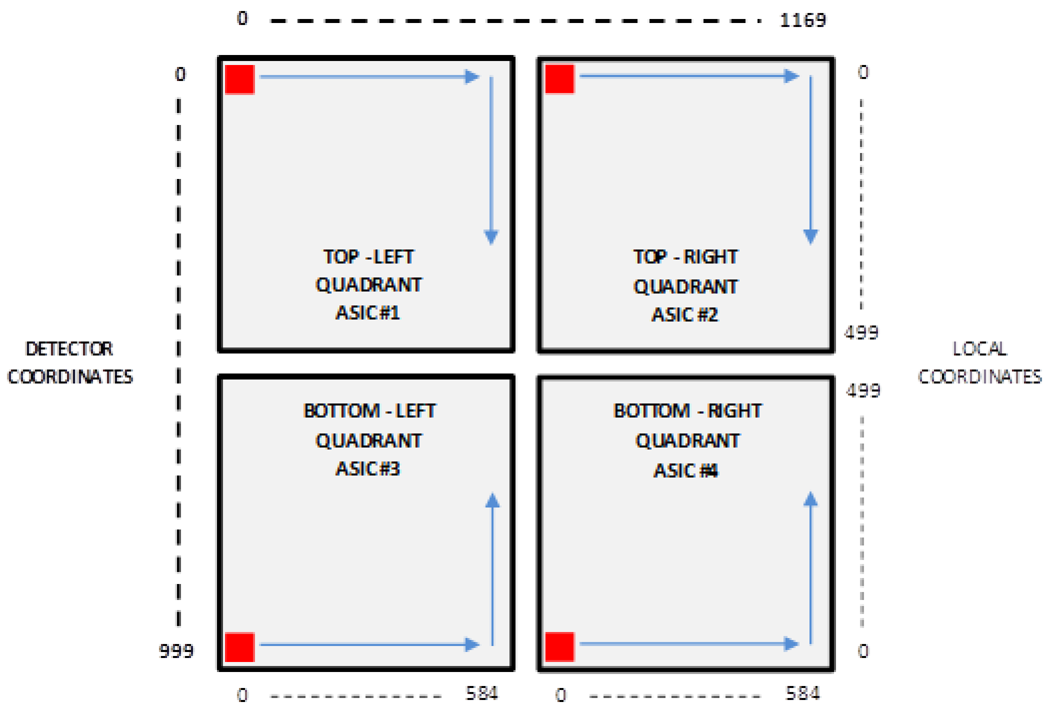


Figure 3.2: quadrant division of detector with local and detector coordinates.

In quadrants #1 and #2 (top-left and top-right), events are sorted by row in ascending order. Events with the same row index are sorted by column, in ascending order. In quadrants #3 and #4 (bottom-left and bottom-right), the events are sorted by row in descending order. Events with the same row index are sorted by column, in ascending order. Events are positioned under one of these quadrants depends to their detector coordinates (y, x) and a flag of neighbors can change the clustering.

Every quadrant of the detector has a dedicated ASIC which convert detected events in raster order to allow the algorithm to start the clustering. The 4 ASICs provide points to FPGA: FPGA contains 4 identical clustering algorithm, one for every quadrant, which work in parallel.

All the events detected by detector and ordering by ASIC are saved in a register FIFO to allow FPGA to provides them, one by one. A requirement specify that ASIC readout is delayed by one frame time, necessary to store all detected events:

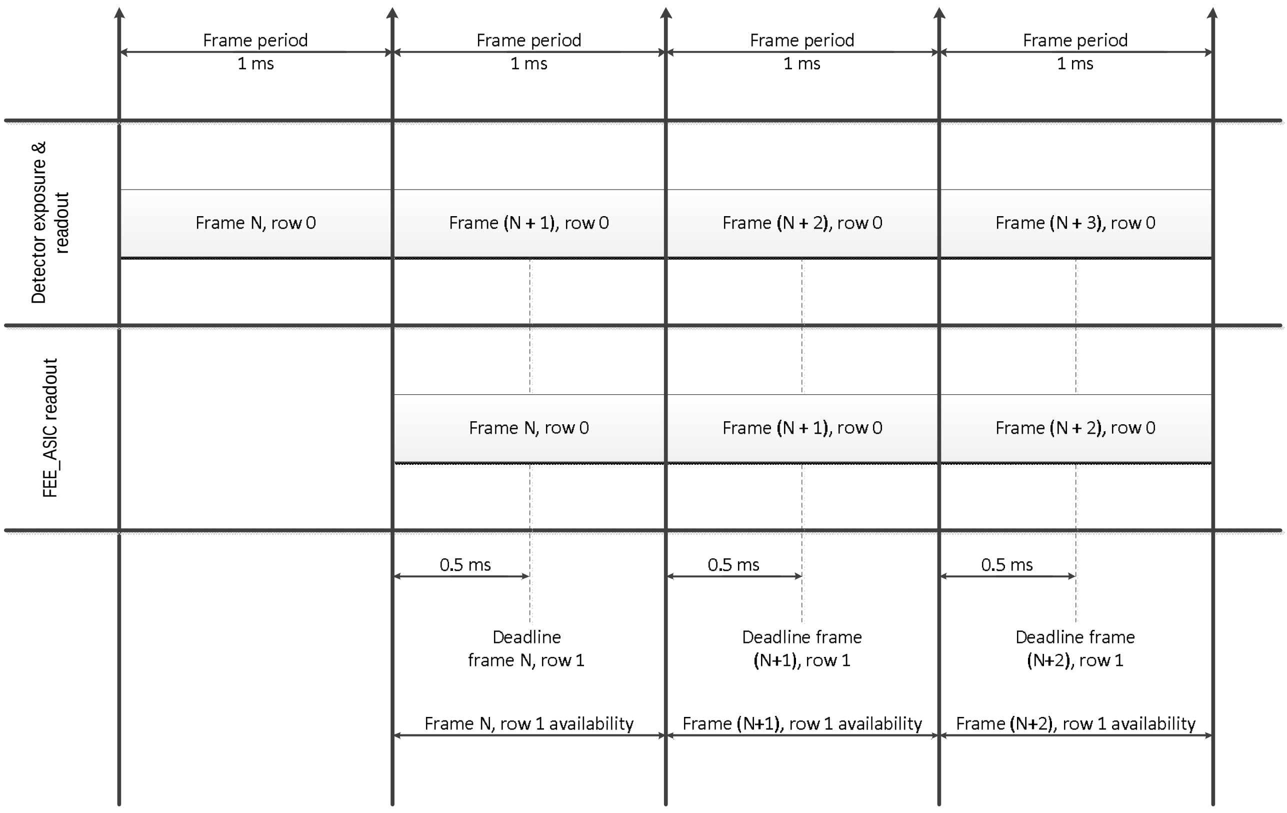


Figure 3.3: timing for every row in detector and FEE_ASIC.

At the end of all detected events in the frame, it is closed and a new frame is elaborated. In FIFO, events of new frame overwrite events of the “old” frame.

The clustering algorithm must be able to elaborate 1000 frames every second, a single frame every 1 ms, and the events coming from row K must be processed within 0.5 ms + K*2 us from the end of the exposure of the first row. The 0.5 ms are the tolerable latency of the algorithm and 2 us is the time of elaboration of every single row:

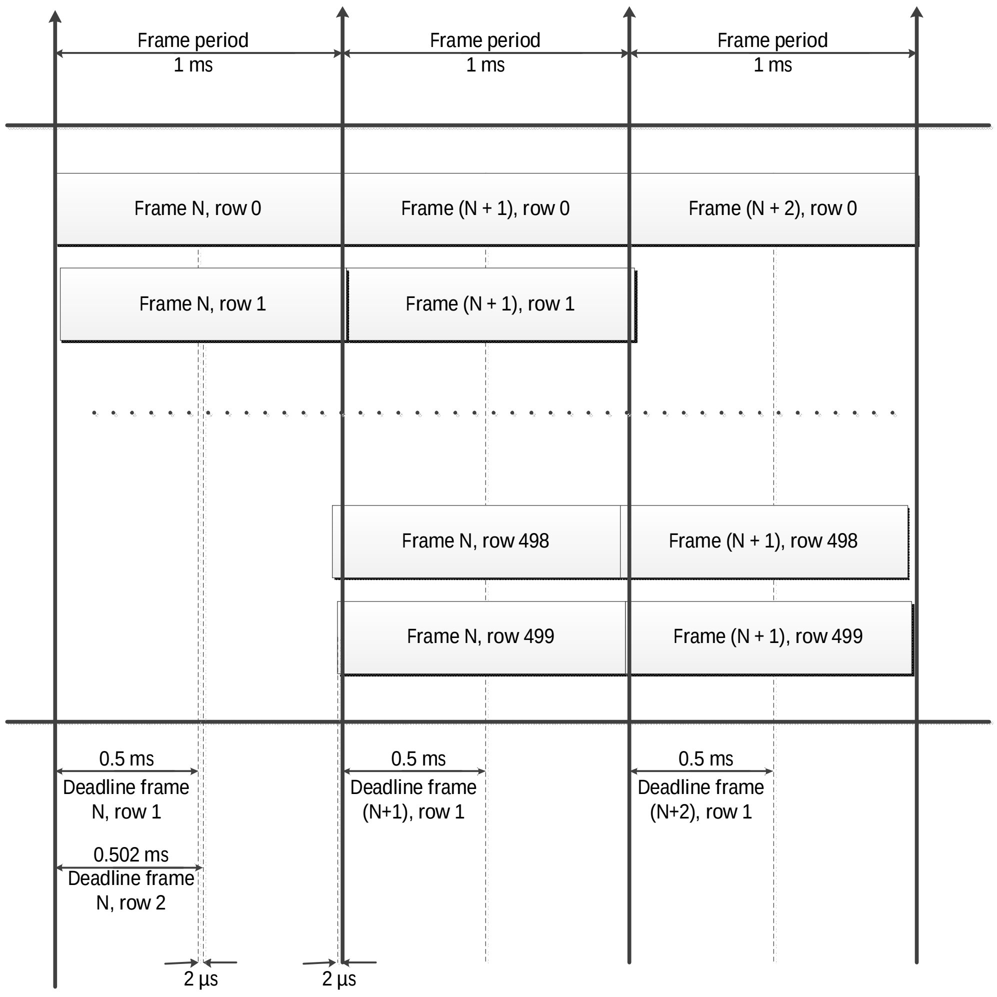


Figure 3.4: timing of all rows in a frame for several frames.

From the formula, the processing for each row must be completed within 0.5 ms since the row is available. But this requirement is difficultly respected when too many events are detected in a single row. This requirement of frame-timing could be violated when there are two consecutive rows, N and N+1, and the first one is full of events so that the clustering time for row N is fully occupied. Let T_e be the event processing time. This situation happens when row N has $0.5\,\text{ms}/T_e$ events. In this case, if row N+1 has more than $2\,\text{us}/T_e$ events, the present requirement is not satisfied. This situation is depicted in figure:

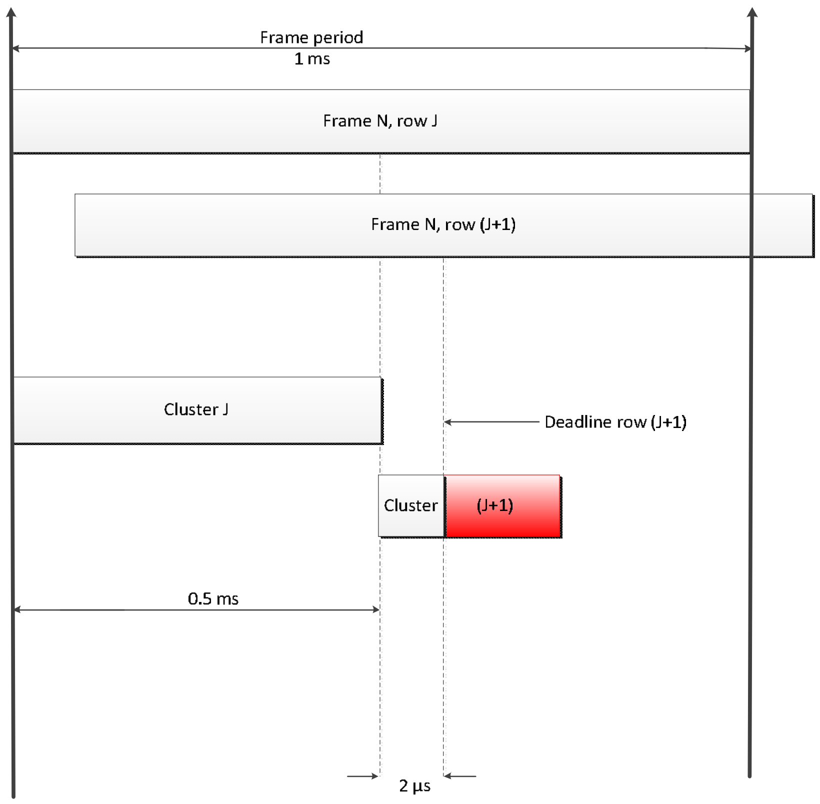


Figure 3.5: critical timing for consecutive rows.

Note that row-timing-requirement could be violated also when a row has more than 0.5 ms/T_e events.

Therefore, even though the global throughput is satisfied, the row specific processing deadline as well as the maximum latency may be violated, depending on the event distribution.

After a break about the principal hardware interface issues, other information about the hardware is exposed. First of all, the FPGA used is an RTAX4000S by ACTEL.

[25] This FPGA contains 4KK equivalent gates, 120 RAM blocks of 540 Kbits everyone, 40320 combinatorial modules (C-cells), 20160 register modules (R-cells) and with a clock frequency of over 350 MHz. Other information will be explained in the VHD code implementation because will be necessary for main choices.

Last part of hardware explain is the interface between ASIC and FPGA. In Figure 3.6, the architecture of the ASIC-FPGA interface:

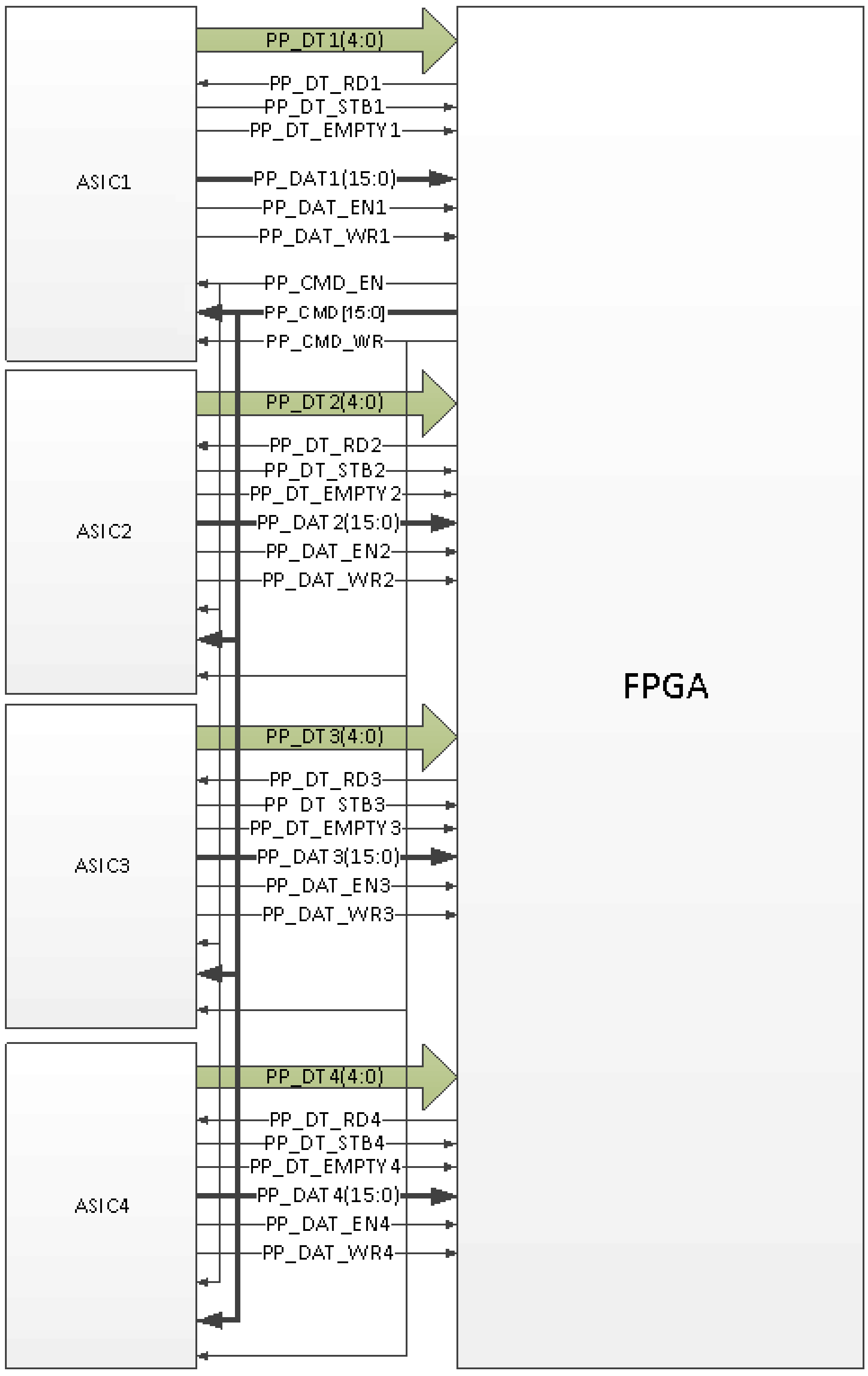


Figure 3.6: signal interface between the four ASIC and FPGA.

In Figure 3.6 there are three kind of interface between ASICs and FPGA:

- Command Interface (PP_CMD): it is used to handle the communication. Signals are:
    - PP_CMD_EN : enable;

- PP_CMD_WR : strobe:
- PP_CMD[15:0] : parallel bus 16-bit wide.

- Data Interface (PP_DATA) to data communication. Signals are:
  - PP_DAT_EN : enable;
  - PP_DAT_WR : strobe;
  - PP_DAT[15:0] : parallel bus 16-bit wide.

- DT Interface (PP_DT) to handle the detected events. Signals are:
  - PP_DT_RD: read request;
  - PP_DT_EMPTY: empty flag (active low);
  - PP_DT_STROBE: strobe;
  - PP_DT[4:0]: data bus 5-bit wide.

DT transmission is quite simple. When requested by FPGA through a PP_DT_RD strobe, the ASIC has to fetch the next Detected Transient from the internal DT buffer and transmit it using 4 consecutive transactions on PP_DT bus (called a DT packet). A DT is uniquely identified by 19-bit containing the row and column coordinates concatenated (9-bit for rows 0 - 499, 10-bit for columns 0 - 584):

# PACKET	PP_DT[4:0]	DESCRIPTION
1	xxxxx	COL[4:0] – DT column coordinate LSB
2	xxxxx	COL[9:5] – DT column coordinate MSB
3	yyyyy	ROW[4:0] – DT row coordinate LSB
4	0yyyy	ROW[8:5] – DT row coordinate MSB

Table 3.1: description of a single packet for DT communication.

An example of timing waveform about a DT packet:

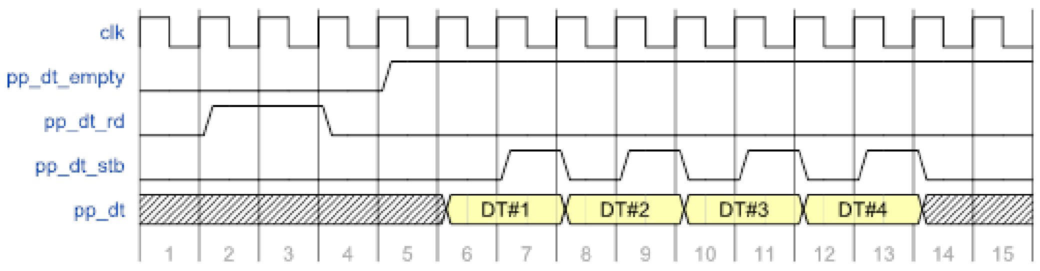


Figure 3.7: example of correct waveform for a DT packet.

After timing and hardware specification, it is necessary to give details about how the algorithm has to work. So, definitions and functional requirements are explained.

3.2 Definitions

First of all, a list of definitions is essential for a better comprehension of functional requirements.

Detector is divided in 4 quadrants, each one with its coordinates:

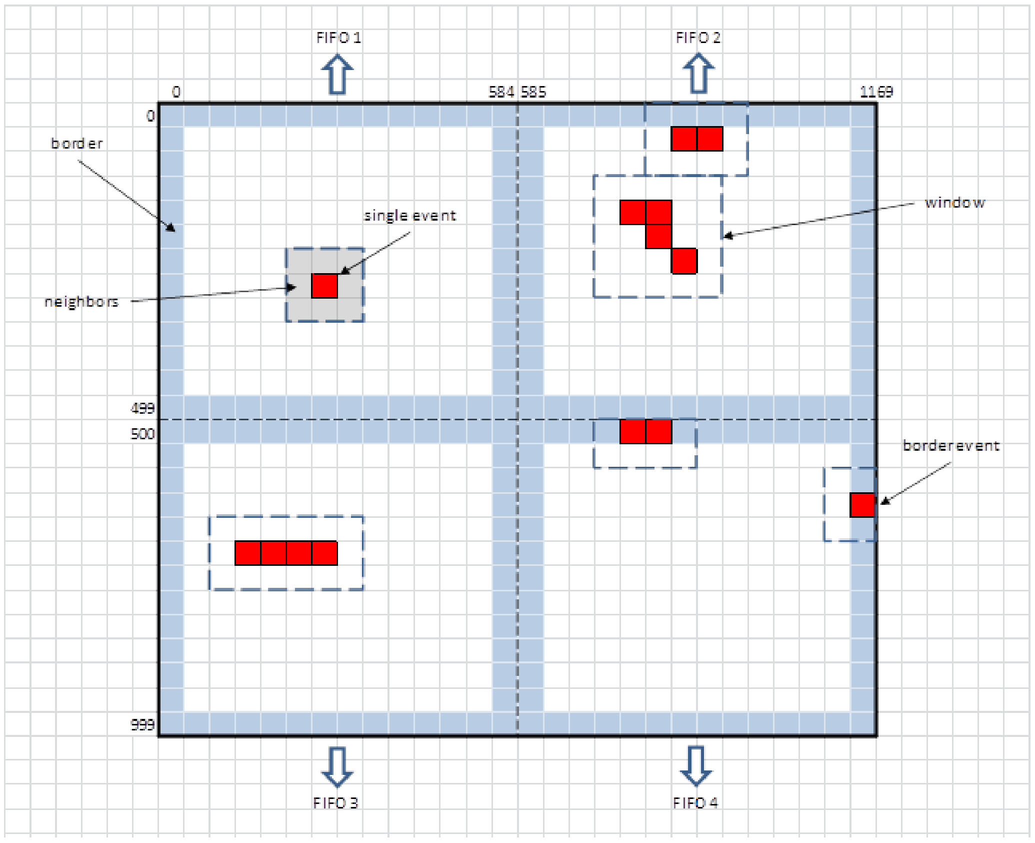


Figure 3.8: view of full detector with clustering algorithm essential elements.

Coordinates of every DT are of two kinds:

- **Detector coordinates.** They identify a pixel in the whole detector area, with row address (y) ranging from 0 to 999 and column address (x) ranging from 0 to 1169.
- **Local coordinates.** They identify a pixel in the local quadrant area, with row address (y) ranging from 0 to 499 and column address (x) ranging from 0 to 584.

A pixel is considered **border** if its x coordinate is 0 or 584 or 585 or 1169 and its y coordinate is 0 or 499 or 500 or 999.

Event or **Event Detected** or **Detected Transient (DT)** is a pixel whose radiance value is over the ASIC threshold: this pixel passes to the clustering algorithm. An event adjacent to a border is called **Border Event**.

One or more Detected Events can be enclosed in a rectangle: this is defined **Window**.

Windows are of two types:

- **Single Event**. An event is considered single if there are no other events or windows in the 3x3 area centered in the event.

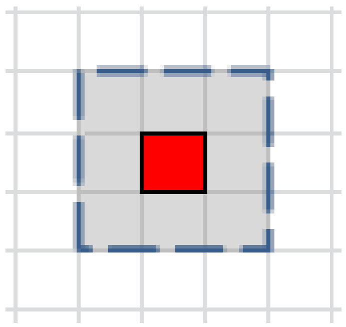


Figure 3.9: example of single event.

- **Cluster**. Cluster is a group of events.

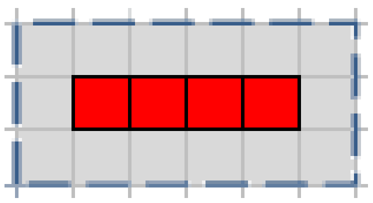


Figure 3.10: example of cluster.

A **Neighbor** is one of the 8 pixels surrounding an event.

3.3 Functional requirements

After these definitions, functional requirements are necessary to specify how the clustering algorithm has to work. Functional requirements are fundamental to implement geometrical functions and open correct windows.

3.3.1 Cluster Window

Cluster window requirement imposes that the algorithm defines just rectangular windows with variable dimensions, from *1x1* to *8x8* pixels, that include:

- all the pixels with an event;
- the 8 neighbors pixels to each event, if the neighbors inclusion is enabled. If the event is a border event, no neighbor pixels exceeding the border are expected to be included in the window.

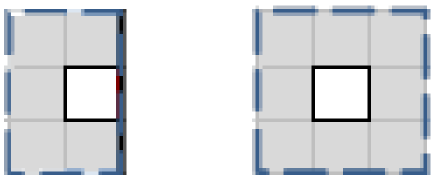


Figure 3.11: difference between border neighbors and 8 neighbors pixel.

But neighbor pixels are limited: in fact geometrical rules impose that all rows and columns of each window contain at least an event pixel or a neighbor pixel (neighbor inclusion is enabled):

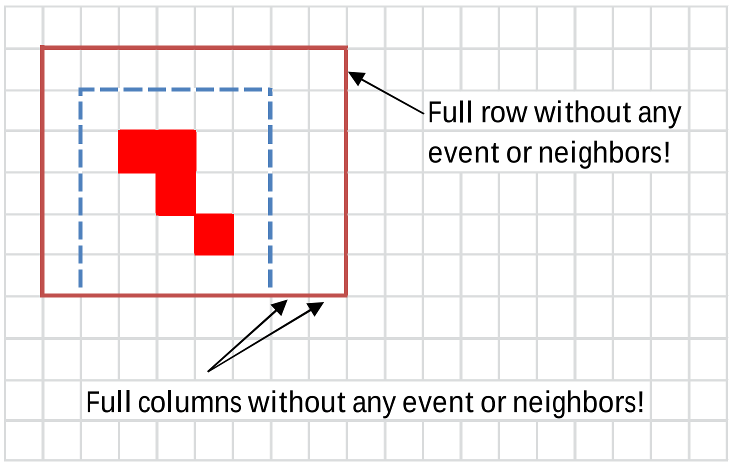

Full row without any event or neighbors!
Full columns without any event or neighbors!

Figure 3.12: requirement for neighbor pixels.

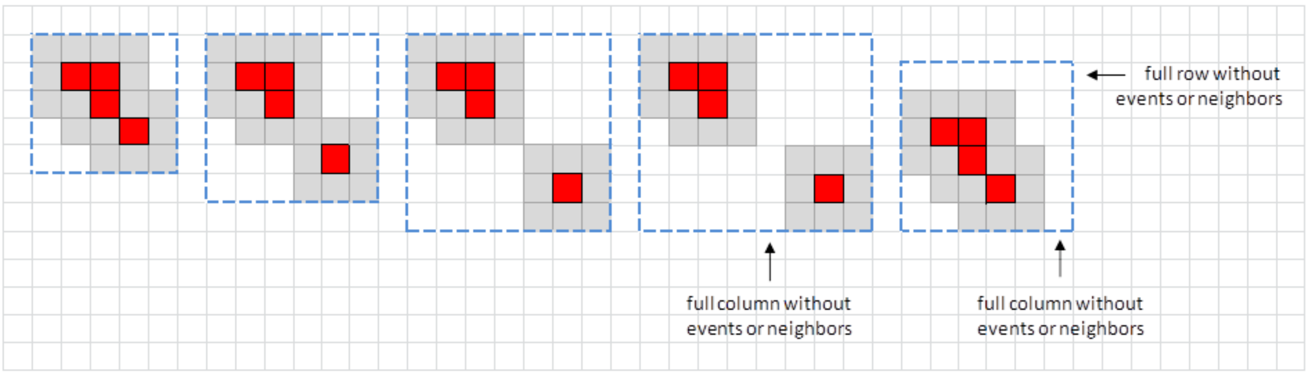

full row without events or neighbors
full column without events or neighbors
full column without events or neighbors

Figure 3.13.: example of allow/not allow neighbors conditions

Furthermore, a Single Event has to be included in a *3x3* window, if neighbor inclusion is enabled and it is not a border event, or in a *1x1* window, if neighbor inclusion is disabled. Windows can overlap only if the neighbor inclusion is enabled.

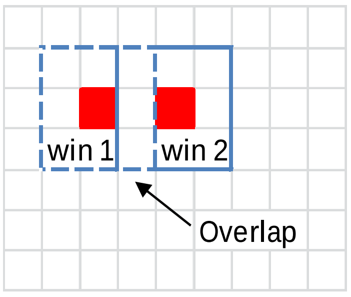


Figure 3.14: overlap of two windows with neighbor enabled.

3.3.2 Window number minimization

This requirement will be considered in analysis of the expansion strategy in paragraph 3.6.3.5 because a possible cause of cluster number could be the direction priority of expansion of a new DT when more window can include it.

First of all, if a Single Event has been included in a window of a near cluster, because of the particular cluster shape, it is not expected to be included also in a dedicated *3x3* (or *1x1*) window containing the Single Event only.

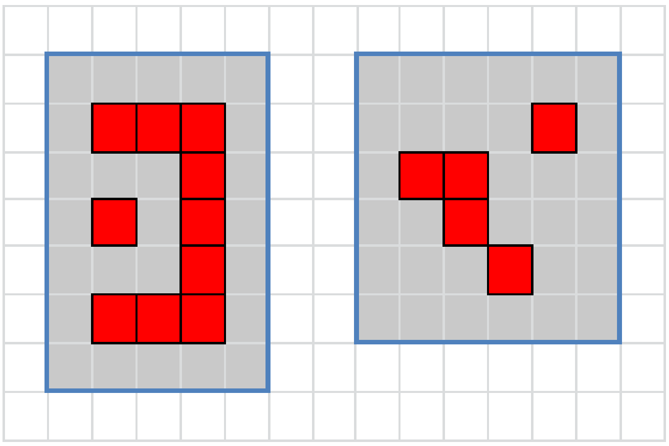


Figure 3.15: single events in a cluster. It's not included in a dedicated *3x3* window.

Single Events should be included in a *3x3* window because the noise filter, described in paragraph 2.2, makes a selection over windows with an area of *3x3* to reject the noise but if Single Events are included in another window, it cannot be considered single and it must not be deleted from the filter.

3.3.3 Window overlap

If the extension of a cluster is bigger than the maximum dimension of a window, the additional window, necessary to include all the outside events of the cluster, can overlap (if the neighbors inclusion is enabled).

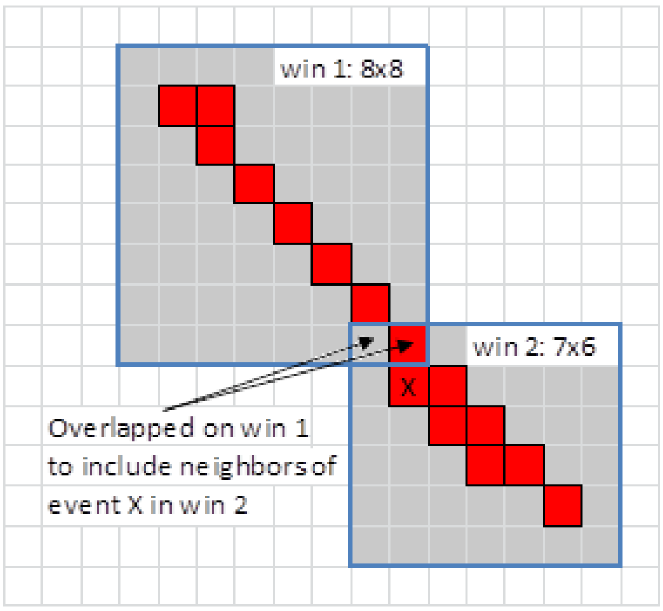

win 1: 8x8
win 2: 7x6
X
Overlapped on win 1 to include neighbors of event X in win 2

Figure 3.16: new window opening in overlap condition with a non-expandable window.

3.3.4 Algorithm throughput

Computational requirement imposes that the algorithm has to be able to process up to 1000 events every millisecond:

- the expected number of single events is 250 every millisecond;
- the expected number of events in clusters is 750 every millisecond.

Furthermore, the number of windows expected to be defined every millisecond is up to 300.

Because of quadrant division, the events distribution is homogeneous and each quadrant has to be able to process up to 250 events and it is expected 75 windows, for every quadrant every millisecond.

3.3.5 Output data format

The unique output requirement is how data information has to exit from clustering algorithm. It has to provide coordinates and size of every window(x, y, Dx, Dy). Coordinates are referred to the Left-Up corner of every window, Dx is the dimension on X axis, Dy on Y axis.

3.4 Algorithm implementation

In order to implement the correct algorithm, it is necessary to explain the difference between three modes of neighbor including.

Input neighbors value can be:

- *0*,in Disable mode;
- *1*, in Enable mode;
- *2*, in Selective Enable mode. This mode adds neighbors only to single events and doesn't add neighbors to clusters.

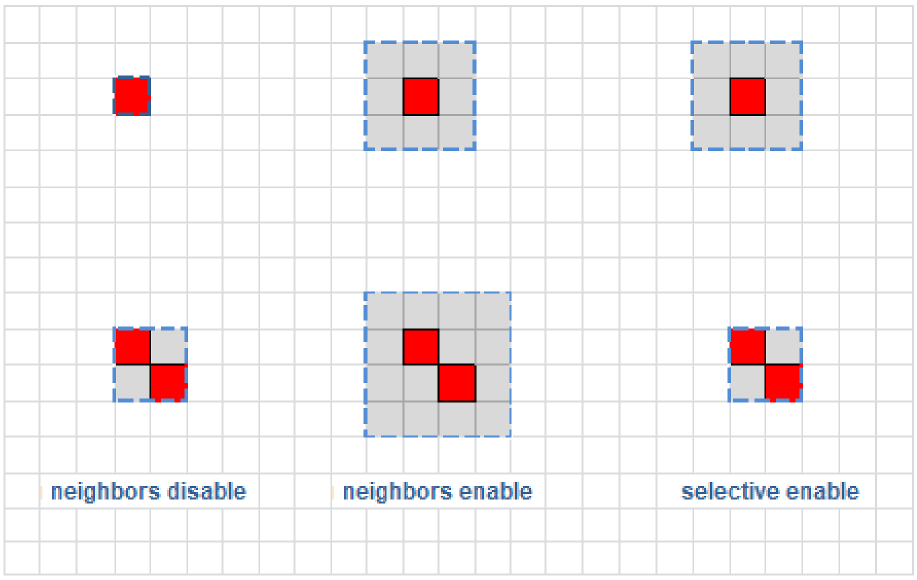


Figure 3.17: difference between three types of neighbors mode.

Geometrical rules used to implement the algorithm are the same for all three modes. Also with overlap, it allows neighbor mode to operate over events even after all DT's elaboration because up to 2 rows or columns can be overlapped between two windows:

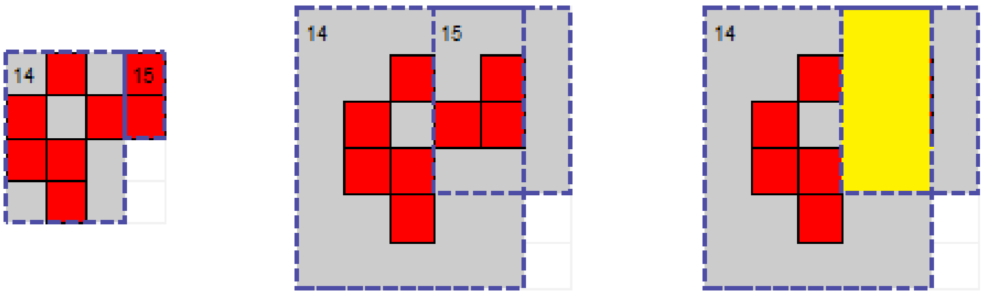


Figure 3.18: example of overlap of two windows. On the left, the windows before neighbors add and in center after the neighbor add (neighbors value is 1). On the right the overlap area: it is up to 2 rows or 2 columns. It is impossible to see a 3x3 or upper area of overlap

So, neighbors can be added after the execution of the algorithm in disable neighbors mode. There is just a single difference in geometrical rules between neighbor modes: when neighbors are enabled, the maximum height and width of a window can be up to 6 pixels instead of 8 because 2 neighbor-pixels must be added later. Therefore, neighbor mode imposes only a check over max size in the process before the add-neighbors post-process.

Neighbor attachment while clustering of events was also considered (dynamic neighbor attachment). Solutions including dynamic neighbor attachment were discarded because they caused problems with total window overlap, Single Event windowing and were also more computationally intensive.

Now the “real” algorithm must be implemented.

Algorithm inputs are just the DT coordinates, X’s and Y’s. The algorithm must expand existing window or open new ones to enclose every DT.

But, how can the algorithm decide if an existing window could include the new DT? And if there are more than one candidate to expansion, how can it choose which cluster is the correct candidate to add the new DT?

To include the new DT in a cluster, it is necessary that at least one of the 8 pixels around the new DT is occupied by a window.

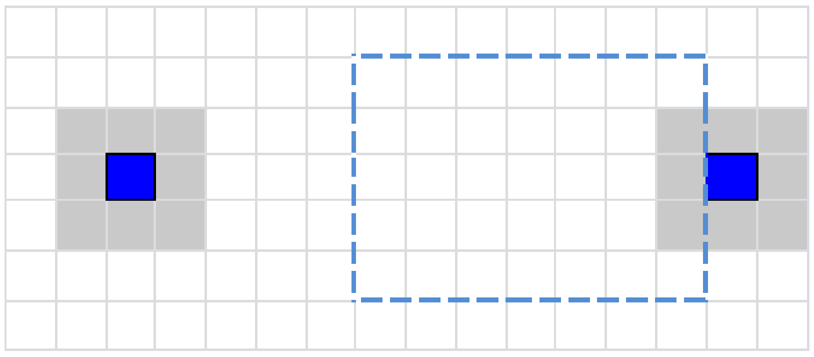


Figure 3.19: new DT (blue) and 8 pixels around it. If at least one window sits in at least one of these 8 pixels, the new DT can be added to a window.

When a new DT is in input at the algorithm, its coordinates is compared with the coordinates of all the already existing windows. The existing windows are stored in a memory; so, all the memory will have to be scanned to find an adjacent window.

A window can be expanded only in an adjacent row or column or both of them. So, there are 8 possible positions of an new DT relative to a window, one for every possible direction of expandability. A special case is when, for a particular configuration of window, a new DT is already enclosed in a window: in this case, the window that include the event has the highest priority over possible adjacent windows and no changes must be applied to the coordinates and size of the selected window:

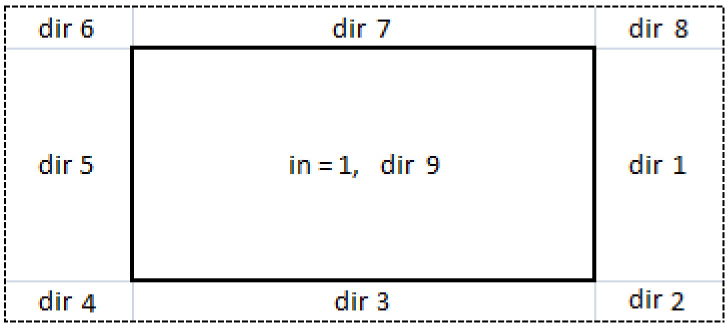


Figure 3.20: 8 possible direction and internal position for new DT relative to a window.

A window adjacent to a new DT is defined *Candidate*.

Therefore, in the worst case, 8 *Candidates* must be checked for the expansion. In fact, it is necessary that a window could change its coordinates and dimensions to include the new DT. It is not possible if the candidate is adjacent to another window in the same direction of the new DT. If no candidate can be expanded, a new window is opened.

So, every *Candidate* must be compared with all the other windows to check the expandability. Expandability is different for every direction of the new DT. Because of the position, not all windows must be checked: far windows are not important. This factor will be very important to minimize the time of elaboration for every window.

In the worst case, all 8 *Candidates* can be expandable. So, it is necessary to choose which one is the most important. For this reason, the second step of the implementation will be the choice of a strategy that will minimize the window number as required. The strategy will select which *Candidate* must will be expanded.

The last step of new DT elaboration is the update of window parameters. Every new DT either changes the coordinates and size of a window or add a new window.

When all the events in a frame are analyzed, neighbors are added. If neighbors value is *0*, no neighbors are added and windows are ready for transmission without changes. If neighbors value is enable neighbors are added to every windows with a "*+1*" in Dx and Dy value and a "*-1*" in X and Y value. In Selective Enable, neighbors are added just to Single Events and windows larger than *1x1* are not modified.

Figure 3.21 represents the flow-chart of the clustering algorithm.

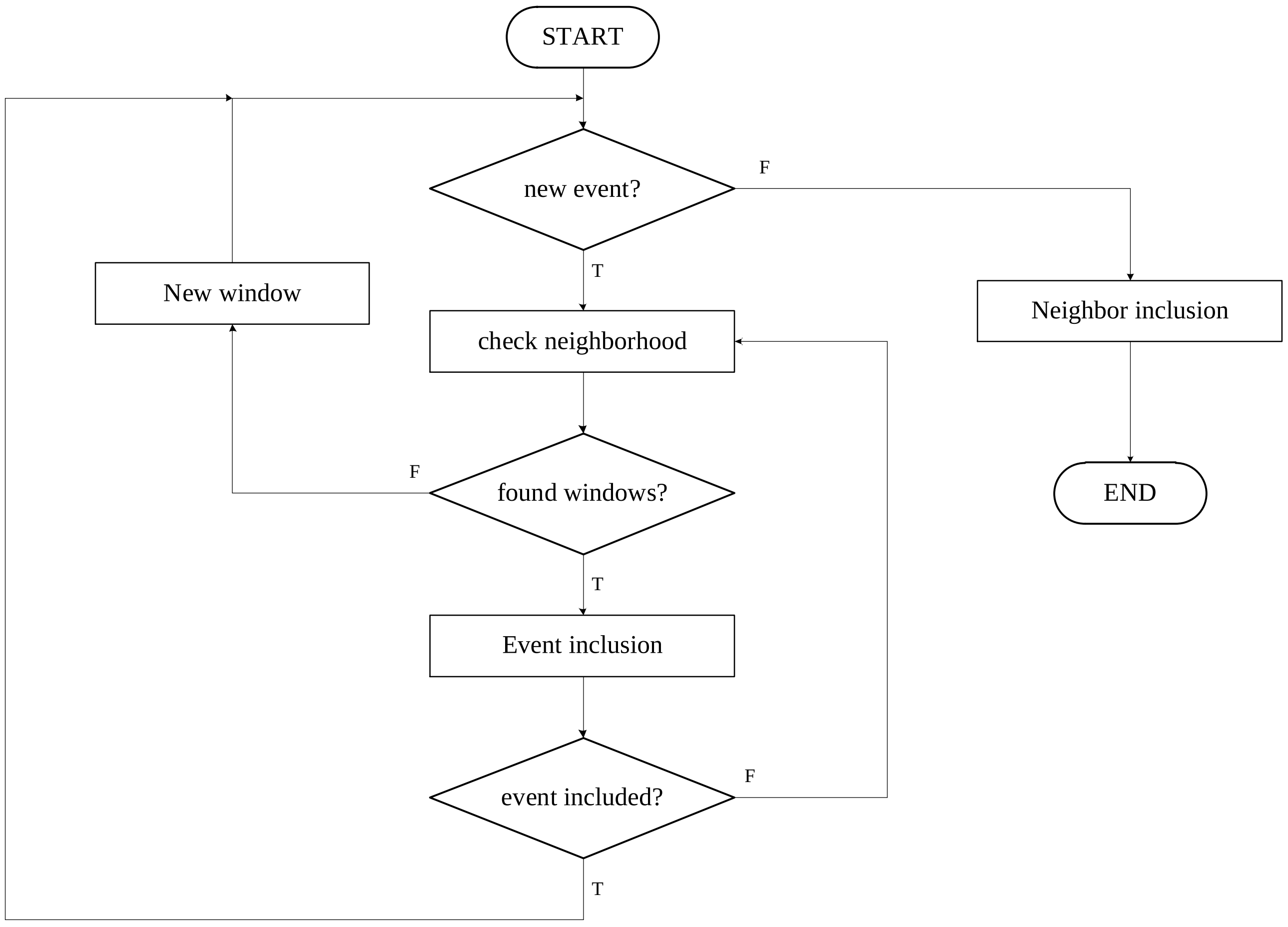


Figure 3.21: global algorithm for every frame.

3.5 Algorithm optimization

The algorithm described in the previous section is the first approach to satisfy all the requirements. In fact, the high number of DTs with the limited HW resources, implies that the algorithm will have to be very fast and efficient. The system cannot wait for the whole image to begin with the clustering. The algorithm has to define windows while the DTs are coming from the detector, in order to maintain the required throughput. So, for the HW implementation of the MATlab code, it is necessary to make some choices for an optimization.

First of all, the positions of a new DT, relative to a window, can be only 4. 8 positions are overestimated because of the raster scan. It allows just new DT over the last existing window/event.

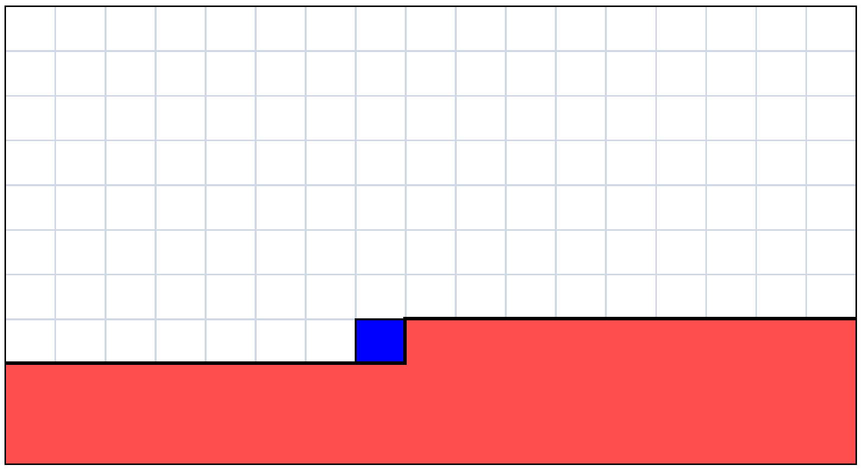


Figure 3.22: example of raster scan order. New DT can have adjacent window or event only in Left, Left-Up, Up and Right-Up directions. Other directions cannot be already occupied because events in this area are not analyzed yet.

There are 8 possible relative positions of new DT to window but, because of the raster scan, only 4 positions are really possible:

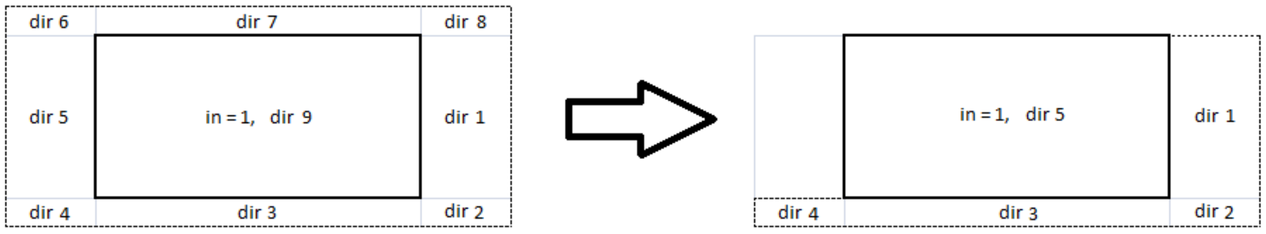


Figure 3.23: 8 and 4 positions of new DT relative to a window.

In fact, no events can be positioned in direction 5, 6, 7 and 8 because all the DT in these positions are already be analyzed.

Because of geometrical law, the expandability check is easier than previously.

First of all, possible direction of expansion are 4 instead of 8. In raster scan, events in Left, Left-Up, Up and Right-Up are already be analyzed and expansions in these directions are already made.

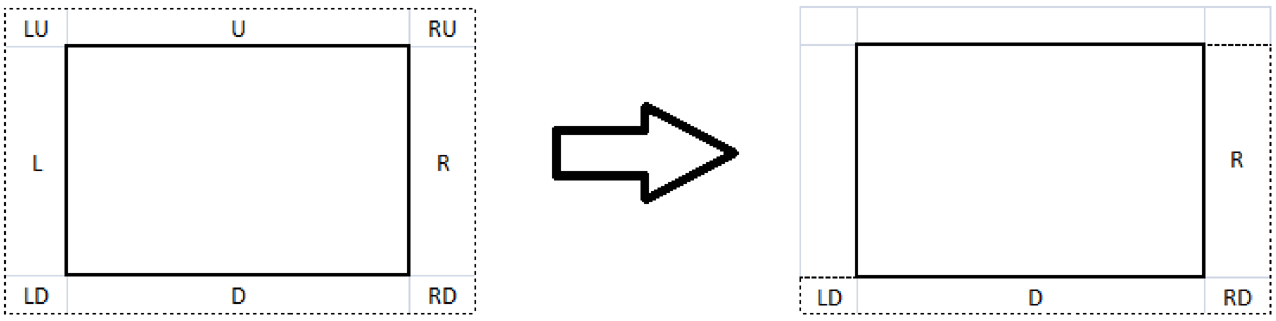


Figure 3.24: real possible expansion directions.

Still because of raster scan, up to 4 windows can be candidates. In fact only 4 of the 8 pixels around a new DT can be already really occupied by a window. Others 4 positions can't be occupied because events or windows in these position are not analyzed:

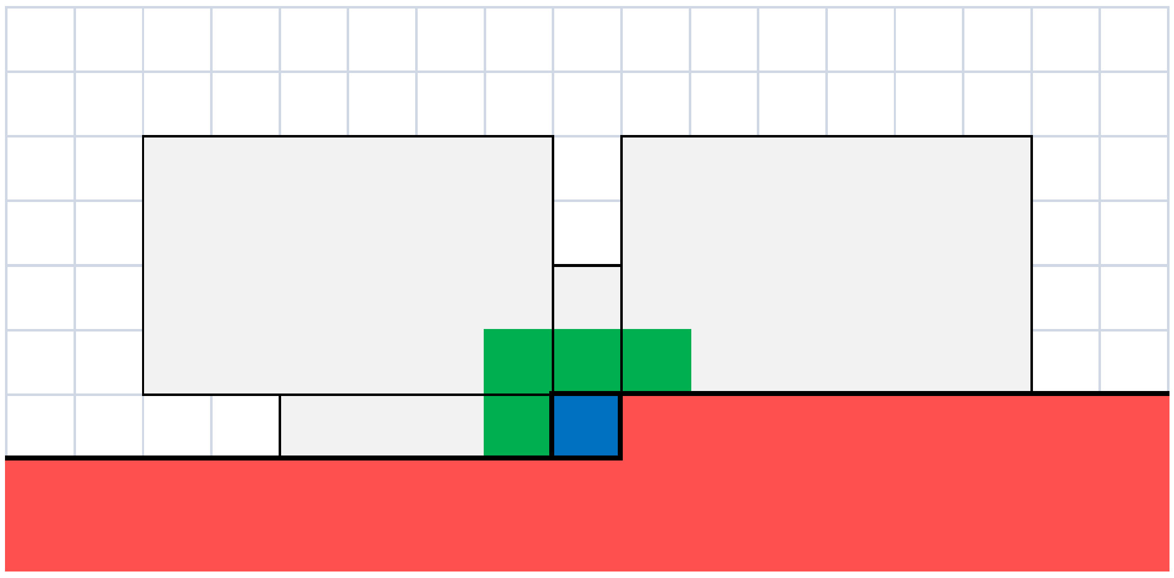


Figure 3.25: image shows which neighbor pixels (green) are valid for the 4 possible candidates (grey). Red zone is the area not scanned yet.

The most important optimization in the expandability check is optimization of the number of comparing between windows. In fact, when a candidate makes expandability check and tries to expand, it must be compared with all the other windows. If an adjacent window exists in the selected direction, the expansion check fails and the candidate is not chosen. This expandability check can be limited to a few number of windows instead of all. These windows are called the *P_windows*, i.e. Proximate windows. To understand how many windows must be checked for expandability, it is necessary to analyze the worst case and deduce geometrical rules. The worst cases are two, one for height and one for width.

The worst case for height is the following:

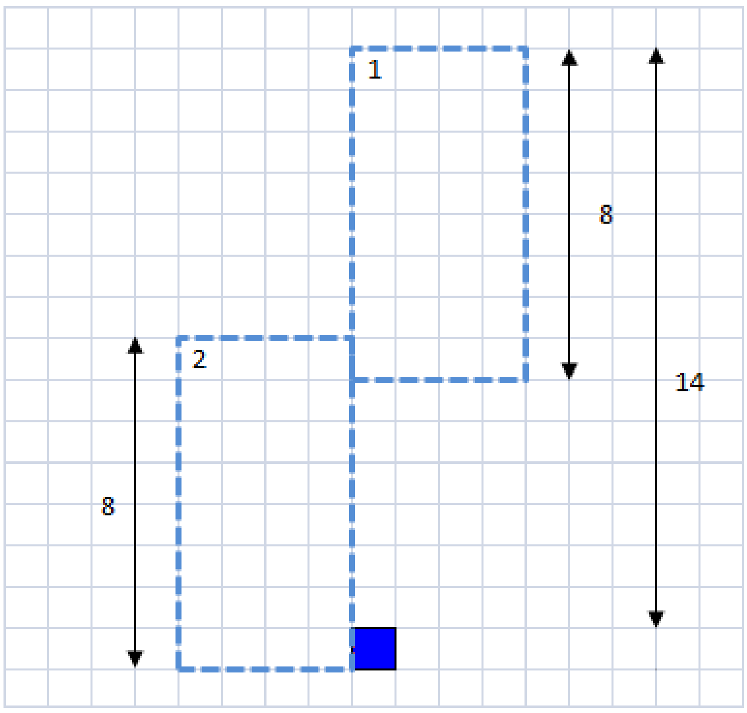


Figure 3.26: height worst case for expandability check.

When the new DT is adjacent on the right side of window 2, expandability check must look for other windows adjacent to this one. If a window exists in the height worst case position, window 2 cannot be expanded in the Right direction. Obviously, the max height is in case of max window height for both of them. So, in Y direction it is necessary to check windows from Y_{DT}-14 to Y_{DT} when X value is X_{DT} where X_{DT} and Y_{DT} are the DT coordinates.

Instead, the worst case for width is:

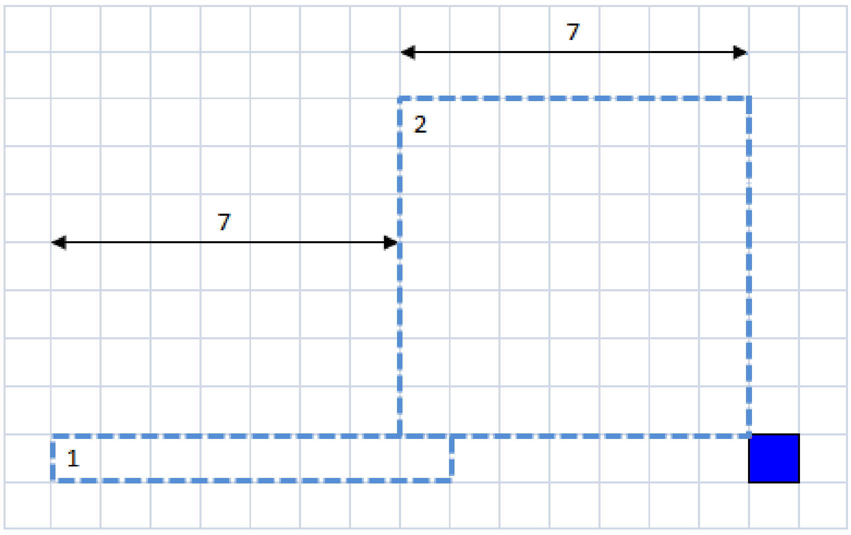


Figure 3.27: width worst case for expandability check.

This is the dual case: when the new DT is adjacent on the right-down side of window 2, expandability check must search other windows adjacent to this one. If a window exists in the width worst case position, window 2 cannot be expanded on Right-Down direction. Once again, the max width is in case of max window width for both of them. So, in X direction it is necessary to check windows from X_{DT}-14 to X_{DT} when Y value is Y_{DT}.

Total area of worst cases are:

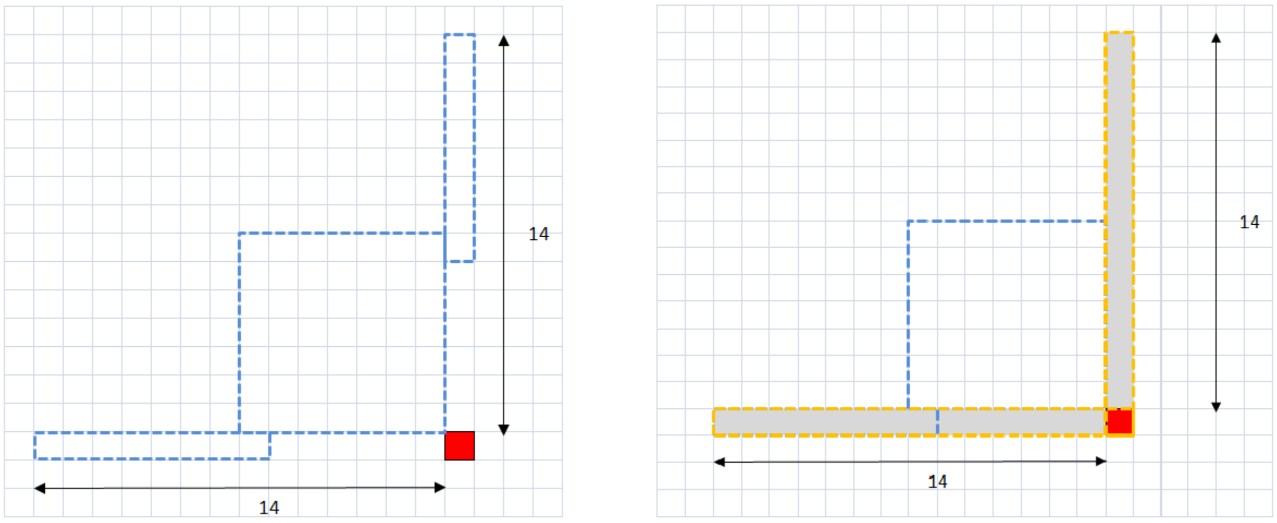


Figure 3.28: two worst cases in the same grid and total area.

The total P_window area is the sum of two areas (and conditions). A window is considered *P_window* when:

$$\begin{cases} Y_{DT} = Y_{window} \\ X_{DT} - 14 \leq X_{window} \leq X_{DT} \end{cases} \quad \text{or} \quad \begin{cases} Y_{DT} - 14 \leq Y_{window} \leq Y_{DT} \\ X_{DT} = X_{window} \end{cases}$$

When a window existing in this total area, it is saved in a dedicate register, the *P_window_register*. At the end the window scan, *P_windows* founded are used for the expandability check. In the worst case there are 14 windows in this global area, one every two pixels.

Expandability depends also on the window size. So, test over the dimension of every windows is sufficient to stop the expandability. In fact, if a window has the max size towards the new DT direction, it is not expandable and cannot be considered a candidate yet. Also a test over quadrant border is made for expandability because no neighbor pixels are allowed over border.

3.6 MATlab High level model

Before the MATlab model description, it is necessary to explain strategies to resolve conflict situations.

3.6.1 Expansion strategies

A classic example of conflict situation is the one depicted in Figure 3.29:

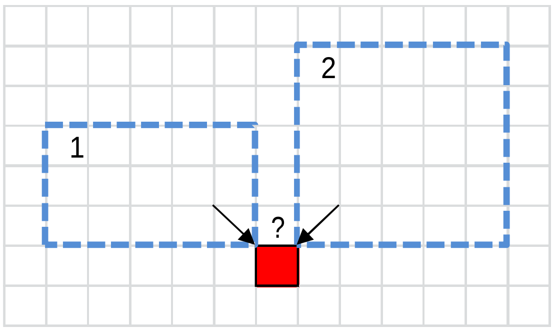


Figure 3.29: window expansion conflict.

The strategy determines which direction is preferred for the expansion of windows. The strategy also impacts on the total number of windows defined. Therefore, the selection of the strategy is important for the minimization of the total window number. As before, since the DT ordering is the usual raster scan over the quadrant, it is not possible that a new DT is on the top edge of a window, i.e. windows cannot be expanded in the *Left*, *Left-Up*, *Up* and *Right-Up* directions:

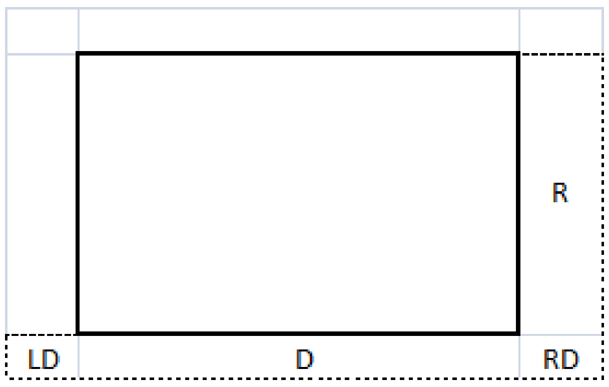


Figure 3.30: allowed directions of expansion.

R, *RD*, *D* and *LD* are the possible expansions for windows. A priority has to be defined between these directions. Referring to Figure 3.29, if an *R*-before-*L* strategy was selected window 1 would have been expanded. On the other hand, with an *L*-before-*R* strategy, window 2 would have been expanded.

Figure 3.31 shows the strategy tree. The possible strategies are given a short name to be recalled in the following: *RDL*, *DLR*, *DRL*, *RDLR*, *DRLR*, according to their direction priority.

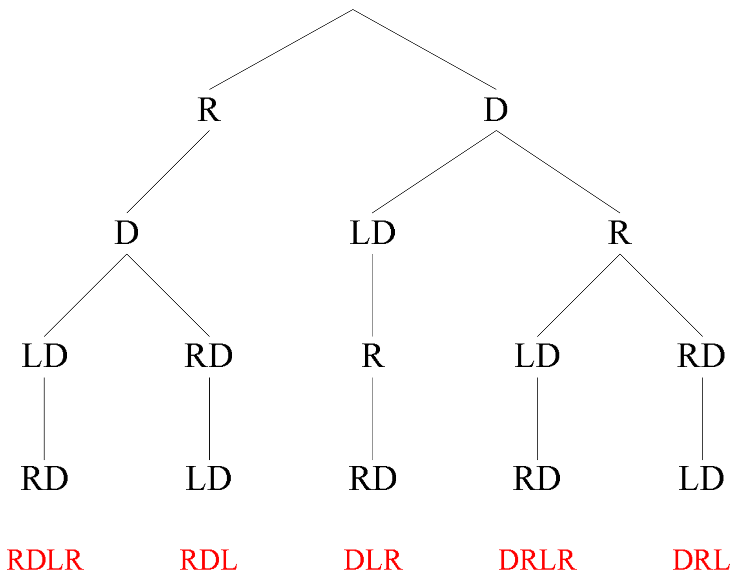


Figure 3.31: possible clustering strategies.

Figure 3.32 shows the order of expansion as a function of the position of the new DT with respect to the window for the different strategies:

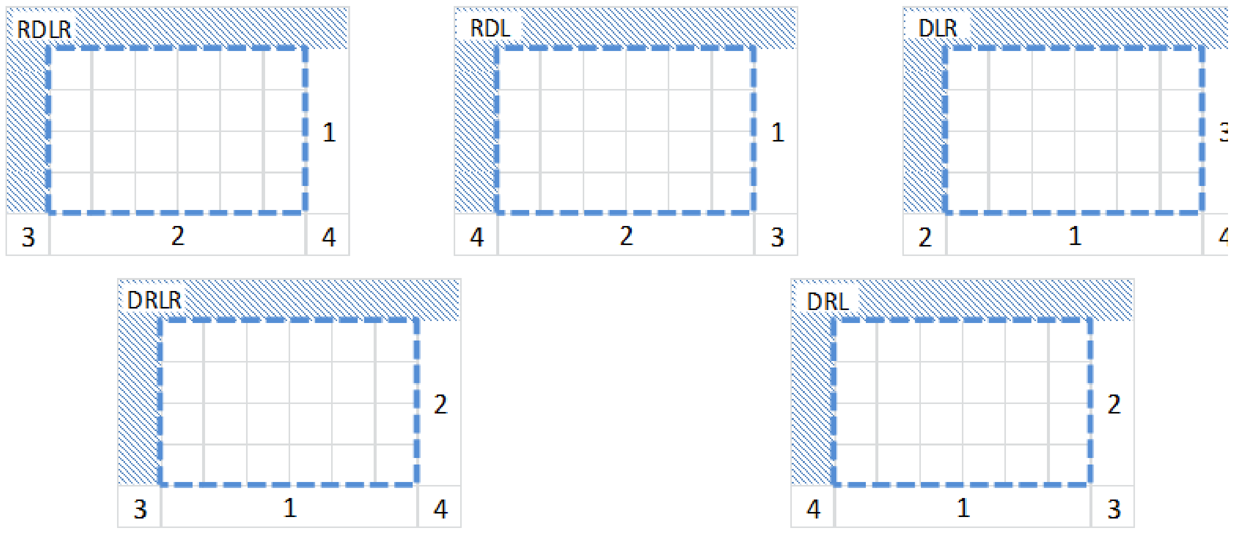


Figure 3.32: priorities directions of expansion.

Strategies are equivalent from the point of view of HW complexity. The best strategy will be evaluated by means of computer simulation over a significant set of test-vectors.

3.6.2 Test-vector generation

The random test-vectors used for the clustering algorithm design and verification were generated with a MATlab simulator.

In order to simulate the randomness of a lightning flash seen from above the clouds, the MATlab model generates random points representing the flash according to a bi-dimensional Gaussian distribution.

First of all, the X and Y coordinates of the points are generated by means of two independent Gaussian distributions. The mean value along X and Y represents the flash centroid, uniformly distributed in the lightning concentration area (max *190x190* pixel). The standard deviation is correlated with the radius of the flash. In a Gaussian distribution, more than 99.7% cases fall in the interval [-3σ, 3σ] Therefore, 3σ can be considered as the radius of the distribution and we can obtain the standard deviation for the generation of flashes as: $\sigma = {}^{R}/_{3}$.

Then, the Y coordinates of the points are shrunk by the factor (1–*e*), where *e* represents the distribution eccentricity. The eccentricity is modeled as a Gaussian random variable with selectable mean and variance.

Finally, the flashes are rotated by an angle ϑ. ϑ is modeled as a uniformly distributed random variable in the interval $[0,2\pi)$.

The resulting distribution skeleton is depicted in Figure 3.33.

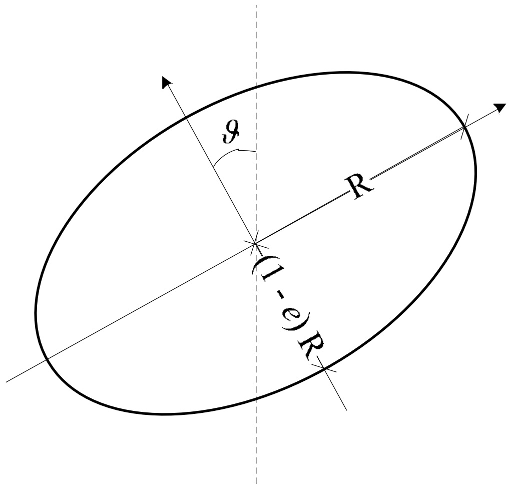


Figure 3.33: distribution skeleton.

Figure 3.34 shows some generated lightning flashes.

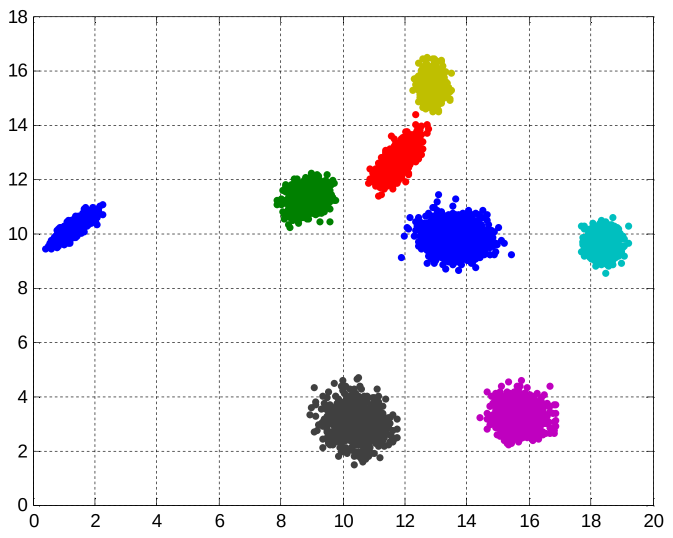


Figure 3.34: generated flashes.

The generated coordinates of points of the flashes are then quantized to integer numbers representing the pixel coordinates, considering that each pixel is about a *10x10* Km^2 square.

An example of resulting test-vector generated with this algorithm is represented in Figure 3.35:

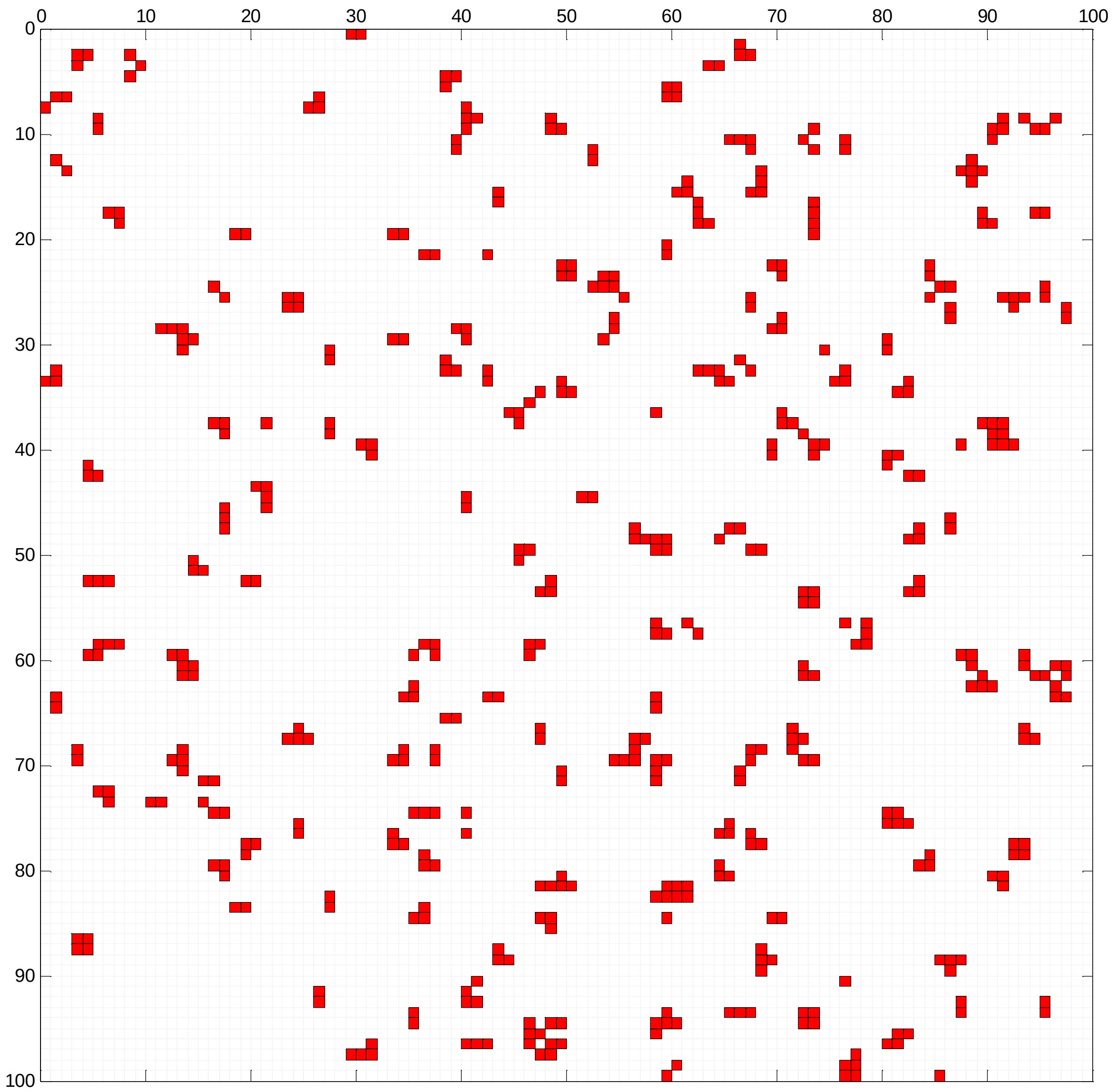


Figure 3.35: example of test-vector.

Test-vector generator parameters are shown in Table 3.2:

PARAMETER NAME	**DESCRIPTION**
FRAME_WIDTH	Width of the frame
FRAME_HEIGHT	Height of the frame
L_AREA_WIDTH	Width of the area which lightning are concentrated into
L_AREA_HEIGHT	Height of the area which lightning are concentrated into
NUMBER_OF_FLASHES_MEAN	Mean number of flashes in a frame
NUMBER_OF_FLASHES_STDDEV	Standard deviation of the number of flashes in a frame
FLASH_ECCENTRICITY_MEAN	Mean eccentricity of flashes
FLASH_ECCENTRICITY_STDDEV	Standard deviation of the eccentricity of flashes
FLASH_RADIUS_MEAN	Mean radius of flashes
FLASH_RADIUS_STDDEV	Standard deviation of the radius of flashes
FLASH_NUMBER_OF_POINTS_MEAN	Mean number of points generated per flash
FLASH_NUMBER_OF_POINTS_STDDEV	Standard deviation of the number of points generated per flash
NOISE_DT_prob	Quantity of noise

Table 3.2: parameters in test-vector generator code.

NOISE_DT_prob is the actual density of events due to noise. For example, it is possible to set *NOISE_DT_prob* = $^{N}/_{(1170 \times 1000)}$ to obtain exactly *N* random noise DTs over the area of *1170x1000.*

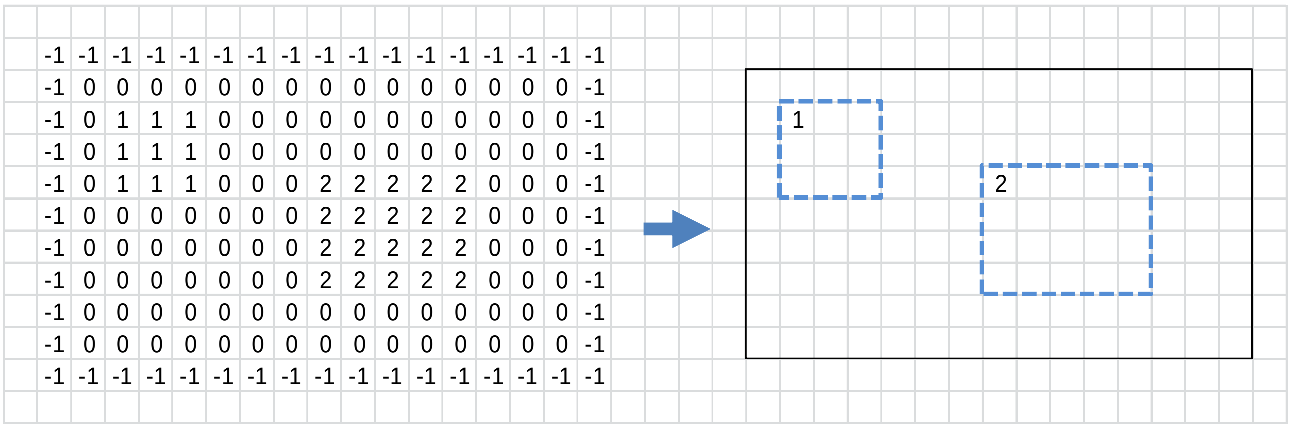
3.6.3 High-level model

A high-level MATlab model of the clustering algorithm was developed in order to test the algorithm functionality and fine-tune the clustering strategy. The high-level model also functions as a reference for the successive steps of implementation.

3.6.3.1 Data structure

The *window* structure contains all the data concerning the windows. The fields of the window structure are:

window.matrix: is the model representation of the clustered quadrant. Basically it is an two-dimensional integer array of height+2 rows and width+2 columns. The extra columns and rows are used to store *-1* value used to borders of the image. The matrix contains the information about the position and the ID of all the windows. Every position in the matrix represents the status of the pixel in the original image. A *0* value pixel means that there is no window there, a value *N* > 0 means that there is a window of ID *N*. A visual representation of the matrix variable is depicted in Figure 3.36:



-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
-1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-1
-1	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	-1
-1	0	1	1	1	0	0	0	2	2	2	2	2	0	0	0	-1
-1	0	0	0	0	0	0	0	2	2	2	2	2	0	0	0	-1
-1	0	0	0	0	0	0	0	2	2	2	2	2	0	0	0	-1
-1	0	0	0	0	0	0	0	2	2	2	2	2	0	0	0	-1
-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Figure 3.36: visual representation of *window.matrix*.

window.number: represents the number of opened windows.

window.x: column coordinate of the left edge of the window.

window.y: row coordinate of the high edge of the window.

window.dx: width of the window.

window.dy: height of the window.

window.max_w: maximum width allowed for a window.

window.max_h: maximum height allowed for a window.

*event_clustered***:** vector containing the ID of the window which every DT is clustered into.

3.6.3.2 Model configuration

Table 3.3 shows the configuration parameters for the algorithm:

NAME	DESCRIPTION
MAX_WINDOWS	Maximum number of windows that the model can use for clustering.
MAX_EVENTS	Maximum number of event treated by the clustering algorithm
MAX_WIN_WIDTH	Maximum width of windows
MAX_WIN_HEIGHT	Maximum height of windows
strategy.dat	File that contains window expansion rules
strategy_masks.dat	File that contains window expansion check masks
new_win_masks.dat	File that contains new window opening strategy
STRATEGY_VECTOR	List of strategies for batch processing
NEIGHBORS	Neighbor inclusion flag. • 0 : no neighbors • 1 : all neighbors • 2 : neighbors on single events only

Table 3.3: algorithm configuration parameters.

Table 3.4 shows the model configuration parameters.

NAME	DESCRIPTION
TEST_VECTOR_PATH	Path to test-vectors
SELEX_TEST_VECTORS	Enables the processing of SES test-vectors
SELEX_TEST_VECTOR_PATH	Path to SES test-vectors
IMG_TEST_VECTOR	Enables the processing of a bitmap image
TEST_VECTOR_FILE_LIST	List of test-vector files for batch processing
height	Height of the area to be processed (quadrant)
width	Width of the area to be processed (quadrant)
STRATEGY_PATH	Path to strategy files
IMAGE_SHOW	Shows the output of the clustering
CLUSTER_STATS	Enables cluster stats evaluation
AREA_HISTOGRAM	Enables the histogram showing the area of windows.
NUMBER_OF_EVENTS_HISTOGRAM	Enables the histogram showing the number of DTs per window.
PRINT_RESULTS	Prints result information on the MATlab console
ERROR_ANALYSIS_PLOTS	When SES test-vector analysis is enabled, this flag enables the visual analysis of the differences between the expected output and the output of the clustering algorithm model.

Table 3.4: model configuration parameters.

3.6.3.3 Strategy Selection

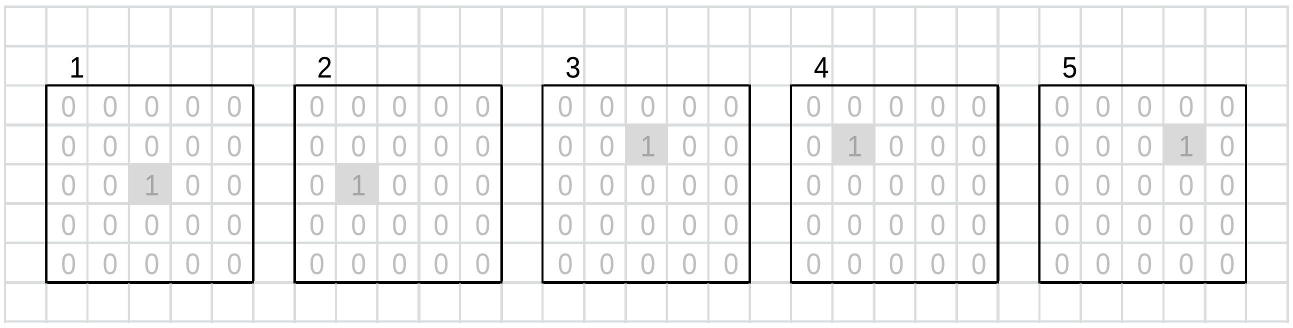
The strategy is selected in the model by means of three configuration files: *strategy.dat, strategy_masks.dat, new_win_masks.dat.*

The chosen approach in the high-level model allows more complicated solutions with respect to the final ones. Dynamic neighbor addition is possible, while in the final solutions the processing is the same for all the neighbor configurations in the DT acquisition phase. The clustering is always carried out without neighbors and they are added at the end of the processing, according to the neighbor inclusion flag.

3.6.3.3.1 Strategy_masks.dat

The file strategy_masks.dat contains the masks used to explore the neighborhood of the event. When a new event is analyzed, the model explores the *5x5* area around it in order to find candidate windows for clustering. The masks contain the pixels that the model has to check on the window.matrix. If all the pixels indicated by the mask are under the same window an expansion can be attempted.

Figure 3.37 shows the strategy masks for the *RDL* strategy.



$$\overset{1}{\begin{matrix}0&0&0&0&0\\0&0&0&0&0\\0&0&1&0&0\\0&0&0&0&0\\0&0&0&0&0\end{matrix}}\quad\overset{2}{\begin{matrix}0&0&0&0&0\\0&0&0&0&0\\0&1&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{matrix}}\quad\overset{3}{\begin{matrix}0&0&0&0&0\\0&0&1&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{matrix}}\quad\overset{4}{\begin{matrix}0&0&0&0&0\\0&1&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{matrix}}\quad\overset{5}{\begin{matrix}0&0&0&0&0\\0&0&0&1&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\end{matrix}}$$

Figure 3.37: strategy masks for RDL strategy.

The new event is always considered to be in the center of the *5x5* matrix.

To make an example, the program starts from the mask number 1. If all the pixels indicated by '*1*' in the mask have the same ID in the window matrix, then they are under the same window. In this case, the event will be contained in a window. If the neighborhood condition number 1 fails, then the model tries with the next one. With the mask number 2 we check if the event is on the right border of a window. If the neighborhood condition number 2 fails the model will proceed with the third mask and so on. If all the neighborhood conditions fail, a new window is opened.

3.6.3.3.2 strategy.dat

The file strategy.dat is composed by two columns indicating the mandatory and optional expansion directions.

The first column indicates the mandatory expansion directions, i.e., used to include the event under the cluster. The second column indicates the optional expansion directions, i.e., used to include the event neighbors under the cluster. With the final approach, in which neighbors are added at the end of the program, the second column is always set to no-expansion.

Each row in this file corresponds to a strategy rule. There is one rule for each of the neighbor checking masks. If the i-th neighbor check succeeds, then the i-th strategy rule is applied. First, the mandatory expansions are attempted. The optional expansions are carried out if the event is included in the window.

The following is an example of the strategy.dat file, for the strategy *RDL*:

```
N       N
R       N
D       N
RD      N
LD      N
```

N stands for no expansion.

For example, if the second neighbor checking is satisfied, it means that the event is on the right border of the cluster as depicted in Figure 3.38. The algorithm will then expand the window by 1 pixel on the right.

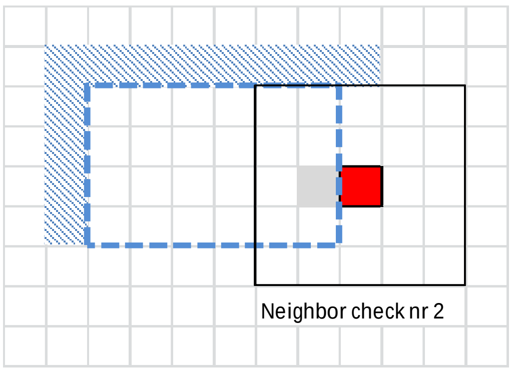


Figure 3.38: algorithm model example.

3.6.3.3.3 New_win_masks.dat

When a new window has to be opened, the *3x3* neighborhood is explored to check the admissible size of the new window, in order to avoid borders and overlap. Since the adopted solution for clustering works without dynamic neighbor inclusion, there is no need to explore the neighborhood and this check is bypassed.

3.6.3.4 Window statistic

The window statistics are calculated on the algorithm output, in order to evaluate the differences among the different clustering strategies. The main window statistics are:

- The total number of windows
- Mean area of windows
- Histogram of the area of windows
- Histogram of the number of DTs per window (Figure 3.39)

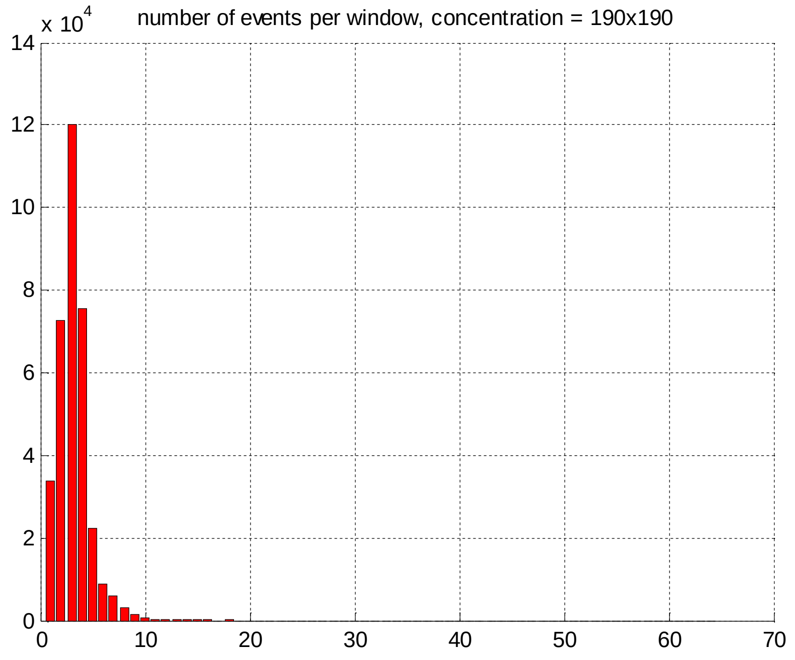


Figure 3.39: histogram of the number of DTs per window.

3.6.3.5 Simulation results

The high-level MATlab model was used to determine the best strategy among *DLR*, *DRLR*, *DRL*, *RLDR*, *RDL*. The results are shown in Figure 3.40.

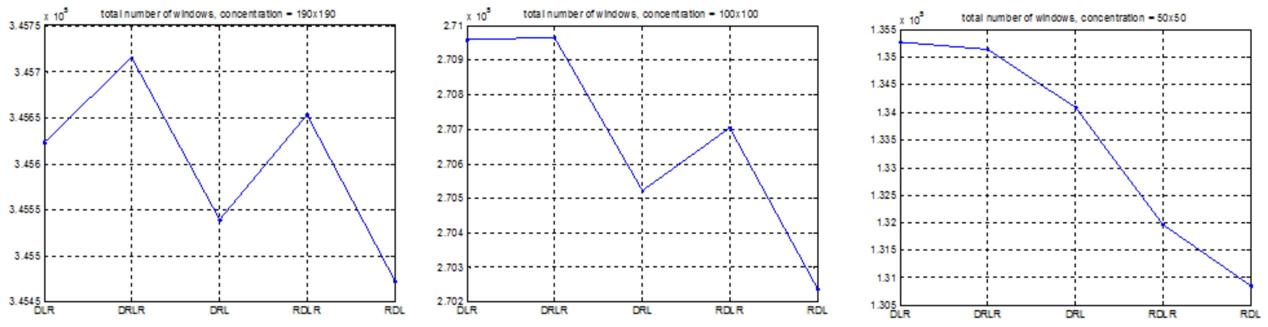


Figure 3.40: Number of windows w.r.t. strategy.

Each result was obtained on 1000 frames without noise. The noise DTs are not necessary for this purpose, since the effects of the different strategies are appreciable only on cluster of events. Different lightning concentration areas were considered: *190x190*, *100x100*, *50x50*. Greater differences are expected on the smaller concentration areas, where the probability of conflicting windows is higher.

RDL is the best strategy in terms of total number of windows. The saving on all the 1000 frames is:

- 243 windows on *190x190* concentration
- 730 windows on *100x100* concentration
- 4420 windows on *50x50* concentration

3.6.3.6 MATlab files

Table 3.5 shows the main files composing the MATlab high-level model.

NAME	PATH	TYPE	DESCRIPTION
LI_clustering.m	high-level	script	Main simulation script
DT_extract_selex_tv.m	common	function	Extract DTs from SES test-vector structure
DT_extract.m	common	function	Extract DTs from image file
expand_window.m	high-level	function	Window expansion
new_window.m	high-level	function	Creates a new window
include_neighbors.m	high-level	function	Includes neighbors into selected windows
image_show.m	common	function	Shows the output of the clustering in a MATlab plot
cluster_stats.m	common	script	Calculates cluster window statistics for clustering strategy analysis

Table 3.5: MATlab model file structure.

3.7 Bit-true MATlab model

After implementation and optimization and the High level MATlab model, the algorithm is divided into blocks for the Bit-true MATlab model. Every block has a particular function in the processing flow of the new DT. Requirements in bit-true MATlab models are the same described in paragraph 3.3.

In MATlab code, every block is implemented like function. Function's syntax in MATlab is:

[*list of function's output, local name*] = <u>*function-name*</u> [*list of function's output, local name*]

Into the function, the functionality of the block is described. Then, in main code, the function is called with syntax:

[*list of function's output, global name*] = <u>*function-name*</u> [*list of function's output, global name*]

Principal blocks implemented in MATlab code are:

- Adjacency Check, *AC*;
- P_Window Check, *PWC*;
- Candidate Window Selection, *CWS:*
  - Candidate Window Selection 1, *CWS_1*;
  - Candidate Window Selection 2, *CWS_2*;
- Expandability Check, *EC*;
- Priority Selection, *PS*;
- Window Update, *WU*;
- Add Neighbors, *AN*.

Algorithm described in Figure 3.21 is implemented with principal blocks in Figure 3.41

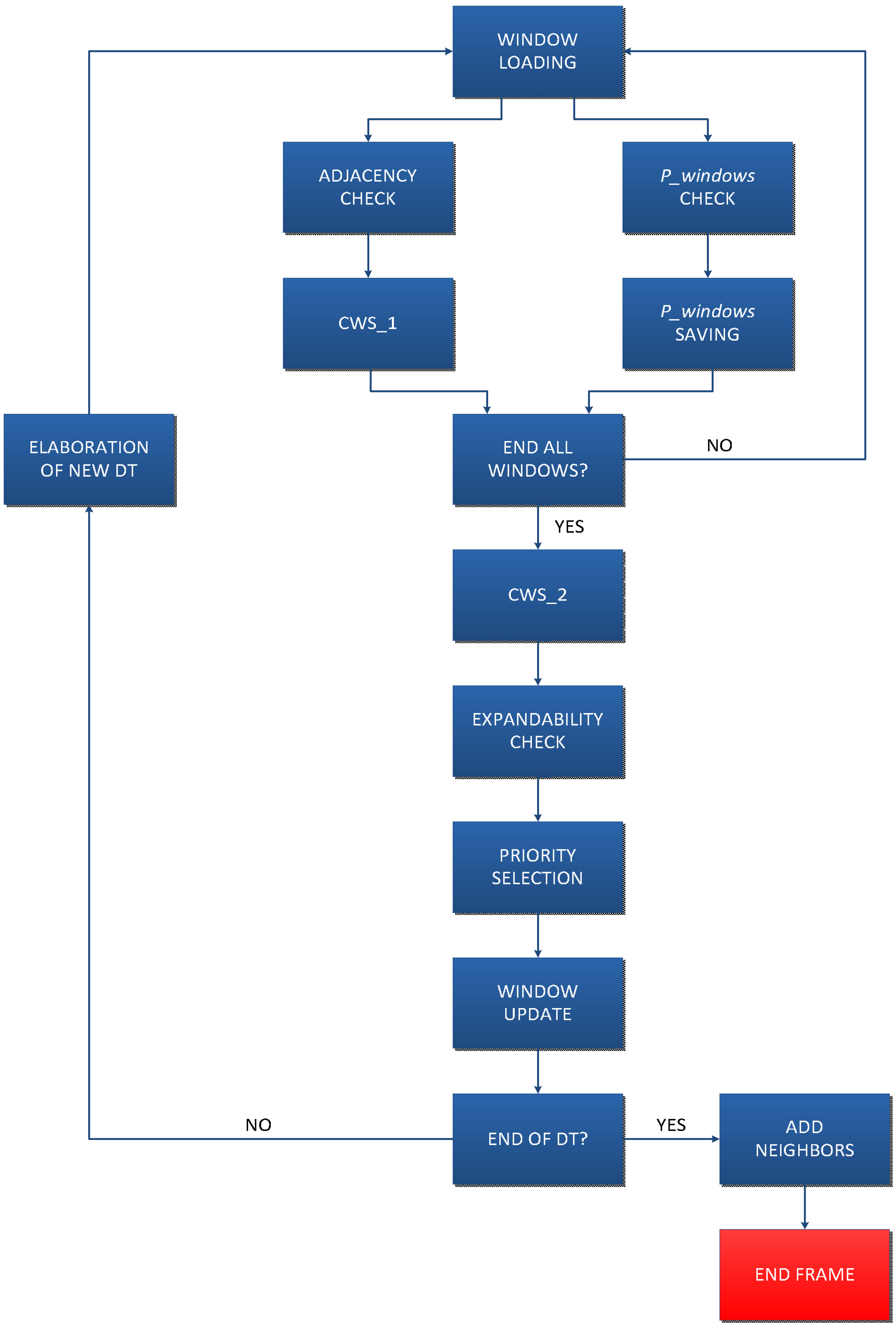


Figure 3.41: algorithm implemented with principal blocks.

In the following, a description of every block and its MATlab bit-true code.

3.7.1 Adjacency Check

Adjacency Check, *AC*, must check if a new DT has one or more adjacent windows and which windows they are. To check a window, this block must compare new DT's coordinates with all window's coordinates and dimensions. The events can be in 4 area, like description in paragraph 3.5:

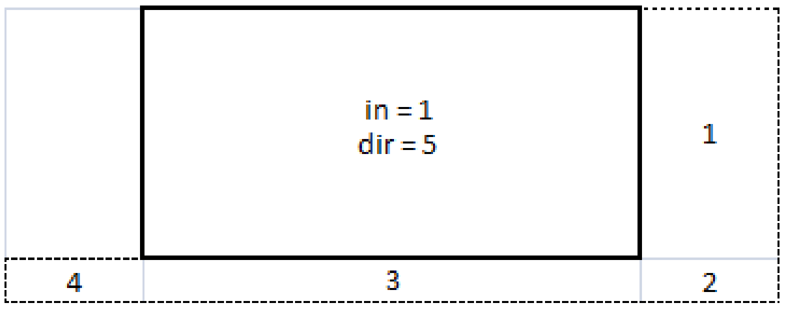


Figure 3.42: allowed new DT position relative to a window.

When a new DT is enclosed in an already existing window, an internal flag *in* is set and dir value, in this case *dir* = 5, is only for information. Because of *dir* position, only one of the directions is possible: *dir*s are mutually exclusive.

Value of coordinates of DT and window in memory are coordinates of high-left corner. So, to check the adjacency, new DT coordinates must be compared with the high-left coordinates of these areas:

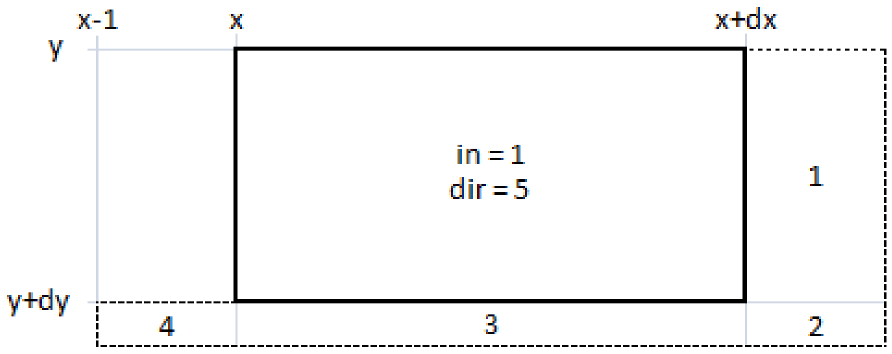


Figure 3.43: coordinates of every adjacent and internal zone.

List of conditions for every direction:

DIRECTION	INTERNAL FLAG	X CONDITION	Y CONDITION
1	0	$X_{DT} = X_{window} + Dx_{window}$	$Y_{window} \leq Y_{DT} < Y_{window} + Dy_{window}$
2	0	$X_{DT} = X_{window} + Dx_{window}$	$Y_{DT} = Y_{window} + Dy_{window}$
3	0	$X_{window} \leq X_{DT} < X_{window} + Dx_{window}$	$Y_{DT} = Y_{window} + Dy_{window}$
4	0	$X_{DT} = X_{window} - 1$	$Y_{DT} = Y_{window} + Dy_{window}$
5	1	$X_{window} \leq X_{DT} < X_{window} + Dx_{window}$	$Y_{window} \leq Y_{DT} < Y_{window} + Dy_{window}$

Table 3.6: conditions for every direction on X and Y axes and internal flag value.

Conditions for every direction must be verified both in X and Y coordinates. If only one of them is not verified for a direction, new DT is not considered in this direction. When no direction is valid, the new DT is not considered adjacent to the window: *dir* and *in* is imposed to *0*.

To maintain the information about the candidate window, candidate *ID* value is assigned to the following registers, called *AC_register*.

In Bit-true MATlab, flags are created to make faster comparison operations between window and new DT coordinates. Every flag is set by 1 if condition is true, to 0 if condition is false.

Flags are resumed in sequent table:

FLAG		COORDINATES CONDITIONS	
AX	1 If	$X_{DT} = X_{window} - 1$	Else 0
BX	1 If	$X_{DT} = X_{window} + Dx_{window}$	Else 0
CX	1 If	$X_{window} \leq X_{DT} < X_{window} + Dx_{window}$	Else 0
DX	1 If	$X_{DT} < X_{window} - 1$ or $X_{DT} > X_{window} + Dx_{window}$	Else 0
AY	1 If	$Y_{DT} = Y_{window} + Dy_{window}$	Else 0
BY	1 If	$Y_{window} \leq Y_{DT} < Y_{window} + Dy_{window}$	Else 0
DY	1 If	$Y_{DT} < Y_{window} - 1$ or $Y_{DT} > Y_{window} + Dy_{window}$	Else 0

Table 3.7: flag's conditions.

Then a series of *IF* structure selects values of *dir*, *in* and *ID* variables. Algorithm for *AC* block is in Figure 3.44:

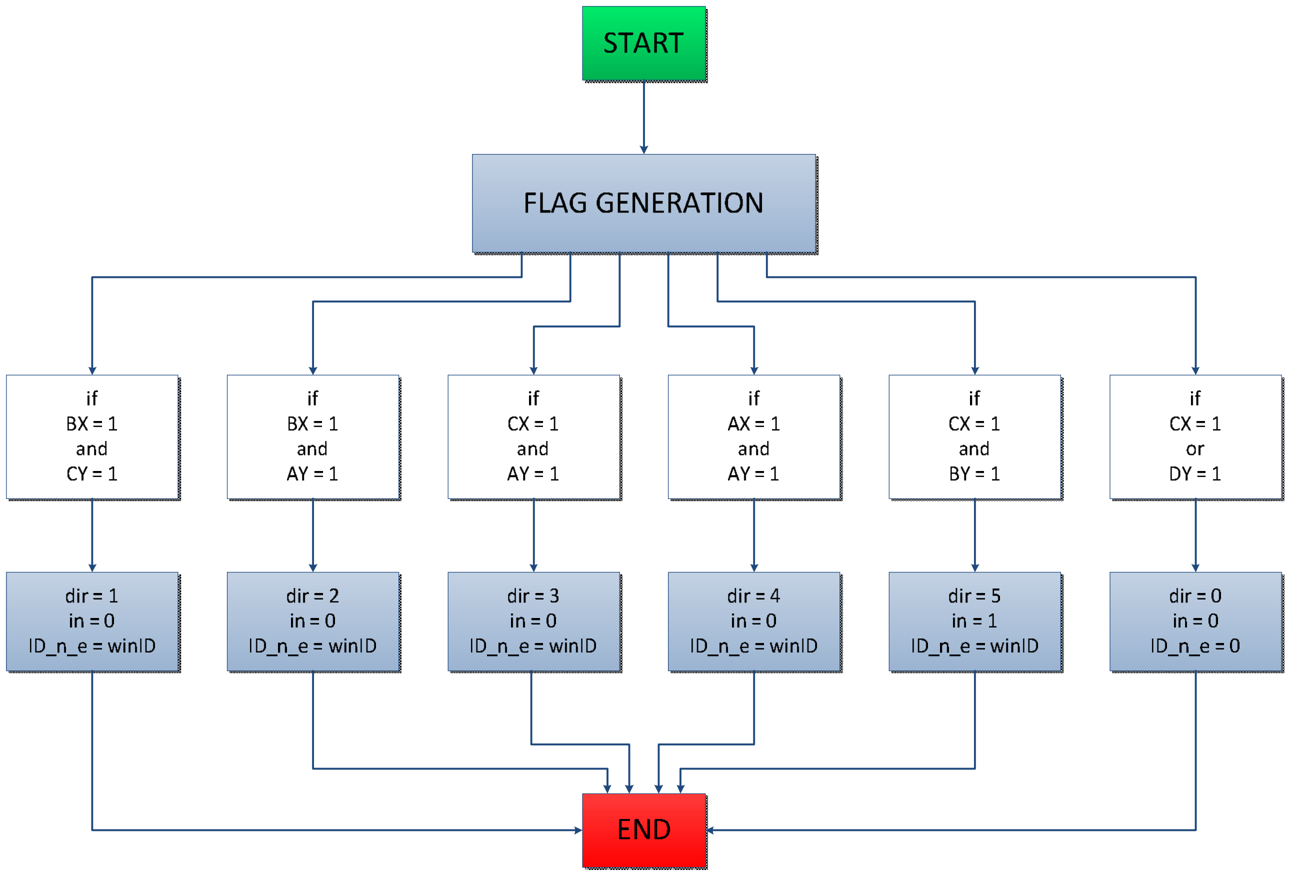

START
FLAG GENERATION
if BX = 1 and CY = 1
if BX = 1 and AY = 1
if CX = 1 and AY = 1
if AX = 1 and AY = 1
if CX = 1 and BY = 1
if CX = 1 or DY = 1
dir = 1 in = 0 ID_n_e = winID
dir = 2 in = 0 ID_n_e = winID
dir = 3 in = 0 ID_n_e = winID
dir = 4 in = 0 ID_n_e = winID
dir = 5 in = 1 ID_n_e = winID
dir = 0 in = 0 ID_n_e = 0
END

Figure 3.44: *AC* algorithm.

As already explained in this paragraph, flag conditions are mutually exclusive and just one *dir* value is assigned. So, only one branch has valid conditions.

Local variable *X_n_e* and *Y_n_e* are the new DT coordinates, *winID*, *window_x*, *window_y*, *window_dx* and *window_dy* are the checked window parameter.

3.7.2 P_Window Check

In the same time of *AC* block, P_Window Check, *PWC*, must check if a window's coordinates is in *P_window* total area. Inputs are the same of *AC* block, coordinates of new DT and coordinates and size of every window, but a counter is necessary to handle the checked *P_windows* in the MATlab dynamic memory. First of all, it is necessary to test if the input window is valid: so, a test on *Dx* and *Dy* value is made. A window is valid if both of them are greater than 0. When a window pass the check, it is stored in a new register called *P_window_register*. Operations in *PWC* block is the same comparison defined in section 3.5:

$$\begin{cases} Y_{DT} = Y_{window} \\ X_{DT} - 14 \le X_{window} \le X_{DT} \end{cases} \quad \text{or} \quad \begin{cases} Y_{DT} - 14 \le Y_{window} \le Y_{DT} \\ X_{DT} = X_{window} \end{cases}$$

If a window passes the check, it is saved in *P_window_register* with a flag of enable (*win_en*) and the counter is increased to allow a right safe of new founded *P_window*. The final step is the *ID* window save in *P_window_register*, together with window coordinates. It will be necessary during the expandability check.

Local variable *A* tests if the window is valid and no empty. Counter and ID variables must have different name in input and output because in the block they change their values.

As shown in Figure 3.41, *AC* and *PWC* blocks must work over all the existing window. For this reason, these two blocks, during the call, must be run in a *FOR* loop from 1 to *MAX_WINDOWS* number. After this loop, *AC_register* and *P_window_register* contain all the information about *Candidate* and *P_window* with ID, coordinates, dimensions and relative position between new DT and window.

3.7.3 Candidate Window Selection

When *AC_register* is ready and all the windows are checked, it is necessary to regroup all the selected windows in a register to make easier the expandability check over *Candidate*: Candidate Window Selection, *CWS*, handle these information. This macro-block is divided in 2 parts: *CWS_1* and *CWS_2*

3.7.3.1 CWS_1

Candidate Window Selection 1, *CWS_1*, takes just valid windows in *AC_register* and save them in a register called *Pos*: every window is positioned in its *dir* position. For example, if a window has *dir* = 2, it is positioned under position 2 in *Pos* register. Because of mutual exclusion between possible direction, only 5 position are available in *Pos* register, one for every direction (from *1* to *5*). Direction *0* is not necessary. In main, this block must be run *MAX_WINDOW* times too, one for every *AC_register* window.

In the code, *AC_register(i,6)* is the *dir* value: if it is greater than *0*, window is copied in *Pos* register and an enable flag is set (*Pos (*,1)*).

3.7.3.2 CWS_2

When *Pos* registers are ready, block Candidate Window Selection 2 (*CWS_2*) takes the windows selected by *CWS_1* and saves them in the *Candidate* register. In main, a *FOR* statement over all the directions, 5, allows to scan all the possible *Pos* window and save them in *Candidate* register. As explained in paragraph 3.5, candidates are only 4.

When a valid window is found, it is copied in one of the four slot of *Candidate* register and *Pos* position is deleted. A counter handles the right position of every transfer.

3.7.4 Expandability Check

Now, 4 or less Candidates are found. For every one of them, it is necessary to check the expandability over all the *P_windows* found: if at least one *P_window* does not allow the expandability of a Candidate, the expandability test fails. A Candidate cannot be expandable if max dimensions are reached too.

As explained in Figure 3.41, outputs of this block are a flag result of expandability test over a Candidate, called *found*, and the variation on the coordinates dependent of positions of new DT and other *P_windows*. These variations are called *delta_x*, *delta_y*, *delta_Dx*, *delta_Dy*. If a Candidate is expandable, at least one of delta values are different from *0*, either all delta are *0.*

MATlab block is implemented for a single Candidate and then it is repeated for all the 4 candidates.

First operation of the block, neighbors flag is tested because it can change window max dimensions, i.e. *MAX_WIN_HEIGHT* and *MAX_WIN_WIDTH*: in fact, if neighbors flag is *1*, max dimension is 6 instead of 8 because of the final neighbors inclusion.

Then, if the direction of Candidate is greater than 0, i.e. Candidate exists, edge values are calculated as shown in Figure 3.45:

- *C_window_x_L*: it is just the value of the right edge of the candidate window because *window_x* is the value of left edge;
- *C_window_y_L*: dually of *C_window_x_L,* it is the value of down edge of the candidate window because *window_y* is the value of high edge;

Edges are necessary to test no-expandability over quadrant borders.

Edges values are used to create some flags: they are assigned with “1” when constrain are violated. Flags of dimension and coordinates constrains are:

- ***W_DY_MAX*** : test *Y* max dimension. If *Dy* is greater than *MAX_WIN_HEIGHT*, it is set;

- ***W_DX_MAX*** : test *X* max dimension. If *Dx* is greater than *MAX_WIN_WIDTH*, it is set;
- ***W_X_0*** : flag is set if left window edge is already on quadrant border;
- ***W_Y_MAX*** : flag is set if down window edge is already on quadrant border;
- ***W_X_MAX*** : flag is set if right window edge is already on quadrant border.

When all flags are assigned, direction boundary flags *R_bound*, *RD_bound*, *D_bound* and *LD_bound* are initialized with logic operation over constrain flags. Boundary flags are set to *1* only if all the conditions over respective edge are *0*; if at least one of condition is *1*, respective boundary flag value is set up to *0*. For example, if *Dy* is equal to *MAX_WIN_HEIGHT, i.e. W_DY_MAX* = 1, *D_bound* is set up to *0* independently of others flag. It means that boundary flags are just a NOR logic operation over respective conditions:

$$LD\_bound = \overline{W\_DY\_MAX + W\_Y\_MAX + W\_DX\_MAX + W\_X\_0}$$

$$RD\_bound = \overline{W\_DY\_MAX + W\_Y\_MAX + W\_DX\_MAX + W\_X\_MAX}$$

$$D\_bound = \overline{W\_DY\_MAX + W\_Y\_MAX}$$

$$R\_bound = \overline{W\_DX\_MAX + W\_X\_MAX}$$

These flags are used to initialize the global expandability flags, because, before the comparison of candidate with all the *P_windows*, it is necessary that conditions over edges of a candidate are verified. Global expandability flags are 4, *R_g*, *RD_g*, *D_g* and *LD_g*, one for every possible direction of expansion. Initialization of Global expandability flags is made with the operations:

$$D\_g = D\_bound$$

$$R\_g = R\_bound$$

$$RD\_g = RD\_bound$$

$$LD\_g = LD\_bound$$

Then, for every *P_window*, the expandability flags are updated. So, a loop of *P_windows* number iteration starts.

A few operations are carried out over a generic *P_window*$_i$ to evaluate Candidate expandability. Principal value of a Candidate are:

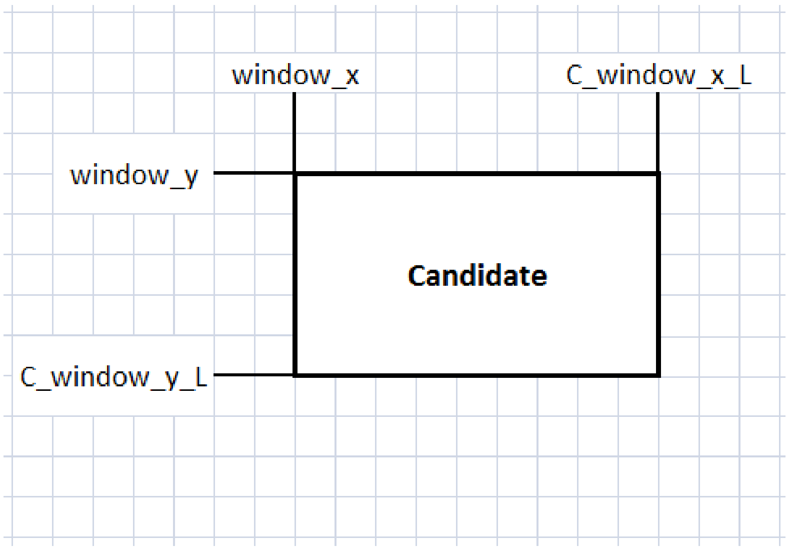


Figure 3.45: edges value of candidate window.

Comparisons between Candidate and a single *P_window*$_i$ are made over all directions with 4 operations:

- ***A*** value is necessary to check on the left side of Candidate. The operation is:

  $A = Px_i + Pdx_i - Window\_x$

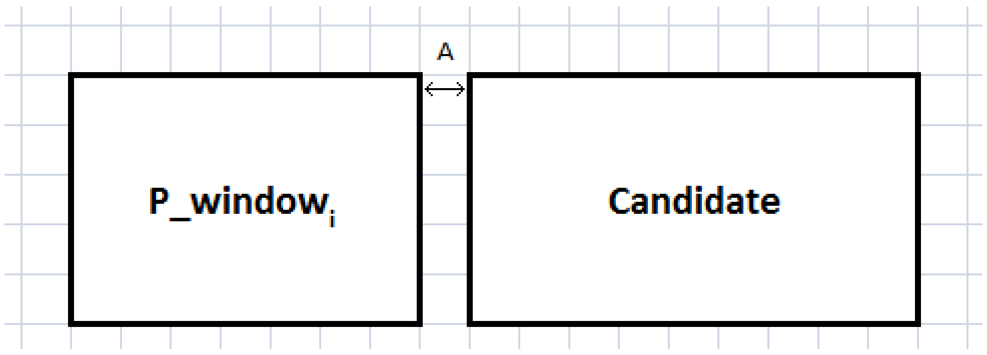


Figure 3.46: schematic of A operation.

- ***B*** is used to check on the right side of Candidate:

  $B = Px_i - C\_window\_x\_L$

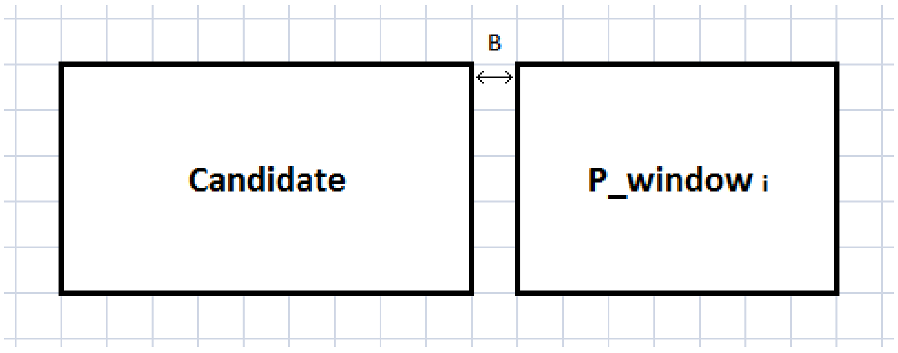


Figure 3.47: schematic of B operation.

- ***C*** to check on up side of Candidate. Dual of A operation:

  $C = Py_i + Pdy_i - Window\_y$

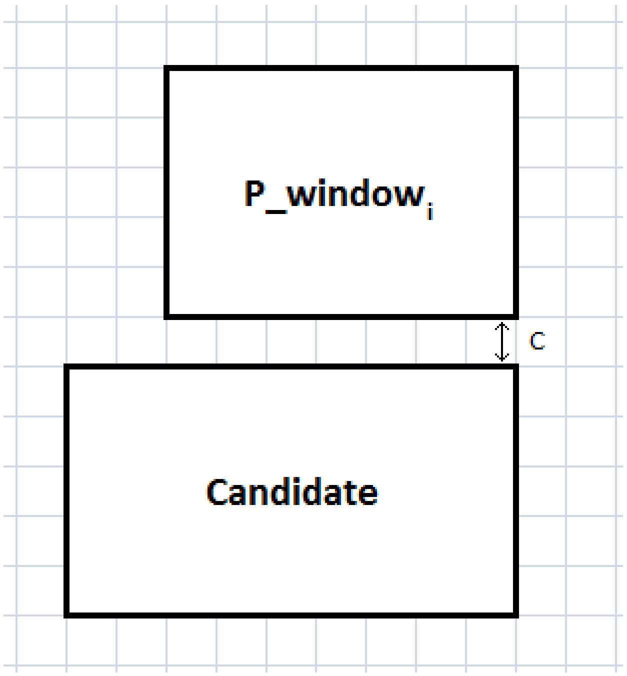


Figure 3.48: schematic of operation C.

- ***D*** for down side of Candidate.

  $D = Py_i - C\_window\_y\_L$

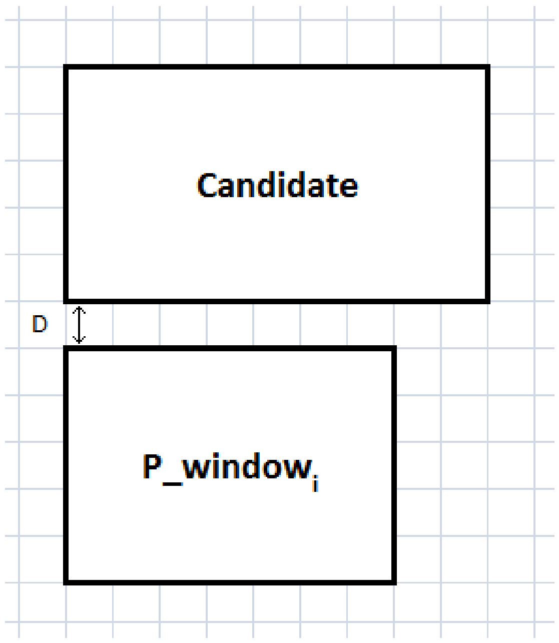


Figure 3.49: schematic of operation D.

From the results of these 4 operations, 7 flags are implemented. They are just to simplify the following operations:

$A_0 = (A < 1)$	$A_1 = (A < 0)$
$B_0 = (B > -1)$	$B_1 = (B > 0)$
$C_0 = (C < 1)$	
$D_0 = (D > -1)$	$D_1 = (D > 0)$

C_1 doesn't exist because possible directions are not symmetric around the Candidate and C_1 is not required.

Thanks to these 7 flags, single *P_window* expandability check *R_win*, *RD_win*, *D_win* and *LD_win* can be calculated. To understand conditions of every flag, it is necessary to subdivide the zone that will allow to pass the expandability check for the current *P_window$_i$*. These zone are highlighted in green in Figure 3.50, Figure 3.51, Figure 3.52 and Figure 3.53:

- ***R_win***. To be expandable in Right direction, *P_window$_i$* must be in at least one of the following positions with respect to Candidate:

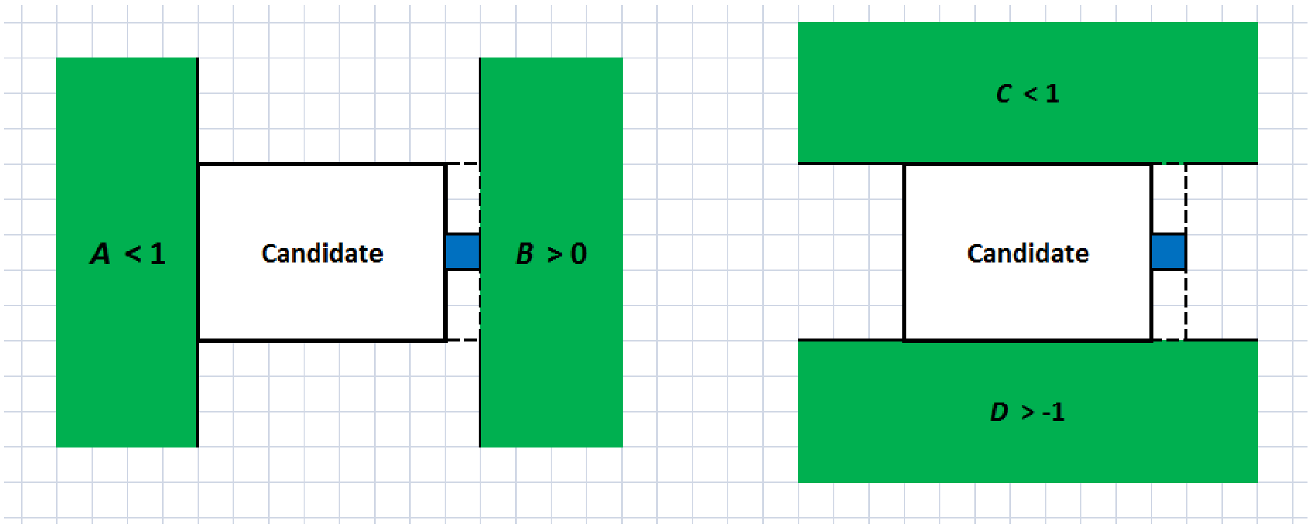


Figure 3.50: allowed *P_window* position for Right expansion and relative conditions.

- ***RD_win***. To be expandable in Right-Down direction, *P_window*$_i$ must be in at least one of the following positions with respect to Candidate:

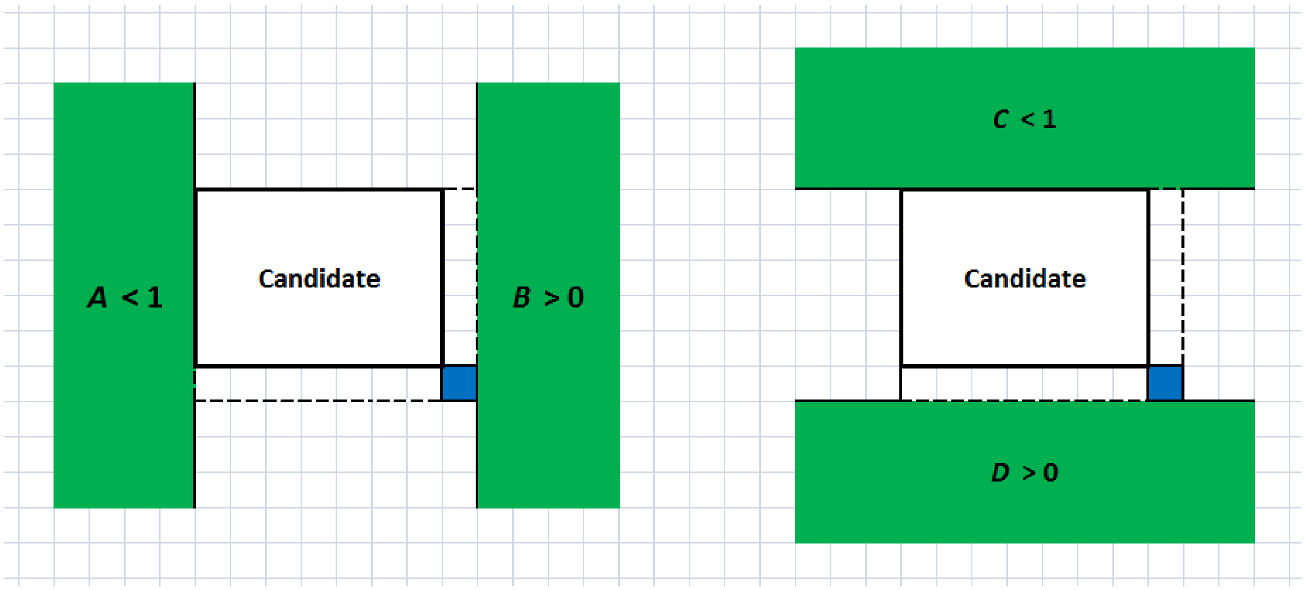


Figure 3.51: allowed P_window position for Right-Down expansion and relative conditions.

- ***D_win***. To be expandable in Down direction, *P_window*$_i$ must be in at least one of the following positions with respect to Candidate:

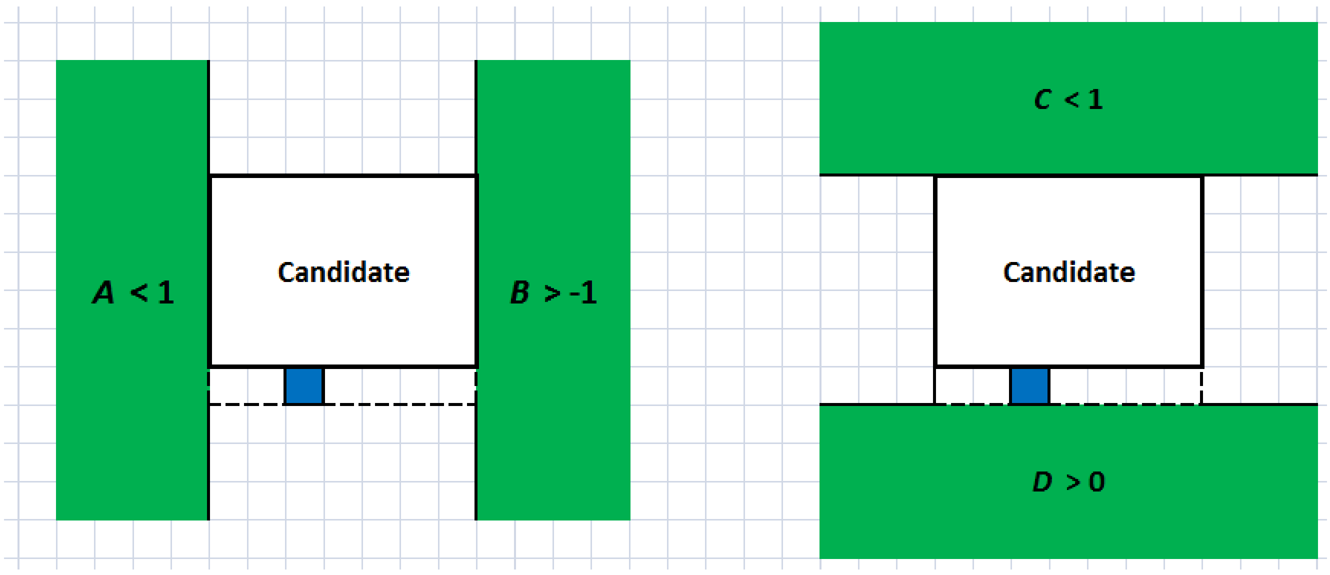


Figure 3.52: allowed P_window position for Down expansion and relative conditions.

- ***LD_win***. To be expandable in Left-Down direction, *P_window$_i$* must be in at least one of the following positions with respect to Candidate:

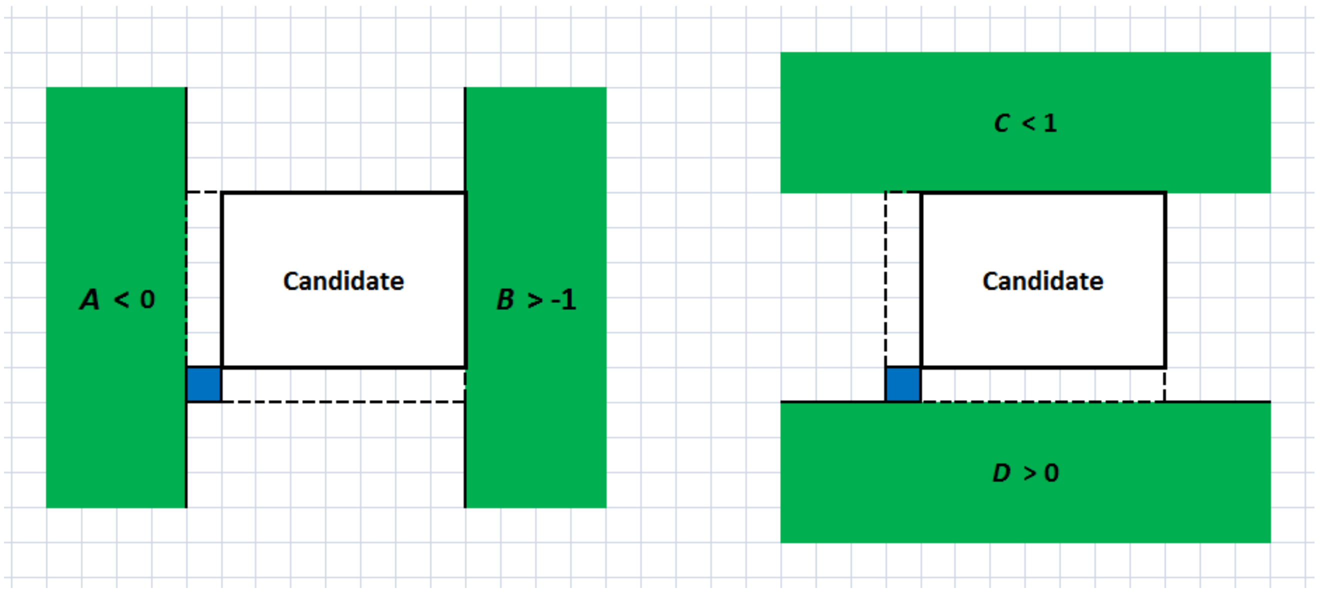


Figure 3.53: allowed P_window position for Left-Down expansion and relative conditions.

So, it is easy to understand that an OR logic between these conditions is sufficient to produce the right value of single *P_window* flags:

$$R\_win \quad = A_0 + B_1 + C_0 + D_0$$

$$RD\_win = A_0 + B_1 + C_0 + D_1$$

$$D\_win \quad = A_0 + B_0 + C_0 + D_1$$

$$LD\_win = A_1 + B_0 + C_0 + D_1$$

When the single flags are calculated, relative global expandability flags are updated with these values. If at least one of the single flags is set to *0*, i.e. just a *P_window* cannot allow expansion to one direction, then the global flag is set to *0* to this direction because, independently on the flag result of all the others *P_window*, this direction is occupied.

For this reason, the last operation of the loop is the update of the global flag with these 4 operations:

$$D\_g \quad = D\_g \quad \cdot D\_win$$

$$R\_g \quad = R\_g \quad \cdot R\_win$$

$$RD\_g = RD\_g \cdot RD\_win$$

$$LD\_g = LD\_g \ \cdot LD\_win$$

The loop is repeated for all the *P_windows*. A *P_window* is considered not valid to update the global flag if is empty , i.e. *win_en = 0*, or if the *ID* value is the same of the Candidate *ID* because the Candidate and the P_window are the same window.

When all *P_windows* are checked, the block is ready to set up the output values. Global flags founded are just the results of comparison between all the *P_windows* and Candidate window but now it is necessary to include the condition over the new DT. This information is resumed in *in* and *dir* values.

Obviously, if new DT is internal to the Candidate, i.e. *in = 1*, the expansion check is automatically passed, *found* flag is set to 1 and *delta* values are zero because the DT is already included in the Candidate window. Otherwise every *dir* value has a mandatory direction of expansion: if this mandatory direction is not allowed, i.e. relative global flag is *0*, Candidate window is not expandable over this direction, *found* flag is set up to *0* and the algorithm check other candidates to look for an expandable window.

A summary of *dir* / mandatory direction is in the table:

dir **VALUE**	**MANDATORY DIRECTION**
1	*R_g = 1*
2	*RD_g = 1*
3	*D_g = 1*
4	*LD_g = 1*

Table 3.8: dir value and relative mandatory direction in expandability check.

The outputs are calculated with a CASE over *dir* variable. Obviously, if no Candidate is expandable and all *found* flags are *0*, a new window is open with the new DT.

For every possible direction of expansion, coordinates and size of the Candidate can change:

EXPANSION DIRECTION	**COORDINATE AND SIZE CHANGES**	**DELTA VALUES**
R	$X_{window} \rightarrow X_{window}$	*Delta_x* = 0
	$Y_{window} \rightarrow Y_{window}$	*Delta_y* = 0
	$Dx_{window} \rightarrow Dx_{window} + 1$	*Delta_Dx* = +1
	$Dy_{window} \rightarrow Dy_{window}$	*Delta_Dy* = 0
RD	$X_{window} \rightarrow X_{window}$	*Delta_x* = 0
	$Y_{window} \rightarrow Y_{window}$	*Delta_y* = 0
	$Dx_{window} \rightarrow Dx_{window} + 1$	*Delta_Dx* = +1
	$Dy_{window} \rightarrow Dy_{window} + 1$	*Delta_Dy* = +1
D	$X_{window} \rightarrow X_{window}$	*Delta_x* = 0
	$Y_{window} \rightarrow Y_{window}$	*Delta_y* = 0
	$Dx_{window} \rightarrow Dx_{window}$	*Delta_Dx* = 0
	$Dy_{window} \rightarrow Dy_{window} + 1$	*Delta_Dy* = +1
LD	$X_{window} \rightarrow X_{window} - 1$	*Delta_x* = -1
	$Y_{window} \rightarrow Y_{window}$	*Delta_y* = 0
	$Dx_{window} \rightarrow Dx_{window} + 1$	*Delta_Dx* = +1
	$Dy_{window} \rightarrow Dy_{window} + 1$	*Delta_Dy* = +1

Table 3.9: resume of coordinates and size change of candidate window and delta output values.

In the code, *C_window_x*, *C_window_y*, *C_window_dx* and *C_window_dy* are coordinates and size of Candidate, *j* is the Candidate ID, *dir* and *in* are position and internal flag of the new DT. *P_window* parameter are:

- *P_window_register(*,1)* is X value;
- *P_window_register(*,2)* is Y value;
- *P_window_register(*,3)* is Dx value;
- *P_window_register(*,4)* is Dy value;
- *P_window_register(*,5)* is ID value;
- *P_window_register(*,6)* is enable flag.

Output are *found* flag, *delta* values and *E_register*, a copy of the candidate window.

Found and all *delta* values are saved in *found_vector* and *delta_*_vector* respectively, to use a loop in *PS* block.

3.7.5 Priority Selection

At the end of all the *EC*, one for every Candidate, expandable windows are found and relative values of delta too. Now it is necessary to decide which direction is the most interesting to expand the window to.

Like explained in section 3.6, direction priority is R -> *RD* -> *D* -> *L*. So, *dir* priority is 1 -> 2 -> 3 -> 4. To check the lowest *dir*, a for loop over all the Candidates is made. The *dir* value and Candidate parameters are saved at each iteration if dir is lower than the saved one. At the end of the loop only the highest priority *dir*, i.e. the highest priority Candidate, can pass to the successive block.

An exception is in case of already internal new DT: if one of the Candidates already contains the new DT, it is the selected candidate and no changes will occur over its coordinates and size.

If no Candidate is expandable, a new window is necessary and an unused ID is assigned.

When the expandable window with higher priority is selected, its delta values are chosen to modify the current coordinates and size and they are saved in *var* variables.

For future VHDL implementation, a signal of *wr_enable* is added. It is useful because if number of window is *MAX_WINDOW*, no other window must be added in memory.

In the code, *Var_x*, *var_y*, *var_Dx* and *var_Dy* are the final values of the selected *delta*.

3.7.6 Window update

In Window Update block *WU*, window parameter is updated.

If all the *found* flags are *0*, it means that no Candidate is expandable and a new window is opened with standard parameter: *X* = *X_n_e*, *Y* = *Y_n_e* and *Dx* = *Dy* = 1.

Otherwise, Candidate parameters are updated with *var* values.

Just a little transformation is necessary: *ID* must be increased by one because *ID* starts from *0* instead *memory* matrix starts to index *1*.

In *WU* MATlab code, *Window_number* is a support variable used to save the global number of windows.

When window values are updated, all the *delta* and *var* variables are reset and *ID* value is decreased by one to return to the initial value.

3.7.7 Add neighbors

At the end of the frame, neighbors must be added. Neighbors modes are already described in section 3.4. In the MATlab code, neighbors mode change parameters of every existing window. A resume is shown in Table 3.10:

MODE	DESCRIPTION	OPERATIONS OVER CLUSTER	OPERATIONS OVER SINGLE EVENTS
Disable	windows don't change their parameter and they are transmitted	-	-
Enable	all the windows increase their size by 2 over the two axes (without border overlap) and left-up corner is translated by one up and left pixel	$X_{window} \rightarrow X_{window} - 1$ $Y_{window} \rightarrow Y_{window} - 1$ $Dx_{window} \rightarrow Dx_{window} + 2$ $Dy_{window} \rightarrow Dy_{window} + 2$	$X_{window} \rightarrow X_{window} - 1$ $Y_{window} \rightarrow Y_{window} - 1$ $Dx_{window} \rightarrow Dx_{window} + 2$ $Dy_{window} \rightarrow Dy_{window} + 2$
Selective Enable	neighbors are added only at single event window (a test over window size is necessary)	-	$X_{window} \rightarrow X_{window} - 1$ $Y_{window} \rightarrow Y_{window} - 1$ $Dx_{window} \rightarrow Dx_{window} + 2$ $Dy_{window} \rightarrow Dy_{window} + 2$

Table 3.10: operations over windows for different neighbors modes.

3.7.8 MATlab main code

The model main calls the functions of the blocks, feeds input DT, handles block loops, stores the output of blocks and gives the results of clustering algorithm.

The main is divided in implementation of algorithm parameter, definition of variables, loading of input test-vector and neighbors mode, algorithm execution and analysis of results.

Internal variables are:

VARIABLE NAME	DEFAULT	BRIEF DESCRIPTION
DTnumber	0	Total number of new DT in the frame
window_number	0	Update total number of window at every iteration
num	1	Counter for CWS_2 block
count	1	Counter for PWC block
index	0	New DT counter
var_x	0	Allow to change X memory value
var_y	0	Allow to change Y memory value
var_dx	0	Allow to change Dx memory value
var_dy	0	Allow to change Dy memory value
ID_val	1	Contain the ID of the high priority expandable window
wr_enable	0	Write memory enable

Table 3.11: variables in main code.

Instead, vector / register used in the code to store information about windows are:

NAME	1	2	3	4	5	6	7
memory	*X*	*Y*	*Dx*	*Dy*	-	-	-
AC_register	*X*	*Y*	*Dx*	*Dy*	*ID*	*dir*	*in*
pos	*en*	*in*	*ID*	*X*	*Y*	*Dx*	*Dy*
candidate	*X*	*Y*	*Dx*	*Dy*	*ID*	*dir*	*in*
E_register	*X*	*Y*	*Dx*	*Dy*	*ID*	*dir*	*in*
P_window_register	*X*	*Y*	*Dx*	*Dy*	*ID*	*en*	-
update_register	*X*	*Y*	*Dx*	*Dy*	-	-	-
transmitted_register	*X*	*Y*	*Dx*	*Dy*	-	-	-
delta_x	*delta_x candidate 1*	*delta_x candidate 2*	*delta_x candidate 3*	*delta_x candidate 4*	-	-	-

delta_y	*delta_y candidate 1*	*delta_y candidate 2*	*delta_y candidate 3*	*delta_y candidate 4*	-	-	-
delta_dx	*delta_dx candidate 1*	*delta_dx candidate 2*	*delta_dx candidate 3*	*delta_dx candidate 4*	-	-	-
delta_dy	*delta_dy candidate 1*	*delta_dy candidate 2*	*delta_dy candidate 3*	*delta_dy candidate 4*	-	-	-
found	*found candidate 1*	*found candidate 2*	*found candidate 3*	*found candidate 4*	-	-	-

Table 3.12: content description of all vectors/registers in the code.

After every DT iteration of the algorithm, registers and vectors are reset to prevent errors in new DT.

Like high-level model, the function *image_show* is used to view clustering results on a grid.

3.7.9 Results

To test the implemented MATlab code, random test-vectors are given in input to the algorithm, An example of DT distribution in the quadrant is shown in the image.

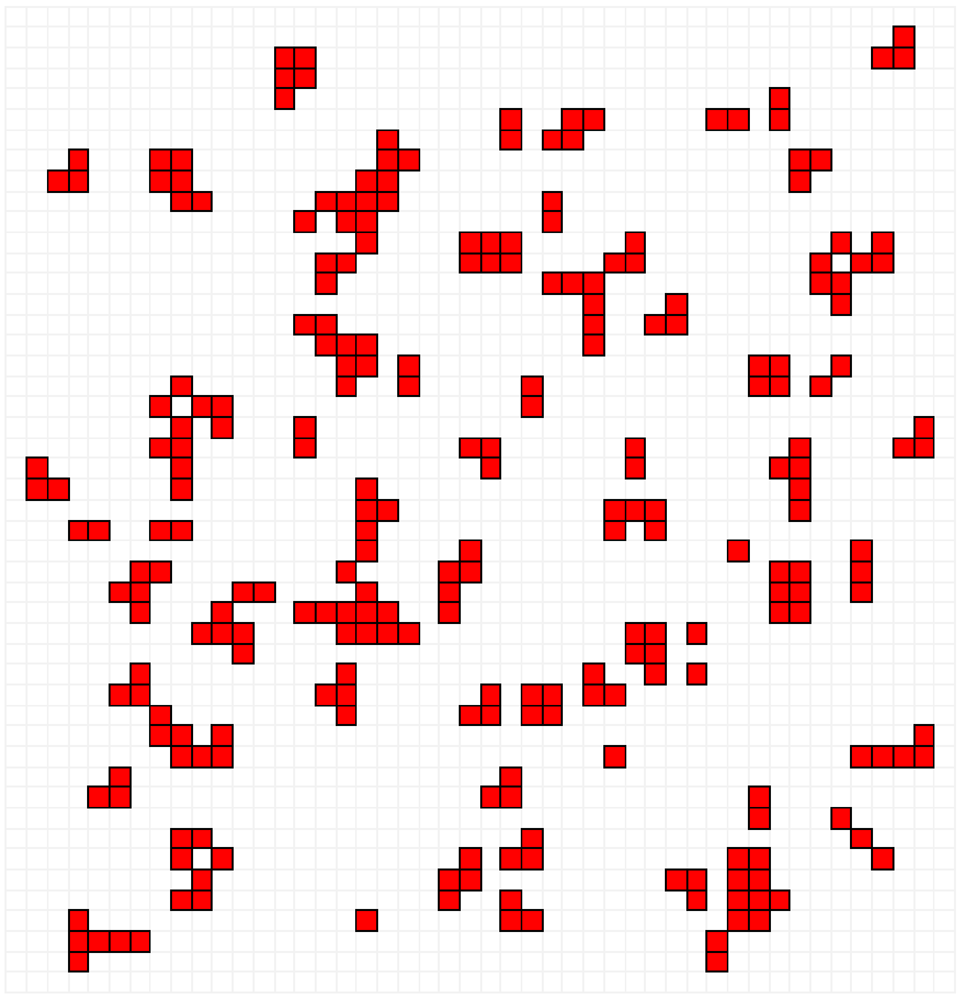


Figure 3.54: DT distribution in a restricted quadrant area. DT's positions in the quadrant are completely random.

The outputs corresponding to this test-vector are 3, one for every neighbors mode.

In neighbors disable mode, added neighbors are not expected:

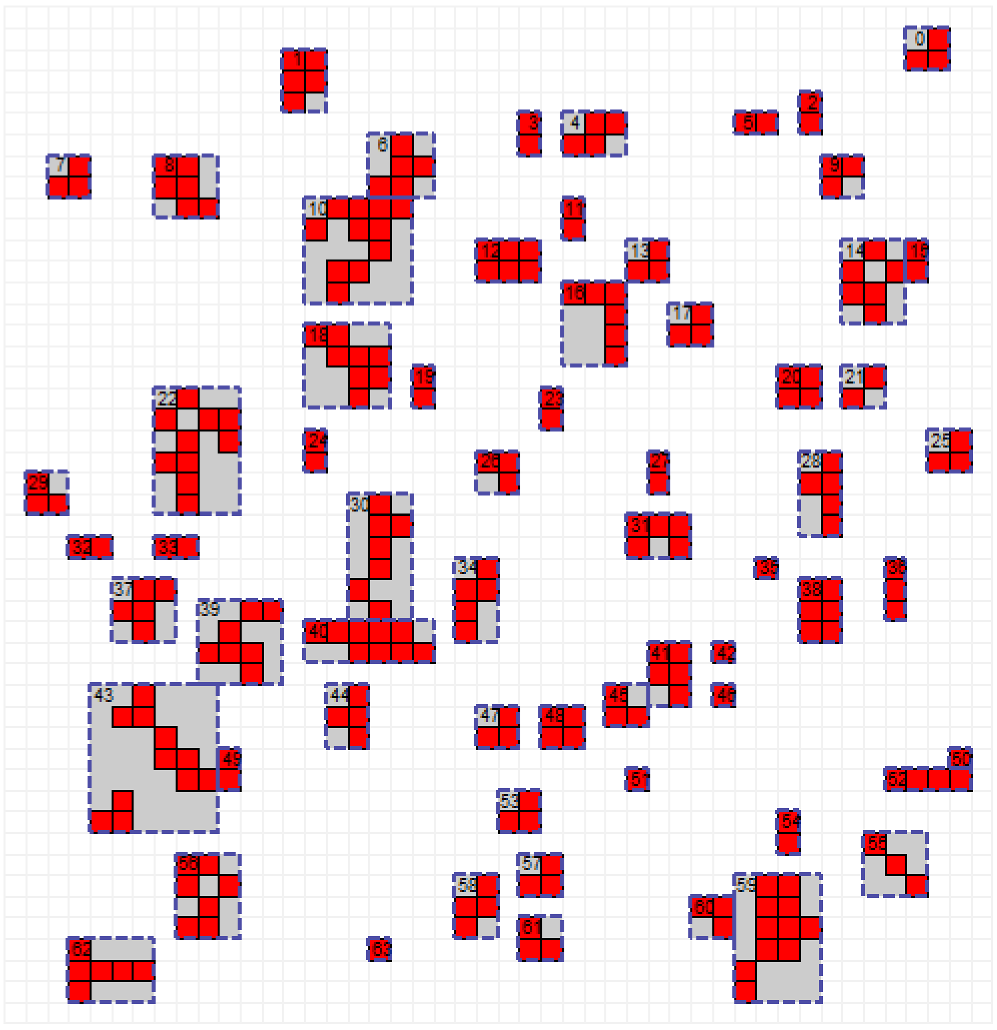


Figure 3.55: algorithm results in neighbors disable mode.

The algorithm meets all the functional requirements. No window exceeds max dimensions, adjacency and geometrical rules are respected and overlap in this mode avoided. The neighbor pixels in Figure 3.55 are only used to maintain rectangular all the windows. ID number start from *0* and increases for every window.

With the same test-vector in input, algorithm gives the output shown in Figure 3.56 when neighbors are enabled:

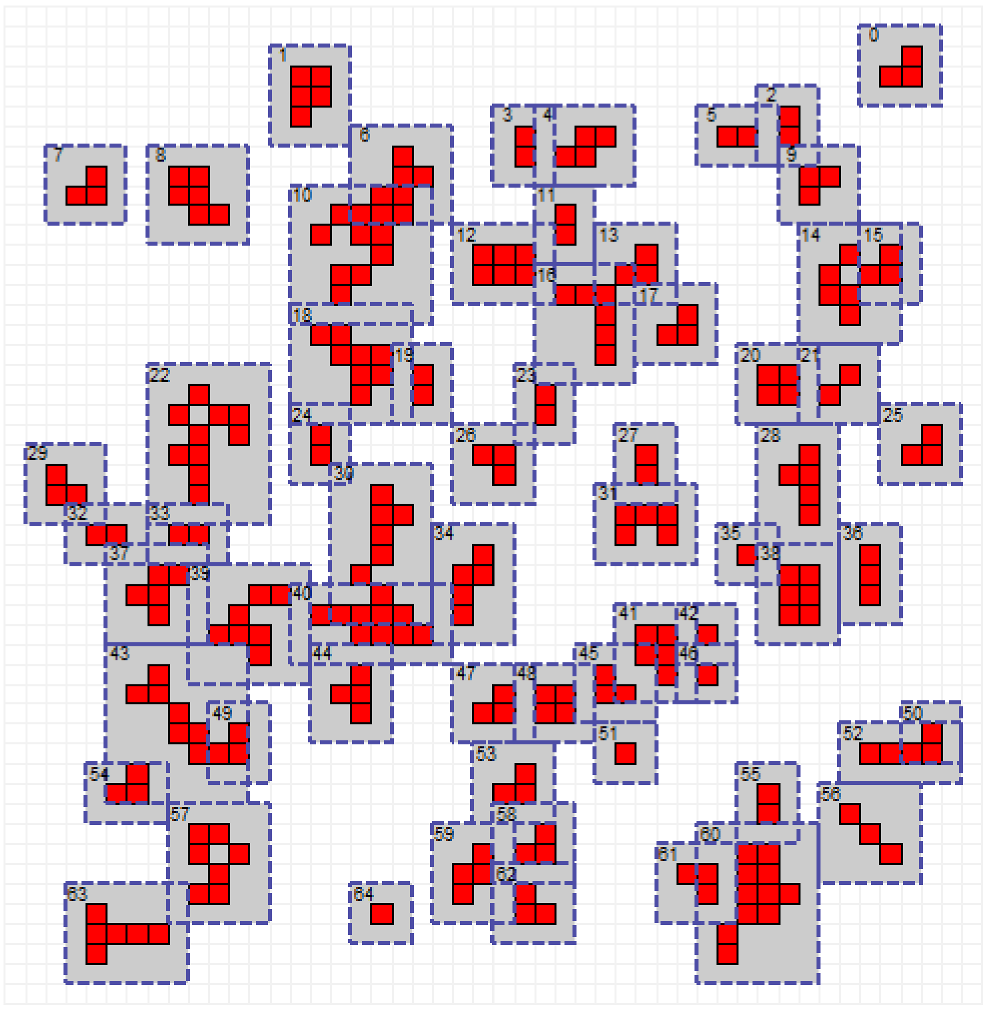


Figure 3.56: algorithm results in neighbors enable mode.

It is possible to see that every window has all the neighbor pixels. In zone of high DT density, windows overlap and overlap dimension is smaller than a *2xN* or *Nx2*with *N* > 2. So, in neighbor enable mode requirements are respected too.

Window number difference between enable and disable mode depends from the max allowed size of a single window: in enable mode max size is 6x6, in disable mode it is 8x8. So, when a window with size 6 can be expanded, if neighbors are disabled, it is expanded but, if neighbors are enabled, a new window is opened. For this reason in disable mode windows are 63 and in enable mode they are 64. In fact, window 43, in disable mode, has a size of 8x7, but this size is impossible in enable mode before neighbors adding: so, a new window is opened.

The third output is the Selective Enable mode, always over the same test-vector:

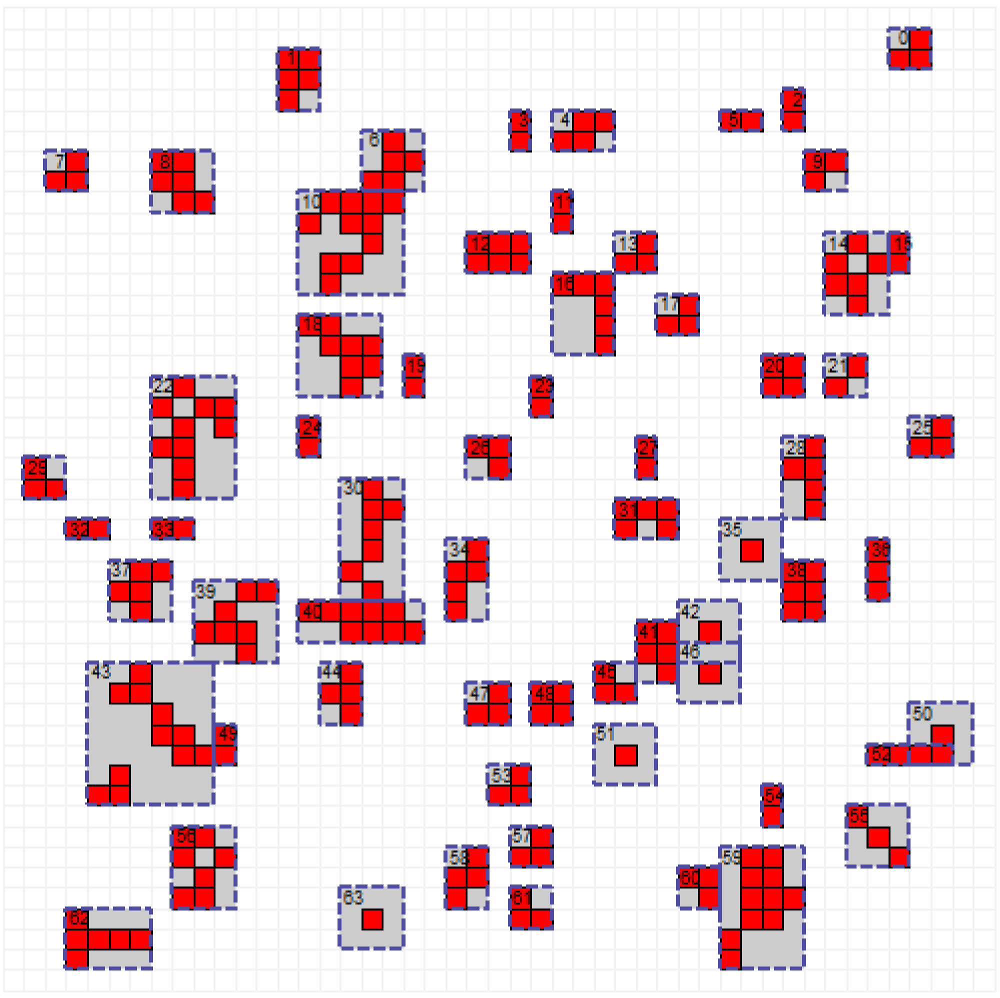


Figure 3.57: algorithm results in Selective Enable mode.

Figure 3.57 shows an output almost equal to Figure 3.55: exceptions are just for Single Events. In disable mode, single events have no neighbors, i.e. hold *1x1* size, in Selective Enable instead they have all the 8 pixels of neighbor, i.e. became *3x3* size.

Now one case is analyzed to prove the fairness of the algorithm.

New DT is in raster order. So, in a particular case, DTs are separated in two windows instead of one. Evolution of data providing is described in Figure 3.58:

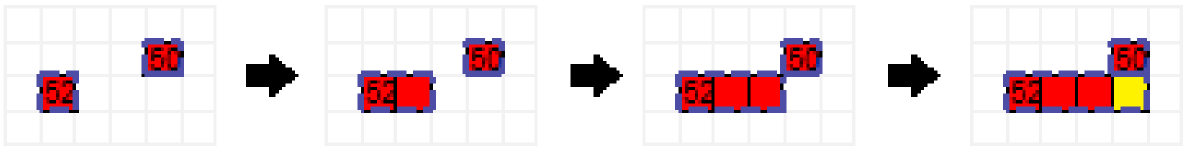


Figure 3.58: evolution of DTs in raster scan and their clustering.

In starting conditions, two windows are present: window 50 and window 52. When a new DT arrives, *AC* found adjacency condition and *EC* allows to window 52 to expand. Same thing for the second new DT. When third event is elaborated, i.e. the yellow pixel, two windows are adjacent to it. Window 50 is adjacent to the new DT in *dir = 3* and window 52 is adjacent to the new DT in *dir = 1*. For *EC* block, both of them are expandable. In this case, *PS* makes a choice over higher priority direction: direction *1* has more higher priority than direction *3*. So, new DT is enclosed in window 52.

3.8 From MATlab bit-true to VHDL model

The VHDL model is a direct consequence of the MATlab bit true model because this MATlab model in low level is implemented to realize the system on FPGA technology. Every block, described in MATlab language in paragraph 3.7, is “translated” in VHDL language because block implementation and rules are the same for both languages but now, all blocks have to be implemented for HW realization.

First of all, window’s parameters have to be saved in few memories. So every parameter needed a standard number of bits. Main parameter for the system are the following, with their range, their equivalent length in bit and length constant name:

PARAMETER NAME	RANGE	BIT LENGTH	LENGTH NAME
X	0 – 584	10	*x_bit*
Y	0 – 499	9	*y_bit*
Dx	0 – 8	4	*d_bit*
Dy	0 – 8	4	*d_bit*
ID	0 – 75	7	*ID_bit*
dir	0 – 5	3	*DIR_bit*
delta	-1 – +2	3	*delta_bit*
in	0 – 1	1	-
neighbors	0 – 2	2	*NEIGHBORS_bit*
en	0 – 1	1	-

Table 3.13: principal parameter in VHDL model.

When all main parameter’s length are defined, it is necessary to make an initial HW description to understand which the main problems and constrains in RTAX4000S architecture are.

The main memory is the memory where all new windows or modified windows are saved. For each window, *X*, *Y*, *Dx* and *Dy* must be stored. The total number of windows is 75, called *N_win_MAX*, in every quadrant. Constant bit’s length of every window, i.e. *x_bit* + *y_bit* + 2**d_bit*, is called *window_reg_width* and its value is 27. So, the total number of bits necessary in main memory are *N_win_MAX* * *window_reg_width*, i.e. 2025 bits. This is a not a big quantity of information but they must be accessed rapidly: the main memory is contained in memory RAM blocks. RAM in RTAX4000S can be accessed with a clock up to 500 MHz. [25] Because of the algorithm high area occupancy in the FPGA architecture, this frequency value is not expected but a 250-200 MHz RAM frequency clock can be reached. All others blocks in the algorithm can be set on a different value of frequency, about 40-50 MHz: a higher value of frequency is not expected. With a clock of 50 MHz, i.e. a clock period of 20 ns, in 1 ms 50000 edges clock are used. These 50000 must be sufficient to elaborate all the 250 new DT in a frame per quadrant but they are not homogenously divided in all the 250 new DT because, for the first DT in a frame, no windows have to be loaded in the algorithm to make comparison and, instead, the last DT imposes that all windows must be loaded in the algorithm. Furthermore, it is impossible to load all *N_win_MAX* windows for every DT in 50000 clock's events: a parallelization is necessary to reduce elaboration time of every DT and optimize the algorithm.

The first idea is to reserve a single clock's event for every step of the algorithm, window load excluded, and then make an optimization to reduce global clock numbers to follow requirements.

A frequency of 200 MHz is imposed for RAM clock and 50 MHz is imposed for all others blocks. So, *4* windows can be loaded in parallel for *AC* and *PWC* comparison as shows in Figure 3.61. Number *4*, i.e. 200/50, is defined as *AC_MAX* in VDHL parameter. But this strategy is difficultly achievable because a too high frequency is required for RAM block. An alternative solution is a new RAM implementation. Instead of a single RAM block with *N_win_MAX* windows with frequency of 200 MHz, 4 block RAM with *N_win_MAX/4* windows, rounded up, can be instantiated and frequency RAM can be decreased to 50 MHz. This strategy is possible because RAM block occupation is very small compared with RTAX4000S resources (120 blocks).

With this strategy, every RAM block can load a different window for every block *AC*/*PWC*:

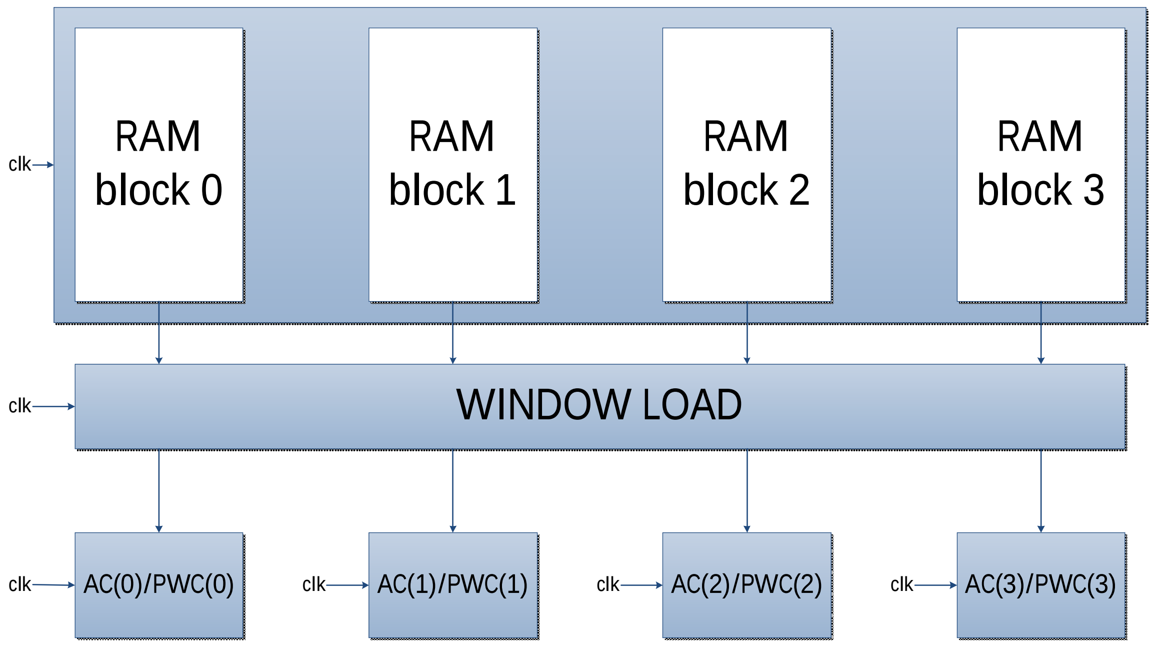


Figure 3.59: solution for *WL* and RAM implementation with frequency of 50 MHz.

Furthermore, with this strategy *WL* block is easier to implement because only one loading-state is necessary. Just the input/output registers are more than before: first of all, in *WL* arrives in the same time 4 windows instead of 1. Moreover, pointers are necessary to loading and updating windows : in fact, RAM division needs a pointer to identify which RAM block contains the window and an offset in the selected RAM block.

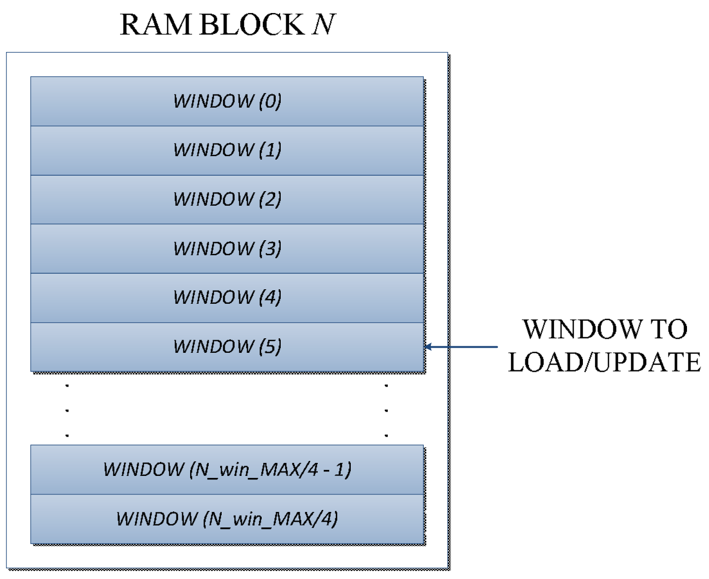


Figure 3.60: localization of a window to load or update.

As shown in Figure 3.60, a RAM block *N* contains the window to load/update in location number 5. So, two variable, *RAM_block_number* and *offset*, are necessary to identify this window. In this case, *RAM_block_number* = *N* and *offset* = 5.

In each algorithm, *AC_MAX* blocks of *AC* and *PWC* can be instanced to work in parallel and compare *AC_MAX* window in parallel for every new DT.

When the output of *AC* and *PWC* blocks is ready , it is saved in registers. Output of *AC* block, called *AC_register* as MATlab model, must contain parameter *X*, *Y*, *Dx*, *Dy*, *ID*, *dir* and *in*: total length of *AC_register*, called *AC_reg_width*, is 38 bits. Same condition in *PWC* output register, called *P_reg_width*: it must contain *en*, *X*, *Y*, *Dx*, *Dy* and *ID* for a *P_reg_width* length of 35 bits.

The output of all the *AC_MAX* blocks in parallel, i.e. *AC* and *PWC* outputs, must be handled to be saved in an ordered and limited register to reduce the elaboration time of successive blocks: this is the function of *CWS* and *PWS* block. In VHDL model, *CWS_1* and *CWS_2* are synthesized in a single *CWS* network to reduce the area and the clock cycles necessary for the processing of every new DT: in fact, *CWS* can make the same function of both *CWS_1* and *CWS_2* in a single clock event instead of two. Furthermore, with *CWS* and *PWS* blocks, external position counters are no more necessary. But *CWS* and *PWS* blocks are high complexity blocks for the redistribution of data: they must be optimized as possible because, with *EC* block, they could reduce the clock frequency and could occupy too much area into the device.

When all the windows are analyzed by *AC-CWS* and *PWC-PWS* branches, *EC* can check the expandability of all the candidates. In output of every branch there is a matrix with the result of the all checks.

Output of the *AC-CWS* branch is an array of *candidate_reg_width*EC_MAX* bits, where *EC_MAX* is the maximum number of possible Candidate, i.e. *EC_MAX* = 4, and *candidate_reg_width* is the bit's length of the register, i.e. 38 bits.

Instead, output of *PWC-PWS* branch is an array of *P_reg_width*N_P* bits, where *N_P* is the maximum number of possible *P_windows*, i.e. *N_P* = 14, and *P_reg_width* is the bit's length of the register, i.e. 35 bits.

Both arrays are used as input in the *EC* block. To have a fast algorithm, the *EC* block can be parallelized: in fact, in the worst case, there are only *EC_MAX* candidates. So, *EC_MAX* block of *EC* are instantiated and they work in parallel over the found candidate.

EC is the block with more logic-arithmetic operations and input's data and this block is instantiated *EC_MAX* times: so, it is the block which must be optimized for area and timing implementation.

In output of every *EC* block there is an array of *E_reg_width* bits, called *E_register*, composed by *X*, *Y*, *Dx*, *Dy*, *ID*, *dir* and *in*, i.e. 38 bits, the *found* flag and *delta* values for coordinates and size changes, *delta_bit*s for each one. *E_register*, *delta* values and *found* outputs of every block are concatenated in a single array: so, for example, global *E_register* is composed by *E_reg_width* * *EC_MAX* bits.

All the outputs of *EC_MAX* blocks *EC* are taken in input on the *PS* block. This one is quite simple but a combinatory network is necessary to find the higher priority Candidate in only a single clock event. In output of *PS* block there are *update_register* of *window_reg_width* bits, selected *delta* values and *found* vector.

The *PS* outputs are passed to *WU* block that give ID and data to RAM memory to change window value or add a new window.

At the end of the frame, the *AN* block change values of window's parameter because of neighbor mode and send results out of the algorithm.

Obviously, all blocks and timing are controlled by a state machine called Global State Machine, i.e. *GSM*. This machine handles *enable* and *ready* signals of every block in the algorithm: these two signals will be used to give a correct clock, a "start" and receive an "end" from every block. The machine must work in parallel with the others blocks because all signals must be supervised by *GSM*.

A concept implementation of the algorithm is shown in Figure 3.61:

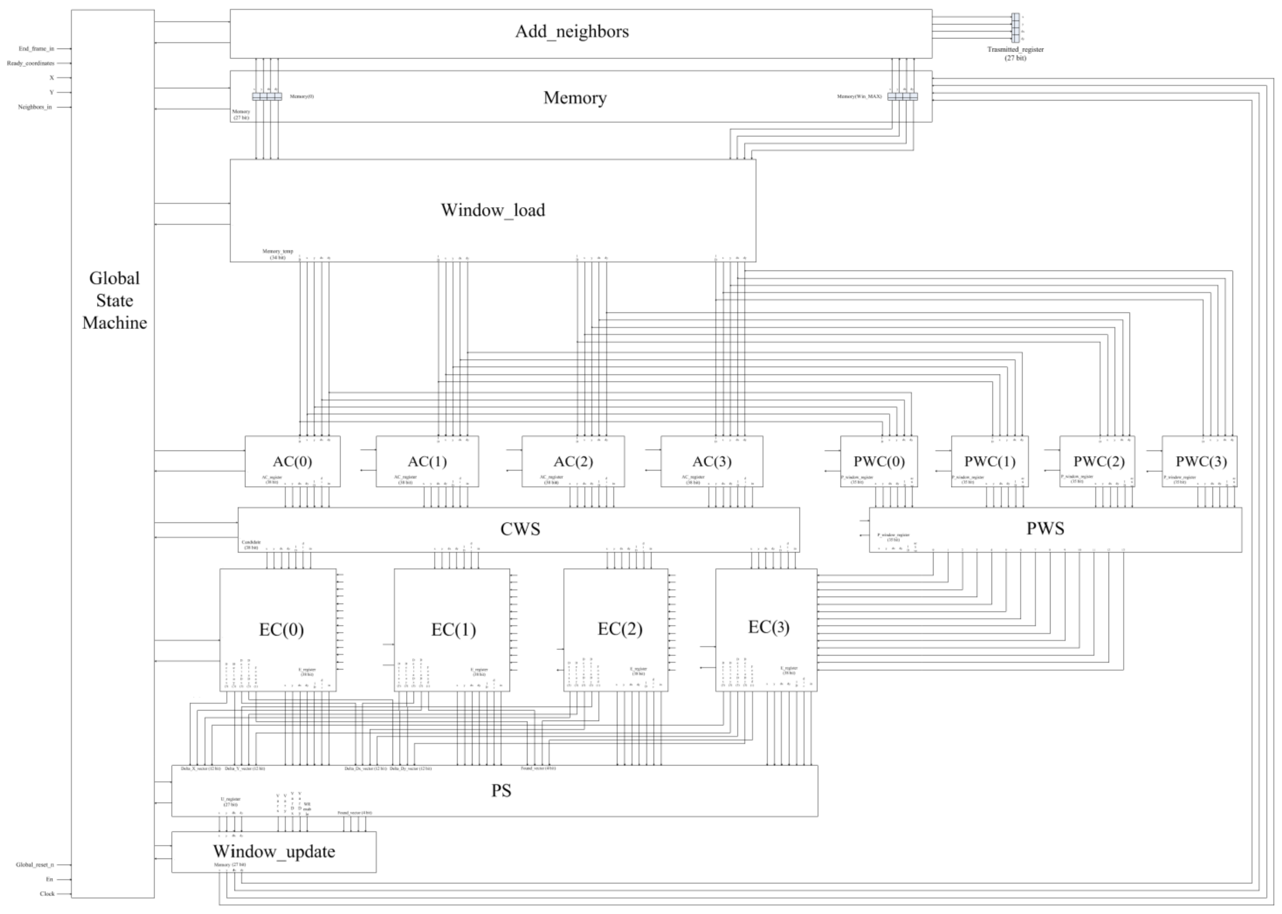


Figure 3.61 : concept of the clustering algorithm implementation. A parallelization is necessary in memory, *AC*, *PWC* and *EC* blocks. Number 4 of parallel *AC* and *PWC* blocks is defined *AC_MAX* instead number 4 of parallel *EC* block is defined *EC_MAX.*

The code is not fully implemented in VHDL model because it is just a prototype of a possible VHDL code of clustering algorithm and the principal function is to evaluate the feasibility of the algorithm in hardware FPGA resources.

Now every block is analyzed to determinate control signal, interface with others blocks and make an area/timing synthesis.

3.8.1Adjacency Check

The function of *AC* block is the same of MATlab model: search in neighbors pixels to found an adjacent window. Also inputs and outputs are the same, but a control interface is necessary to give a right timing to the block. Inputs control are *clk*, *reset_n* and *en*, output control is just a ready signal, *ready_AC*.

This block must be executed in a single clock cycle: so, internal operations and comparisons must be in combinatorial logic. Registers are used only for inputs and outputs.

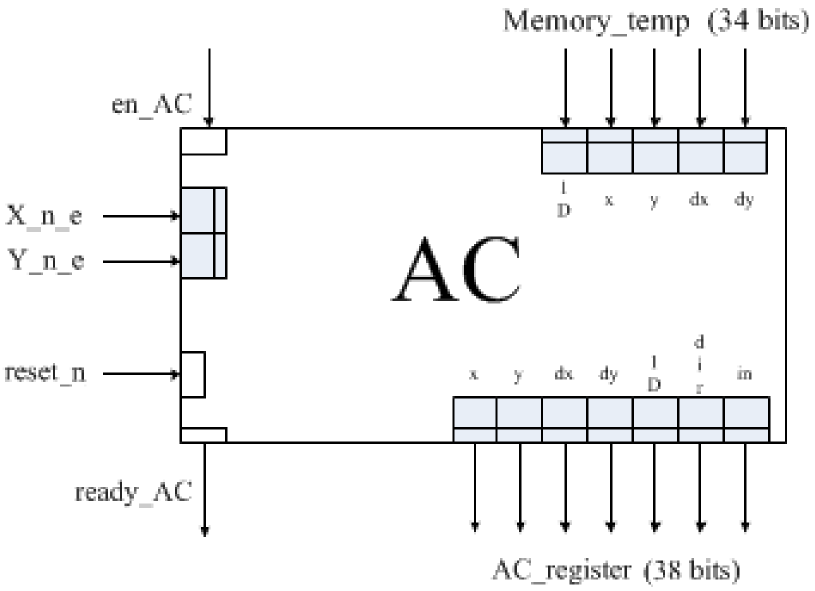


Figure 3.62: black box of *AC* block with its interfaces.

en_AC is a "start" signal given by *GSM* and *ready_AC* is a "ready" signal for *GSM*. *Clk* is the timing signal and *reset_n* is necessary to restart the block when the frame is finished Others inputs and outputs are the same of MATlab model. *AC_MAX* blocks are instantiated in the algorithm, each one with their *enable* and *ready* signals: enables are the same for all blocks but ready signal is different for every block. Just when all the *AC_MAX* blocks are ready, the *AC* step can be considered finished. So, an *AND* logic operation between the *AC_MAX* ready is necessary to give the *ready_AC_tot*:

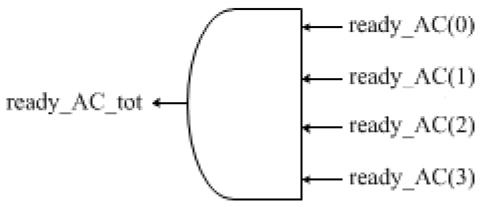


Figure 3.63: logic AND operation for *ready_AC_tot*.

Instead, *clk* signal is the same for every *AC* block because they must work in parallel and their temporization must be the same.

Operations in the block are almost the same of MATlab model with the difference that operations are executed in processes, flags, i.e. the same of MATlab bit-true model, are concatenated in an array and a case is made over it instead of an *IF* chain.

A special attention should be given to arithmetic operations: in fact, the correct number of bits must be used for operators and results.

3.8.2 P_Window Check

As *AC* block, also *PWC* block is the same of MATlab model. Interfaces with *GSM* machine and timing requirements are the same of *AC*; obviously, operations and outputs are different but the structure of the block is equivalent.

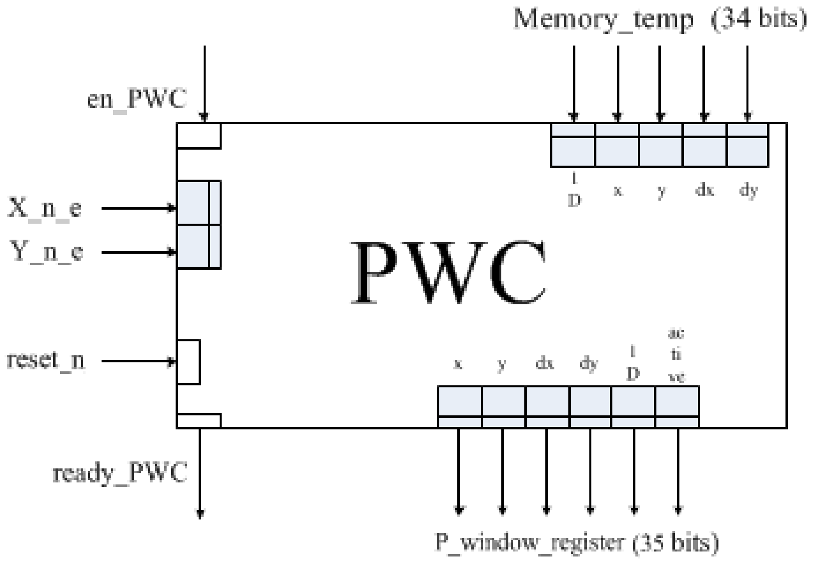


Figure 3.64: black box of *PWC* block with its interfaces.

How Figure 3.64 shows, “start” and “ready” signals, i.e. *en_PWC* and *ready_PWC*, *clk* and *reset_n* are equivalent to *AC* block.

Also in this block, “start” are the same for all the *AC_MAX* blocks instantiated and *PWC* step is considered “ready” when all blocks are “ready”. In Figure 3.65 is possible to see that operation:

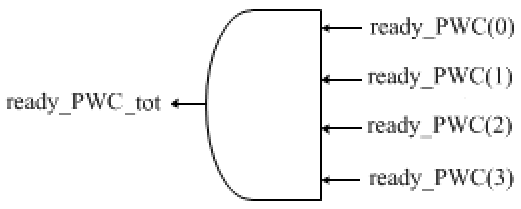


Figure 3.65: logic AND operation for *ready_PWC_tot.*

3.8.3 Candidate Window Selection

As explained in introduction of Paragraph 3.8, *CWS* becomes a single block to improve the optimization of the algorithm. This block must order results of the *AC_MAX* parallel *AC* blocks in continuous registers to give correct results to *EC* blocks. Operation is explained with an example. Consider the initial condition in Figure 3.66:

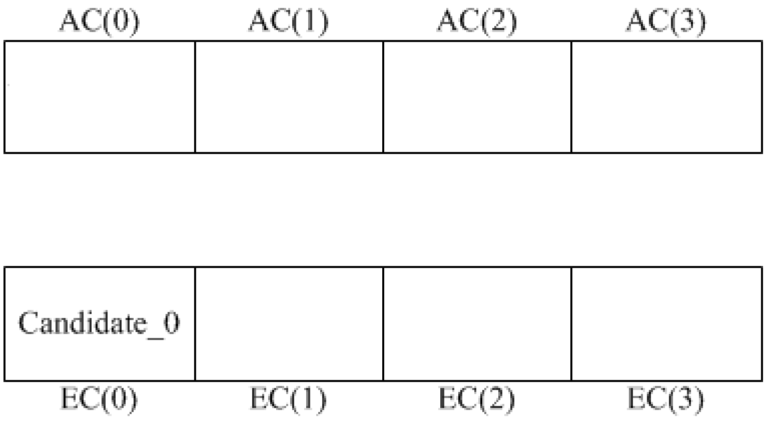


Figure 3.66: initial condition before *CWS* run.

If *AC* blocks found two possible Candidate, they must be inserted in successive register's locations after *EC(0)*:

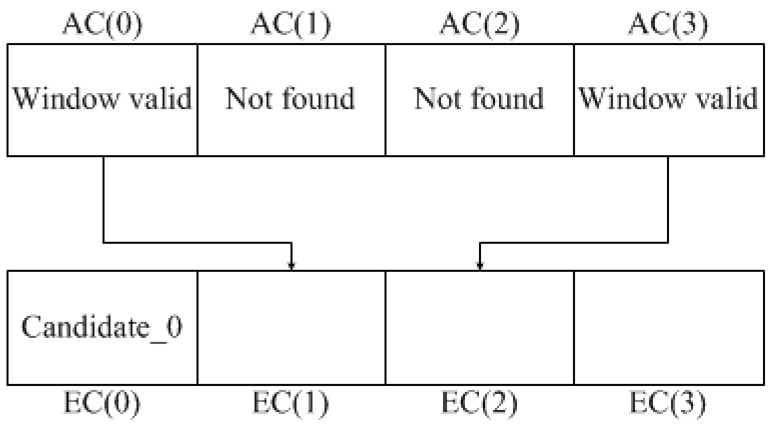


Figure 3.67: correct position of founded Candidates in candidate register.

So, the correct situation after *CWS* run is shown in Figure 3.68:

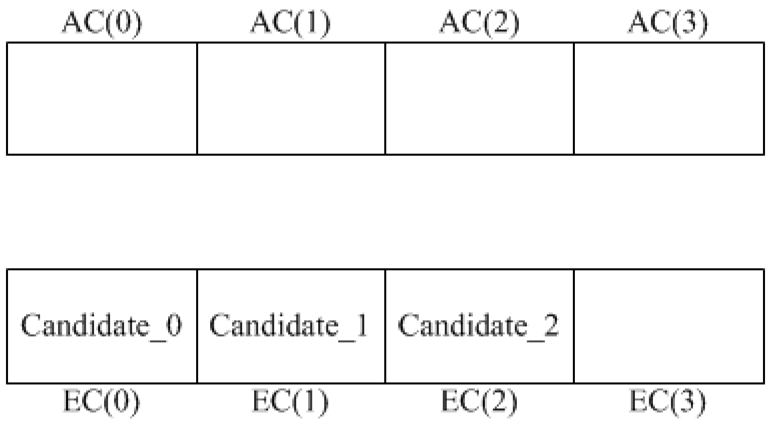


Figure 3.68:final situation of founded Candidate after *CWS* run.

CWS is composed by two part: a network to choose the right *AC* position and a network to write output registers.

To have a correct *AC* position, two arrays are necessary:

- *num*: it is used as counter to memorize total founded Candidates. This value is token in output and then re-put in input for the successive iteration of *AC_MAX* windows;
- *en_vector*: this array is necessary to save which *AC* block is valid, i.e. a window is found;
- *address_vector*: it is used to memorize the address of empty location to memorize found Candidates.

If an *AC* block finds a Candidate, for example *AC(2)*, *num* is increased by one, *en_vector(2)* is set to 1 and *address_vector* is increased by one. When *AC* valid windows are identified, a logic network re-address the *AC_register*s in the *candidate register*s.

CWS block has the same interface of others blocks:

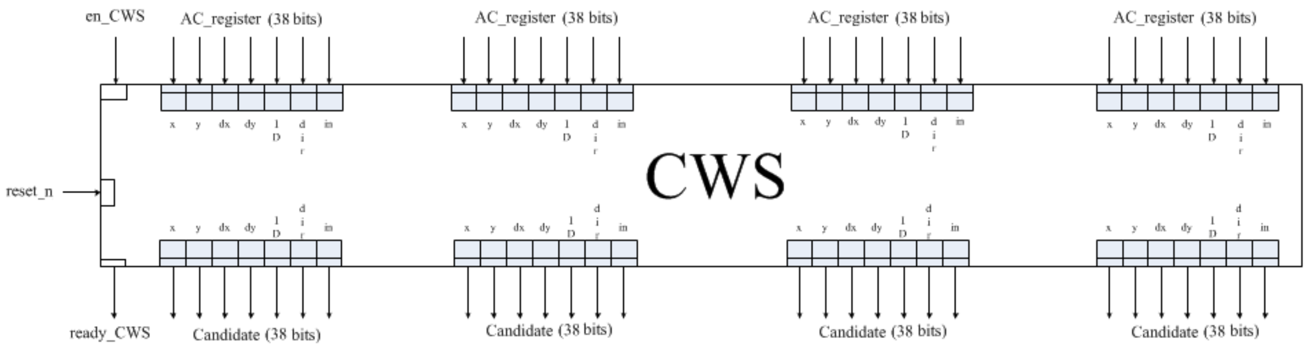


Figure 3.69: *CWS* block with interface.

A successive optimization must be made to meet the requirements of area and timing.

3.8.4 P_Window Selection

PWS is the dual block of *CWS* for *P_windows*. For this reason, *PWS* and *CWS* are implemented with same network. Differences are outputs: *PWS* has *N_P* registers of *P_reg_width* bits instead of *EC_MAX* registers of *candidate_reg_width* bits.

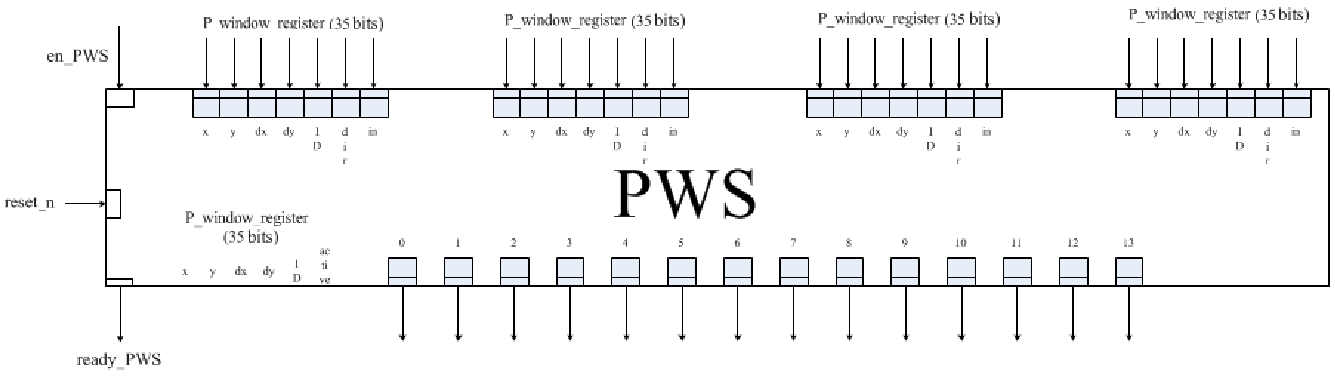


Figure 3.70: *PWS* block with interface.

3.8.5 Expandability Check

EC is the most difficult block for computational logic. It has the highest number of inputs and a lots of operations are carried out by this block.

Operations are almost the same of MATlab model but they must be made in only one clock cycle. For this reason, arithmetic operations are broken in easier operation and their results is used in an array. A better understanding of *EC* is postponed in the optimization paragraph.

Control interface is the same for every block:

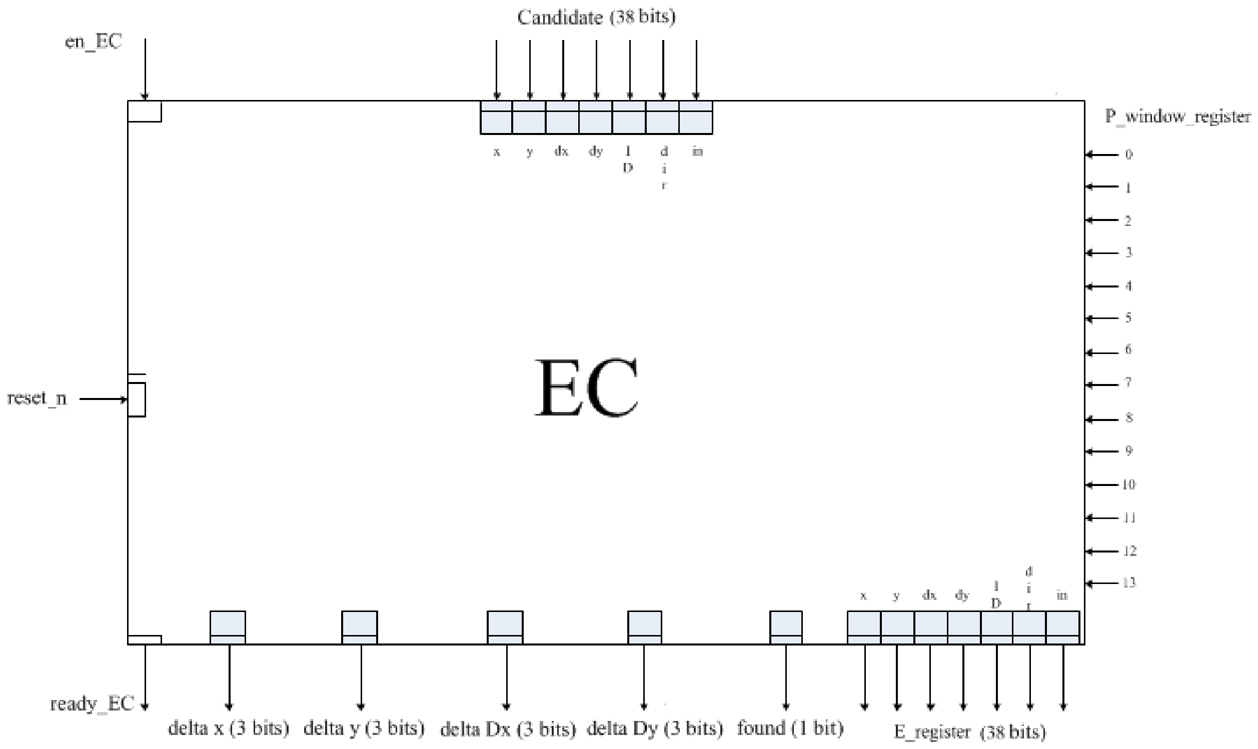


Figure 3.71: *EC* interfaces.

3.8.6 Priority Selection

PS block is a selection network: with some criteria, it chooses one of the Candidate or open a new window. So, the network is just a “big mux” with multiple inputs and outputs. Control interface is shown in Figure 3.72:

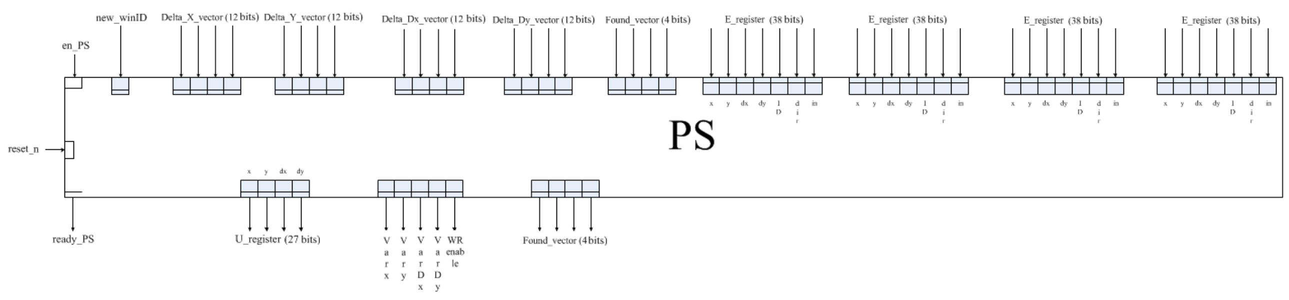


Figure 3.72: *PS* block.

3.8.7 Window update

WU is a block of window loading in principal memory. It has control interface and address signal in addition to inputs and outputs in MATlab model. Address is necessary to save the window in the correct position.

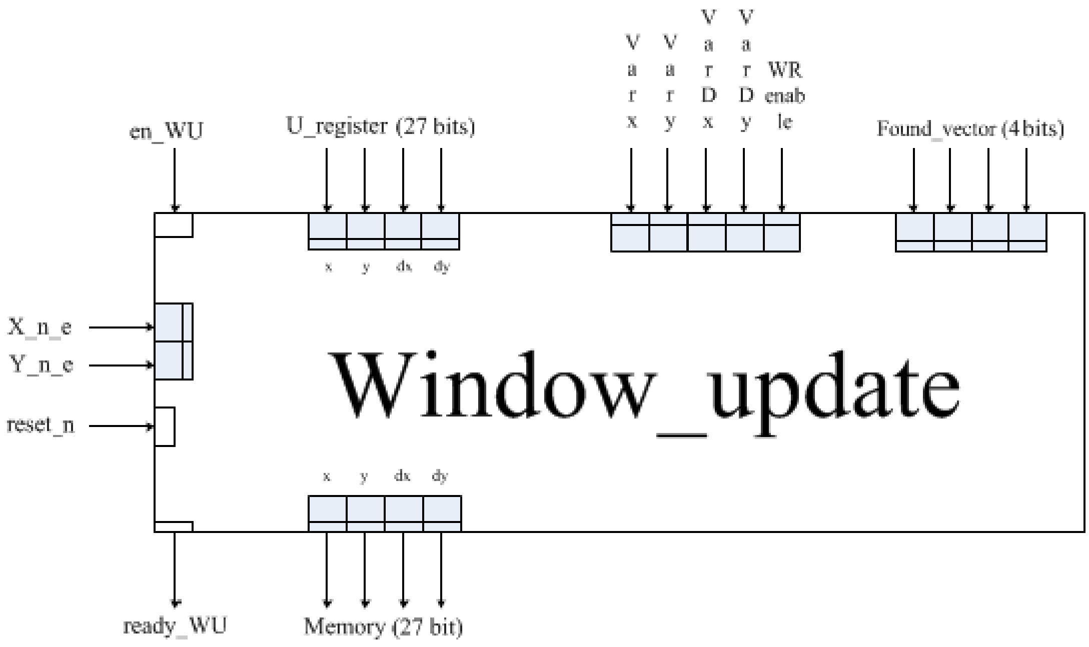


Figure 3.73: black box of *WU* block.

3.8.8 Add Neighbors

AN function is to add neighbors at the end of the frame. A number of clock's event equal to *N_win_MAX* is necessary to update neighbors to all windows. *Address* and read enable *re* interfaces are necessary to provide windows from RAM memory. *Address* is controlled into the block: *address* is increased at every clock's event to load in *AN* all windows of RAM memory. This *address* signal is provided in output to control RAM address but it is re-inserted in input to AN block for the successive increasing.

All others control signals are used as other blocks.

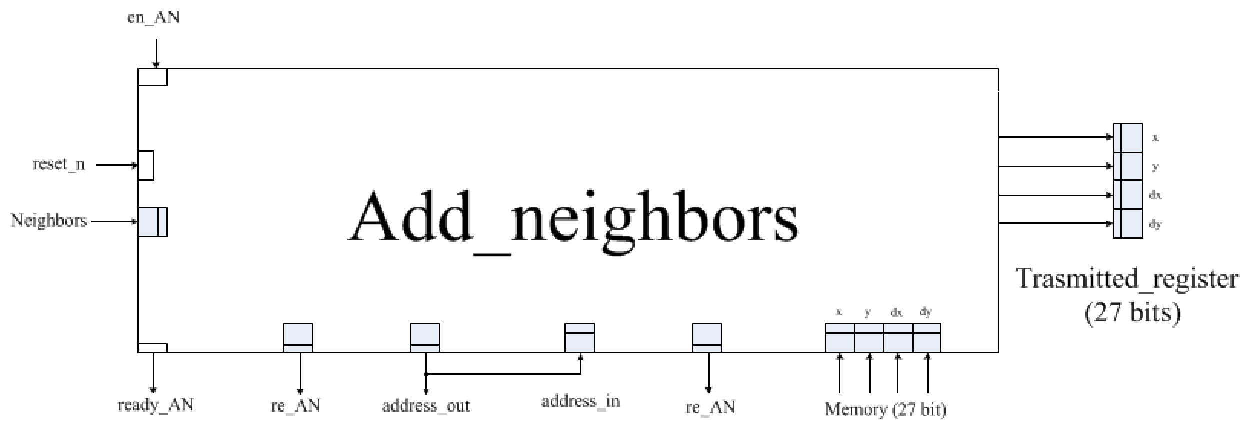


Figure 3.74: *AN* block with its interfaces.

3.8.9 RAM memory

All windows are saved in principal memory, in RAM blocks. RAM blocks need *address*, *read enable* and *write enable* to be controlled and a *data* ports in and out to pass information. RTAX4000S has dual port RAM blocks and, for a better optimization, both ports are used.

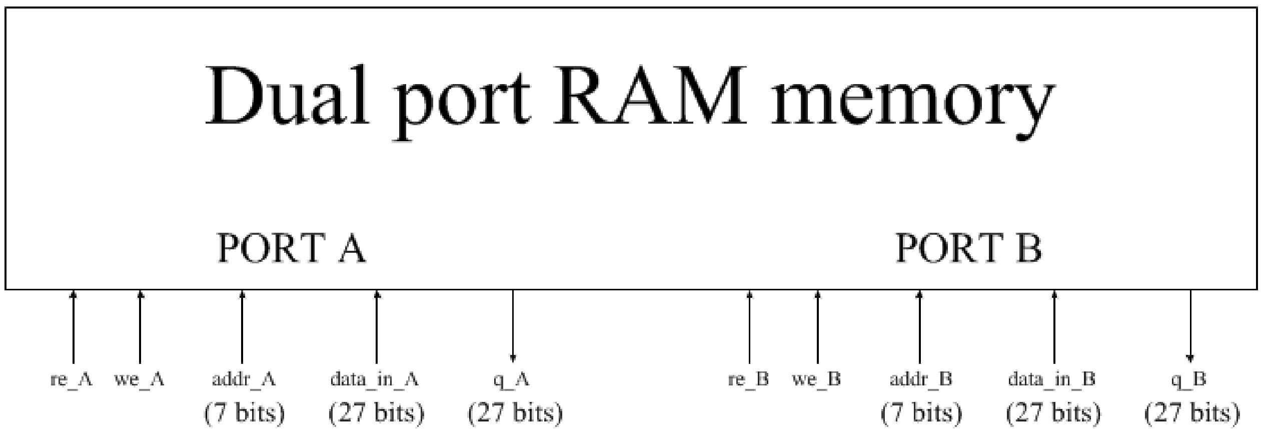


Figure 3.75 : Dual port RAM interfaces.

But RAMs in RTAX4000S are not “true” dual port: they can be used as dual port if a port is used only in write mode and the other only in read mode:

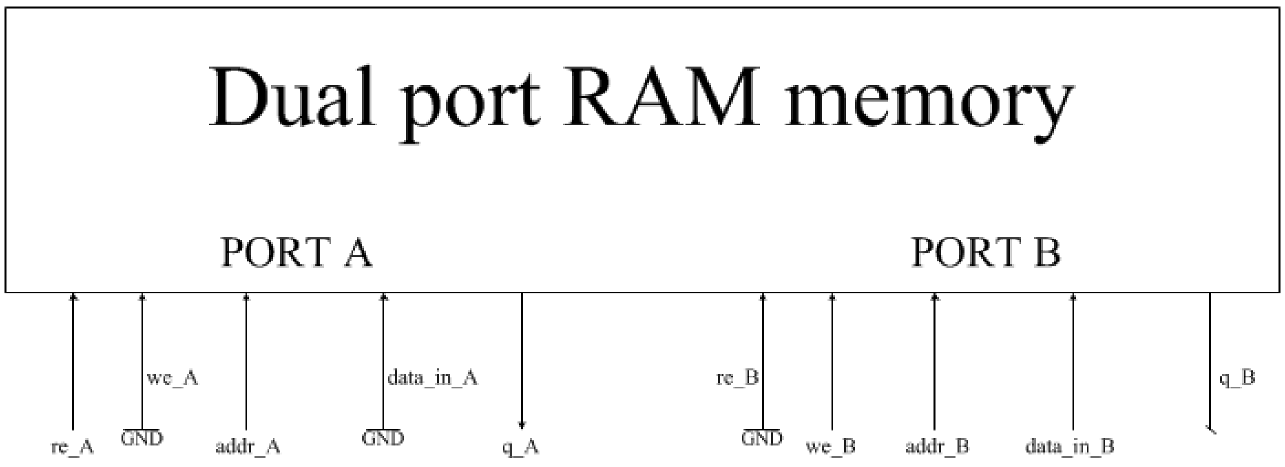


Figure 3.76: interfaces connection for dual port RAM. Interface called *q_B* is not connected.

For this reason, one port is reserved to loading window in *AC-PWC* and in *AN*, i.e. read only, the other port is reserved in write mode only for *WU*. Problem of data sharing between blocks is not present because at least one block accesses to RAM information in the same time. There is only a sharing problem for address in first port: this is passed with a *GSM* test over *ready* signals of *AN* block: in fact *GSM* machine handles blocks to run blocks one at a time. To control RAM signals, a control network is used. It handles *write* and *read enable*, *address* and *data* ports for every port.

Clock for this block is 50 MHz as explained in introduction of paragraph 3.8. *Reset_n* is not necessary too.

3.8.10 Window Load

WL is a new block used to handles RAM loading in *AC* and *PWC* blocks. In MATlab mode, this function is made by a simple *FOR* cycles. As explained in the introduction of paragraph 3.8, to parallelize as possible the algorithm, *AC_MAX* blocks work in parallel with frequency clock of 50 MHz. *WL* loads data in input of all *AC_MAX* blocks: for this reason, *WL* must load *AC_MAX* windows in parallel. *WL* uses only the read port of RAM memory and it must be switched with *AN* to allow dual port mode.

WL is a state machine that handle writing data from RAM to *AC-PWC* blocks. An internal register is necessary to hold the value of the last window valid in RAM: for this reason, *GSM* provides to *WL* block values of current ID with two signals:

- ***AC_WIN***;
- ***BLOCK_MAX***.

These values are found with the equation:

$$Current\_ID = AC\_WIN + BLOCK\_MAX * AC\_MAX$$

State machine *WL* must load blocks of *AC_MAX* windows, change block when *AC_MAX* windows are loaded and stop last valid window is loaded. Then, for every new DT, all valid windows in RAM must be loaded. *WL* is in stand-by mode only when *EC*, *PS*, *WL* and *AN* is working. This period is very short in comparison to the full period of elaboration of every single new DT: for this reason, *WL* must be optimized in area and timing constrains.

A state diagram of *WL* blocks is in Figure 3.77:

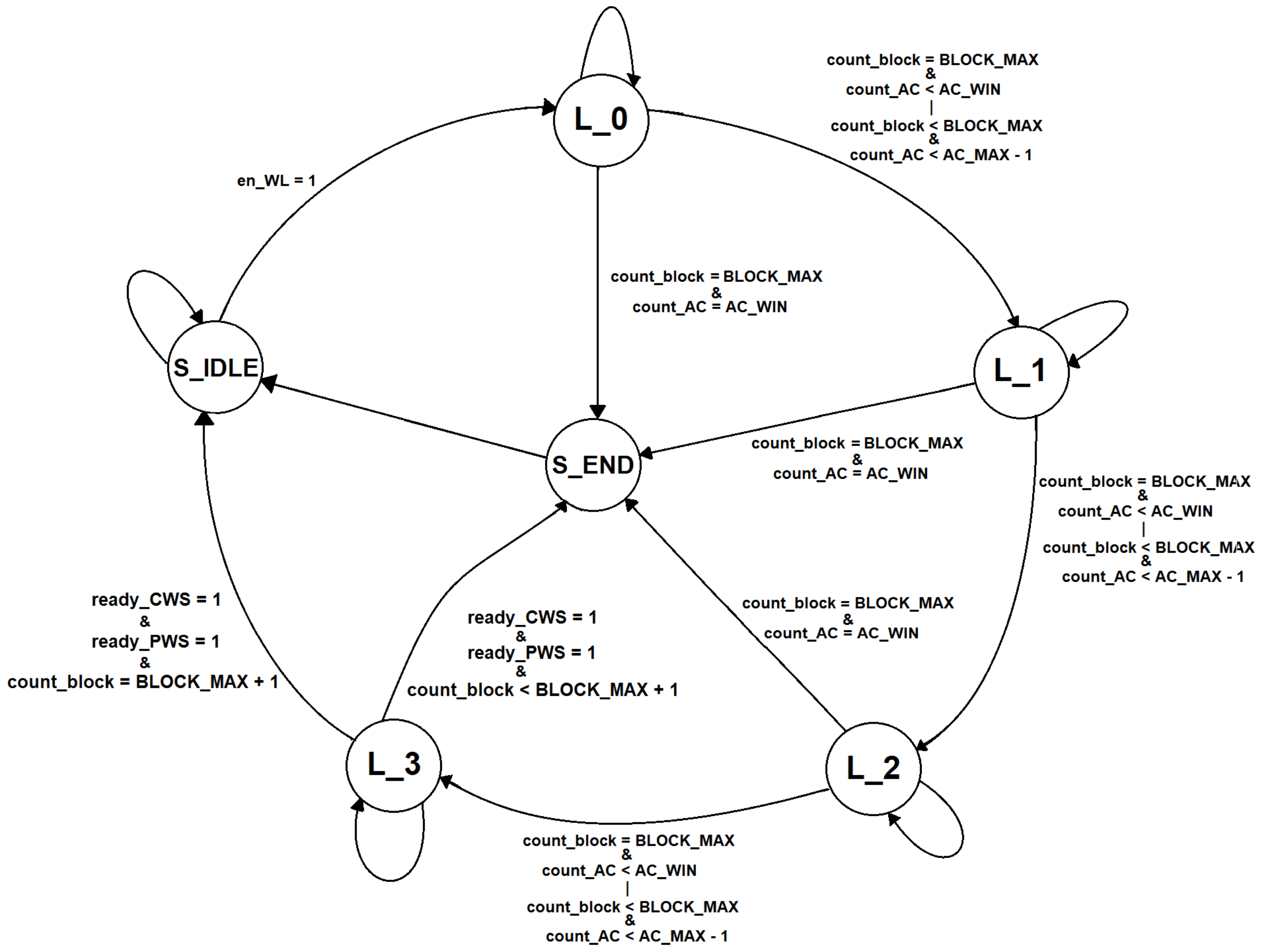

L_0
count_block = BLOCK_MAX & count_AC < AC_WIN | count_block < BLOCK_MAX & count_AC < AC_MAX - 1
en_WL = 1
count_block = BLOCK_MAX & count_AC = AC_WIN
S_IDLE
L_1
S_END
count_block = BLOCK_MAX & count_AC = AC_WIN
count_block = BLOCK_MAX & count_AC < AC_WIN | count_block < BLOCK_MAX & count_AC < AC_MAX - 1
ready_CWS = 1 & ready_PWS = 1 & count_block = BLOCK_MAX + 1
ready_CWS = 1 & ready_PWS = 1 & count_block < BLOCK_MAX + 1
count_block = BLOCK_MAX & count_AC = AC_WIN
L_3
L_2
count_block = BLOCK_MAX & count_AC < AC_WIN | count_block < BLOCK_MAX & count_AC < AC_MAX - 1

Figure 3.77: *WL* state diagram.

The starting state of *WL* is S_IDLE: when *en_WL* = 1, machine starts to load window in *AC* and *CWS*. Every *L_n* state loads a window in *memory_temp* output. When all memory is loaded, *WL* stand in stand-by mode, waiting for the "ready" signals of *AC_CWS* and *PWC_PWS* branches. Two counters, *count_block* and *count_AC*, are used to update the ID of the loading window: if the ID is equal to *current_ID* (*count_block* = *BLOCK_MAX* and *count_AC* = *AC_WIN*), *WL* must stop itself because all valid windows are loaded and successive state will be *S_END*. Else, i.e. *count_block* = *BLOCK_MAX* and *count_AC* < *AC_WIN* or *count_block* < *BLOCK_MAX* and *count_AC* < (*AC_MAX – 1*), loading continued with the successive window. During all states *L_n*, counter *count_AC* is increased to update the ID of the loaded window. In *L_3*, *count_AC* is set to 0 and *count_block* is increased by 1. When all the *AC_MAX* (4) windows are loaded, *WL* next state will be *S_END* if the frame is not ended, i.e. *ready_CWS* = 1 and *ready_PWS* = 1 and *count_block* < (*BLOCK_MAX* + 1), and *S_IDLE* if the frame is ended, i.e. *ready_CWS* = 1 and. *ready_PWS* = 1 and *count_block* < (*BLOCK_MAX* + 1).

WL uses only *clk*, *reset_n*, *enable* and *ready* signals, aside from *AC_WIN* and *BLOCK_MAX*.

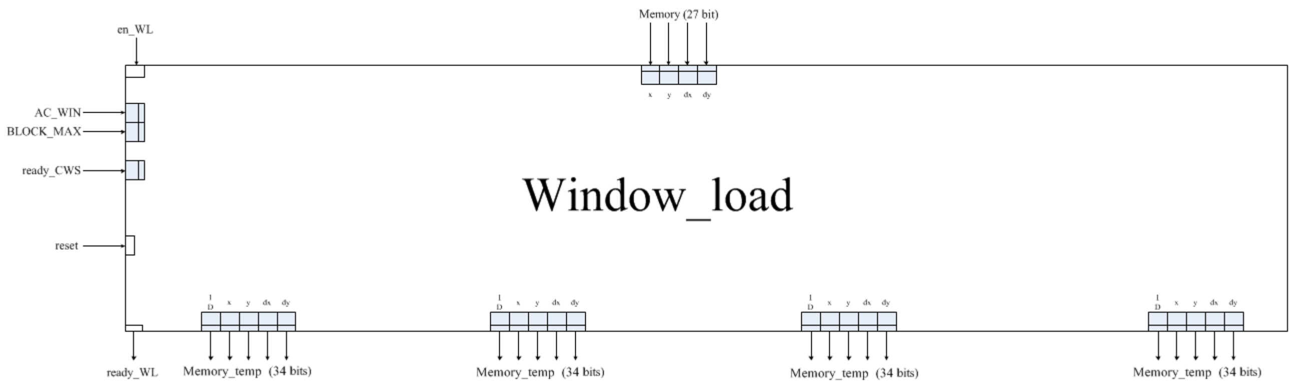


Figure 3.78: *WL* interfaces.

3.8.11 Global State Machine

GSM is the state machine that handle "start" and "stop" signals of every block. The state diagram of *GSM* will be shown in optimization paragraph because every change in the other blocks will change the state diagram.

Inputs of *GSM* are the input of the algorithm and *ready* signals of every block; outputs instead are control signals for every block, *enable* included.

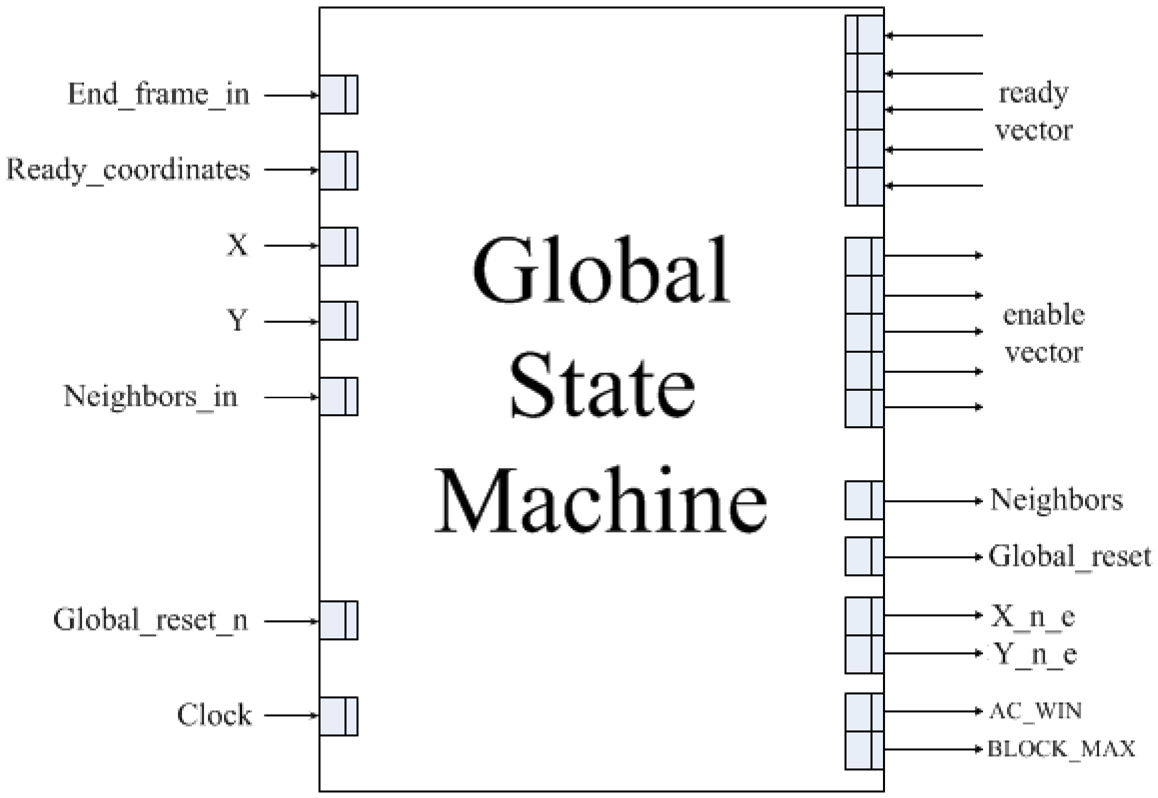


Figure 3.79: *GSM* block with inputs and outputs.

3.9 Algorithm optimization

After a first review of every block, they are inserted on a tool called ***Precision*** to have an idea of area occupation and max frequency, before of each block and then of the whole algorithm. Obviously every block is just a prototype but they can be as real as possible. If a block is too big for FPGA capacity or if frequency is lower than 50 MHz, it is optimized to meet the constrains: for timing requirements, critical paths are analyzed to break it and increase block working frequency, for area requirements, processes in the block are reordered to have easier operations. In state machines, internal variables and states are optimized to improve the efficiency and reduce dead times. Every block is given an area occupation and a max work frequency before and after the optimization.

The purpose of the optimization is to have the least clock cycles to elaborate a new DT in order to process as many DTs as possible and allow FPGA RTAX4000S to contain 4 clustering algorithm instances.

Timing analysis in description of paragraph 3.8 force all blocks to work on frequency of 50 MHz. Instead, area constrain for every algorithm is to occupy up to 25% of the whole area because 4 instances of the algorithm, one for quadrant, must be implemented in the FPGA RTAX4000S.

3.9.1Window Load and RAM block

As explained in paragraph 3.8.9, RAM memory is not a true dual port. Controller of RAM memory must handle signals to allow a dual port configuration and allow a work frequency of 50 MHZ. Port A is switched between *AN* and *WL* blocks, only read mode, while port B, only write mode, is reserved for *WU*, i.e. *PS_WU* block.

For an easier implementation of RAM, RAM controller and *WL* are enclosed in a top level called *WL_RAM*. It allows a better handling of the Port A controller because read enable and address must be switched from the output signals of *WL* and *AN*.

For a correct timing, *WL* provide the enable signal to *AC_CWS* and *PWC_PWS* branches and wait for their ready signals to load others *AC_MAX* windows. So, *GSM* must implement only start signal for *WL* and must wait for ready signals from *WL*, *AC_CWS* and *PWC_PWS*.

WL_RAM top level is already optimized because *WL* and RAM blocks are implemented from the beginning with fast and easy operation and a simple state diagram, already shown in Figure 3.77. The only optimization is the implementation of flags to check the conditions to change state in diagram:

```
over       <="1" when (((count_block=BLOCK_MAX_I)and(count_AC<AC_WIN_I))or ((count_block< BLOCK_MAX_I)and(count_AC<AC_MAX_VAL))) else"0";
stop       <="1" when ((count_block = BLOCK_MAX_I) and (count_AC = AC_WIN_I))     else "0";
ready      <="1" when (ready_CWS = "1" and ready_PWS = "1")                      else "0";
block_over <="1" when (count_block < BLOCK_MAX_I + 1)                            else "0";
```

Before the implementation of these flag report is in Table 3.14:

% Global area	% Register area	% Combinatory area	Max work frequency (MHz)	% RAM block
0,62	0,61	0,62	118,680	0,83

Table 3.14: WL_RAM before flag optimization.

Just a single RAM block of 120 is used to store all windows.

Report of optimized code is in Table 3.15:

% Global area	% Register area	% Combinatory area	Max work frequency (MHz)	% RAM block
0,4	0,76	0,22	120,106	0,83

Table 3.15: report for top level *WL_RAM*.

All area goals are met and timing goal too. In fact, a work frequency of 50 MHz is largely met to allow the parallelization shown in Figure 3.38.

3.9.2 AC_CWS branch

AC_CWS branch, as *PWC_PWS* branch, is the most important step to be optimized. In fact these 2 blocks are repeated for every window loading until the last valid window in memory. The purpose is to run this branch in only 2 clock cycles with a limited area utilization. A single top level from *CWS* and *AC_MAX AC* blocks is implemented.

AC block is quite simple: it is simply "translated" from MATlab without changes. Its synthesis results are reported in

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
0,67	0,49	0,76	73,959

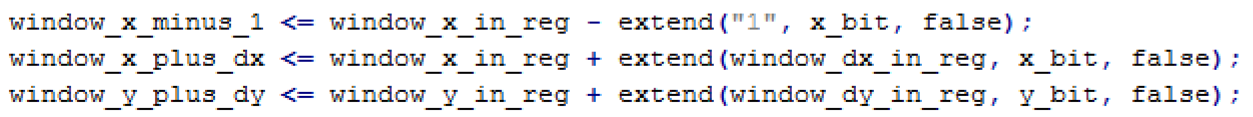
Table 3.16:

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
0,67	0,49	0,76	73,959

Table 3.16: results of first synthesis for *AC* block.

Frequency work is not optimal and occupancy can be reduced, mainly because there are *AC_MAX* blocks of *AC* in a single algorithm, *AC_MAX**4 in the whole FPGA.

For this reason, critical path is optimized. First of all, dedicated arithmetic operations are implemented:



```
window_x_minus_1 <= window_x_in_reg - extend("1", x_bit, false);
window_x_plus_dx <= window_x_in_reg + extend(window_dx_in_reg, x_bit, false);
window_y_plus_dy <= window_y_in_reg + extend(window_dy_in_reg, y_bit, false);
```

Then, four new processes are implemented, two just used to found adjacency flags and the others two for *ID_n_e* and *ready* variables:

```
adj_x_flag_proc : process(x_n_e_reg, window_x_minus_1, window_x_plus_dx, window_x_in_reg)
begin

    AX <= '0';
    BX <= '0';
    CX <= '0';
    DX <= '0';

    if(x_n_e_reg = window_x_minus_1) then
        AX <= '1';
    elsif(x_n_e_reg = window_x_plus_dx) then
        BX <= '1';
    elsif(x_n_e_reg >= window_x_in_reg and x_n_e_reg < window_x_plus_dx) then
        CX <= '1';
    else
        DX <= '1';
    end if;
end process;


adj_y_flag_proc : process(y_n_e_reg, window_y_plus_dy, window_y_in_reg)
begin

    AY <= '0';
    BY <= '0';
    DY <= '0';

    if(y_n_e_reg = window_y_plus_dy) then
        AY <= '1';
    elsif(y_n_e_reg >= window_y_in_reg and y_n_e_reg < window_y_plus_dy) then
        BY <= '1';
    else
        DY <= '1';
    end if;
end process;
```

Furthermore, with this strategy, two compare elementary blocks are no longer needed and area utilization is reduced too.

Then, all flags are concatenated in a vector:

```
flags <= AX & BX & DX & CX & AY & DY & BY;
```

and a case over it allow to find *dir* and *in* values used, as MATlab code, to indentify the position of a DT.

With these changes, the AC block is simpler and synthesis results are better (Table 3.17):

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
0,54	0,48	0,58	88,020

Table 3.17: report data after *AC* optimization.

AC block is simpler and synthesis results are better than before.

Now, *CWS* blocks is optimized. This block must be optimized as well as possible because it is critical for speed and area. Before the optimization, it is composed by two parts and then they are assembled in a top level. With this procedure, *CWS* block is bigger and it is necessary two clock's events for a single run. Synthesis report is in the

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
2,81	3,45	2,68	80,205

Table 3.18:

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
2,81	3,45	2,68	80,205

Table 3.18: *CWS* report before optimization. Max frequency is with a run of two clock's events.

So, an optimization is made to reduce the area utilization and allow the execution in a single clock cycle of 50 MHz.

First of all, two blocks are united in a single block to reduce register and allow to run it in a single clock cycle. After that, *address_vector* is deleted because it is equal to *num*-1 array. Integer variables is limited in their respective range and others changes are made in the algorithm to reduce the complexity of operations. The results of optimization is resumed in the table:

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
1,83	1,65	1,91	85,616

Table 3.19: *CWS* report after optimization.

When the two blocks are optimized, a top level *AC_CWS* with *AC_MAX* blocks of *AC* and a *CWS* is instantiated. It is necessary to reduce the complexity of global top level for a single algorithm and to verify that ***Precision*** tool can compress area and evidence critical path to optimize the FPGA utilization.

If the tool is not able to reduce the area utilization, the total area occupation of the top level *AC_CWS* should be the sum of every single block and the work frequency should be the lower between the max work frequency of all blocks:

Name	% Global area	% Register area	% Combinatorial area	Max frequency (MHz)
AC	0,54	0,48	0,58	88,020
CWS	1,83	1,65	1,91	85,616
Expected *AC_CWS*	3,99	3,57	4,23	85,616
Real *AC_CWS*	3,22	2,32	3,66	66,636
Difference	0,77	1,25	0,57	-

Table 3.20: report expected and real report for *AC_CWS* branch.

How shows in Table 3.20, the area occupation is reduced with a single top level *AC_CWS.* It is a great vantage for the project because also an optimization of every block cannot guarantee the capacity of the algorithm to be contained in the FPGA.

These parameters meet the constraint of timing, i.e. 66,636 MHz is greater than 50 MHz, and the covered area is small.

3.9.3 PWC_PWS branch

Even *PWC_PWS* branch is critical as *AC_CWS* but the number of windows in outputs of *PWS* are more than windows in *CWS* outputs, i.e. *N_P* is greater than *EC_MAX*. For this reason, it is expected that *PWC_PWS* branch is more complex than *AC_CWS* with a greater utilization of area and a lower value of working frequency.

First of all, optimizations of every block are discussed.

PWC is not much optimized because this block is very simple. Report of synthesis is in Table 3.21:

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
0,48	0,44	0,50	87,009

Table 3.21: *PWC* before optimization.

Only the implementation of combinatory variable for arithmetic operations can reduce the area:

```
valid <= "1" when (window_dy_in_reg > "0000" and window_dx_in_reg > "0000")                                    else "0";
x_14  <= "1" when ((std_logic_vector(unsigned(x_n_e_reg) - 14) <= window_x_in_reg) and (x_n_e_reg >= window_x_in_reg)) else "0";
x     <= "1" when (x_n_e_reg = window_x_in_reg)                                                                  else "0";
y_14  <= "1" when ((y_n_e_reg >= window_y_in_reg) and (std_logic_vector(unsigned(y_n_e_reg) - 14) <= window_y_in_reg)) else "0";
y     <= "1" when (y_n_e_reg = window_y_in_reg)                                                                  else "0";
```

With this strategy, parameter of PWC changed in:

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
0,47	0,44	0,49	92,285

Table 3.22: *PWC* after optimization.

Synthesis results are almost the same but an increment of the work frequency is a good news for the successive implementation of *PWC_PWS* branch.

PWS block has the same architecture of *CWS*. Before the optimization, the report is:

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
5,68	4,79	6,21	66,273

Table 3.23: *PWS* before optimization.

So, the same optimization is implemented for this block: *PWS* is reduced in one block instead of two to allow the run in just one clock cycle ,*address_vector* internal variable is deleted and some operations are simplified. The result is in Table 3.24:

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
4,54	3,17	5,23	57,894

Table 3.24: *PWS* optimized.

After the optimization, *PWS* area occupation is greatly reduced but the work frequency is reduced too. For this reason , it is important to implement a top level *PWC_PWS* to have an reliable report of the branch. In fact, in *AC_CWS* branch, the tool has reduced the occupation and optimized critical path.

AC_MAX blocks of *PWC* and a *PWS* are instantiated in the *PWC_PWS* top level. The expected results is the arithmetic sum of the area and a work frequency equal to the lowest work frequency of the two different blocks:

Name	% Global area	% Register area	% Combinatorial area	Max frequency (MHz)
PWC	0,47	0,44	0,49	92,285
PWS	4,54	3,17	5,23	57,894
Expected *PWC_PWS*	6,42	4,93	7,19	57,894
Real *PWC_PWS*	5,78	4,02	6,66	58,241
Difference	0,64	0,91	0,53	-

Table 3.25: report data for *PWC_PWS* branch.

The tool reduces the area occupation of the branch and increase the max work frequency: this is the perfect condition of optimization. Frequency is higher than before because the tool has optimized the path of information and the assignment of registers in *PWS* block. Now, the branch is optimized with a lower area occupation and a working frequency higher than 50 MHz. The execution of the branch is completed in two clock cycles.

3.9.4 Expandability Check optimization

EC block is the most important and complex block in the algorithm. For this reason, this block is implemented, from the beginning, with a high optimization.

The optimization is explained with the VHDL code.

First of all, two variables, right and down borders, are calculated. *Extend* function is used to equalize the number of bits.

```
C_window_x_L <= C_window_x_reg  + extend (C_window_dx_reg, x_bit, false);
C_window_y_L <= C_window_y_reg  + extend (C_window_dy_reg, y_bit, false);
```

Then, boundary checks are made to compare the window size with the edges of the quadrant:

```
W_DY_MAX <='1'when ((unsigned(C_window_dy_reg)=unsigned(MAX_WIN_DIM_b)) or ((unsigned(C_window_dy_reg)=unsigned(MAX_WIN_DIM_b2))and(neighbors_reg="01")))else'0';
W_DX_MAX <='1'when ((unsigned(C_window_dx_reg)=unsigned(MAX_WIN_DIM_b)) or ((unsigned(C_window_dy_reg)=unsigned(MAX_WIN_DIM_b2))and(neighbors_reg="01")))else'0';
W_X_0    <='1'when (unsigned(C_window_x_reg) = 0)                  else '0';
W_Y_MAX  <='1'when (unsigned(C_window_y_L) = (unsigned(height_b)))  else '0';
W_X_MAX  <='1'when (unsigned(C_window_x_L) = (unsigned(width_b)))   else '0';
```

Boundary flags are initialized with results of these checks:

```
D_bound     <= not(W_DY_MAX or W_Y_MAX);
R_bound     <= not(W_DX_MAX or W_X_MAX);
RD_bound    <= not(W_DY_MAX or W_Y_MAX or W_DX_MAX or W_X_MAX);
LD_bound    <= not(W_DY_MAX or W_Y_MAX or W_DX_MAX or W_X_0);
```

When boundary flags are initialized, the comparisons between Candidate window and all *P_windows* are made. First, operations *A*, *B* ,*C* and *D* are made and the results are compared with standard values as explained in paragraph 3.7.4. Then window flags are updated. These operations are repeated for every *P_window*. Every window saves the results of every operation in an array, i.e. *A, B, C, D,* A_0, A_1, B_0, B_1, C_0,, D_0, D_1, *D_win, R_win, RD_win and LD_win* are arrays:

```
flag_generate : for jjj in 0 to N_P-1 generate
    A (jjj) <= P_window_x(jjj) +    extend (P_window_dx(jjj), x_bit, false) - C_window_x_reg;
    B (jjj) <= P_window_x(jjj) -    C_window_x_reg                          - extend (C_window_dx_reg, x_bit, false);
    C (jjj) <= P_window_y(jjj) +    extend (P_window_dy(jjj), y_bit, false) - C_window_y_reg;
    D (jjj) <= P_window_y(jjj) -    C_window_y_reg                          - extend (C_window_dy_reg, y_bit, false);

    A0  (jjj) <= '1' when (A(jjj) <  1)  else '0';
    A1  (jjj) <= '1' when (A(jjj) < 0)   else '0';

    B0  (jjj) <= '1' when (B(jjj) > -1)  else '0';
    B1  (jjj) <= '1' when (B(jjj) > 0)   else '0';

    C0  (jjj) <= '1' when (C(jjj) <  1)  else '0';

    D0  (jjj) <= '1' when (D(jjj) > -1)  else '0';
    D1  (jjj) <= '1' when (D(jjj) > 0)   else '0';

    D_win  (jjj)  <= (A0  (jjj) or B0  (jjj) or C0  (jjj) or D1  (jjj)) when ((P_window_ID(jjj)  /= ID_in) and (P_window_active(jjj)  = "1")) else '1';
    R_win  (jjj)  <= (A0  (jjj) or B1  (jjj) or C0  (jjj) or D0  (jjj)) when ((P_window_ID(jjj)  /= ID_in) and (P_window_active(jjj)  = "1")) else '1';
    RD_win  (jjj) <= (A0  (jjj) or B1  (jjj) or C0  (jjj) or D1  (jjj)) when ((P_window_ID(jjj)  /= ID_in) and (P_window_active(jjj)  = "1")) else '1';
    LD_win  (jjj) <= (A1  (jjj) or B0  (jjj) or C0  (jjj) or D1  (jjj)) when ((P_window_ID(jjj)  /= ID_in) and (P_window_active(jjj)  = "1")) else '1';

end generate;
```

Window flags are array for a faster comparison: in fact, t is sufficient made an AND-tree to have the result of the expandability:

```
D_g      <= "1" when (D_bound = '1' and D_win = ones)     else "0";
R_g      <= "1" when (R_bound = '1' and R_win = ones)     else "0";
RD_g     <= "1" when (RD_bound = '1' and RD_win = ones)   else "0";
LD_g     <= "1" when (LD_bound = '1' and LD_win = ones)   else "0";
```

General flags *D_g*, *R_g*, *RD_g* and *LD_g* are the final flags used to found the right window.

As final operation, the same comparisons of the bit-true model are made.

The optimization allows to *EC* to meet the constraints: in fact results of the report are in Table 3.26

% Global area	**% Register area**	**% Combinatorial area**	**Max work frequency (MHz)**
4,53	2,88	5,35	64,263

Table 3.26: *EC* report.

A better optimization of the *EC* block is very difficult for the algorithm. Results respect the timing goals and area are quite limited for the complexity of the block.

3.9.5 PS and WU

Function of *WU* is just to load in the RAM the found window with higher priority and *PS* has to find this window. To optimize the algorithm, *PS* and *WU* are implemented in the same block. In fact *PS* can directly load the result of the priority selection in the RAM. With this strategy, a clock event is saved and *WU* area too. Before the enclosing in the same block, report for *PS* and *WU* are:

Name	% Global area	% Register area	% Combinatorial area	Max frequency (MHz)
PS	1,35	1,30	1,38	67,516
WU	0,47	0,52	0,45	103,466

Table 3.27: *PS* and *WU* report before optimization.

When two blocks are merged, report is:

Name	% Global area	% Register area	% Combinatorial area	Max frequency (MHz)
Expected PS_WU	1,82	1,80	1,83	67,516
Real PS_WU	1,35	1,30	1,38	67,516

Table 3.28: Expected and real report of *PS_WU* block.

In area occupation, the saving is limited but the block is run in a single clock cycle instead of two. So, goals are largely achieved.

3.9.6 Add Neighbors

AN block is executed at the end of the frame. It must add neighbors to all windows saved in RAM memory. Controller for RAM memory switches only-read *port A* between two inputs, as explained in paragraph 3.8.9. *AN* is one of these inputs. So, the block has to provide *address* and *read enable* to read windows in RAM and made arithmetic/logic operation over them.

Before the optimization, *AN* made lots of operations in a single process and result of report is in Table 3.29 :

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)

0,78	0,71	0,79	67,833

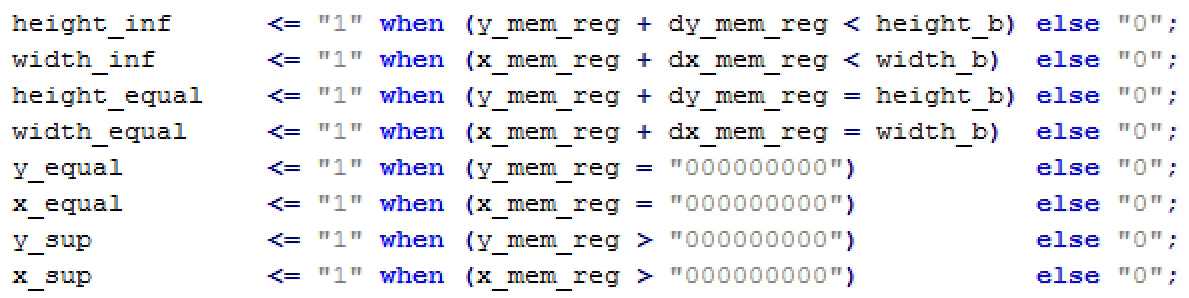
Table 3.29: report of *AN* before optimization.

Then, *AN* critical path is broken with the creation of dedicated operations. With this strategy, the *AN* block becomes:

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
0,41	0,36	0,43	71,083

Table 3.30: report of *AN* after optimization.

When the critical path is optimized, work frequency is increased and area occupation is reduced of about 50%. Area occupation is reduced with the division conditions in easier operations:



```
height_inf      <= "1" when (y_mem_reg + dy_mem_reg < height_b) else "0";
width_inf       <= "1" when (x_mem_reg + dx_mem_reg < width_b)  else "0";
height_equal    <= "1" when (y_mem_reg + dy_mem_reg = height_b) else "0";
width_equal     <= "1" when (x_mem_reg + dx_mem_reg = width_b)  else "0";
y_equal         <= "1" when (y_mem_reg = "000000000")           else "0";
x_equal         <= "1" when (x_mem_reg = "000000000")           else "0";
y_sup           <= "1" when (y_mem_reg > "000000000")           else "0";
x_sup           <= "1" when (x_mem_reg > "000000000")           else "0";
```

This block is run when a frame is completed: so, it is run *N_win_MAX* times at the end of the frame, one for every window that must be provided in output. When *address_in_reg*, the output register of *address_in*, is over *N_win_MAX*, valid windows are finished and no more neighbors must be added.

Timing goal is achieved and the block is ready for top level implementation.

3.9.7 Global State Machine

As block description of paragraph 3.8.11, the *Global State Machine* (*GSM*) handles all blocks with start and stop signals and provides them signals from the algorithm inputs. When the optimization of all blocks is complete, the GSM state diagram can be implemented.

Output start signals are:

- *en_WL*: start for *WL* machine. Starts for *AC*, *CWS*, *PWC* and *PWS* are not necessary because will be *WL* that will give them;
- *en_EC*: enable for *EC*. This signal are the same for all parallel *EC* blocks;

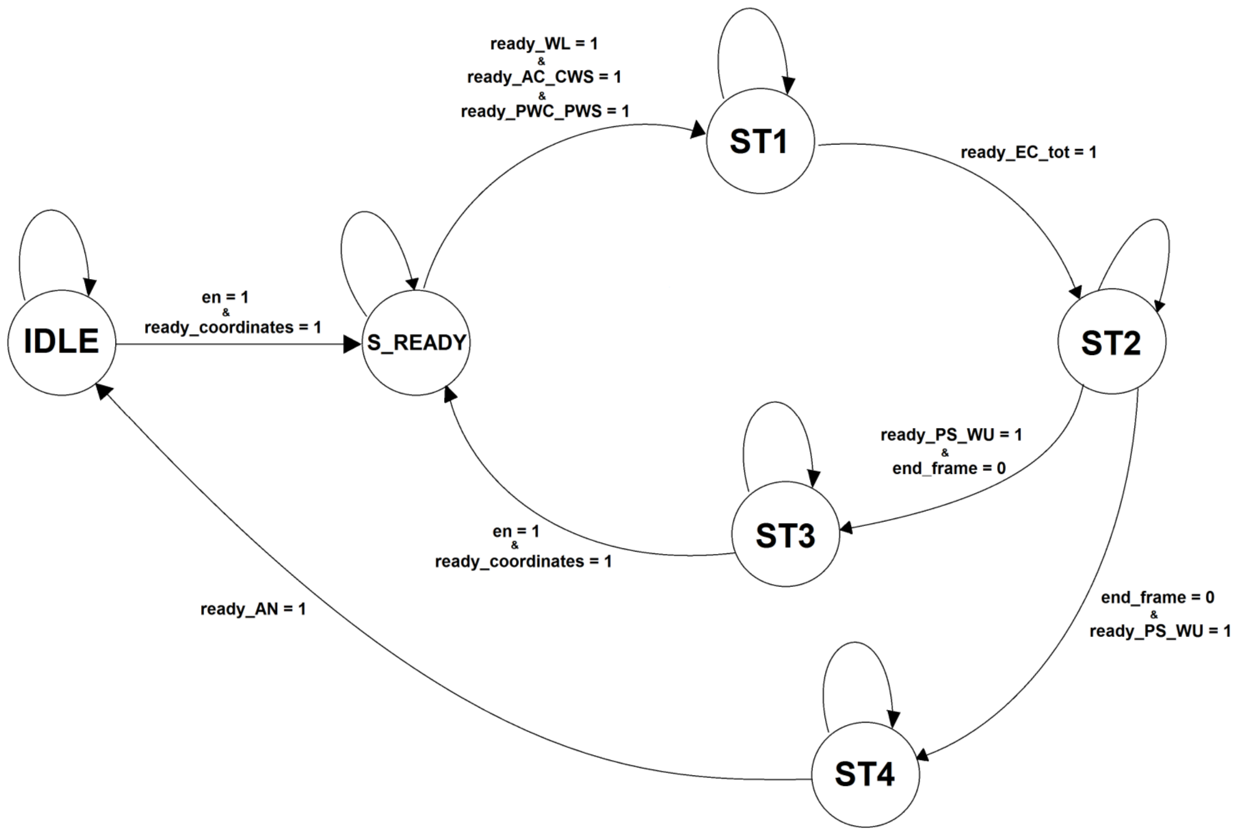
- *en_PS_WU*: enable for the optimized block *PS_WU*;
- *en_AN*: enable for AN.

Instead, ready signals in input are more than start signals:

- *ready_WL*: ready of machine WL;
- *ready_AC_CWS*: ready of *AC_CWS* branch;
- *ready_PWC_PWS*: ready of *PWC_PWS* branch;
- *ready_EC_tot*: ready for *EC* block;
- *ready_PS_WU*: ready for *PS*_WU;
- *ready_AN*: ready of *AN* block.

GSM must wait also ready signals for *AC_CWS* and *PWC_PWS* branches. When *WL* has loaded the last block of *AC_MAX* windows, i.e. its ready signal becomes "1", also ready signals of *AC_CWS* and *PWC_PWS* blocks must be waited to be certain of the correct elaboration of all *AC_MAX* windows: in fact if a Candidate is in the last block of loaded *AC_MAX* windows and ready signals for *AC_CWS* and *PWC_PWS* are not waited, *EC*s elaborate incorrect information because one Candidate is not yet ready for the expandability check.

State diagram of *GSM* is shown in Figure 3.80:



ready_WL = 1 & ready_AC_CWS = 1 & ready_PWC_PWS = 1
ST1
ready_EC_tot = 1
IDLE
en = 1 & ready_coordinates = 1
S_READY
ST2
ready_PS_WU = 1 & end_frame = 0
ST3
en = 1 & ready_coordinates = 1
ready_AN = 1
end_frame = 0 & ready_PS_WU = 1
ST4

Figure 3.80: state diagram of *GSM*.

Arrows that enter in the same exiting block is for no change of state because conditions for change state are not verified. *IDLE* is the initial state for every new frame, in *S_READY* the algorithm is ready to run and others states *ST1*, *ST2*, *ST3* and *ST4* are for the different steps of the algorithm.

When *GSM* is in *IDLE*, just valid data can start the algorithm, i.e. the system is enabled and input coordinates are ready. Then, for every step, ready signals from others blocks are checked and, if conditions are verified, the state changes. When the algorithm is in ST2, if *PS_WU* is ready, i.e. *ready_PS_WU* = 1, *end_frame* signal is checked: if *end_frame* = 1, *AN* is enabled and the neighbors adding is started. Else, if *end_frame* = 0, algorithm restarts for a new DT.

In the diagram, no arrow for *global_reset_n* signal is present but, when *global_reset_n* = 0, next machine state will be *IDLE*.

In *GSM* block is inserted also the clock enable generator for all *clk_en* of blocks with frequency of 50 MHz and two register counters used to save last valid window's ID, *AC_WIN* and *BLOCK_MAX*. All these values are saved in dedicated registers and they change with the clock. So, for example, *counter* resister for clock enable generator is increased every clock's event and if its value is equal to *AC_MAX*-1, *clk_en* is set to 1. In the successive clock's event, *clk_en* returns to 0.

```
process (clk, global_reset_n)
begin
    if global_reset_n = '0' then
        clk_en          <= '0';
        counter         <= 0;
    elsif (clk'event and clk = '1') then

        if (counter = AC_MAX-1) then
            clk_en  <= '1';
            counter <= 0;
        else
            clk_en  <= '0';
            counter <= counter + 1;
        end if;
    end if;
```

Registers for the last valid window ID are updated after a test over the found vector in output of *PS_WU*: if found_vector is equal to 0, the update is made with a "+1" over old valid window's ID. As explained in paragraph 3.8.10, this ID is found with an operation over two signals, *AC_WIN* and *BLOCK_MAX*, stored in *AC_WIN_reg* and *BLOCK_MAX_reg*. But *AC_WIN* value is between 0 and *AC_MAX*-1: if *AC_WIN* = *AC_MAX* – 1 and its value must be increased, *AC_WIN* is set to 0 and *BLOCK_MAX* is increased by 1.

```
process (clk, global_reset_n)
begin
    if global_reset_n = '0' then

        AC_WIN_reg      <= "00";
        BLOCK_MAX_reg   <= "00000";

    elsif (clk'event and clk = '1') then

        when IDLE | S_READY | ST1 | ST3 =>

            AC_WIN_reg      <= AC_WIN_reg;
            BLOCK_MAX_reg   <= BLOCK_MAX_reg;

        when ST2 =>
                if (add_win_value = "1") then

                    if(AC_WIN_reg = AC_MAX_VAL_b) then

                        AC_WIN_reg      <= (others => '0');
                        BLOCK_MAX_reg   <= BLOCK_MAX_reg + "00001";

                    else

                        AC_WIN_reg      <= AC_WIN_reg + "01";
                        BLOCK_MAX_reg   <= BLOCK_MAX_reg;

                    end if;
                else
                AC_WIN_reg      <= AC_WIN_reg;
                BLOCK_MAX_reg   <= BLOCK_MAX_reg;

                end if;

            when ST4 =>

                AC_WIN_reg      <= "00";
                BLOCK_MAX_reg   <= "00000";

            when others =>

                AC_WIN_reg      <= AC_WIN_reg;
                BLOCK_MAX_reg   <= BLOCK_MAX_reg;

        end case;
    else

    end if;
end process SEQ;
```

GSM outputs are the ready signals already described, *new_ID* for *PS_WU* block, *AC_WIN* and *BLOCK_MAX* for *WL* and all signals form algorithm inputs as *X*, *Y* and *neighbors*. Reset signal *reset_n* is enabled in *IDLE* state and it is necessary to reset all blocks at the end of the frame, i.e. *reset_n* is then sent to all others blocks.

The synthesis report is resumed in Table 3.31:

% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
0,08	0,07	0,09	201,086

Table 3.31: *GSM* report.

Area occupation is very small and clock frequency is much more greater than the goal of 50 MHz. So, *GSM* block largely meets the constrains.

3.9.8 Global top level

When all blocks are optimized in occupation and timing, a *Global_top_level* is implemented. Blocks are connected with internal signals and inputs and outputs of the algorithm are inserted in the port map.

To estimate the gain of area occupation and clock frequency, a comparison between optimization and no-optimization code in *Global_top_level* is made. After, a "real" *Global_top_level* is implemented to have a report and valuate the goals of timing and area.

In case of no-optimization, total occupation of area is estimated in the sum of area occupation of every single no-optimized block:

Name	**% Global area**	**% Register area**	**% Combinatorial area**	**Max frequency (MHz)**
WL-RAM	0,62	0,61	0,62	118,680
AC	0,67	0,49	0,76	73,959
CWS	2,81	3,45	2,68	80,205
PWC	0,48	0,44	0,50	87,009
PWS	5,68	4,79	6,21	66,273
EC	4,64	2,88	5,47	62,089
PS	1,35	1,28	1,38	96,516
WU	0,47	0,52	0,45	103,466
AN	0,78	0,71	0,79	67,833
GSM	0,11	0,12	0,12	203,088
Expected Global_top_level	34,98	26,72	39,17	62,089

Table 3.32: estimate of area occupation as sum of no-optimized blocks and frequency estimate.

Obviously, without optimization, values of area occupation don't cover constrains because all values must be at least lower than 25% to allow to 4 clustering algorithm to be contained in the FPGA. Instead, frequency value respects the initial goal.

In case of optimization, it is expected a lower area occupation for *Global_top_level*:

Name	% Global area	% Register area	% Combinatorial area	Max frequency (MHz)
WL-RAM	0,40	0,76	0,22	120,106
AC_CWS	3,22	2,32	3,66	66,636
PWC_PWS	5,78	4,02	6,66	58,916
EC	4,53	2,88	5,35	64,263
PS_WU	1,35	1,28	1,38	67,540
AN	0,41	0,36	0,43	71,083
GSM	0,08	0,07	0,09	201,086
Expected Global_top_level	29,36	20,33	33,84	58,916
Difference	5,62	6,39	5,33	-

Table 3.33: report of *Global_top_level* with optimized blocks.

In Table 3.33 it is possible to see that area occupation is reduced of about 15%-20% but it is not sufficient yet because occupation is over 25% for combinatorial and global area.

In *AC_CWS*, *PWC_PWS* and *PS_WU* cases, ***Precision*** has reduced the area occupation with an internal optimization of FPGA space. For this reason, it is expected that this kind of optimization could be made also for *Global_top_level*. So, VHDL code for *Global_top_level* is implemented and a synthesis is made. Report for *Global_top_level* is in Table 3.34:

Name	% Global area	% Register area	% Combinatorial area	Max work frequency (MHz)
Real Global_top_level	17,18	5,74	22,90	58,548
Difference	12,18	14,59	10,94	-

Table 3.34: report for VHDL *Global_top_level* code.

Area occupation is reduced of about 40% respect to the "external" optimization. It is easily explained with the structure of the FPGA RTAX4000S. In fact, FPGA is composed by elementary cells connected to each other to create a block: these cells are indivisible and when one block of the algorithm occupies a part of this cell, the remaining unoccupied part is no longer available. Instead, when more than a block is insert in the VHDL code and all necessary resources are estimated, the tool optimizes the occupation of these cells and reduces to minimum the waste of cells. For example, *AC_CWS* occupy the 30% of the elementary cell and *PWC_PWS* the 40%: if they are separated in two different VHDL code, they occupy a total of two whole cells with an high wasted area but, by the internal optimization, they can occupy just the 70% of a single cell with a gain of a whole cell. So, it is simple to understand the great gain of *Global_top_level* respect to single blocks implementation. Obviously this strategy is different for R-cells and C-cells but the idea is the same for the whole FPGA.

When a single algorithm is implemented and optimized, a top level with the 4 instances of the clustering algorithm is synthesized. The expected occupation is about the area occupation of 4 algorithms.

The synthesis report is in Table 3.35:

% Global area	**% Register area**	**% Combinatorial area**	**Max work frequency (MHz)**
68,72	22,96	91,60	58,548

Table 3.35: area occupation and frequency value for the whole FPGA.

The strategy is optimal also for the timing goal because the maximum clock frequency stays over the goal of 50 MHz and area occupation is quite under the 100%.

3.9.9 Timing analysis

The final step of the analysis is the estimate of computational power of the algorithm. In fact it is important to check if all the 250 new DTs can be elaborated in the time requirement of paragraphs 3.1 and 3.3.4.

An estimate of clock cycles is made for a single frame composed by indeterminate *N_DT_MAX* new DTs; then, the result is multiplied by the clock period: the result must be lower than 1 ms. With this strategy, the maximum number of DTs elaborate in 1 ms can be evaluated.

The estimate is made for the worst case, i.e. with maximum window loading because this requires the maximum number of clock cycles for the elaboration. Total new DTs are *N_DT_MAX*, total windows are 75, i.e. *N_win_MAX*. So, the worst case is considered when the memory is filled faster, or rather, when the first 75 windows are Single Events and fill all memory and the other (*N_DT_MAX* - 75) imposes to load from the memory all the 75 windows. In theory, this is the worst case but it is a hardly possible case because, for the raster scan and the position, the last (*N_DT_MAX* -75) new DTs should be enclosed in the windows present in the last elaborated row as shown in Figure 3.81.

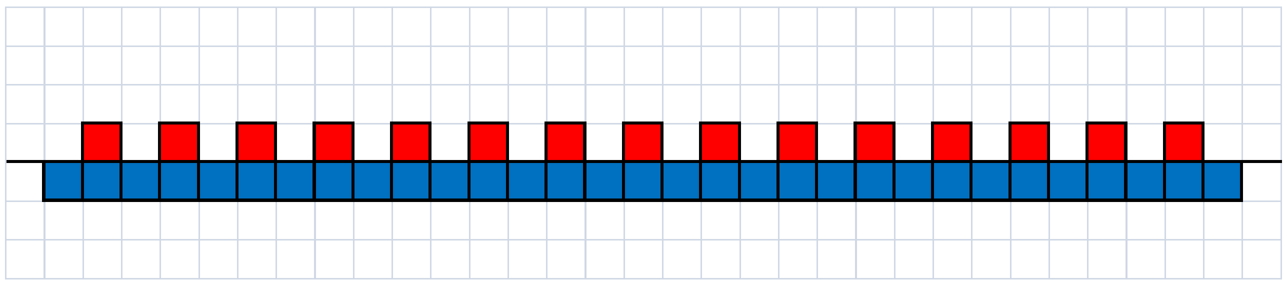


Figure 3.81: all red DTs are Single Events present on the last row of the analyzed area. Remaining *N_DT_MAX*-75 events, in blue, should be on the successive row to create the worst case just described.

First of all, an estimate of clock cycles for *WL* is necessary. *WL* loads *AC_MAX* windows in the same time. So, from 1 to 4 windows in memory, a single loading is sufficient, from 5 to 8 windows, two loading and so on. In mathematical terms, number of windows loading is equal to rounded up of the last valid ID divided by *AC_MAX* (4). Until to the maximum capacity of the window memory, *N_win_MAX*, window loading is increasing as precedent rule. Between *N_win_MAX+1* and *N_DT_MAX*, windows loading are always the rounded up of *N_win_MAX* / 4 i.e. 19.

Total of window loading are equal to the clock cycles because a single loading of *AC_MAX* is made in a single clock cycle.

So, for the whole frame (all the *N_DT_MAX* new DTs),total number of clock cycles are found with the formula:

$$WL_{frame\ clock\ cycles} = \sum_{i=0}^{75} \lceil i/AC\_MAX \rceil + \lceil 75/AC\_MAX \rceil * (N\_DT\_MAX - 76)$$

$$WL_{frame\ clock\ cycles} = 741 + 19 * (N\_DT\_MAX - 76)$$

$$WL_{frame\ clock\ cycles} = 741 + 19 * N\_DT\_MAX - 1444$$

$$WL_{frame\ clock\ cycles} = 19 * N\_DT\_MAX - 703$$

For *AC_CWS* and *PWC_PWS* branches are necessary twice $WL_{frame\ clock\ cycles}$ clock cycles because every branch uses two clock cycles for every memory loading. So, $AC\_CWS_{frame\ clock\ cycles} = PWC\_PWS_{frame\ clock\ cycles} = 2 * WL_{frame\ clock\ cycles}$ for the whole frame.

EC block is executed in a single clock cycles and *PS_WU* in a single cycle too. So, for the whole frame, it is necessary:

$$EC_{frame\ clock\ cycles} = N\_DT\_MAX$$

$$PS\_WU_{frame\ clock\ cycles} = N\_DT\_MAX$$

Last block *AN* uses one clock cycle for a single window to update. It is necessary *N_win_MAX* clock cycles for the updating of the windows in the whole frame.

$$AN_{frame\ clock\ cycles} = N\_win\_MAX \ = 75$$

GSM uses just one clock cycles for every state for each DT:

$$GSM_{frame\ clock\ cycles} = 6 * \ N\_DT\_MAX$$

Clock cycles in a frame for every block are resumed in Table 3.36:

BLOCK	**CLOCK CYCLES PER FRAME**
$WL_{frame\ clock\ cycles}$	19 * *N_DT_MAX* - 703
$AC\_CWS_{frame\ clock\ cycles} = PWC\_PWS_{frame\ clock\ cycles}$	2 * (19 * *N_DT_MAX* – 703)
$EC_{frame\ clock\ cycles}$	*N_DT_MAX*
$PS\_WU_{frame\ clock\ cycles}$	*N_DT_MAX*
$AN_{frame\ clock\ cycles}$	75
$GSM_{frame\ clock\ cycles}$	6 * N_DT_MAX

Table 3.36: resume of clock cycles necessary to every block for the whole frame.

With a simple sum it is possible to find the total number of clock cycles for a whole frame. The result will be function of the parameter *N_DT_MAX*.

$$GLOBAL_{frame\ clock\ cycles} = 3 * (19 * \ N\_DT\_MAX - 703) + 2 * N\_DT\_MAX + 75 + 6 * \ N\_DT\_MAX$$

$$GLOBAL_{frame\ clock\ cycles} = 3 * 19 * \ N\_DT\_MAX \ - 2109 + 2 * N\_DT\_MAX \ + 75 + 6 * \ N\_DT\_MAX$$

$$GLOBAL_{frame\ clock\ cycles} = 57 * \ N\_DT\_MAX \ - 2034 + 8 * N\_DT\_MAX$$

$$GLOBAL_{frame\ clock\ cycles} = 65 * \ N\_DT\_MAX \ - 2034$$

This is the total of clock cycles necessary for the elaboration of a whole frame. The result must be multiplied for the clock period of 20 ns: the condition imposes that the full period must be lower than 1 ms.

$$1\,ms > \left(65 * \ N_{DT_{MAX}} - 2034\right) * \ 20\,ns$$

The inequality is verified when:

$$N\_DT\_MAX \leq 800$$

The algorithm guarantees the perfect functionality in timing requirements for a number of DTs equal to 800. It is a number higher than the required 250 new DTs for frame.

Chapter 4
Conclusion

The purpose of this thesis is the implementation of a new kind of clustering algorithm for a satellite lightning imager that operates in real time with stringent requirements over specific resources.

An introduction with an overview of orbits, agencies, satellites and missions are made to describe the history of meteorological space missions.

An overview of the *MTG* series and its main instrument, the Lightning Imager, is presented to explain the function of clustering algorithm.

At first, a High Level MATlab model is implemented for the design of the algorithm. This model was used to verify that all functional and output requirements are met, to find the best strategy of clustering and have a first running prototype. Furthermore, this model produces statistic, e.g. the number of windows over a number of test-cases..

Then, the hardware architecture is designed and a Bit-true MATlab model is implemented. The Bit-True model is tested and the results are analyzed to confirm the compliance with the requirements. The Bit-true model also serves for test-vector generation, to use for VHDL model verification.

In the next step, a VHDL prototype is created on the basis of the Bit-True MATlab model. Every block is “translated” in VHDL and the whole algorithm is controlled by a state machine with signals added for HW implementation.

The hardware is optimized with the parallelization strategy to improve the throughput keeping in mind the hardware resources utilization. Four instances of the clustering algorithm must fit in the RTAX4000S.

The final step is the prototype optimization of all blocks to respect all constrains and an analysis of timing and area for the top level block.

Chapter 5

References

[1] http://spaceradioandmore.blogspot.it/2009/02/le-orbite-dei-satelliti.html

[2] http://www.meteotrentino.it/analisiMM/I_satelliti_meteorologici.pdf

[3] http://it.wikipedia.org/wiki/EUMETSAT

[4] http://en.wikipedia.org/wiki/Meteosat

[5] http://www.eumetsat.int/website/home/Satellites/PastSatellites/index.html

[6] EUMETSAT Satellite Program.pdf, Marianne Konig & many colleagues.

[7] MSG's GERB Instrument.pdf, H-J Luhmann.

[8] 4.2 Koenemann EUMETSAT GEO Systems.pptx, Joaquin Gonzalez on behalf of Ernest Koenemann.

[9] 4.Biron_AGILE_10th_WS.pdf, Daniele Biron.

[10] MTG_LI-Finke-stw.pdf, Ullrich Finke, Jochen Grandell and Rolf Stuhlmann.

[11] MTG-LI_Status_update_stuhlmann.pdf, Rolf Stuhlmann.

[12] 2012_Lorenzini_Optical Design of the Lightning Imager for MTG.pdf, S. Lorenzini, R. Bardazzi, M. Di Gianpietro.

[13] Meteosat Third Generation Lightning Imager, MTG-LI_Biron_introduction.pdf, Daniele Biron.

[14] MTG-LI_CWG_Grandell.pptx, Jochen Grandell.

[15] MTG-LI_Status_update-garndell.pdf, Jochen Grandell, Marcel Dobber, Hartmut Höller and Rolf Stuhlmann.

[16] Meteosat.pdf.

[17] www.rational-systems.net/files/eum/devweb/v250/page_content_mfg_mission_overview.html

[18] http://www.eumetsat.int/website/home/Satellites/CurrentSatellites/Meteosat/index.html

[19] http://www.eumetsat.int/website/home/Satellites/FutureSatellites/MeteosatThirdGeneration/MTGDesign/index.html

[20] http://it.wikipedia.org/wiki/Orbita_terrestre_bassa

[21] http://www.arianespace.com/news-mission-update/2012/912.asp

[22] directory.eoportal.org/web/eoportal/satellite-missions/m/meteosat-first-generation

[23] http://directory.eoportal.org/web/eoportal/satellite-missions/m/meteosat-second-generation

[24] 2012_MTG-LI_Biron_introduction.pdf, Daniele Biron.

[25] http://www.actel.com/documents/RTAXS_DS.pdf.