Università degli Studi di Perugia

Master Thesis

A Flexible simulation and verification framework for next generation hybrid pixel readout chips in High Energy Physics

Author:
Sara Marconi

Supervisor:
Ph.D. Eng. Pisana Placidi

Assistant supervisors:
Ph.D. student Elia Conti

Eng. Jorgen Christiansen

A thesis submitted in fulfilment of the requirements for the degree of Electronics and Telecommunication Engineering

in the

Electronics Research Group
Engineering Department

Academic year 2012/2013

CERN-THESIS-2014-066
//2014

-

“Strive for perfection in everything you do. Take the best that exists and make it better. When it does not exist, design it.”

Sir. Henry Royce

Acknowledgements

I would like to express my gratitude to my supervisor Pisana Placidi for the continuous help and for the useful comments and remarks from the very beginning up to the end of this work.
Moreover, this paper would not have been possible without my assistant supervisor Jorgen Christiansen and his precious teachings and guidance through the learning process of this master thesis.
Furthermore I would like to thank the Ph.D. student Elia Conti for introducing me to the topic as well as for the (not only) technical support on the way. Also, I am very grateful to all my supervisors and to all the other people, between whom Andrea Scorzoni and Gianmario Bilei, that have helped me in having the opportunity of working in a international and stimulating environment.
In my daily life I have been pleased by a friendly and cheerful group of students and professionals from building 14 (that can not all be listed), especially Sebastian, Manoel, Marco, Marcos and Matteo.
Also I would like to thank my exceptionally close high school and university friends Alessandra, Angela, Federica, Giulia and Valentina who are always there for me in the various stages of my life (through thick and thin!). I cannot leave apart all the other university friends that have made the last years pleasant even in some tought moments.
Last, but not least, I would like to thank my family members, especially my parents, my sister Giulia, my aunt Manuela and my grandparents that have always been there for me every step of the way and have supported me through all of my decisions.

Contents

List of Figures

List of Tables

Abbreviations

ADC	**A**nalog-to-**D**igital **C**onverter
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
ALICE	**A** **L**arge **I**on **C**ollider **E**xperiment
ATLAS	**A** **T**oroidal **L**HC **A**pparatu**S**
BX	**B**unch crossing
CAE	**C**omputer-**A**ided **E**ngineering
CERN	European Organization for Nuclear Research
CMOS	**C**omplementary **M**etal **O**xide **S**emiconductor
CMS	**C**ompact **M**uon **S**olenoid
DAC	**D**igital-to-**A**nalog **C**onverter
DAQ	**D**ata **A**c**Q**uisition
DUT	**D**esign **U**nder **T**est
EDA	**E**lectronic **D**esign **A**utomation
EOC	**E**nd **O**f **C**olumn
ESL	**E**lectronics **S**ystem **L**evel
FIFO	**F**irst **I**n **F**irst **O**ut
FSM	**F**inite **S**tate **M**achine
HEP	**H**igh **E**nergy **P**hysics
HDL	**H**ardware **D**escription **L**anguage
HDVL	**H**ardware **D**escription and **V**erification **L**anguage
IC	**I**ntegrated **C**ircuits
IO	**I**nput **O**utput
LHC	**L**arge **H**adron **C**ollider
OOP	**O**bject **O**riented **P**rogramming
OSCI	**O**pen **S**ystem**C** **I**nitiative

MAPS	**M**onolothic **A**ctive **P**ixel **S**ensors
MSIE	**M**ulti-**S**napshot **I**ncremental **E**laboration
OVM	**O**pen **V**erifiication **M**ethodology
PC	**P**ixel **C**hip
PUC	**P**ixel **U**nit **C**ell
RNM	**R**eal **N**umber **M**odeling
ROC	**R**ead-**O**ut **C**hip
RTL	**R**egister **T**ransfer **L**evel
SV	**S**ystem **V**erilog
TL	**T**ransaction **L**evel
TLM	**T**ransaction **L**evel **M**odeling
TOA	**T**ime **O**f **A**rrival
TOT	**T**ime **O**ver **T**hreshold
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **Circuits**
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage
UML	**U**nified **M**odelling **L**anguage
UVM	**U**niversal **V**erifiication **M**ethodology

Introduction

Next generation pixel detector systems and ASICs will have to face many technical challenges, including smaller pixels to resolve tracks in boosted jets, much higher hit rates (1-2 GHz/cm^2), unprecedented radiation tolerance (10 MGy), much higher output bandwidth, and low power consumption. Their electronics will also have to work reliably for years under extremely hostile radiation conditions. A collaboration, named RD53, has started to design the next generation of hybrid pixel readout chips to enable the phase 2 pixel upgrades of the ATLAS (A Thoroidal LHC ApparatuS) and CMS (Compact Muon Solenoid) expertiments. This formal collaboration has participating universities and research institutes from Europe and USA. In detail, in this thesis I worked in the implementation of a pixel verification and simulation framework for the optimization of the architecture of next generation pixel chips.
In the remainder of this section I will describe the contents and the organization of the thesis. In Chapter 1 an introduction on the state of the art of such detectors and on next generation requirements will be introduced. Moreover, a big challenge will be the growing complexity of such systems, both at the level of the single blocks specifications and at the top level. This requires a large effort to design and qualify front-end electronics following a modular approach and to encourage re-usability between different experiments for saving time and resources. For those reasons it is necessary also in High Energy Physics applications to start to look into system level design flow and new high level languages largely used in industry: a summarily description of SystemC, SystemVerilog and of the Universal Verification Methodology will be provided in Chapter 2. In order to prove that the tools used are able to deal with the system target complexity, initial study on scalability and an evaluation of some commercial simulation tools have been done and are reported in Chapter 3. Parallel design of singular blocks, of the system top

level and of a framework capable of simulating it, optimizing its architecture and finally verifying it is encouraged for facing the design complexity. In Chapter 4, the work done for obtaining a model for the basic building block of a pixel chip is described and the different architectures that have been simulated at the system level in the framework are introduced. The overall structure of the simulation and verification environment will be presented in Chapter 5 and details will be provided on its interfaces to the chip and on the role of each verification component. A specific one, in charge of generating meaningful hit stimuli, will be the focus of Chapter 6. Finally, in Chapter 7, it will be presented a test case where the developed framework and hit generator have been used for the optimization of buffering architectures. Simulation results will be compared with the ones obtained from an implemented statistical/analytical model of the same buffering architecture.

Chapter 1

Electronic circuits in pixel detectors for High Energy Physics

The notion of pixel comes from image processing applications and it describes the smallest discernible element in a given process or device. A pixel detector is therefore a device able to detect an image and the size of the pixel corresponds to the granularity of the image. For high energy physics (HEP) applications, the focus is put on pixel detectors which are particularly fast and able to detect high-energy particles and electromagnetic radiation. Pixel detectors are used in High Energy Physics (HEP) applications as radiation sensors that are distributed in complex systems used for tracking high energetic particles. Accelerators generate elementary particle collisions at a rate of 10–100 MHz, with particles emerging from every collision. Some rare, but interesting particles live about 1 ps and then decay into a few daughter particles [1]. An example of such a decay is sketched as in Figure 1.1, the collision vertex (V) and the decay vertex (D) are also indicated.

Therefore they are high granularity tracking detectors which provide unambiguous and precise 3D measurements in the harsh environment close to the interaction point. The charged particles traversing the sensor releases electron-hole pairs that can be separated and drifted to collection electrode with high voltage biasing ($\sim$100V).

The main requirements of such detectors are:

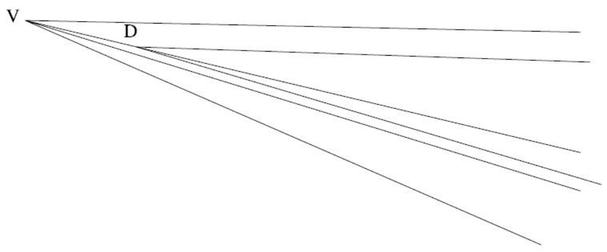


FIGURE 1.1: Topology of a short-lived particle decay, with ordinary particles emerging from the same collision [1].

- space and time resolutions for unambiguous detection of short-lived particles;
- capability of coping with the increasing interaction rates and energies of modern particle accelerators;
- high level of radiation hardness, since they are placed in an extremely hostile environment;
- fast readout (depending on technology);
- selection of events of interest.

Pixel detectors are used at the heart of the trackers in the experiments of the Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN), e.g: A Large Ion Collider Experiment (ALICE), A Thoroidal LHC ApparatuS (ATLAS) and Compact Muon Solenoid (CMS). They are also key elements for the upgrades, as will be more presented more in detail in the following sections.

Different approaches exist for pixel detectors when it comes to the way the sensible part (i.e. the sensor) is linked to the readout Application Specific Integrated Circuit (ASIC). As regards the sensor material, there are potentially multiple options: planar silicon, 3D silicon, diamond (both planar and 3D), CMOS sensor, etc. The material most commonly used at the state of the art for pixel detectors is silicon. Some groups have looked into the possibility of building both electronics and sensor in the same technological process, integrating the charge generation volume into the ASIC itself. This leads to the so-called Monolithic Active Pixel Sensors (MAPS), whose basic block is shown in Figure 1.2. Such an approach aims to reach lower cost, higher resolution, lower mass but has also drawbacks, starting from the fact that silicon substrate used for electronics chips in most cases is not ideal as silicon detector, where low resistivity would be the optimal choice. The connection to the detector in the substrate is also critical, as also getting sufficient charge collection (and speed) efficiency.

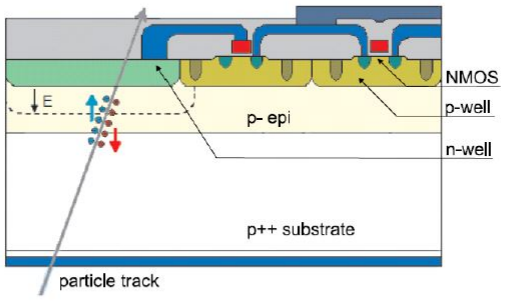


FIGURE 1.2: Traditional MAPs building block [2].

For those reasons, at the state of the art the baseline approach used for the running detectors are the Hybrid Pixel Detectors (HPD). They are composed of a sensor (typically a semiconductor photodiode), where information related to the passage of a particle is generated, decoupled from a high density ASIC circuit for data readout and processing. The adjective “hybrid” comes from the fact that those two blocks are fabricated separately and are then joined together through a process called bump bonding, as shown in the basic building block shown in Figure 1.3. Such a process is characterized by very high cost, even if there is hope that it could improve in the future profiting from bonding techniques from 3D technologies. Hybrid pixels are capable of standing high radiation levels and are also reliable for high rate applications.

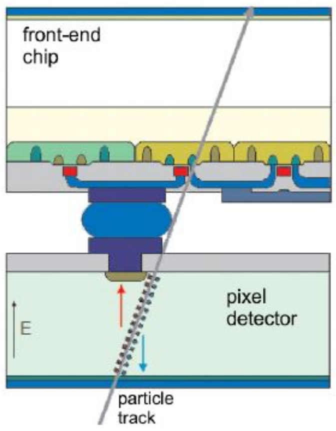


FIGURE 1.3: Hybrid Pixel Detectors: single building block [2].

Matrices made of hundreds of thousands of pixel cells can be implemented in few square centimeter surfaces thanks to planar integration technology. Such a matrix of elementary

Pixel Unit Cells (PUCs) is called Pixel Chip (PC): it can be divided into an active area which contains a repetitive matrix of nearly identical pixels and the chip periphery from where the active part is controlled, where data are buffered and global functions common to all pixels are located, as shown in Figure 1.4.

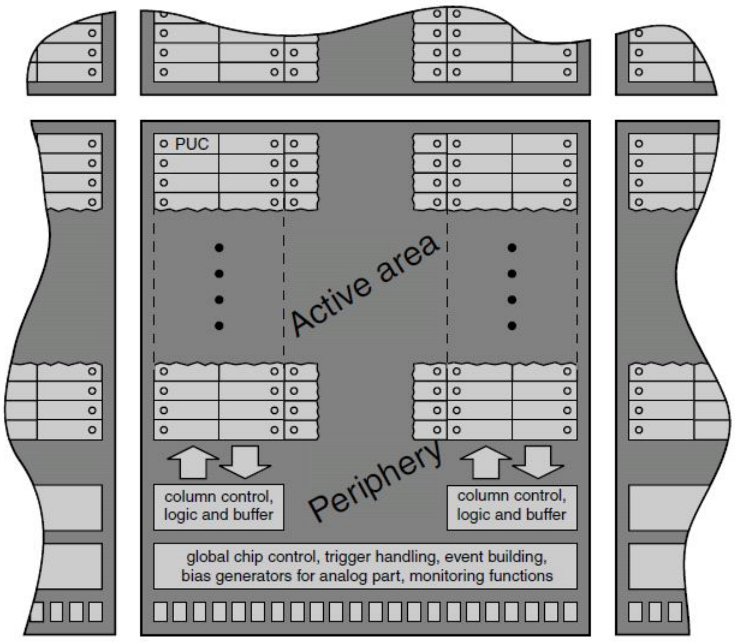

PUC
Active area
Periphery
column control, logic and buffer
column control, logic and buffer
global chip control, trigger handling, event building, bias generators for analog part, monitoring functions

FIGURE 1.4: Generic pixel detector: active area and periphery circuitry [1].

Matrices are then joined together and arranged in complex modules that cover the inner surface of the collider. Hybrid pixel detector modules are made from multiple pixel Read-Out Chips (ROC) bump bonded to a single pixel sensor. Readout and control signals plus power are connected from the ROCs through wire-bonding to a thin and light printed circuit board glued on the back side of the pixel sensor. Just to give an example, in the CMS tracker sensors are arranged in concentric cylinders around the interaction region of the LHC beam. A sketch of this structure can be seen in Figure 1.5. The center of the tracker is at the left-bottom corner of the drawing, the horizontal axis, to which the detector has a cylindrical symmetry, points along z, and the vertical axis points long the radius r. The LHC beams are parallel to the z axis [9].

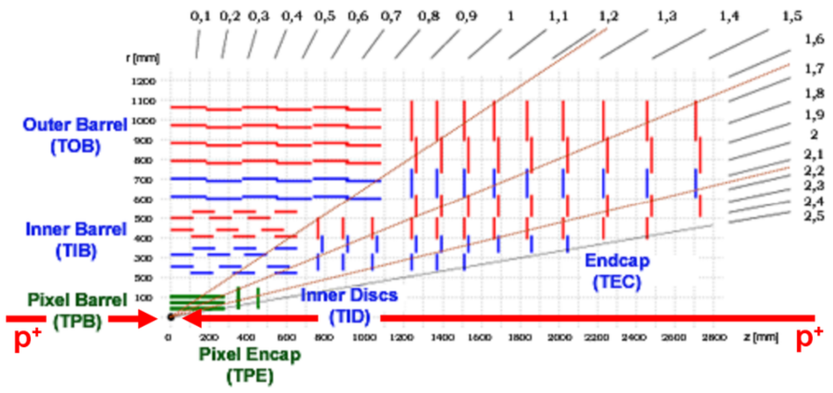


FIGURE 1.5: Quarter of the r-z slice of the CMS tracker.

1.1 Basics on hybrid pixel detector design

Some basic concepts which are important in order to understand the features and the design challenges of HPDs are mentioned in this part. For more detailed descriptions references that can be found in literature will be provided.

1.1.1 The Front-end electronics

The classical approach for the analog front-end of a single unit cell is made of a combination of pre-amplification circuitry and a discriminator(whose threshold can be set through a Digital-to-Analog Converter) that produces a binary output, to be read out from the digital front-end. Such a generic structure, with an optional shaping filter in addition, is shown in Figure 1.6. Details on this architectures can be found in [1].

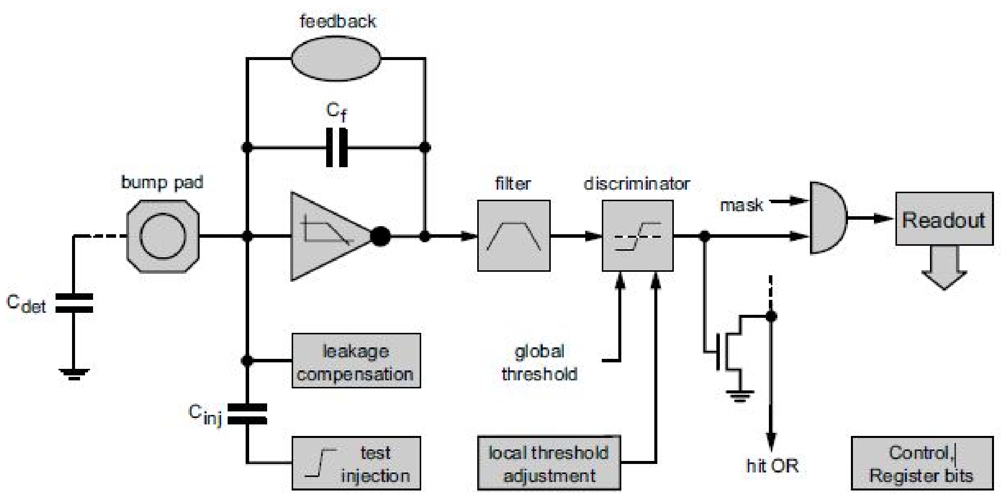


FIGURE 1.6: Components of a generic PUC [1].

In order to perform the analog to digital conversion of the charge information different approaches can be evaluated, i.e. the use of a Analog-to-Digital Converters (ADC, individual or shared between different PUCs, with just one or more bits) or a Time Over Threshold (TOT) measurement. The key concept behind each of them can be observed in Figure 1.7. While the first is a quite generic way of achieving analog-to-digitl conversion, the latter is particularly used for HEP applications: the TOT is the time during which the signal is higher than the discriminator threshold. It can be measured using a clock signal and a digital counter.

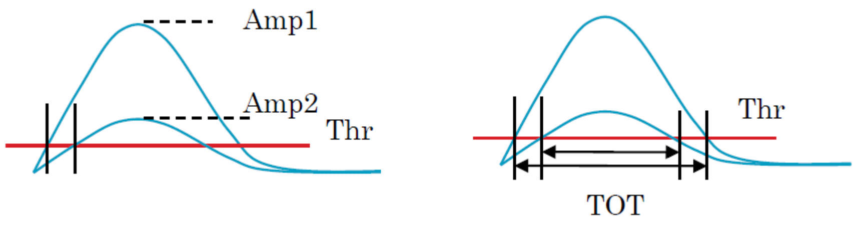


Figure 1.7: Comparison between charge digitization methods: ADC and TOT approaches [3].

From the Figure 1.7 it can also be observed the time-walk problem: in the analog front-end particles that deposit higher charge produces a faster response than those that produce a lower charge. The response time of the discriminator (in combination with the rising time of the pre-amplifier) is crucial in applications where the arrival time of a signal must be detected with high precision. In the case of very low-energy particles the time walk can become higher than the minimum required time resolution, causing the particle arrival to be associated to the following cycle. Compensation strategies are then needed in order to correct it. For details on the concept of response time of the discriminator the reader can refer again to [1].

1.1.2 Readout architectures

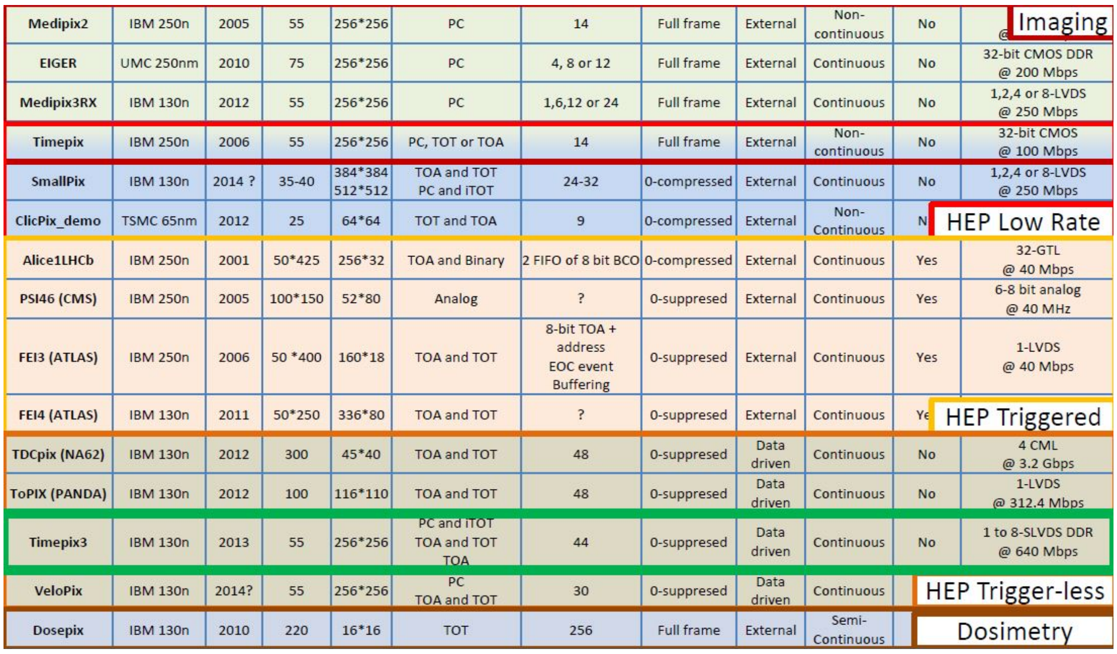
The readout architecture of the ASICs of HPDs depends very much on the target application. Position, time and possibly the corresponding pulse amplitude, of all hits belonging to an interaction must be usually provided in HEP. This requires a timing precision at least equal to the bunch crossing interval (25ns) for the detectors at LHC. An exhaustive description of the readout architectures is beyond the scope of this work and material on existing detectors can be found in [1] and [10]. Just an introduction on the distinction between trigger-less and triggered architectures will be herein provided since it is a key concept to be introduced for the understanding of this work. Some experiments with low input rates allow the readout of each event to be done immediately after the interaction, while when higher rates are involved on-detector data reduction is needed in order to obtain a feasible data rate towards the Data Acquisition (DAQ). For this purpose, usually a trigger signal, that is generated by other parts of the detector (e.g. calorimeters or muon chambers), is used for selection of hits of interest. Since the delay between the time when a particle is detected and when the trigger signal is provided to the digital logic that performs the selection can be high (latency is currently in the order of around 100 interactions and will be incremented in the future detectors, see Table 1.2 in the following section), some storage logic is required and the limited buffering space available is a source of loss of hits. Also the choice on in which part of the pixel chip to locate the buffers (End Of Column (EOC), single PUC, region of certain number of PUCs) leads to different architectures. Some details on existent and planned buffering strategies to be used can be again found in Table 1.2 in the subsequent section.

1.2 State of the art and future of hybrid pixel detectors

Running detectors are all based on the hybrid approach and this is still the baseline choice for next generation pixel chips as well. Indeed, even if MAPs offer many advantages (smaller pixels, lower capacitance, less material, lower cost, etc...), hybrid detectors are still considered the most reliable way of coping with very high particle rates and fluence [11] that add additional challenges to the designers (e.g. radiation hardness requirements, need to integrate high rate logic and large buffers, cross-talk problems coming from in-pixel logic).
The LHC community tends to refer to pixel readout chips in terms of generations. The present ATLAS and CMS detectors contain the so-called 1st generation chips. 2nd generation chips have been designed and fabricated and will become operative for the so-called phase0/1 upgrades, after the end of the first Long Shutdown, that is taking place during the years 2013-2014 [12]. The longer-term plan would lead to the 3rd generation chips, that are supposed to become operative for the phase 2 upgrade, that will follow the second Long Shutdown (2017-2018). A straightforward classification of the state of

the art of hybrid pixel ASICs, taken from a seminar on the Timepix3 pixel readout chip [13], can be observed in Table 1.1.

Table 1.1: Hybrid pixel ASICs classification [2].



Medipix2	IBM 250n	2005	55	256*256	PC	14	Full frame	External	Non-continuous	No	@ Imaging
EIGER	UMC 250nm	2010	75	256*256	PC	4, 8 or 12	Full frame	External	Continuous	No	32-bit CMOS DDR @ 200 Mbps
Medipix3RX	IBM 130n	2012	55	256*256	PC	1,6,12 or 24	Full frame	External	Continuous	No	1,2,4 or 8-LVDS @ 250 Mbps
Timepix	IBM 250n	2006	55	256*256	PC, TOT or TOA	14	Full frame	External	Non-continuous	No	32-bit CMOS @ 100 Mbps
SmallPix	IBM 130n	2014 ?	35-40	384*384 512*512	TOA and TOT PC and iTOT	24-32	0-compressed	External	Continuous	No	1,2,4 or 8-LVDS @ 250 Mbps
ClicPix_demo	TSMC 65nm	2012	25	64*64	TOT and TOA	9	0-compressed	External	Non-Continuous	N	HEP Low Rate
Alice1LHCb	IBM 250n	2001	50*425	256*32	TOA and Binary	2 FIFO of 8 bit BCO	0-compressed	External	Continuous	Yes	32-GTL @ 40 Mbps
PSI46 (CMS)	IBM 250n	2005	100*150	52*80	Analog	?	0-suppresed	External	Continuous	Yes	6-8 bit analog @ 40 MHz
FEI3 (ATLAS)	IBM 250n	2006	50 *400	160*18	TOA and TOT	8-bit TOA + address EOC event Buffering	0-suppresed	External	Continuous	Yes	1-LVDS @ 40 Mbps
FEI4 (ATLAS)	IBM 130n	2011	50*250	336*80	TOA and TOT	?	0-suppresed	External	Continuous	Ye	HEP Triggered
TDCpix (NA62)	IBM 130n	2012	300	45*40	TOA and TOT	48	0-suppresed	Data driven	Continuous	No	4 CML @ 3.2 Gbps
ToPIX (PANDA)	IBM 130n	2012	100	116*110	TOA and TOT	48	0-suppresed	Data driven	Continuous	No	1-LVDS @ 312.4 Mbps
Timepix3	IBM 130n	2013	55	256*256	PC and iTOT TOA and TOT TOA	44	0-suppresed	Data driven	Continuous	No	1 to 8-SLVDS DDR @ 640 Mbps
VeloPix	IBM 130n	2014?	55	256*256	PC TOA and TOT	30	0-suppresed	Data driven	Continuous		HEP Trigger-less
Dosepix	IBM 130n	2010	220	16*16	TOT	256	Full frame	External	Semi-Continuous		Dosimetry

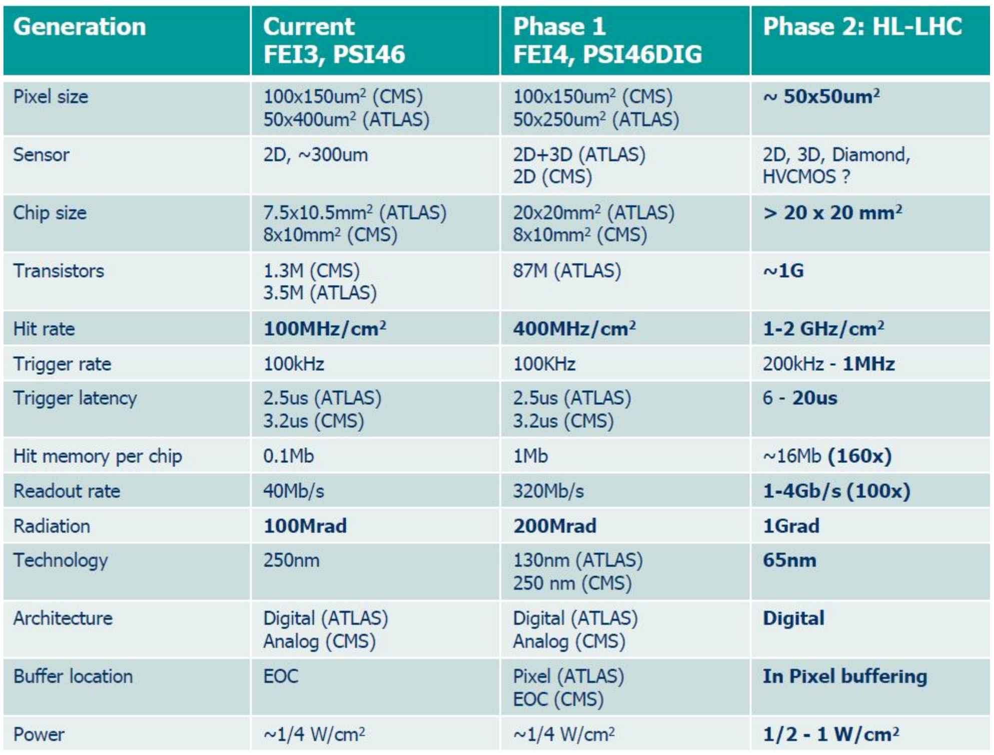
Hybrid pixel ASICs are therein classified depending on the application: imaging, High Energy Physics with Low Rate, High Energy Physics with triggering (Triggered) or without triggering (Trigger-less), dosimetry. The target application affects significantly the design challenges and choices. For example, while buffering resources are a more a critical aspect for triggered than trigger-less architectures, since all the incoming data need to be read out in the latter case.

As regards the technology node of the detectors at the state of the art, it can be noticed still in Table 1.1 that 1st generation chips like the current ATLAS and CMS chips [14, 15] are fabricated in a 250 nm technology. What can be considered the state of the art are chips under construction or at advanced design stage for ATLAS and CMS, as well as the Medipix and Timepix chips [16, 17]. As concerns the technology used, the CMS pixel upgrade is based on a modified version of the 1st generation chip in 250 nm CMOS [18], while the other examples are all 2nd generation chips in 130 nm CMOS. Between those ones it is worth to mention the FE-I4 chip being used to build the Insertable B-Layer upgrade of ATLAS [19].

1.2.1 RD53 Collaboration

Now the attention will be focused on next generation pixel chips. The Phase 2 pixel detector systems and ASICs will have to support unprecedented high hit rates (∼1-2 GHz/cm^2) and radiation levels. The pixel detector electronics will have to work reliably for 10 years under extremely hostile radiation conditions of up to 10 MGy and 10^{16} hadrons/cm^2. This is an unprecedented radiation level for electronics of such complexity. A large part of the required design efforts will be assigned to design and qualify front-end electronics to work reliably under such harsh radiation conditions. Besides, they are meant to achieve better resolution thanks to the use of smaller pixels and to keep power consumption the same or possibly lower. The increase in trigger latency (of a factor of ∼10) combined with the growing hit rate will bring to ∼100 times higher buffering requirements. A substantial increase (factor of 100) in the readout data rate will also have to be faced by future designers. A comparison of the 3rd generation chips with the state of the art can be observed in Table 1.2 for the ATLAS and CMS pixels: some parameters of crucial importance are highlighted.

Table 1.2: Pixel chip generations [7].



Generation	**Current FEI3, PSI46**	**Phase 1 FEI4, PSI46DIG**	**Phase 2: HL-LHC**
Pixel size	100x150um^2 (CMS) 50x400um^2 (ATLAS)	100x150um^2 (CMS) 50x250um^2 (ATLAS)	**~ 50x50um^2**
Sensor	2D, ~300um	2D+3D (ATLAS) 2D (CMS)	2D, 3D, Diamond, HVCMOS ?
Chip size	7.5x10.5mm^2 (ATLAS) 8x10mm^2 (CMS)	20x20mm^2 (ATLAS) 8x10mm^2 (CMS)	**> 20 x 20 mm^2**
Transistors	1.3M (CMS) 3.5M (ATLAS)	87M (ATLAS)	**~1G**
Hit rate	**100MHz/cm^2**	**400MHz/cm^2**	**1-2 GHz/cm^2**
Trigger rate	100kHz	100KHz	200kHz - **1MHz**
Trigger latency	2.5us (ATLAS) 3.2us (CMS)	2.5us (ATLAS) 3.2us (CMS)	6 - **20us**
Hit memory per chip	0.1Mb	1Mb	~16Mb **(160x)**
Readout rate	40Mb/s	320Mb/s	**1-4Gb/s (100x)**
Radiation	**100Mrad**	**200Mrad**	**1Grad**
Technology	250nm	130nm (ATLAS) 250 nm (CMS)	**65nm**
Architecture	Digital (ATLAS) Analog (CMS)	Digital (ATLAS) Analog (CMS)	**Digital**
Buffer location	EOC	Pixel (ATLAS) EOC (CMS)	**In Pixel buffering**
Power	~1/4 W/cm^2	~1/4 W/cm^2	**1/2 - 1 W/cm^2**

In order to cope with those unprecedented requirements (a detailed list can be found in [6]), that are similar for the ATLAS and CMS experiments, a new cross experiment

R&D collaboration, named RD53, that aims to develop the next generation of pixel ROCs has started. This formal collaboration involving 18 participating institutes from both ATLAS and CMS experiments is focused on the required radiation testing/qualification of the proposed Integrated Circuit (IC) technology and developing the required IC to build the next generation pixel ASIC. Large pixel chips are going to be designed and prototyped within this collaboration framework. In particular, the collaboration is focused on different subjects, for each of which a dedicated working group has been created: qualification of a technology able to survive in the harsh radiation environment, definition of a top-level architecture, development of a simulation framework, development of rad-hard Input Output (IO) cells, evaluation and design of the analog front-end, design of specific IP blocks.

As regards the pixel size, two basic ones of 50x50 μm^2 and 25x100 μm^2, that bring to the same ROC area, have been found to be the most promising compromise between significantly improved tracking performance and what can realistically be implemented in a next generation pixel readout chip and pixel sensors. This pixel size is also compatible with available bump bonding technologies. Elongated pixels (25x100 μm^2) can be optimal in some parts of the pixel detector (e.g. end of barrel layers with inclined tracks) and square pixels (50x50 μm^2) can be optimal in other parts (middle of barrel and forward disks with near perpendicular tracks). With an appropriate bump-bonding pattern the same ROC can be used for the two different pixel aspect ratios.

As concerns the technology node to be used, currently 65 nm is currently being investigated by both the CMS and ATLAS phase 2 pixel upgrade projects as baseline technology, even if it needs further testing for higher radiation levels. This technology is quite mature and long-term available in industry and thanks to the scaling process it can also guarantee the higher speed and density (the latter is vital for obtaining smaller pixels and to ensure more logic capability). Going to smaller technology nodes (e.g. 40 nm or 28 nm are existing ones) has also its drawbacks: higher costs of engineering and of Mixed-Project Wafer runs together with a higher design complexity. The 65 nm technology is therefore considered a good compromise.

As regards the object of this work, it can be collocated in the working group (WG3) that is dedicated to the development of a pixel verification and simulation framework for the actual optimization of the architecture of next generation pixel chips.

Chapter 2

System Description and Verification Languages

In the industrial context the increasing complexity of system-on-chips (SoCs) and time-to-market pressures, have lead to an attempt to raise the abstraction level of the description of the systems [20]. By designing at the system level, it becomes possible for hardware engineers to avoid gate-level semantics, to manage growing system complexity, to speed up simulation, to support system-level verification and possibly HW/SW co-design. All this advantages enable design productivity to increase, development costs and risks to reduce and time-to-market to accelerate.

Even in HEP applications, the evolution to new generation pixel chips leads to an increase of complexity. For this reason the community has started to look into available and appropriate languages and tools to enable simulations at both very high level and detailed gate level using well established toolkit components for all the major Computer-Aided Engineering (CAE) tool suppliers [6]. A brief introduction about existent high-level design languages and tools is provided in this chapter and the motivations of the decisions that have been currently taken are described.

2.1 SystemC and Transaction Level Modeling

SystemC is defined and promoted by the Open SystemC Initiative (OSCI), and has been approved by the IEEE Standards Association as IEEE 1666-2005 [21]. It has increasingly been used for system-level modeling, architectural exploration, functional verification, and high-level synthesis. It uses a set of C++ classes enabling a designer to simulate concurrent processes, each described using C++ syntax and, if needed, Object Oriented Programming (OOP) features. In certain aspects, SystemC deliberately resembles Hardware Description Languages (like VHDL and Verilog), but is more appropriate to describe it as a system-level modeling language.
This language is often associated with the concept of Transaction Level Modeling (TLM). In such models, the details of communication are separated from the details of computation components. Communication is not modeled through signals resembling the HW, but by high level channels where information is passed through transactions. Details of communication and computation are hidden in a TLM and their addition is postponed to following steps of the design. This is claimed to be an approach that speeds up simulation and that allows architecture exploration to be easier and faster [22]. On the other hand, it can be observed that such models than need to be detailed in order to obtain a synthesizable description of the chip and no gate-level description can be used. So a clear design flow and appropriate synthesis design tools need to be available and proved to be effective.
Even if not excluded as a possible system description level language, it has not been chosen in the context of this work. The reason of the choice that has been made will be clarified in the following sections.

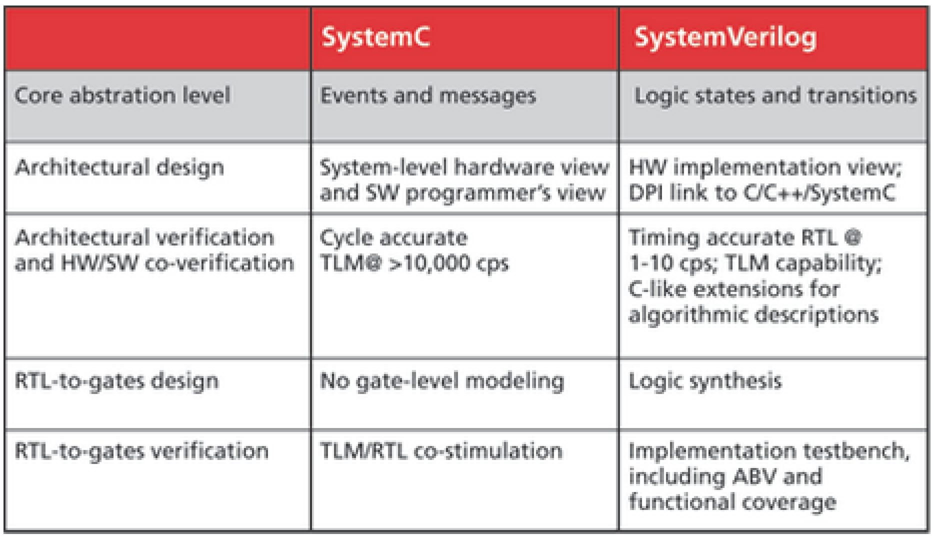
2.2 SystemVerilog

Another language that is spreading in the industry for system-level modeling is SystemVerilog (SV). Differently from SystemC, it has been developed as an extension of an established Hardware Description Language (HDL). Indeed, it has its bases on IEEE 1364-2005 Verilog Standard [23] with which many additional features (coming from SUPERLOG, C, C++ and VHDL languages, along with OVA and PSL assertions), are integrated in order to provide improved specification of design,

conciseness of expression, together with unification of design and verification. Like SystemC, SystemVerilog supports Object Oriented Programming, that adds both the possibility of obtaining high level description of systems and a new approach for verification. For this reason it is often referred as a Hardware Description and Verification Language (HDVL). What has to be underlined is that with SystemVerilog these multiple levels of description can not only co-exist, but also be developed using a unique language.
It is so clear how in the context of RD53, where this work can be collocated, for the development of a dedicated pixel verification and simulation framework (capable of simulating alternative pixel chip architectures at increasingly refined level [6]), SystemVerilog has been taken into account as a particularly valuable option.

A comparison between some fundamental capabilities of SystemC and SystemVerilog is reported in Table 2.1.

Table 2.1: SystemC and SystemVerilog complementary design capabilities [8]



	SystemC	SystemVerilog
Core abstration level	Events and messages	Logic states and transitions
Architectural design	System-level hardware view and SW programmer's view	HW implementation view; DPI link to C/C++/SystemC
Architectural verification and HW/SW co-verification	Cycle accurate TLM@ >10,000 cps	Timing accurate RTL @ 1-10 cps; TLM capability; C-like extensions for algorithmic descriptions
RTL-to-gates design	No gate-level modeling	Logic synthesis
RTL-to-gates verification	TLM/RTL co-stimulation	Implementation testbench, including ABV and functional coverage

2.2.1 SystemVerilog for Design

A brief overview on SV capabilities when it comes to chip design will be herein provided since they have been used for the developed model of the pixel chip, as it will be described in the following parts. It is herein reminded that the design can be described at different levels (not all directly synthesizable, since just a subset of the SV constructs are):

- Gate Level (GL): describes the logic gates and the interconnections between them. It it is always synthesizable;
- Register Transfer Level(RTL): a model that describes the data flow between registers and how a design processes these data. It is normally done using synthesizable code;
- Behavioural level: a model that implements a design algorithm in high-level language, describing how a particular design should respond to a given set of inputs. It is not in general guaranteed to synthesize (this implies that the designer is aware of synthesizable SV constructs);
- TLM: this approach is intended to be used for high level system description and cannot be directly synthesized.

Among the new features, SV offers conciseness of expression achieved through the addition of coding shortcuts from C, the simplification of port expressions, and the collection of related data together. Those new capabilities have been used in the work that has been done for the description of the Design Under Test (DUT) part of this work. The main improvements when comparing it with Verilog are listed below [24]:

- extended built-in data types and enhanced ways of specifying literal values;
- user-defined and enumerated data types;
- support for array, structures and unions;
- enhancement to Verilog procedural blocks and tasks/functions;
- enhancement to Verilog procedural statements;
- enhanced modelling of Finite State Machines (FSM);
- enhanced modelling of interconnections.

It is not purpose of this work to cover all of them, and it is recommended to make reference to [24] and to the Language Reference Manual (LRM) [25]. Just the ones that have been used in this project will be summarized if needed.

2.2.2 SystemVerilog for Verification

With designs getting bigger and more complex, verification is starting to take the most part of the design effort. In industry there is furthermore a strong need to reuse existent code, to use more efficient coding constructs and to have ways to measure verification progress. In this context, SystemVerilog itself provides many features to create complete verification environments at a higher level of abstraction than what is possible to achieve with a standard HDL. This makes also possible a verification approach that goes beyond traditional simulations based on directed testing. Some of the typical features of this HDVL that distinguish it from a HDL are summarized in the list below [26]:

- constrained-random stimulus generation;
- High level structures, especially Object Oriented Programming and Transaction Level Modeling;
- multi-threading and inter-process communication (IPC);
- assertions;
- functional coverage, that measures the progress of all tests in fulfilling the verification plan requirements.

Also in this case, it is not a goal of this work to give a complete description of all of them. When needed for the understanding of the following sections details will be provided.

Directed testing is a very automatic and unobtrusive way to collect test coverage, by measuring which tests have passed and which have failed, and performing failure analysis. Even if it makes easy to measure progress, protocol specific stimulus can often require much rework under architectural changes. Another significant problem with this approach is that only defects with specific features are normally found, while the methodology does not cover unforeseen bugs. Advanced RTL verification techniques start to look into use of constrained random stimuli, that enables testing to be capable of identifying more bugs and faster. At the same time it is difficult to see what has been actually tested because the testbench is by definition randomized. Coverage driven verification is added to visibly see what is tested and what is not during this randomization process. Functional coverage has been seen to produce huge amounts of coverage data: it can make difficult to analyze which design features belong to which coverage point. Coverage driven has for this reason some limitations that affect its usability and scalability. The newest approach is to use the so-called Metric Driven Verification that aims to use different "metrics" rather than just coverage, including: checks, assertions, software, and time-based data points. This can be achieved starting from a clear and organized verification plan, that helps to manage the wealth of data captured by all the tools involved in the execution of a verification project. It is goal of this (and of the future) work to achieve such an efficient, scalable, productive, and predictable verification process.

2.2.3 Universal Verification Methodology

On top of SV, specific verification methodologies have also been defined based on industry practices. The more recent ones are the Open Verification Methodology (OVM) and the Universal Verification Methodology (UVM), where the latter is becoming a mature standard used by a growing community. For getting started with UVM there are many available references: the UVM user guide [4] and online resources offered by industries (e.g. [27] are valuable and free ones). This book [28] can also be recommended since it provides with practical advices. Compared to standard SystemVerilog, UVM offers a set of more solid and documented base classes for all the building blocks of the environment, as it can be noticed in Figure 2.1. High configurability, highly customizable reporting features and the possibility of building reusable environments are other well-known UVM capabilities.

Components can be instantiated hierarchically and their operation is controlled through an extendable set of pre-defined phases that are executed in a strict order. These phases are potentially many, but the most relevant ones (that have been mostly frequently used) are the following:

- an initial *build_phase()*, used to construct various components and configure them;
- a *connect_phase()*, during which the components are linked one to the other;
- the actual *run_phase()*, when the parallel simulation of all the components takes place;

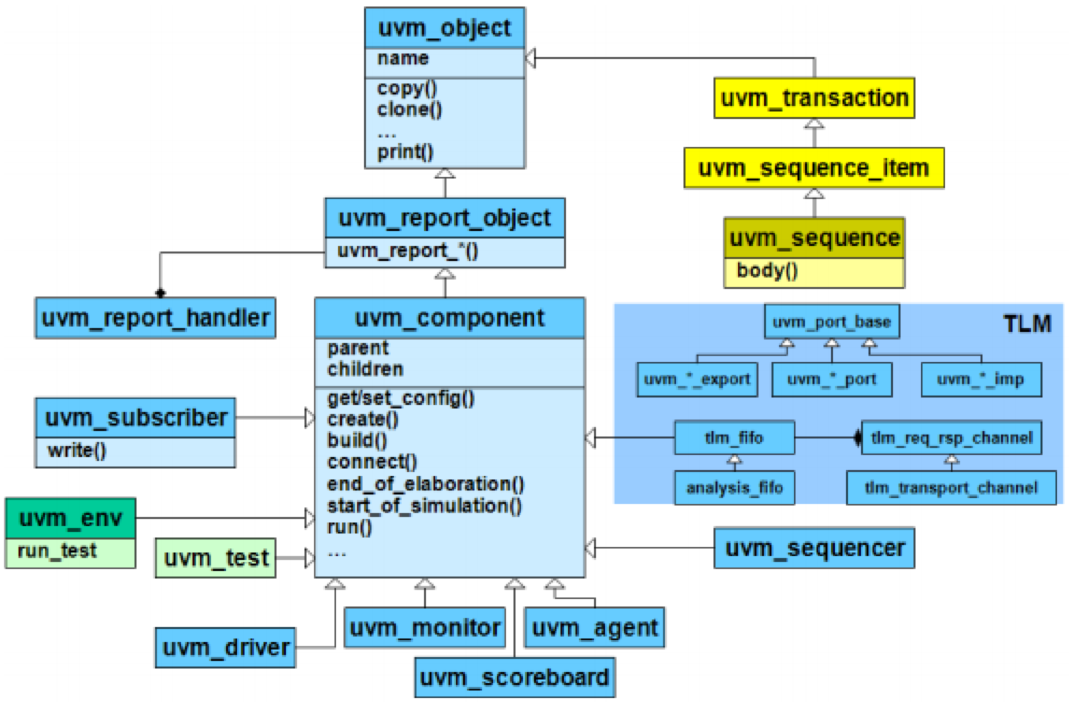


FIGURE 2.1: Partial UVM class library [4].

- the *report_phase()*, finally used for reporting the pass/fail status and for eventually dumping result of analysis performed in the environment.

Potentially the run phase itself could be furthermore divided in a quite long list of different phases, that are reported in the UVM Class Reference [29], but the use of those is not recommended for re-usability [28] since it is not a well-established approach and it is likely to change in the future.
From the experience gained working on this thesis, it can be reported that the initial learning phase of the UVM coding constructs has been rewarded by the pre-existent features offered by the methodology, as it has been partially introduced. Indeed, as an initial step in this work just self-defined SV classes had been used when building the environment (some examples of such components can be found in the last chapters of [26]), but they have clearly much less evolved features and they simplify much less the task of building a very configurable and generic environment if compared with the UVM ones. For those reasons it has been decided to adopt this advanced methodology in this work.
The description of the role of the different UVM components will not be herein provided, apart from what will be described in the context of the environment description.
What is anyway worth to be highlighted is the UVM capability of performing different

tests on a certain DUT reusing a unique verification environment for all of them. The test case-specific code is kept separate from the testbench and it is the code intended to be used by verification engineers that have not been developing the environment itself and may not know the details on how it has been designed. This aspect is of fundamental importance in industry where the team of “test-writers” engineers can be separate from the people that design the environment, but also in the context of a collaboration between multiple experiments and institutes like RD53.

Chapter 3

Evaluation of software tools

Every valid methodology requires the existence of tools capable of dealing with problems of growing complexity in a shorter time. For this reason, the Electronic Design Automation (EDA) tools have evolved during the years from simple simulators to complex frameworks capable of guiding the designer also in high-level choices [20]. As concerns the design verification role, SystemVerilog and UVM are widely used in industry and the largest EDA vendors have incorporated it into their mixed language HDL simulators. The three main and EDA vendors are:

- Cadence, with the simulation tool *Cadence Incisive* Cadence Incisive [30];
- Methor Graphics, whose HDL simulator is *Questa/Modelsim* [31];
- Synopsys, with the tool *Synopsys VCD* [32].

After an initial period during which the different vendors were proposing different verification methodologies, the birth of UVM has finally brought to a unique (and possibly compatible) framework.

It has to be reported that while working on the definition of the verification environment in some cases significant simulation speed problems have been hit. For this reason, before going on with the design of the simulation framework an initial study on scalability and also an evaluation of some commercial simulation tools has been performed. Some details on this study are reported in the sections of this chapter. It is anyway highlighted that the tool mainly used for this thesis has been Cadence Incisive, since it was the software more easily available in the context of the work.

3.1 Preliminary study on scalability

As anticipated, while developing the first versions of the verification environment significant simulation time issues have been hit. It is not purpose of this chapter to describe them in details, since just the latest version of the environment will be presented in chapter 5 . A quick overview on the problems that have been encountered will be anyway given in order to understand their entity and to introduce the expediences used to analyse and possibly solve them. The initial simulation had been run with a very preliminary and partial version of the environment, shown in Figure 3.1, which will not be described here. Its overall architecture should anyway become quite clear after the reading of the dedicated chapter.

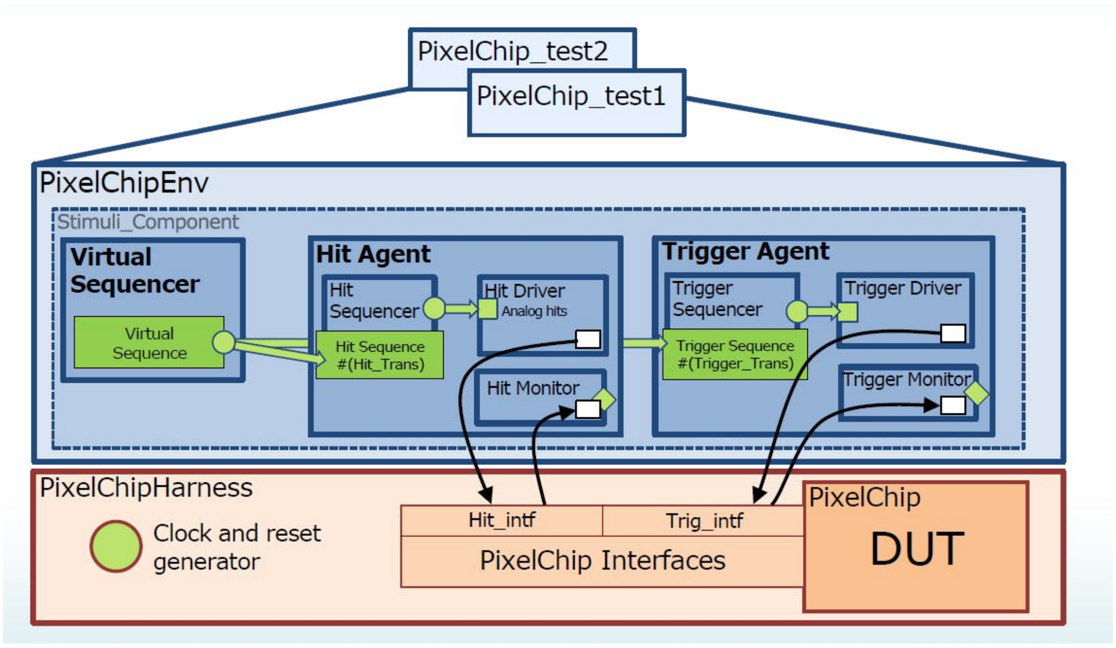


FIGURE 3.1: Initial (and partial) structure of the Verification Environment used for study on scalability

While starting to build the environment the first very simple approach used to generate random hits to be sent to the pixel chip was the following:

- every bunch crossing cycle a complete bi-dimensional matrix (since 1024x256 is an option mentioned in [6] this size has been used), with a correspondence 1:1 between the elements of the array and the pixels of the chip;
- the whole array has been randomized with a constrained probability of being hit.

Table 3.1: Simulation time results obtained from the initial version of the environment.

Number of bunch crossing cycles	Simulation time	CPU usage	Memory usage
100	143.0s (~2min30s)	99.0%	1134.4M total
10000	12217.1s (~3h30min)	100.0%	1134.4M total

Simulations with such a simple generation approach have been run with:

- DUT not actually instantiated in the simulation (further development was at the time still needed)
- All debug messages turned off since it has been observed that they cause simulation time to increase significantly

In order to study simulation performance a tool used in IC design for this purpose has been used, i.e. the so-called profiler. In particular, Cadence HDL simulator contains a utility that generates an output file with information that can guide users in understanding and possibly optimizing the environment, also improving simulation performance. The profiler internally keeps track of the number of what are called "hits" in the running activities: this is a quantity that approximates the amount of CPU time spent in each of them. In particular, Cadence also offers the possibility of running a profiler specifically designed to be used with UVM environments. This is the Incisive Advanced Profiler, that has been herein used, as in other steps of the design, each time it was considered useful.

Some simulation time results have been reported in Table 3.1 , highlighting the number of BX cycle being run, the total simulation time needed, the CPU and memory usage. It has to be said that these results were obtained already after some optimization of the code done thanks to the profiler analysis.

The analysis performed with the profiler has shown that the use of resources was mainly due to randomization of the bi-dimensional array made of around 256K pixels. If still sustainable when simulating small models of the chip, such performance is not acceptable when running only a subset of the environment, i.e. the stimuli generator. For this reason it was planned to change the approach used for hit generation to a higher level one, that would have not required to randomize every single element of the matrix. This high level strategy will be presented in chapter 6.

3.1.1 SystemVerilog arrays

When it comes to optimizing performance of the environment for scalability, an essential aspect that needs to be taken into account is the how data are represented in the framework. To this end, SystemVerilog adds several enhancements to Verilog for representing large amounts of data [25]. The Verilog array construct has been extended and other ways of representing collections of variables have been introduced. Classic verilog arrays hold a fixed number of equally-sized data elements. Arrays can be classified as fixed-sized arrays (or static arrays) whose size cannot change once their declaration is done, or dynamic arrays, which can be re-sized. In addition to them, other “special” dynamic arrays like queues and associative arrays have been introduced in SystemVerilog in order to achieve an optimized managing of storage resources in different situations. It is important to underline that all dynamic arrays are not synthesizable, and are mainly intended to be used for verification or for a high level modelling of the design for initial studies, that will then need to be described through synthesizable blocks when the architecture of the design will be defined. Among the dynamic arrays, in this work the queue and associative array constructs have been used. The first one enables the simulator to automatically perform a run-time managing of space allocation, while for dynamic arrays an explicit variation of the array dimension needs to be performed. This way of storing data is particularly useful when the dimension of the array continuously grows and shrinks during the simulation. Queues can be used to model a last in, first out buffer or first in, first out buffer and they support insertion and deletion of elements from random locations using an index. Queue elements are by the way stored in contiguous locations, so it is efficient to push and pop elements from the front and back, like it is required for a First In First Out (FIFO) buffer. Adding and deleting elements in the middle of the queue requires shifting the existing ones to make room. In this case the time need grows linearly with the size of the queue [26].

As concerns the associative arrays, they are meant to be used while entries are stored in a very large but sparse matrix, were most of the elements do not actually need to be written or accessed at a single time. In this case allocating, initializing and possibly randomizing all the elements of such an array for every clock cycle of the simulation would just waste resources, as has already been shown (static arrays where being used when generating the simulation results presented in 3.1). For this reason, as it will be presented later, such a data structure has been used in this work for the generation and randomization process.

3.2 Study of commercial simulation tools performance

Since the framework that will be developed is intended to be used to verify and simulate very wide matrices of pixel chips, potentially described also at gate level, the feasibility of a detailed simulation and its performance are clearly between the main issues that need to be taken into account from the very beginning.
To this end, a study has been carried out using an available designed pixel chip provided by ATLAS in the context of the collaboration. The pixel chip is FEI4[19], already mentioned as one of the most relevant examples of existent chips for the future upgrades. In particular, the version B of it has been simulated. The use of such a complete design was necessary in order to obtain meaningful results on performance. The entire architecture of the DUT is reported in Figure 3.2, where also the full chip size of 336x80 pixels organized in 336 rows and 40 double columns can be observed.

The verilog description of the different modules of such a DUT has been provided at two abstraction levels: Gate level and RTL. Unfortunately those where not available in a double version for all of them, not enabling the comparison to be completely effective. So what has been actually simulated is:

- a mainly RTL description: the approach used has been to first compile the gate level description and then override most of the modules with a behavioural description;
- a mainly GL description: the only exception made for the Phase Locked Loop and Double Digital Column modules.

An already fully developed and very DUT-specific Verification Environment based on OVM has also been made available for those initial studies. Between the different simulators that a reasonable option for such applications, it has been possible to use have *Cadence Incisive* and *Modelsim/Questa* to obtain more significant results.

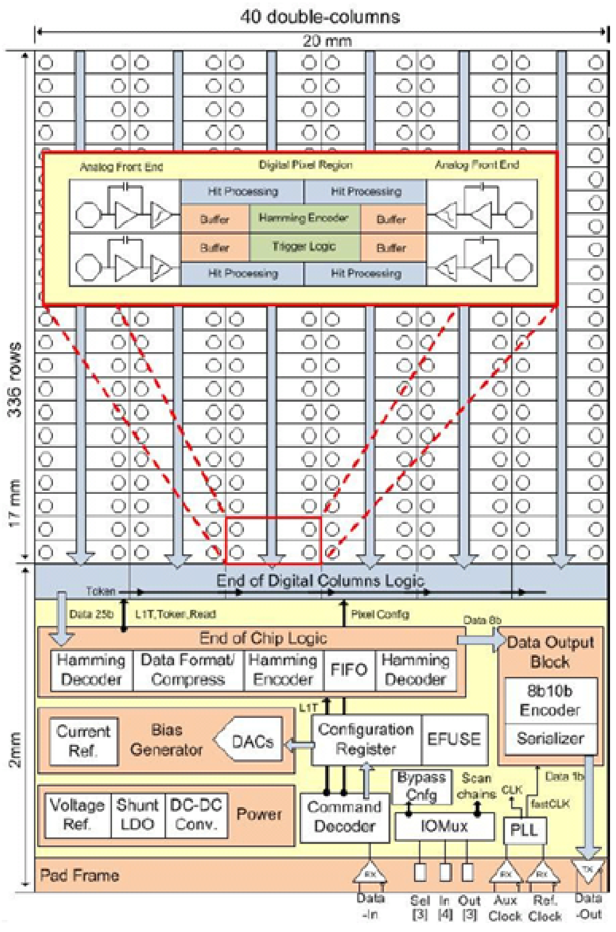


FIGURE 3.2: FE-I4 full chip diagram [5].

3.2.1 Design flow

The design flow offered by the two vendors are clearly similar, even if not identical. The main steps that required for the simulation can be considered anyway three: compilation, elaboration/generation of the design hierarchy and actual simulation.

The "setup" of the tools that has been done in this particular context is herein highlighted.

The design flow used for simulations with Incisive Cadence is the following:

1. initial compilation of project libraries;
2. Multi-Snapshot Incremental Elaboration (MSIE) has been performed. With Cadence software, elaboration is the step which creates the simulatable model of the complete design and testbench environment called Snapshot. Such a step can

turn out to be heavy and lenghty, as the complexity and size of the SoCs increases. It is particularly undesired when performing minor changes in the snapshot requires re-elaboration of complete DUT and environment. Using MSIE it is useful not only at the design level (to separate the part of the design that is stable from the one that is intended to change often), but also for verification engineers, since it allows them to avoid to re-elaborate the whole complex DUT model each time the testbench is modified. What is performed in this work is:

- a primary snapshot is generated considering only the top-level DUT;
- a secondary snapshot with the whole class-based Verification Environment. It is then combined by the tool with the primary snapshot(s) for execution: in this way rebuilding the whole environment is extremely fast when change happens to incremental portion of the code;

3. actual simulation.

The difference between the flow used for the first run (when also the first snapshot needs to be generated) and the following ones is shown in Figure respectively in 3.3 (a) and Figure 3.3 (b).

Another aspect that is characteristic of the simulator used concerns the optimization, performed automatically by the tool if it is not differently indicated through command options. In this work no optimization made by the tool has been turned off.
In the case of Questa/ModelSim, the user can intervene more the optimization process using dedicated arguments. Optimization options have been specified through simulation commands. As regards the simulations performed with the tool of this vendor, it has been chosen an optimized two-step Flow (due to pre-existent scripts that have been made available), whose main steps are shown in Figure 3.4 and herein summarized:

1. Initial compilation of project libraries (modules can keep gate level description or can be overwritten using RTL);
2. Simulation phase, that can be in turn divided into two steps:
   - the loading of compiled design units from their libraries and generation of optimized code (a new top level is generated);

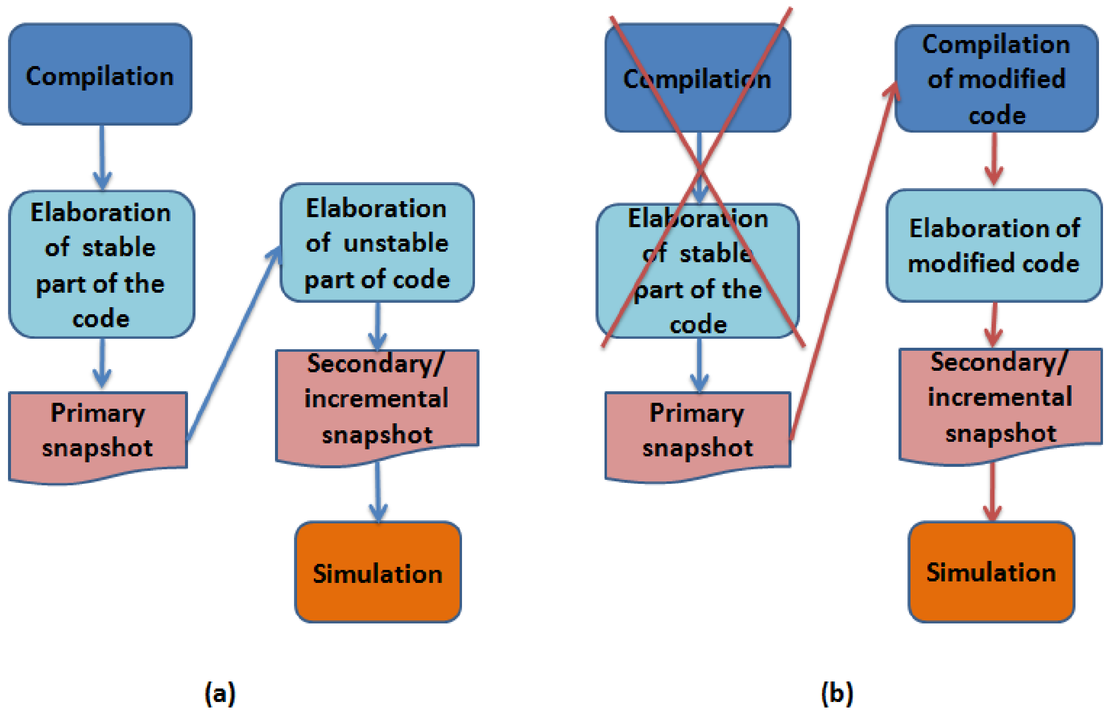
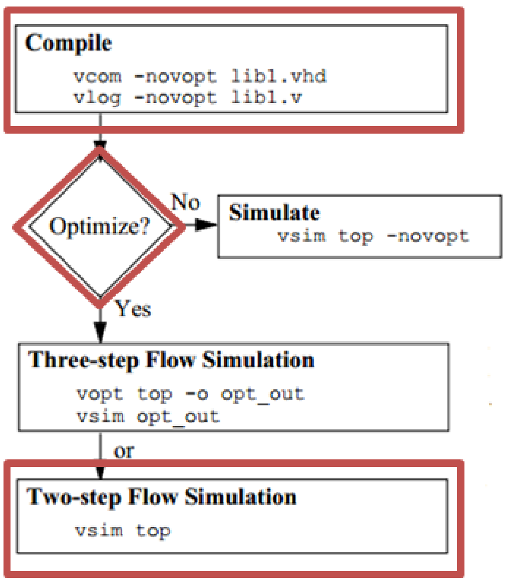

Compilation
Elaboration of stable part of the code
Primary snapshot
Elaboration of unstable part of code
Secondary/ incremental snapshot
Simulation
(a)
Compilation
Elaboration of stable part of the code
Primary snapshot
Compilation of modified code
Elaboration of modified code
Secondary/ incremental snapshot
Simulation
(b)

FIGURE 3.3: Multi Snapshot Incremental Elaboration flow: comparison between the first run (a) and the following ones (b).



```
vcom -novopt lib1.vhd
vlog -novopt lib1.v
```

```
vsim top -novopt
```

```
vopt top -o opt_out
vsim opt_out
```

```
vsim top
```

FIGURE 3.4: Flow diagram of the optimized two-step modelsim flow.

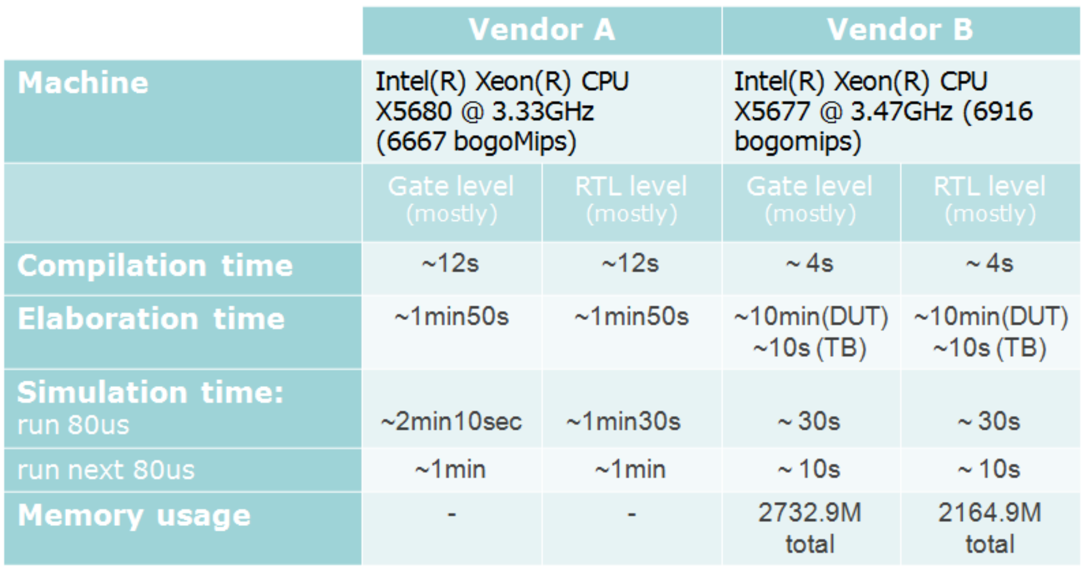
- actual simulation of the optimized code.

Other settings that have been done for the simulation and that need to be highlighted before presenting the results of the study are the following:

- simulations have been run on different machines, but with close performance (on the processor and the *BogoMips* indicator details will be provided);
- simulation have not be run granting access to all the signals (/nets/ports/cells..), but only to a subset of them that had been considered of interest by the designers of the chip;
- no Standard Delay Format (SDF) annotation of the timing delays has been performed. Such a file normally includes path delays, timing constraint values, interconnect delays and high level technology parameters;
- simulations have been performed in a single-threaded fashion, without running them in multiple cores since such an option was not available for both the vendors.

The results of the test on performance have been quite comforting. Both the tools used have demonstrated to be capable of dealing with a full pixel-chip (of 336x80 pixels) without hitting insurmountable compilation or simulation time problems. This has been both possible with the (partially) gate level and RTL descriptions of the chip. The use of memory observed while running them has also been encouraging. More detailed results are reported in Table 3.2, clearly without giving details on which vendor assures certain performances. This is also motivated by the fact that simulation time can be potentially affected a lot by the settings done by the user, so even if (as presented) in this context it has been tried to keep the comparison "fair enough", it is not excluded that more experienced designers could use different settings of the parameters to obtain different and possibly better performance for one vendor or for the other. To explain slightly more in detail what is presented in Table 3.2, it can be noticed that the machine used for testing both the simulation tools coming from different vendors is reported, together with the compilation, elaboration and simulation time observed for both the Gate level and Register Transfer Level (RTL) description of the DUT. Simulations have been run at steps of 80 μs. Finally, when it has been possible to obtain them, also indications on memory usage have been reported.

Table 3.2: Results of simulation performance tests.



	Vendor A		Vendor B	
Machine	Intel(R) Xeon(R) CPU X5680 @ 3.33GHz (6667 bogoMips)		Intel(R) Xeon(R) CPU X5677 @ 3.47GHz (6916 bogomips)	
	Gate level (mostly)	RTL level (mostly)	Gate level (mostly)	RTL level (mostly)
Compilation time	~12s	~12s	~ 4s	~ 4s
Elaboration time	~1min50s	~1min50s	~10min(DUT) ~10s (TB)	~10min(DUT) ~10s (TB)
Simulation time: run 80us	~2min10sec	~1min30s	~ 30s	~ 30s
run next 80us	~1min	~1min	~ 10s	~ 10s
Memory usage	-	-	2732.9M total	2164.9M total

As a conclusion of this preliminary work, it can be said that it has encouraged to keep using the tools that had been chosen without necessarily considering different (and maybe more high-level) ones. It is anyway important to highlight that the chip simulated was smaller that the one that is planned to build and that the environment used was anyway DUT-specific and may not had some advanced features and analysis capabilities that one may want to obtain for a more generic framework. To this end, it is good practice every time a decision on how to proceed is made, to always consider its impact on simulation time and memory usage.

Chapter 4

Design under test: description of the system

Before presenting the work that has been done in the verification environment, it is important to introduce the architecture that is supposed to be simulated at the system level. It should be clear that, as it often happens in the industry, the design of the chip and of the verification environment are being carried out in parallel in the context of RD53 collaboration. Moreover, the simulation framework itself is supposed to be used as a design tool to optimize the architecture of the future pixel chip, and to then verify it up to the final versions of the design. A common pixel chip architecture, that is fully digital after the basic threshold detection and charge digitization in the analogue pixel cell, has been defined: the block diagram of its hierarchical organization is shown in Figure 4.1. It can be noticed how pixels are grouped in regions, how columns are made of a replication of regions and how several columns form a full matrix.

Digital hit processing including the critical trigger latency buffer is implemented within the pixel array in local pixel regions followed by data merging, data formatting and readout after the first level trigger accept. As it can be seen, the hit information is supposed to be stored locally at the different stages of the data readout. It is of primary importance to optimize such a storage through the use of shared buffers in order to achieve compact circuitry and low power. Indeed, storing information from multiple hits from the same cluster together translates into significant savings in required storage

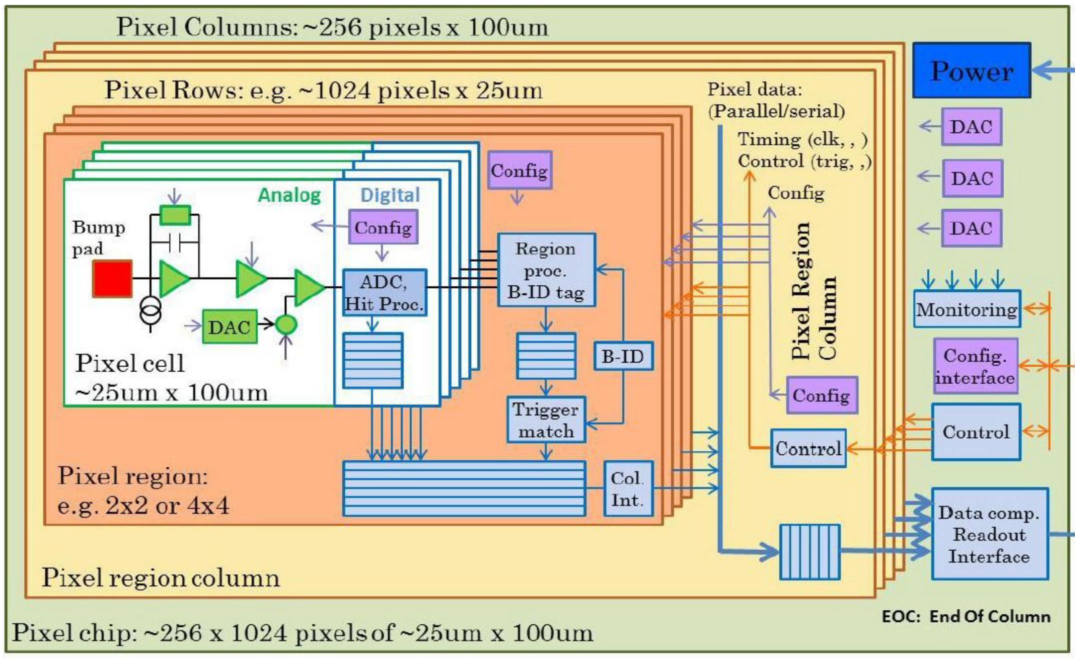


FIGURE 4.1: Pixel chip hierarchical organization [6].

resources [6]. On this topic also preliminary statistical studies have been done and the most convenient Pixel Region (PR) shapes and sizes to adopt in the design of next generation pixel chips have been considered without taking into account implementation related issues (e.g. routing, occupation of area, power consumption). This study [33] has been carried out with simple assumptions on the shapes of clustered hits, as it will be described in Chapter 7. Such statistical analysis needed to be cross-checked with simulation results, performed starting from initial architectures of the pixel chip that could be afterwards optimized. Some preliminary results obtained using VEPIX53 will be also presented in 7. Also considerations on the layout of the analog part would anyway play a separate but fundamental role in the choice [6]. It has to be highlighted that my personal activity has not been focused on the design of the hierarchical pixel chip architecture. My main contribution has been on the definition of a behavioural model for a single PUC. For this reason, a section will be dedicated to this part, while the complete architectures being simulated will be described at a higher level in the subsequent sections.

4.1 Development of a model for the Pixel Unit Cell

Since the purpose of the framework is to be capable of simulating different kind of architectures, it has also been chosen to start with an interface from the framework to the chip as generic as possible, so no assumptions have been done on how the analog-to-digital conversion is performed in each PUC. As it has been mentioned in subsection 1.1.1 on the Front-End electronics, possible options are ADCs (independent or shared) or a TOT converter. For this reason the input of the single PUC has been considered as an “analog” amplitude, being in particular represented as a SystemVerilog *real* data type, i.e. a double precision floating-point type (correspondent to C *double*). This choice, even if not really of fundamental importance at the early stage of the design, has been taken because the use of such a data type is advised for modeling analog signals in SystemVerilog when Mixed-Signal simulations of the complete architecture are performed. Real Number Modeling (RNM) turns out indeed to be a new “trend” when it comes to modeling analog signals (i.e. current, voltage), with an acceptable level of accuracy, in a digital domain, where analog modules are simulated through a behavioural description, in order to allow designers to perform system simulations making only use of the digital solver: for big and complex designs this is claimed to assure the best trade off between simulation speed and accuracy [34].

A simple block diagram of the PUC that has been developed can be seen in Figure 4.2. As previously stated, it can be noticed how the main input to such a cell is the “analog” value representing the amplitude of an hit.

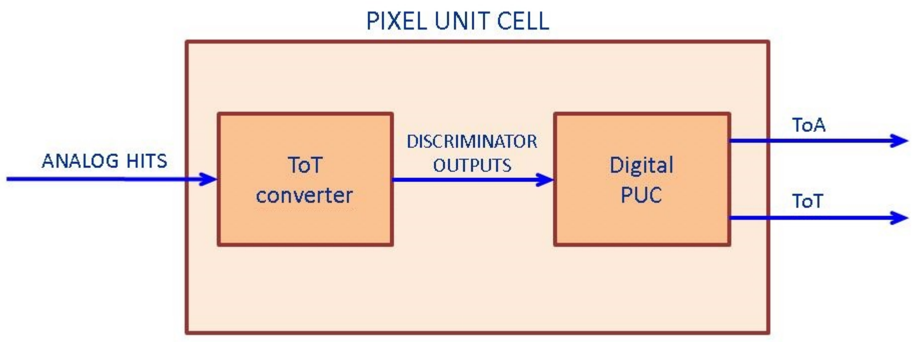


FIGURE 4.2: Block diagram of the pixel unit cell.

As concerns its architecture, a TOT-based one has been taken into account. In particular, the single PUC is composed of:

- a simple TOT converter module which abstracts the behavior of the analog front-end;
- the actual digital part of the cell, that outputs information on the Time Of Arrival (TOA) of the hit (corresponding to the bunch crossing cycle when it has been detected) and on its amplitude, represented by the Time Over Threshold (TOT).

As regards the TOT converter, it is a very simple model that is meant to produce a binary output (*discr_out*) whose value is high for a number of clock cycles equal to the "amplitude" of the analog hit. A timing diagram of this module is reported in Figure 4.3, in correspondence of an incoming hit. A limitation that is intrinsically part of such an "analog" front-end is the dead time due to the TOT accumulation. Indeed, while the discriminator output is high the pixel unit is practically blind if new particle arrive. Depending on the actual implementation of the analog section, their charge could be just neglected or accumulated to the previous one causing a higher energy hit to be observed. It can be noticed in the timing diagram the presence of two internal flag signals that are

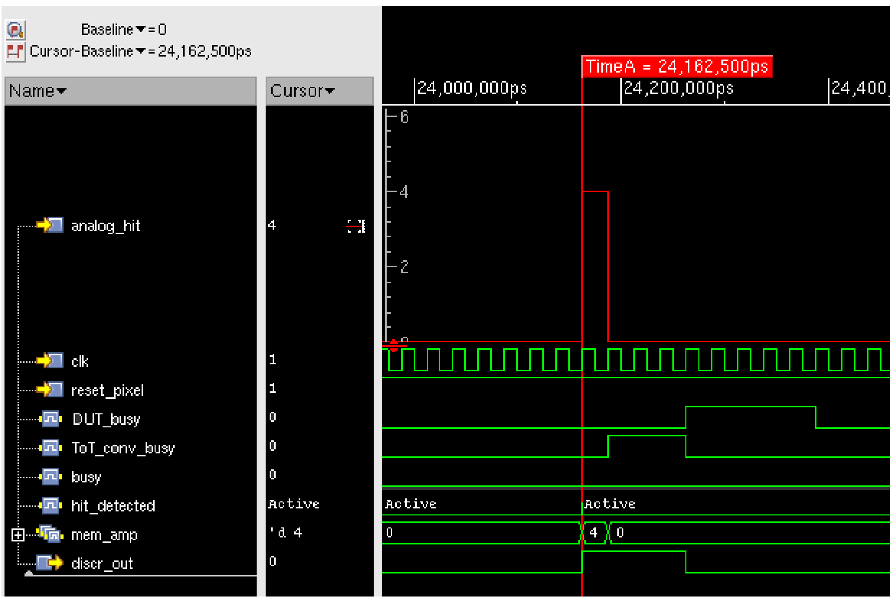


FIGURE 4.3: Timing diagram of the TOT converter.

also probed to inform the Verification Environment about the deadtime of the PUC. It can be in particular seen how the number of clock cycles that follow the detection of the hit (but are still needed to measure the TOT) are marked by the *TOT_conv_busy* signal high. At the same time, it has been chosen to avoid starting to measure a new incoming hit TOT during the fixed time during which the PUC is busy in digitally processing the previous one. Such a time is identified as *DUT_busy*. Even if it would be interesting to obtain more sophisticated simulations in the future, in this version no precise conversion from a "charge" deposited in a sensor to the TOT measured in the different PUCs is done, and simply the TOT value is being passed as the amplitude. When needed, this module could be modified in order to obtain a more sophisticated model of the analog block. A hardware limitation that has instead already partially modeled from a "digital point of view" has been the already mentioned timewalk. Instead of actually modeling a slow response of a discriminator module, what has been done is the following: for hits whose amplitude is lower than a certain parameterized threshold, this causes the binary discriminator output to be raised at the subsequent (and so wrong) bunch crossing cycle. Anyway, since the study of the limitations of the front-end is not the primary focus at the early stages of the design, for the initial simulations such a threshold has been fixed to zero.

4.1.1 Digital PUC: use of SystemVerilog constructs

The architecture of this basic module will be presented together with timing diagrams obtained with Incisive Cadence, with the aim to focus on the SystemVerilog features that have been used and that are meant to improve the digital design with respect to Verilog approach.

The block diagram of this module is displayed in Figure 4.4. Before going further in the description, it is pointed out that for the actual digital part all the inputs and outputs have been defined with the SystemVerilog 4-state data type *logic*. The Verilog language uses the *reg* type as a general purpose variable for modelling hardware behaviour. It is known to be a misnomer since it reminds the association to a "register", while there is no correlation between this variable and the kind of hardware inferred (combinational or sequential logic). SystemVerilog uses the more intuitive *logic* keyword to indicates indicating that the signal can have 4-state values (and can be used with both *wire* and *var* types, even if it is a var if not differently specified). A *logic* signal can be used anywhere a *wire* is used, except that a *logic* variable cannot be driven by multiple structural drivers (this is done in order to prevent design errors). Rather than trying to choose between *reg* (variable) and *wire* it is a recommended approach [24] to declare all signals as logic at a first time, and check if compilation errors are produced for the presence of multiple drivers: if it was expected, one can change it into *wire* only where it is actually necessary. This is the reason why the *logic* variable has been used mainly for the design.

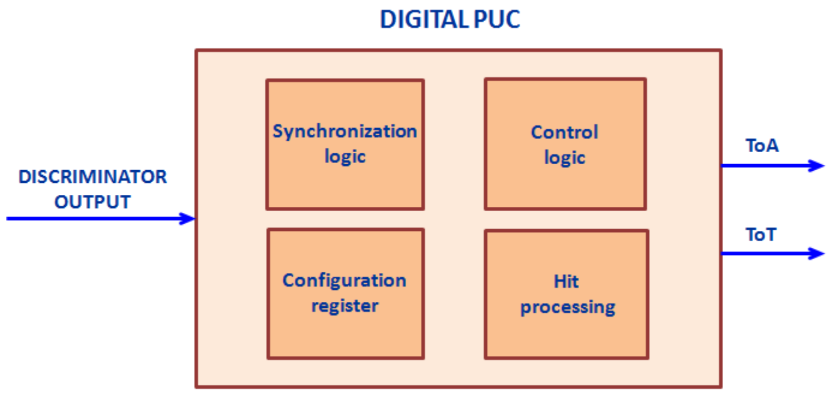


FIGURE 4.4: Block diagram of the digital part of a PUC.

The digital PUC contains the following sub-modules:

- Synchronization logic, which takes care of the synchronization with the 40 MHz input clock;
- Pixel configuration register, which is a rudimental description of the configuration block inside the pixel;
- Pixel core, which gathers both the control logic and the hit processing logic. It is in turn composed of some sub-modules:
  - a 16-bit TOA register, which synchronously loads the value taken as a parallel input from the external reference counter (resolution: 25 ns) and outputs it in a parallel fashion;
  - a 16-bit counter with parallel output, the function of which is to measure the Time Over Threshold;
  - a Finite State Machine (FSM) which represents the control logic of the pixel.

The synchronization logic is basically made of a D-flip flop that transforms the synchronous or asynchronous signal coming out from the discriminator in the synchronous output, meant to be sent to pixel core. This is clearly visible from Figure

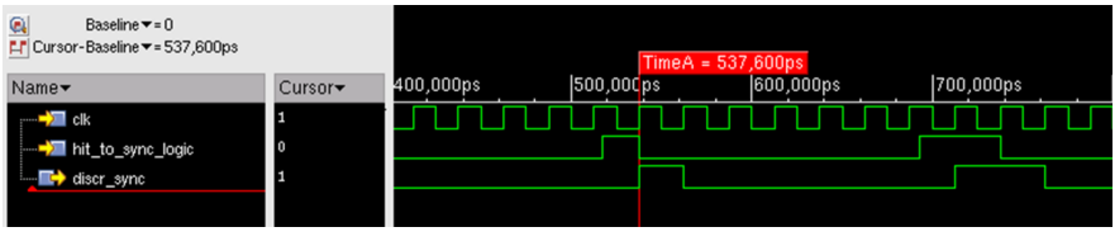


FIGURE 4.5: Timing diagram of the synchronization logic module.

4.5. As concerns the pixel configuration register, since it is not a priority for architectural studies, just a support for the pixel mask bit has been included as a basic functionality. In the timing diagram present in 4.6, two subsequent loadings of the mask register are shown: the value passed from *mask_in* (respectively 1 and 0) is copied in the register (see *mask_out*) at the clock rising edge. When the value of the mask bit is 1 every hit coming from the discriminator output is not detected by the synchronization logic, as shown on the last two signals in grey. As regards the

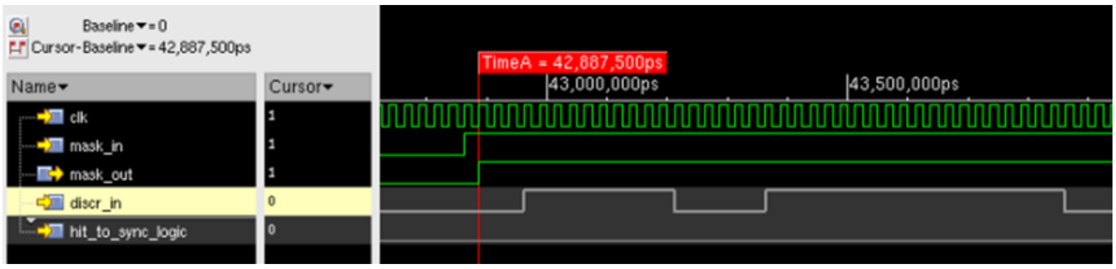


FIGURE 4.6: Timing diagram of the configuration register.

sub-modules of the pixel core, it has been mentioned the presence of a register for the TOA and a counter for measuring the TOT. The timing diagram of the former is shown in Figure 4.7, while the latter is reported in Figure 4.8. The behaviour of the

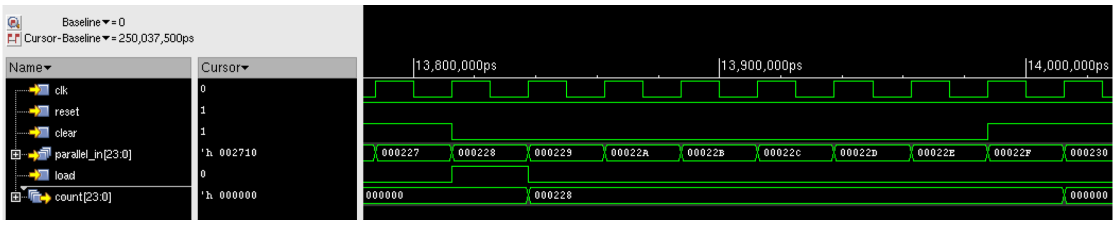


FIGURE 4.7: Timing diagram of the TOA register.

TOA register that can be noticed from the simulation is the following: when the load input is active (in correspondence of an incoming hit) the value of the parallel input (provided from an external counter present in the End Of Column) is loaded into the register, providing an absolute TOA measurement, that is initially stored and then sent out from the PUC.

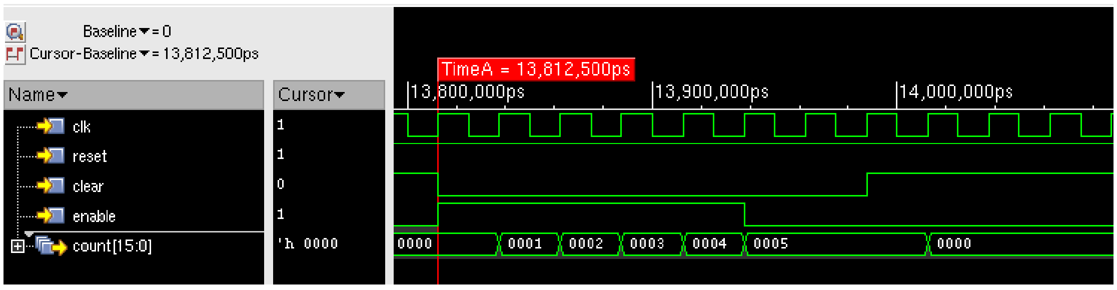


FIGURE 4.8: Timing diagram of the TOT counter.

From the timing diagram concerning the TOT counter, it is evident how it starts increasing the *count* value progressively when the *enable* input is asserted, i.e. for the time during which a hit is seen as an input to the PUC. This module can also be synchronously reset and cleared acting respectively on the *reset* (active low) and *clear* input (active high). Even if their effect on the counter is the same, these two signals have been kept separated in order to distinguish the clear needed enable a new counting to start from the general reset of the whole PUC.

As far as the modelling of FSMs is concerned, the consistency of how different software tools interpret the Verilog models can be increased thanks to SystemVerilog. The main SystemVerilog features used to achieve this goal are:

- using enumerated types for modelling FSMs;
- using enumerated types with FSM *case* statements;
- using *always_comb* with FSM *case* statements.

Normally what is used to code a FSM is a three procedural block modelling style: one is used to increment the state machine to the next state, one to determine the next state and the last to set the output values. By using enumerated types, one can assure (and check) that the possible values assumed by the state are only the possible ones listed. Additional use of the unique/priority case statements instructs the tool on how to interpret a case statement. Verilog *case/casex/casez* statements allow designers to select one branch of logic among multiple choices. For example they can be used inside an *always_comb* block that is intended to behave in a different way depending on the current state of a FSM. In Verilog standard it is defined that the case selection has to be evaluated in the order in which they are listed. This normally implies that the compiler has to optimize with some additional logic in order to reach the priority encoding.

SystemVerilog introduces special *unique* and *priority* modifiers to be put before the case statements in order to directly specify whether a priority is needed or not in the selection. A *unique case* asserts that there are no overlapping *case* items and hence that it is safe for them to be evaluated in parallel, while a *priority* one acts only on first match, implying a selection with priority. In the combinatorial processes of the FSM of the project (for next state and outputs update), *unique case* statements have been used since no priority scheme was required. The defined FSM is quite simple, it has just three possible states that are summarily described in the following:

- IDLE (00), i.e. the initial state, where both the TOA register and TOT counter are cleared;
- COUNTING (01), to which the FSM passes when the discriminator output signal goes high (the rising edge of this signal is detected by an internal edge generator that rises a *r_edge* pulse). In this state the TOA register and TOT counter are enabled to perfom their respective measurments;
- READY(10), final state which signals the conclusion of the operations and allows the FSM to go back to the IDLE state when *ack* is asserted from the higher level logic. During such a state the content of the single PUC gets written into the indipendent PUC or PR buffer (depending on the architecture).

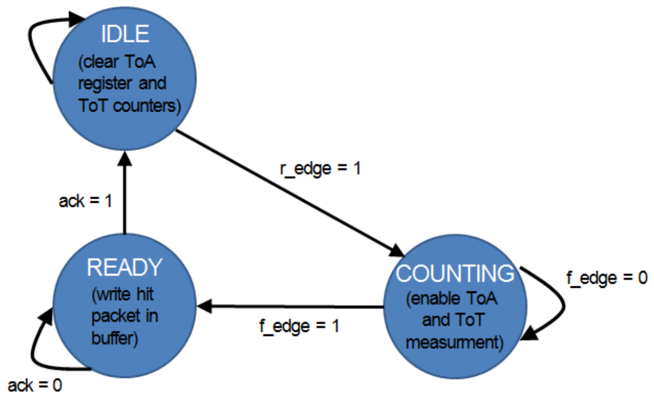

IDLE
(clear ToA register and ToT counters)
r_edge = 1
ack = 1
READY
(write hit packet in buffer)
ack = 0
f_edge = 1
COUNTING
(enable ToA and ToT measurment)
f_edge = 0

FIGURE 4.9: Timing diagram of the TOT counter.

A simplified state diagram and a timing diagram of the state evolution when a hit is received are also reported respectively in Figure 4.9 and 4.10.

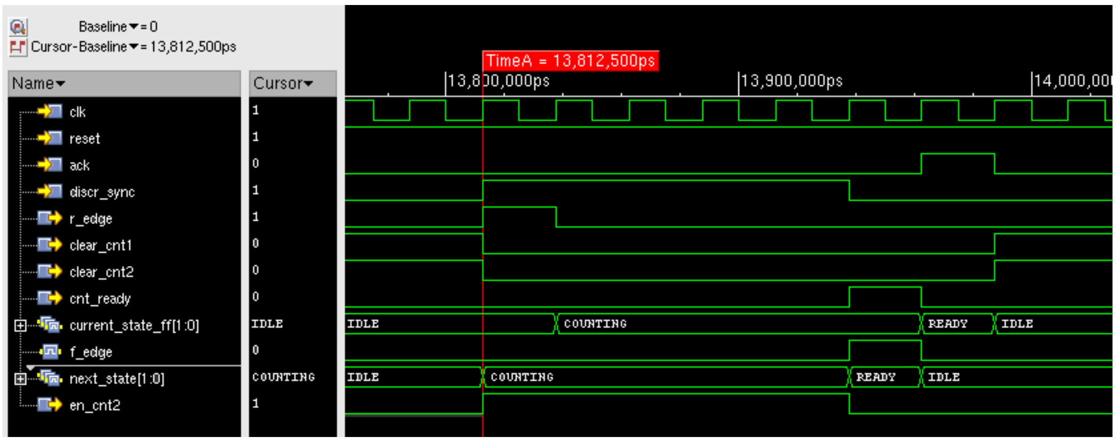


FIGURE 4.10: Timing diagram of the TOT counter.

The actual description of the presented modules has not be done at the gate level, but at a more concise (but synthesizable) level. When dealing with sequential and combinational logic in the description of the design, the specific SystemVerilog *always* processes have been used. Indeed, while in pure Verilog it was a common mistake to use an always block to model combinational logic and to forget an else (leading to an unintended latch), SystemVerilog has added specialized *always_comb*, *always_latch* and *always_ff* blocks, which indicate design intent to simulation, synthesis and formal verification tools. *Always_comb* is for the specification of combinational logic. Synthesis tools can check that the hardware synthesized from the block does not contain any sequential elements. Additionally, the sensitivity list for an *always_comb* block is calculated by tools rather than by the user, ensuring that simulation will behave closely with the synthesized result. *Always_latch* is another form of always block designed to express latch logic (it also determines its sensitivity list automatically). The *always_ff* block was added to ensure that the hardware being described contains a flip-flop. These blocks also limit the block to a single activation expression, so that an implicit state machine will not be inferred. Additional explanations on this topic can be found in [25].

When dealing with design hierarchy issues, SystemVerilog offers some shortcuts to make the code lighter and more readable. A simple (but useful) feature is the possibility of specifying an ending name for modules and any kind of block (tasks, functions, loops. . .): this is a good coding style that makes it easier to identify and separate the different parts. When connecting ports of different modules that are instantiated inside another, one can normally do it either connecting them by name or by order: the first way is clearer but quite verbose, while the second one can easily lead to some errors. A nice (and fast) solution is possible with SystemVerilog: the dot-star " .* " connection can be used together with a connection by name: that means that all the ports that have the same name get automatically connected without listing them in the module instantiation, while the ones that have different names in different modules can be specified by name to avoid mistakes. Even if dealing with small and simple modules, this approach has been extensively used in the design hierarchy of the project.

The simulation output of such a PUC is reported in Figure 4.11. The two different parts, i.e. the TOT converter and the digital PUC can be distinguished since the correspondent signal names are shown in different colors, respectively green and orange. It is therein shown that when a hit whose amplitude (*analog_hit*) is 8 is received, the detection takes place and the discriminator output (*discr_out*) is kept high for an equal number of bunch crossing cycles. In the digital part instead, at the rising edge of the clock after the *discr_out* signal has been asserted, the value of the external counter (*ext_count_in*) is copied in the TOA register and immediately available at the output though the *parallel_out*, while the TOT counter starts counting and stops only when the discriminator ouput goes low again. At that point the counter informs the higher level logic that it is ready (*cnt_ready*) and only after the arrival of the *read_pixel* signal both the TOA register and the TOT counter are reset. The time when a new hit can be received is highlighted by the marker.

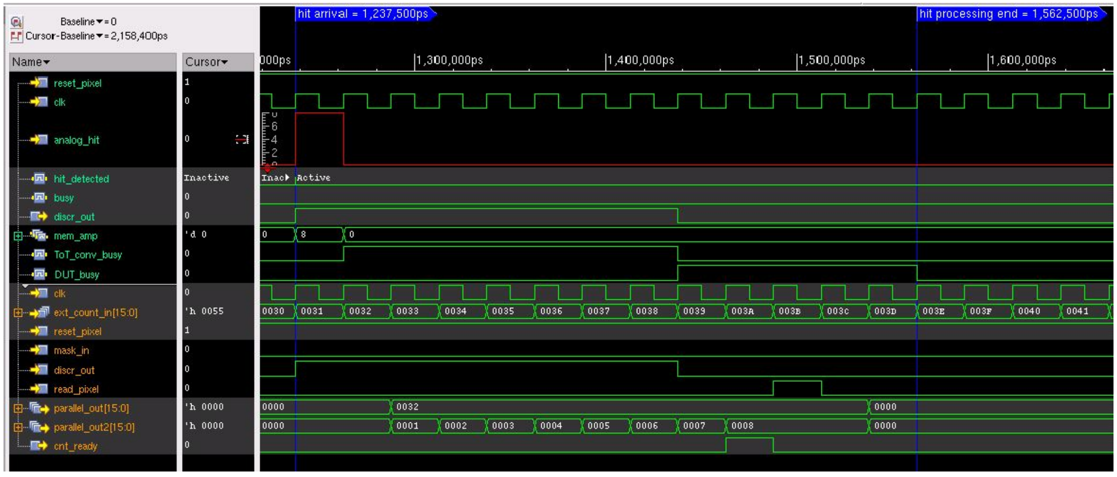


FIGURE 4.11: Timing diagram of a single PUC (both TOT converter and digital front-end).

4.2 System architecture

With the aim of optimizing the pixel chip architecture, some preliminary models have been designed in the context of the collaboration. It is once more reminded that such a design has not be directly subject of my work, but it is herein reported as a description of the DUT used in the simulation and verification environment. Initially, the focus has mainly been put on obtaining an efficient use of the buffering resources that will be likely located in the pixel region for the 3rd generation. At the time of writing, two different architectures have been taken into account in order to investigate how one can better profit from the use of logic shared within the pixel region. For this purpose the first models of the DUT that have been taken into account do not contain a whole pixel chip matrix but just a group of pixel unit cells (PUCs), i.e. a pixel region (PR), of parameterized size. The general structure of the simulated pixel chip is shown in 4.12. It can be noticed as just one PR is so far instantiated, but also that column arbitration logic is already present at the end of column.

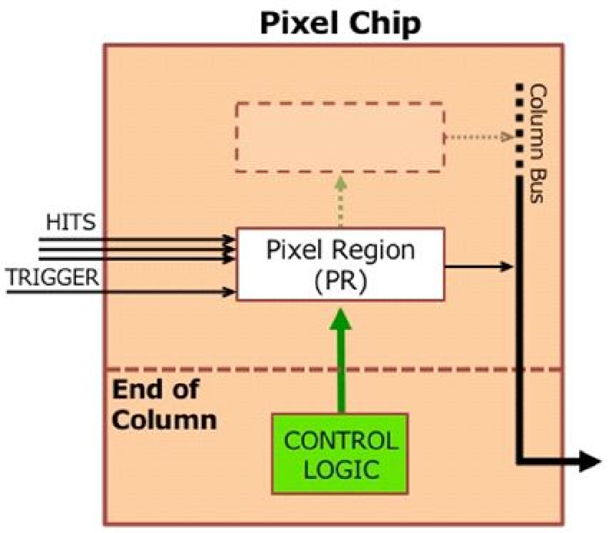


FIGURE 4.12: Block diagram of the pixel chip architecture.

In the current status of the work a behavioural and non directly synthesizable description of the DUT has been used, since at this stage of the design is more reasonable to aim to use models that are described in a simple (and generic) fashion that can be easily studied, understood and simulated in less time instead of directly start from complex and elaborated architectures. For all the architectures the starting point for the model of a single PUC has been the one described in the previous section, even if some modifications have been done to adopt it for the second architecture.

4.2.1 Independent pixels architecture

The very first architecture taken into account has been one where the different PUCs can be basically considered as independent. In order to use it for comparison with more complex schemes, this architecture has been developed as the simplest possible. Its block diagram can be observed in Figure 4.13.

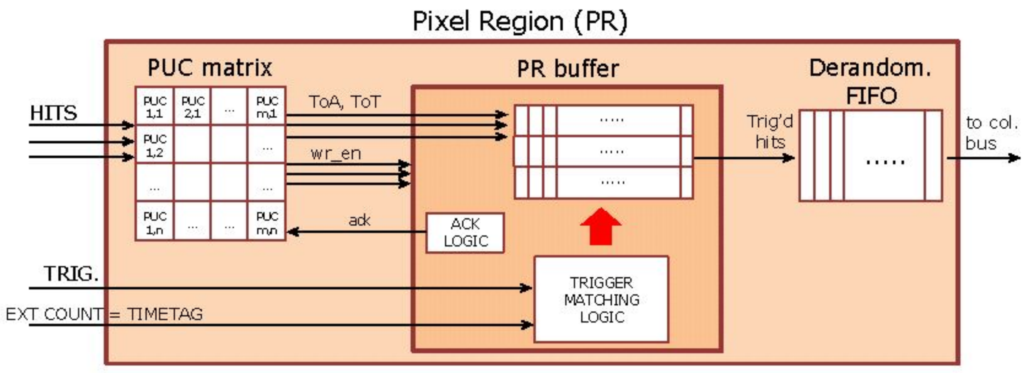


FIGURE 4.13: Block diagram of a PR containing indipedent PUC buffers.

The main functions defined for the pixel region can be summarized in the following:

- conversion of the amplitude value of a hit into discriminator output for each PUC in the matrix;
- computation of hit time of arrival TOA and amplitude TOT separate for each PUC in the matrix;
- storage of incoming hits in the so-called pixel region buffer;
- trigger selection of hits depending on the input trigger signal (performed by the trigger matching logic);
- a derandomizing FIFO to send data from the PR to the column bus.

From the block diagram in Figure 4.13 also the main input and outputs of such a model can be noticed. As regards the inputs, an “analog” hit signal is sent to each

PUC in the PR, a parallel signal coming from an external counter is used to perform the TOA measurement and a yes-no trigger signal is provided to the trigger matching logic. As concerns the outputs, the only physical one is represented by the output of the derandomizing FIFO, that gathers the hit information for each PUC, as it will be more precisely reported. It can be highlighted that, although the code used to describe the pixel chip uses some non-synthesizable constructs, the description has not been completely decoupled from implementation, since the logical function of each block is defined. The main non-synthesizable structure that is being used is the so-called SystemVerilog queue, used to implement both the buffer and the derandomizing FIFO. As already mentioned in the previous subsection 3.1.1, when the dimension of the array is expected to grow and shrink during the simulation it is advised use of such a structure for handling memory allocation. In this application buffer occupancy is a parameter that needs to be investigated in order to understand the number of buffering location required to keep the overflow probability of such a buffer under a certain value, since such an overflow translates in hit loss. The maximum width of a queue can potentially be unspecified, but in this case an upper bound has been fixed through a dedicated and settable parameter. When the interest is to study (potentially) all the possible values for buffer depths, the approach is simply to use a very high value that is never reached during the simulation. Some details are provided regarding the organization of the pixel region buffer. Even if represented as a unique block, it is in reality made of a buffer for each PUC and they work completely independently one from the other. To be more precise, each PUC has a dedicated FSM that controls the writing into the PUC buffer and also the output of the single PUCs can be all read in parallel at the same time. The structure of the packets that are written on such a buffer is shown in Figure 4.14.

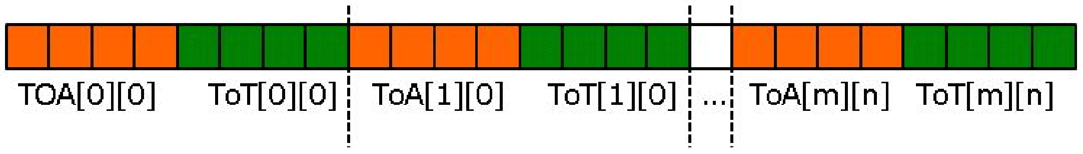


FIGURE 4.14: Structure of the hit packets contained in the PR buffer for the independent pixel architecture.

It can be noticed as both the TOA and the TOT of the hits are separately stored by each PUC in a totally independent fashion. As can be seen, the number of unit cells is parameterized in a bi-dimensional fashion, reproducing the physical matrix structure.

4.2.2 Zero-suppressed FIFO architecture

In order to study the advantages that a shared pixel region architecture brings in terms of saving storage resources, another model of the DUT has been developed in the context of the simulation working group of the RD53 collaboration. It can be said that from a functional point of view, such a module performs the same operations of the previous architecture. In order to be more precise, a block diagram of the pixel region architecture is reported in Figure 4.15 and the main functionalities of the PR are listed below, highlighting the differences with the previous model:

- conversion of the amplitude value of a hit into discriminator output for each PUC in the matrix;
- computation of hit amplitude TOT is separate for each PUC in the matrix, while for the TOA it is shared;
- the storage of incoming hits in a single and shared pixel region buffer and the access to the buffer is arbitrated by the "write logic" block;
- trigger selection of hits depending on the input trigger signal (performed by the trigger matching logic);
- a derandomizing FIFO to send data from the PR to the column logic.

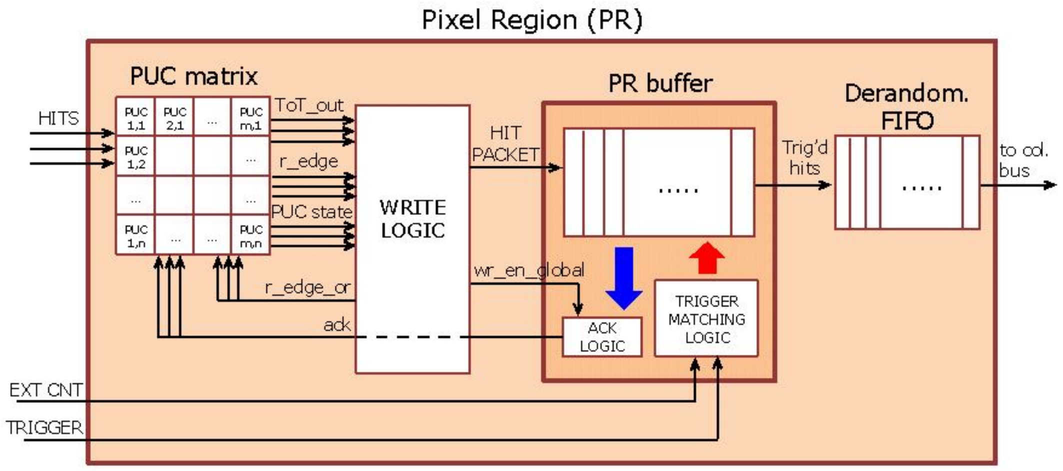


FIGURE 4.15: Structure of the hit packets contained in the PR buffer for the zero-suppressed FIFO architecture.

As for the previous model, the hardware description level is non-synthesizable mainly because of the use of SystemVerilog queues. In Figure 4.15 it can be noticed how the inputs and outputs of the block diagram are, as for the previous case, the hit signals, the trigger, the signal of an external counter and the data output. Clearly the latter is in this case organized in a different fashion with respect to the independent PUCs architecture: for a straightforward comparison in terms of the hit packet being stored in the buffer (and then sent out to the column logic) one can refer to Figure 4.16. It is

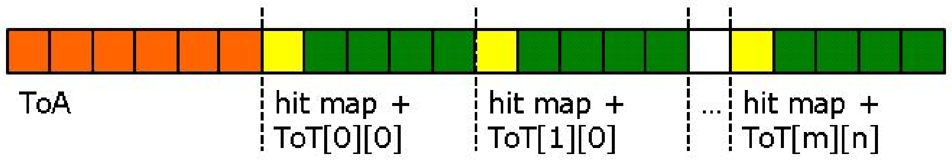


FIGURE 4.16: Structure of the hit packets contained in the PR buffer for the zero-suppressed FIFO architecture.

shown as a single location of the buffer contains a packet where the TOA measured value is shared between all the PUCs in the pixel region while just the TOT value is measured independently to maintain the spatial resolution of the measurement. Together with the TOT, in Figure 4.16 also a potential hit map bit is represented: it is, at least theoretically speaking, meant to be used to distinguish whether which PUC has been hit (hit map value equal to 1) and which not (hit map value equal to 0). This approach is anyway redundant, and it is likely to be substituted with the storage of just the TOT values, where a hit map value equal to zero is simply represented by a TOT value of zero. Further compression may be reached with more evolved schemes.

Chapter 5

UVM simulation and verification framework: VEPIX53

The first version of the simulation and verification framework for the RD53 collaboration has been developed [35]. With obvious reference to the context in which it is collocated, it has been named VEPIX53. Such an environment is a preliminary version of the final one. As stated in the RD53 Proposal itself [6], the goals of such a platform are:

- to be capable of simulating alternative pixel chip architectures;
- to enable designers to simulate and verify pixel chip architectures at increasingly refined level as design progresses;
- to perform automated verification functions;
- to be capable of dealing with different kinds of data sets:
  - realistic (and extreme) pixel hits and triggers (to be generated within the framework itself);
  - particle hits from external full detector/experiment Monte Carlo and/or detailed sensor simulations (to be imported in the framework);
  - mixed data (external and generated in the framework).

The final goal of the simulation working group is therefore to obtain a framework that provides all the features shown in the one in the block diagram in Figure 5.1. It can be therein noticed as some master timing should be capable of controlling the generation of the hits (e.g. random, tracks and Montecarlo data) and of the trigger signal. The generation of the signals needed to configure the pixel chip should also be provided, as long as a high level control/sequencer in order to decide how to inject such stimuli to the pixel chip. Also, it should be capable of reading the pixel chip outputs (*readout interface*), to compare them with the ones generated by the *reference model* and to finally produce information on errors/warnings together with performance indicators. As a last point, the possibility of performing directed tests should also still be an option in order to check possible expected bugs of the design, even when using such a complex and automated framework. As it has been already highlighted, the HDVL chosen for

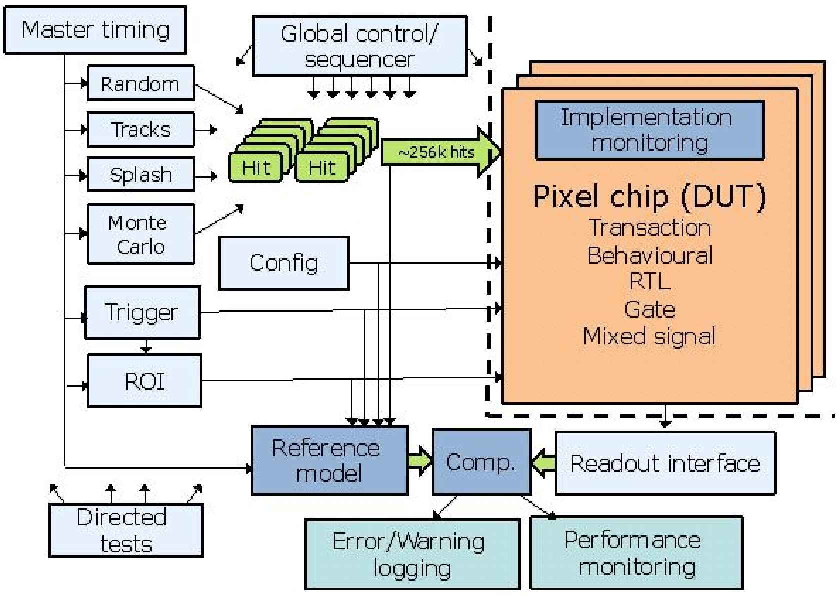


FIGURE 5.1: Block diagram of a dedicated simulation and verification framework for next generation pixel chips.

such a framework is SystemVerilog. It has been also made use of the UVM library for the sake of standardization and re-usability.

Not all the features have currently been developed, but a working "engine" to use for preliminary architecture studies and will be presented in this chapter.

In the remainder of this work it will be highlighted what has mainly been developed without my personal contribution.

5.1 Overall architecture of the simulation and verification framework

The block diagram of the overall simulation and verification framework (VEPIX53) is reported in 5.2. Two main parts can be distinguished in such an architecture:

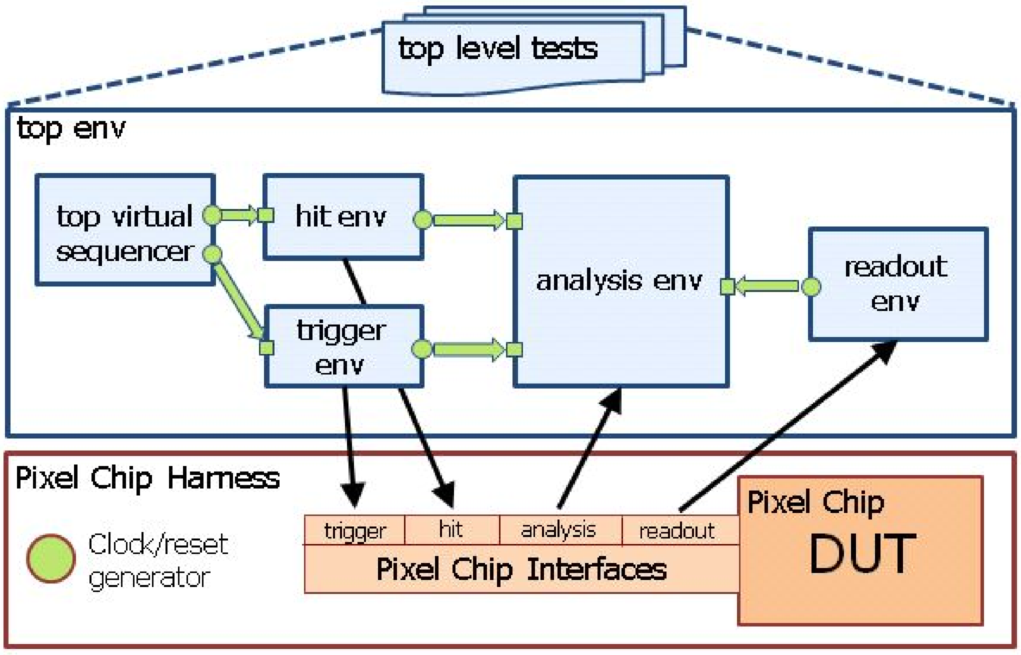


FIGURE 5.2: Block diagram of VEPIX53.

1. the DUT, its interfaces to the environment and a generator for fundamental signals (i.e clock and reset) on the bottom, highlighted by the use of an orange colour and wrapped by the *Pixel Chip Harness* module;
2. the actual simulation and verification environment (*top env*) whose blocks are represented with a blue colour.

A major difference between the two parts has to be highlighted: while the first is described in a standard Verilog module-based fashion (even if SystemVerilog constructs are used), the verification part is described at a higher level, the so-called transaction level, using the capabilities of OOP. This means that the components that are shown in 5.2 are specific instances of classes. To be more precise, in a UVM environment with different types of instances of classes can be distinguished (see Figure 2.1 for identifying them in the UVM library):

- classes that are directly inherited from *uvm_object*s are actually dynamic and continuously allocated and deallocated during the simulation. An example of such objects are the *uvm_sequence_item*s, i.e. the transactions that are passed between the channels of the different components to allow them to communicate.
- *uvm_component*s are instances that can be considered quasi-static. Even if they are also instances of classes, that need to be built in the UVM *build_phase()*, they are "stable" and cannot in general be removed from the environment during a single simulation. Such components provide some additional capabilities, e.g. phasing and execution control, hierarchy information functions, configuration methods, factory methods, hierarchical reporting control [28].

As concerns the *Pixel Chip Harness* it should be reminded that the description of the DUT has been performed by using a time-based and behavioral description. Even if different levels are also being considered (e.g. a TLM description) at the time of writing the framework is not capable of interfacing all the different verification components with them. That means that some additional work needs to be performed to achieve better flexibility and generality. To conclude the overview of the block diagram, it can be noticed the presence of some interfaces (*Pixel Chip Interfaces*) between the chip and the verification environment. Such modules can be considered the "link" between the module-based and the class-based worlds. It can also be observed that an "environment" (*env*) has been defined for generating input and/or reading the outputs of each one of the pixel chip interfaces. While the DUT architecture has been exhaustively presented in the Chapter 4, further details on the verification components and on the interfaces to the chip will be provided in the following sections and subsections.

5.1.1 Interfaces

Just to provide a brief introduction on SystemVerilog interfaces, they were specifically created to encapsulate communication between blocks. At its lowest level, an interface is indeed simply a named bundle of nets or variables. The interface is instantiated in a module and can be accessed through a port as a single item but also the single nets or variables can be referenced if needed. The ability to replace a group of signals by a single name can significantly reduce the size of a description and improve its maintainability. Additional power of the interface comes from its ability to encapsulate functionality as well as connectivity. Indeed, also tasks and functions can be declared within an interface and called from modules that are connected to the interfaces. Moreover, clocking blocks and modports can also be used when dealing respectively with synchronous signals and with different blocks in the environment accessing the same interface with different roles (these last features have been used in this work). Further details that are not herein presented can be found in dedicated chapters both in [24] and in [26], as well as in various online resources. As regards VEPIX53, following UVM guidelines, the *interface* construct has been used to bundle the DUT signals that need to be connected to the testbench. It has to be highlighted that, since they are actually nothing less than modules, they can be instantiated in a module but not inside a class. For this reason, SystemVerilog provides with an additional *virtual* keyword to allow the verification engineer to place a pointer to the physical interface in a class [28]. In this work, the interfaces have been instantiated in the harness block and the pointers to them are registered in the UVM configuration database. In such a way, references to them are available for the class-based verification components. So far, four interfaces have been defined for the current versions of the DUT, that is to say:

1. *hit interface*: it contains a bi-dimensional array of "analog" (as presented in section 4.1, the *real* datatype has been used for each of them) hit signal that represent the amplitude of the hit that is fed to each pixel of the pixel chip; it has been so far defined as a synchronous interface using clocking blocks. An asynchronous version is also defined for initial studies done on the introduction of asynchronous delays or analog baseline noise to the incoming hits. Such aspects have not been completely developed at the time of writing;

2. *trigger interface*: it contains the physical trigger, represented with a simple yes-no signal. Also in this case synchronous access to the interface has been defined;

3. *readout interface*: it is meant to communicate with the pixel chip output. At the current status, its actual implementation can slightly vary depending on the specific version of the DUT being simulated. In any case, for both the independent pixel and the zero-suppressed FIFO architectures, it consists of a buffer output

packet, whose structure has been previously shown in Figure 4.14 and 4.16. The access to such an interface has been defined as synchronous;

4. *analysis interface*: this is a peculiar interface, since it does not contain actual inputs or outputs of the DUT, but it is used to probe virtual flag signals related to DUT status. This approach has been chosen for collecting statistical information on pixel deadtime due to TOT and to digital hit processing (such signals have been shown in the timing diagram in Figure 4.11) and buffer occupancy. As for the other physical signals, such internal flags are accessed in a synchronous fashion. Further monitoring of the DUT status could be performed adding additional and meaningful signals to such an interface.

5.1.2 Project organization

In this subsection the way the project directories of the UVM framework are organized is described. When accessing the svn repository three main directories are (by default) present: branches, tags and trunk. VEPIX53 project has been uploaded in the latter. Therein, one can find two main directories that are linked to such a project, shown in the Figure 5.3. As it can be noticed, a directory is used for gathering all the source files,

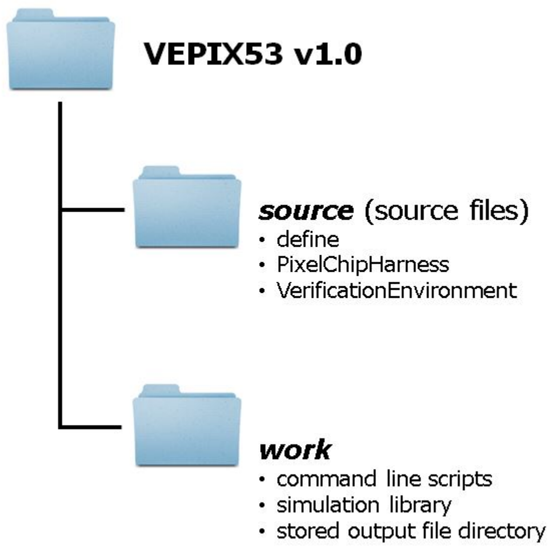


FIGURE 5.3: Project organization: first level of directories.

i.e.:

- the *define* folder contains both files where the `*define* compiler directives are declared and a file which groups all the user-defined data types;
- the module-based parts of the environment are grouped in the *PixelChipHarness* directory;
- all the files related to the verification part are gathered in the *VerificationEnvironment* directory.

As regards the *work* directory, that is the location where the simulation is supposed to be run. It contains the command line scripts, a directory for possibly storing the output files of the simulation and the actual simulation library automatically generated after each simulation. Further details on such files will be provided at the end of the chapter. It is instead worth it to focus on how the *VerificationEnvironment* directory is organized, since it has a strong relation with the framework architecture, as presented in Figure 5.4. As recommended in [28] and in a lot of online resources, a different directory

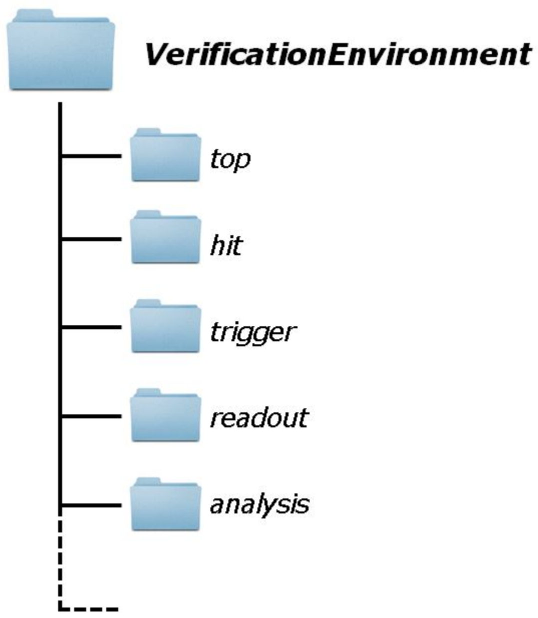


FIGURE 5.4: Project organization: directories of the verification environment.

is defined for each "environment" of the framework. With this term one refers to a reusable verification block which is in general an encapsulated, ready-to-use, configurable verification component for an interface protocol, a design submodule, or a full system [4]. The recommended approach is to define a directory for each interface protocol of the DUT. In this case for each interface between the environment and the DUT, already described in subsection 5.1.1, a directory has been used: hit, trigger, readout and analysis can be distinguished in Figure 5.4. For each of them the SystemVerilog source files are contained in the *sv* folder and a package file (identified by the *_pkg* suffix) is used to group them in a unique compilation scope, while keeping separate files makes the code more clear and maintainable even when growing in complexity. Besides, a top directory is used to wrap the top level environment and all the other top level classes, together with the UVM tests: the role of such components will be presented more in details in the following sections. When other interfaces to the DUT will be defined in the future, it is good practise to dedicate a folder to each of them, adding elements to the provided list. Separate packages are used for the top level classes and the tests.

5.2 Verification components

As for the project directories, also the definition of the different verification components of the framework is highly related to the existent interfaces between the DUT and the environment: it has been extensively highlighted in the block diagram of the framework in Figure 5.5. Even with some differences depending on the specific

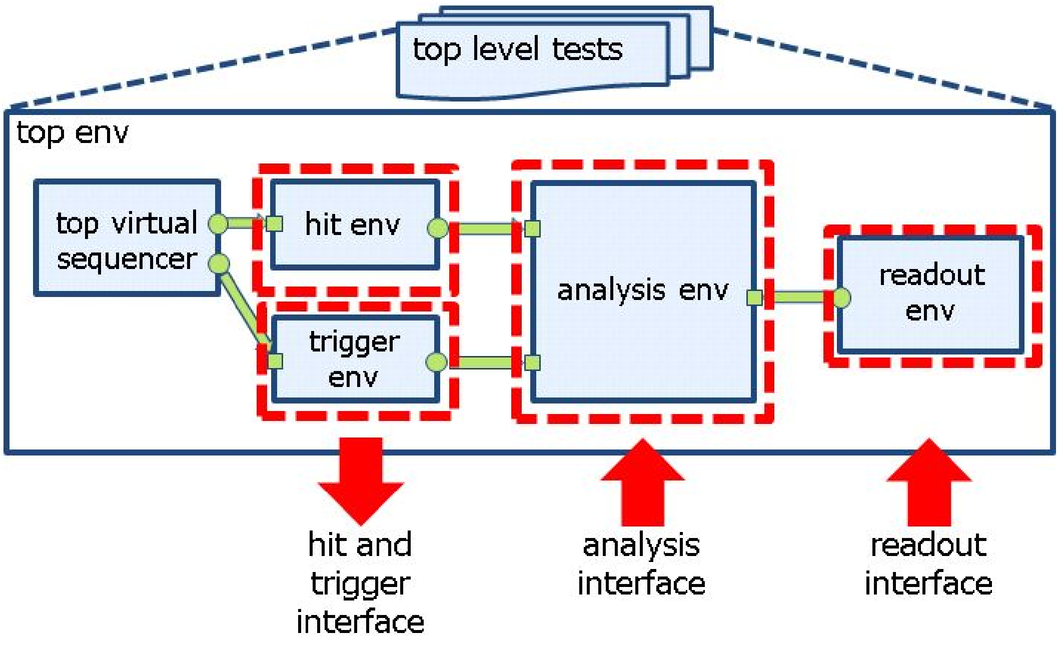


FIGURE 5.5: Block diagram of the verification components of VEPIX53, with focus on their link to the DUT interfaces.

interface, each environment of the design is composed of some basic UVM verification components that are recurrently used. Moreover, within an environment different instances communicate at high level through channels, where objects are passed at the transaction level. Normally, a certain type of transaction is used for internal communication in each of the environments. In UVM testbenches, such structures are represented extending the *uvm_sequence_item* class. Data items basically represent the input and output of the DUT at a higher level of abstraction than that of the physical signals that are actually driven to them. Other types of objects are usually part of UVM testbenches:

- *uvm_sequence*s, i.e. a user-defined sequence of data items and/or sequences;
- configuration objects, used to gather configuration fields of each one of the different verification environments. Potentially they could also be randomized.

As regards the verification components, the main ones that have been used all over the project are the following:

- a stimulus generator (*uvm_sequencer*) to create transaction-level traffic to the DUT. For this purpose it runs a *uvm_sequence*;
- a driver (*uvm_driver*) to convert these transactions to signal-level stimulus at the DUT interface. Connection between the sequencer and the driver is done through a point-to-point port-export connection, realized at the Transaction Level (TL);
- a monitor (*uvm_monitor*) to recognize signal-level activity on the DUT interface and convert it into transactions: it is a passive element since it does not drive any signal towards the DUT;
- an analysis component, such as a coverage collector or scoreboard, to analyze transactions. In the current project, for such components the suffix *subscriber* has been used;
- an agent (*uvm_agent*) is an abstract container with a monitor and (if active) also a driver and a sequencer. If passive it is intended just to monitor DUT outputs.

Following the common approach of OOP, all those classes are extended from a common base one: *uvm_object*. The detailed hierarchical relationship can be found in

Figure 2.1, where also the distinction between dynamic *uvm_objects* and quasi-static *uvm_components* can be noticed. Further information on the specific use done of such classes in VEPIX53 will be provided in the following subsections.

5.2.1 Hit environment

The hit environment is supposed to inject and monitor hits sent to the pixel chip matrix. The basic hit environment block diagram is already reported in Figure 5.6. The structure that has already been introduced can be recognized: in the *hit*

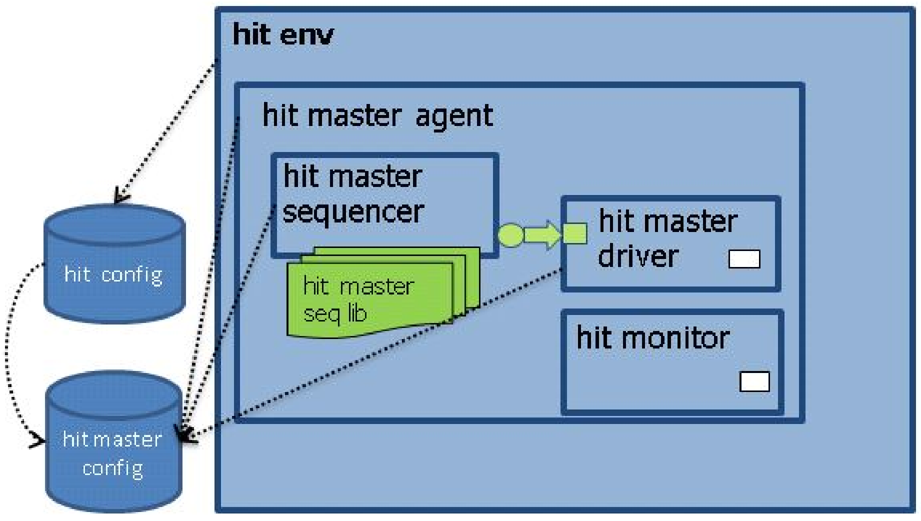


FIGURE 5.6: Hit environment block diagram.

environment, a *hit master agent* can be identified as a wrapper for three other main blocks: the *hit master sequencer*, *driver* and *monitor*. All those components are inherited from the UVM specific and correspondent classes. In the figure the TL connection between the sequencer and the agent is represented with the green arrow, while the library of possible sequences that can be run by the *hit master sequencer* is shown under its block with a similar green colour. The modules that have also a link to a physical DUT interface contain a white rectangle that represents the virtual interface that points to it. It can be also noticed as a hierarchy has been defined for the configuration objects. Indeed, the *hit master config* object is the one to which all the lower level components (wrapped in the agent) point, while the higher level

components (in this case just the *hit environment*) point to the *hit config*, that contains itself a reference to the lower level configuration object. Although quite redundant, this approach has been used following the UVM guidelines. With such a structure it would be possible to define other lower level configuration objects (for other components in the hit environment) and to point to each of them using the higher level configuration object.
As regards the types of transactions used in such an environment, an unusual distinction has been done between the ones used at the driving level and the ones used by the monitor, in particular:

1. *hit_trans* is used at the generation side;
2. *hit_time_trans* is used at the monitoring side.

The class diagram of both of them is reported respectively in Figure 5.7 and 5.8, following the Unified Modelling Language (UML) scheme. The most relevant fields of the hit_trans

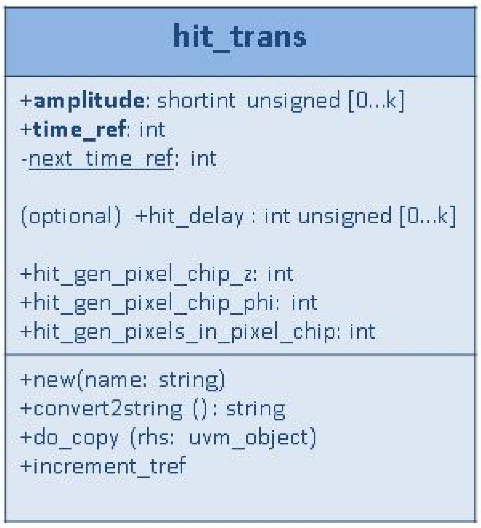

hit_trans
+amplitude: shortint unsigned [0...k]
+time_ref: int
-next_time_ref: int
(optional) +hit_delay : int unsigned [0...k]
+hit_gen_pixel_chip_z: int
+hit_gen_pixel_chip_phi: int
+hit_gen_pixels_in_pixel_chip: int
+new(name: string)
+convert2string (): string
+do_copy (rhs: uvm_object)
+increment_tref

FIGURE 5.7: UML class diagram of the *hit_trans*.

have been highlighted in bold. To the hit transaction that it is meant to be sent to the pixel chip, it is associated a time reference value, so far correspondent to the bunch crossing cycle (BX), and an array of amplitude values. Since pixel chips are very large pixel matrices, where most of the elements do not actually need to be accessed at a single time an associative array has been chosen. It has already been described in section 3.1.1 how this can significantly save simulation resources. Such a type of array is forced to

have just one-dimension. This means that when driving the hit signals to the actual bi-dimensional DUT a conversion from the 1D indexes to the 2D ones is performed, based on the parameters set for the pixel chip size. For a similar reason the information on the pixel chip matrix simulated at the generation level (not necessarily equal to the actual pixel chip matrix, as will be clarified in Chapter 6) is also memorized in such an item in the remaining fields to be available for the transaction methods. As regards the datatype used for the array of amplitude values a *shortint* was initially suggested in the collaboration. This means clearly that randomization is done on a subset of all possible *real* values that may be driven to the hit interface; at the time of writing there was no interest on complicating the hit generator in order to perform real value randomization, it indeed requires particular expedients and can slow down the simulation. An additional field of such a transaction, classified as "optional" since it is selected through a compiler directive, is a preliminary model for the delay of the hits going to each pixel with respect to the synchronous bunch crossing cycle. Such a feature has been developed just at the generation level and in order to use it also at the monitoring level further development is needed.
As regards the transaction methods, they are just the standard ones that normally need to be overwritten for *uvm_sequence_item*s:

- the constructor *new()*, normally used in OOP to create an object, even if in UVM it is not recommended to directly use it (the *create()* function is instead usually adopted since it delegates the call to *new()* to the UVM factory);
- the method used to print details of the item (*convert2string()*);
- the *do_copy()* function used to create a so-called "deep copy" of the object (i.e. a complete new pointer to a new but identical object).

Such methods could be also directly inherited from the parent classes, but it is good practise to override them. A summary of the standard transaction methods and of the right way of invoking them can be found in the dedicated section of [27]. The only additional method is a very simple one used to increment the time reference value from the *hit master sequence*s when generating a sequence of transactions. As regards the hit_time_trans diagram reported in 5.8, it can be noticed that in this particular case a slightly different approach has been used at the monitoring level: a bi-dimensional

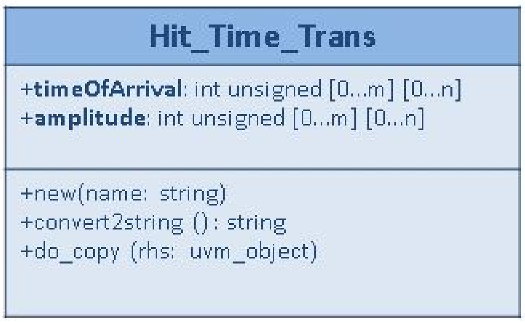

Hit_Time_Trans
+timeOfArrival: int unsigned [0...m] [0...n]
+amplitude: int unsigned [0...m] [0...n]
+new(name: string)
+convert2string (): string
+do_copy (rhs: uvm_object)

FIGURE 5.8: UML class diagram of the *hit_time_trans*.

array has been used to store both the monitored information of the TOA of a hit and its amplitude. It means that there is a 1:1 correspondence between the bi-dimensional arrays and the pixel chip matrix. This kind of approach has been kept mainly for simplicity, but it is not excluded that when starting to simulate larger DUTs it may turn out to be worth it to switch to more optimized ways of storing data. Once more, from a quick look at the transaction methods it can be noticed that they are simply the standard ones that have already been described. As concerns the operations of the blocks in the hit environment which are functional for the simulation, they can be summarized in the following way:

- the *hit master sequencer* is used to run user-defined sequences of hit transactions which are collected in the sequence library *hit_master_seq_lib*. It includes 3 sequences, but it may be extended adding more of them:

    1. the *hit sequence* generates a user-defined number (*num_trans*) of transactions, one for each BX cycle. In any of them a randomization is separately done for the amplitude value of each pixel in order to decide whether to hit it or not. The probability of hitting each pixel is set through the *is_hit_prob* field, while the maximum and minimum allowed amplitude for the generated signals are specified through dedicated variables. As it will be better shown in subsection 5.2.6, such parameters can be accessed by the test writer from a unique test file. In the context of the sequence they are simply *get* at the beginning of the so-called *body()* task, that is the main one for an object, assuming that they have been *set* in the test. The *get* and *set* terms refer to the actual functions used to access the UVM configuration database, that enables

configuration parameters to be exchanged between different hierarchical levels of the environment;

2. the *hit_sequence_poisson* also generates a parameterized number (*num_trans*) of hit transactions. For each BX a randomization is separately done for each pixel in order to decide the amplitude of the generated hits. For this particular and very simple model the amplitude have been set to follow a Poisson distribution with specified seed and mean. There is no correlation of such an example sequence with realistic ones;
3. the *hit_sequence_cluster* creates a sequence of a certain number *num_bx_cycles* of hit transactions, one for each BX. It is anyway much more complex compared to the previous and it contains a high number of fields. For this reason a more extensive description of it will be provided in Chapter 6;

In order to impose constraints to the fields of the generated transactions in a flexible and time-saving way (some studies about how the SV constraint solver slows down the simulation when having to randomize in parallel big arrays can be found in [36]) they have been imposed using dedicated tasks in the sequences, executed each time a new hit transaction needs to be created and randomized. Such an approach would also in the future allow designers to write into the transaction not randomized data, but input coming from other sources, like from physics simulations. It can be also highlighted that the specific sequence to run in the sequencer is something that can be defined at a higher level, without having to look in the lower level detailed file. See 5.2.5 for more information;

- the *hit master driver* main operations can be found in the *run_phase()*: after an initial waiting period (set in the hit configuration object) it starts getting transactions from the sequencer for each clock cycle. Its role is to translate the high level information contained in the transaction into the the hit interface signals. The *drive()* task is used to do it in parallel for each pixel of the DUT. Parallel-threading has been achieved thanks to the SystemVerilog *fork...join_none* construct, that makes it possible to launch a sequence of operations and to continue with the execution of the code without waiting for any of them to conclude (see [26] for details on Multi-threading with SystemVerilog). The possibility of generating a sinusoidal baseline noise and of adding delays to the driven hits has also been investigated at the driving level for potential further developments: since it was not considered as a priority, the related code has been isolated with the use of compiler directives). A timing diagram obtained by Incisive Cadence showing such hit signals with an added baseline noise is reported in Figure 5.9 for two PUCs. For the use of the mathematical functions access to the C library has been done through the SystemVerilog DPI interface as presented in [37];

- the *hit monitor* senses the hit interface: when hits are seen, it measures the time and amplitude for every pixel, converts the time in the bunch crossing cycle information and gathers such data in the hit_time_trans in order to send them to the analysis components.

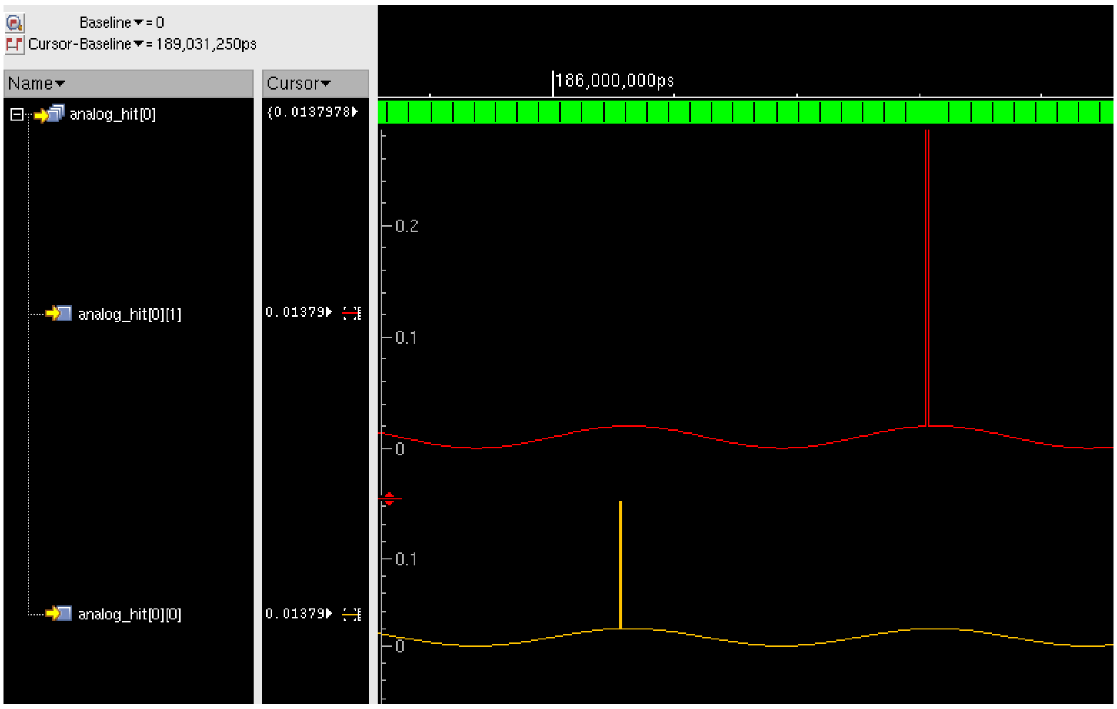


FIGURE 5.9: Timing diagram of two PUCs with analog hit signals with an added baseline noise.

As regards the *hit subscriber* and the hit configuration objects, they will be described in detail in Chapter 6, since their role is much linked to the topic therein presented.

5.2.2 Trigger environment

The trigger environment is in charge of driving and monitoring the trigger signal used by the pixel chip to select hits of interest. The block diagram of such a component is

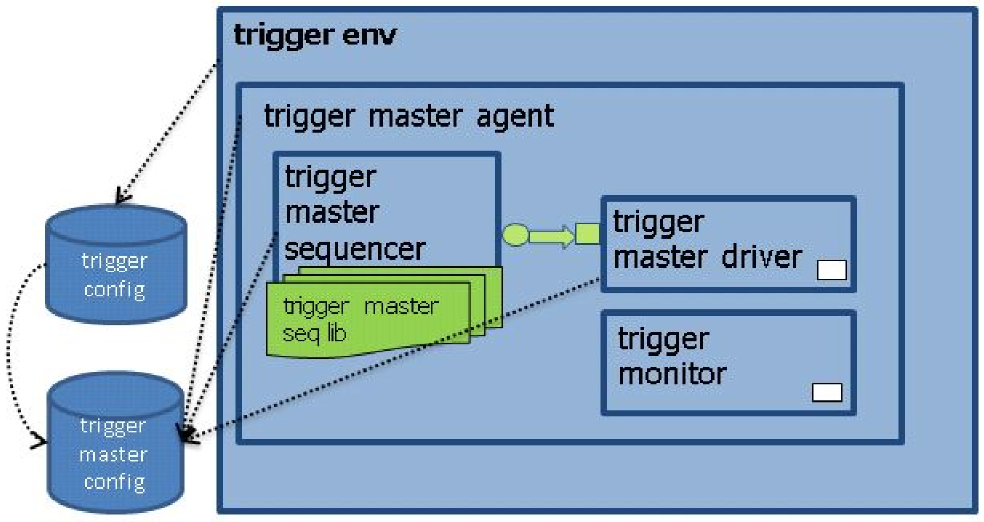


FIGURE 5.10: Trigger environment block diagram.

reported in Figure 5.10. It can be noticed as its structure is basically identical to the hit environment one, since it is also responsible of generating inputs to the chip and according to UVM re-usability principle. The same approach has been also followed for the definition of transaction items used in such a component, i.e.:

1. *trigger_trans* is used at the generation side;
2. *trigger_time_trans* is used at the monitoring side.

Their UML class diagram can be observed in Figure 5.11 and 5.12. As it can be noticed,

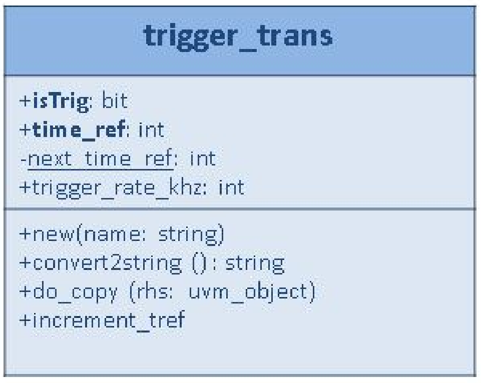


FIGURE 5.11: UML class diagram of the *trigger_trans*.

each trigger transaction contains few fields:

- a *isTrig* bit, used to decide whether or not to generate a trigger signal for a certain BX; such a decision is taken depending on the *trigger_rate_khz* field;
- a time reference value, correspondent to the BX of the associated hits (no information on the latency is added at this level);
- in addition, a field is used to pass to each transaction the trigger rate specified from the sequence. In this way it can be used to impose constraints on the *isTrig* value from the transaction itself, using the standard SystemVerilog constraint solver, since in this case there is no need to use special expedients as for dealing with randomization of big arrays.

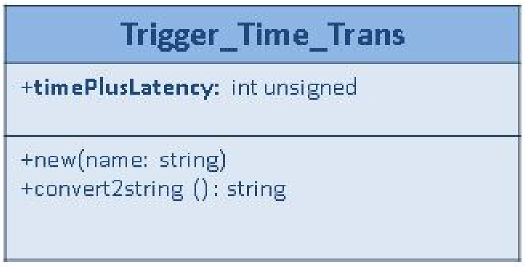


FIGURE 5.12: UML class diagram of the *trigger_time_trans*.

As regards the trigger_time_trans diagram reported in 5.12, it contains just one field. Indeed, since such a transaction is produced only if the trigger signal is sensed, the only interesting information is related to the time when it has been detected. Even if still represented in number of clock cycles, the *timePlusLatency* value already adds the trigger latency. As concerns the methods of both the presented items, they are exactly the same standard and non-standard ones that have been already described for the hit environment. In the trigger environment reference is made to a configuration object. The same hierarchical structure used for the hit configuration objects has been used. The *trigger_master_config* is referenced by the components wrapped in the master agent, while for the higher level components a pointer to the *trigger_config* object is adopted. Such an item so far just contains the reference to the lower level configuration object, but other pointers could be potentially added if needed for configuring other new components. The fields of the *trigger_master_config* can be seen in Figure 5.13. It can be notice the presence of:

- a configurable name for the object;

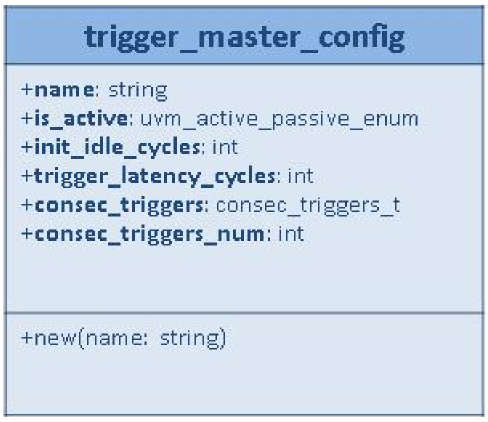

trigger_master_config
+name: string
+is_active: uvm_active_passive_enum
+init_idle_cycles: int
+trigger_latency_cycles: int
+consec_triggers: consec_triggers_t
+consec_triggers_num: int
+new(name: string)

Figure 5.13: UML class diagram of the *trigger master config*.

- a variable to specify whether the trigger master agent is active or passive;
- an initial number of cycles before the start of operations of the driver;
- a variable to define the trigger latency, expressed in number of BX cycles;
- a variable to define if any number of consecutive triggers can be generated (*consec_trig* = UNLIMITED) or if such a value should be limited (*consec_trig* = LIMITED) to a certain value (such a data type is user-defined, and can be found in dedicated file the *define* folder, as introduced in the section 5.1.2);
- an integer to specify the number of allowed consecutive triggers (if limited).

The operations performed by the main components of the trigger environment are herein listed:

- the *trigger master sequencer* is used to run sequences of trigger transactions. The *trigger master seq lib* contains just one simple sequence (*trigger_sequence*). It has two fields, used to specify the number of bunch crossing cycle to run (*num_trans*) and the trigger rate. Both those two values can be set from the test file;
- the *trigger master driver* waits for an initial period (set in the trigger configuration object) and afterwards begins to get trigger transactions from the sequencer at each clock cycle in order to translate them to the trigger physical signal. Indeed, such a component has a pointer to the trigger interface. Once more, the *drive()* task is used for such a goal. It also takes into account possible limitation to the number

of consecutive triggers that it is allowed to send to the DUT, as specified by the configuration object;

- every time a trigger signal is detected in the trigger interface, a trigger_time_trans is built by the *trigger monitor* and sent to the analysis ports that are in charge of driving it to the analysis environment.

5.2.3 Readout environment

The development of the readout environment has not been the focus of my personal work. It will be anyway described for completing the overview on the whole framework. Its block diagram can be seen in Figure 5.14. As it can be seen, its structure is extremely

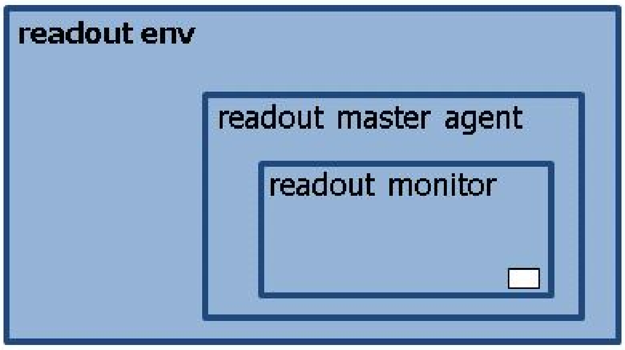


FIGURE 5.14: Readout environment block diagram.

simple: it basically contains one passive agent, used to wrap the monitor. Such a component is indeed not meant to drive any signal to the DUT, but just to sense its outputs from the readout interface and to group them into a *readout_trans*. Even if its role is generic, the specific structure of such a transaction is so far dependent on the format of the DUT output. It is straightforward to notice the correspondence between the output packets of the two different DUT architectures presented in the previous chapter in the Figures 4.14 and 4.16 with the UML class diagram reported respectively in Figure 5.15 and 5.16. It can be seen as in the first case an array is used for both the TOA and the TOT (amplitude) of the hits and as it refers to an architecture with independent pixels. As regards the methods of such a transaction, it can be noticed the presence of some standard ones and only a simple additional one (*get_ToA()*) used to obtain the maximum TOA value contained in the array. Comparing it with the second class diagram, it is clear how the TOA is stored just once per each PR, while a separate

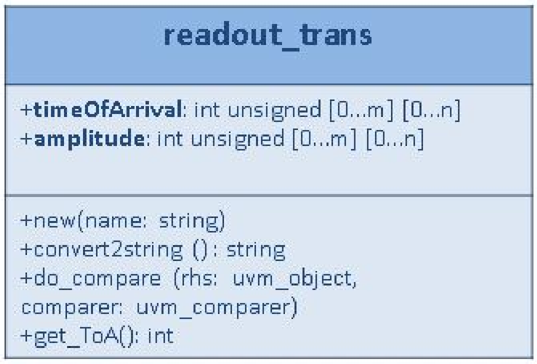

readout_trans
+timeOfArrival: int unsigned [0...m] [0...n]
+amplitude: int unsigned [0...m] [0...n]
+new(name: string)
+convert2string (): string
+do_compare (rhs: uvm_object, comparer: uvm_comparer)
+get_ToA(): int

FIGURE 5.15: UML class diagram of the *readout trans* for the independent pixels architecture.

TOT value is present for each PUC in the region. As concerns the role of the *readout*

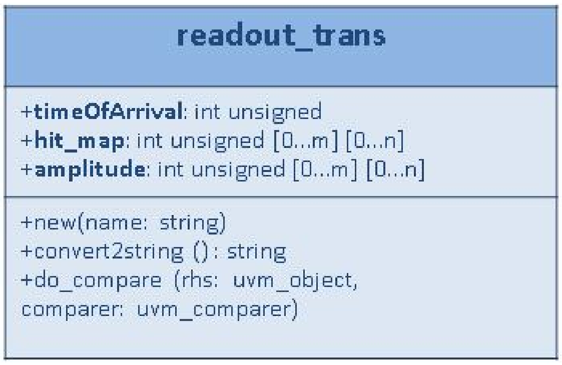

readout_trans
+timeOfArrival: int unsigned
+hit_map: int unsigned [0...m] [0...n]
+amplitude: int unsigned [0...m] [0...n]
+new(name: string)
+convert2string (): string
+do_compare (rhs: uvm_object, comparer: uvm_comparer)

FIGURE 5.16: UML class diagram of the *readout trans* for the zero-suppressed FIFO architecture.

monitor, it basically checks at each bunch crossing cycle if the output of the PR buffer has changed and if it is the case case senses the signals of the readout interface and uses them to build one *readout trans*.

5.2.4 Analysis environment

I have have not been personally involved on the development of this component, apart from some minor aspects. It will be anyway presented for completeness without providing many details. The block diagram of such an environment is shown in Figure 5.17. Besides the usual master agent wrapping the monitor, many other components can be noticed:

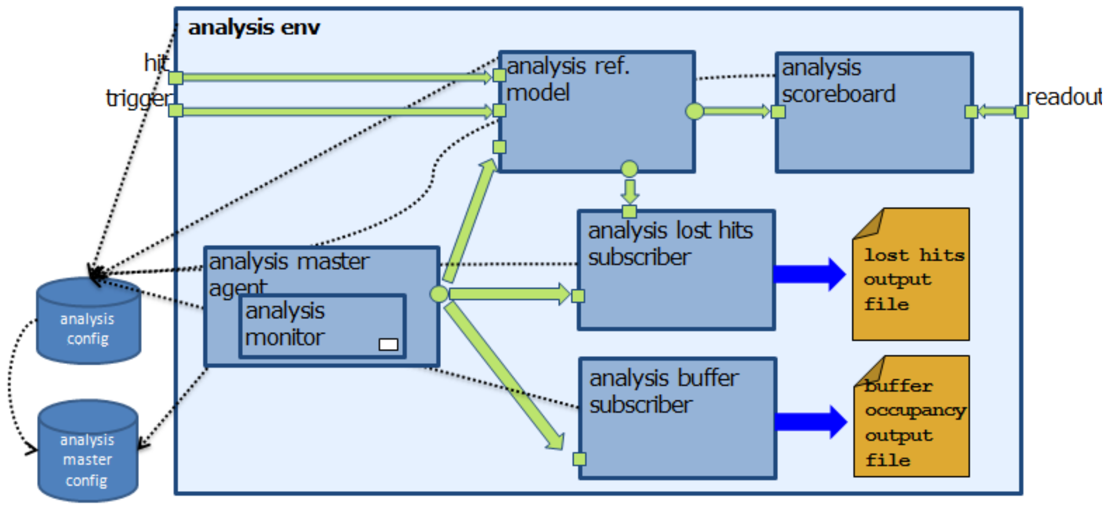


FIGURE 5.17: Block diagram of the analysis environment.

- a reference model that, fed with the same hit and trigger inputs, emulates the ideal behaviour of the DUT and predicts the expected output;
- a scoreboard for conformity checking between predicted and actual DUT outputs: results on matches and mismatches are reported in the simulation log file;
- a lost hits subscriber for collecting classification of lost hits and printing it to a text output file during the UVM *report_phase()*;
- analysis buffer subscriber, for collecting histograms of buffer occupancy of the PR buffer and printing such information to a text file during the UVM *report_phase()*.

It can be also noticed the presence of the usual hierarchy for the configuration objects. As regards the transaction (*analysis_trans*) created by the *analysis_monitor* and sent to the other components of the environment, it has the role of grouping signals used to monitor the DUT status: its field depend on the DUT. Indeed, to be capable of classifying sources of hit loss and providing statistics collection, the signals that need to be monitored can vary from one DUT to another. Moreover, such signals are also used by the reference model to emulate the DUT behaviour without predicting DUT outputs that are already seen to be lost. That would otherwise cause a sequence of mismatches in the scoreboard (since it simply compares the reference model output with the actual DUT output) without any possibility of recovering. At the time of writing, modifications on the reference model itself are likely to be necessary when changing the DUT under simulation. The UML class diagram of the *analysis_trans* used for both the independent PUC and the zero-suppressed FIFO architectures are shown, respectively in Figure 5.18 and 5.19. Status signals probed for both the architectures are:

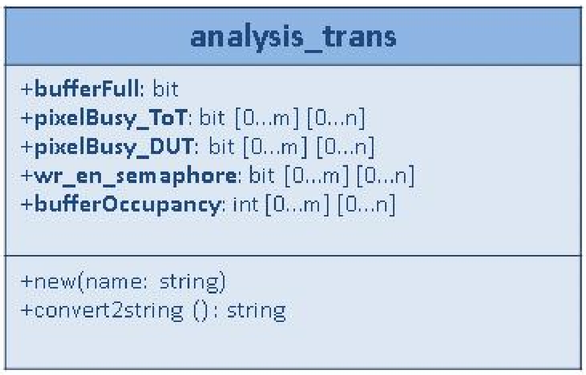

analysis_trans
+bufferFull: bit
+pixelBusy_ToT: bit [0...m] [0...n]
+pixelBusy_DUT: bit [0...m] [0...n]
+wr_en_semaphore: bit [0...m] [0...n]
+bufferOccupancy: int [0...m] [0...n]
+new(name: string)
+convert2string (): string

FIGURE 5.18: UML class diagram of the *analysis_trans* for the independent PUCs architecture.

- the buffer overflow flag;
- the flag used to monitor when each PUC is busy measuring the TOT;
- the flag used to check if the PUC is "blind" because digitally processing a hit;
- the flags used to get information on buffer occupancy: the value itself and the *wr_en_semaphore* that decides when to actually monitor it (only on write).

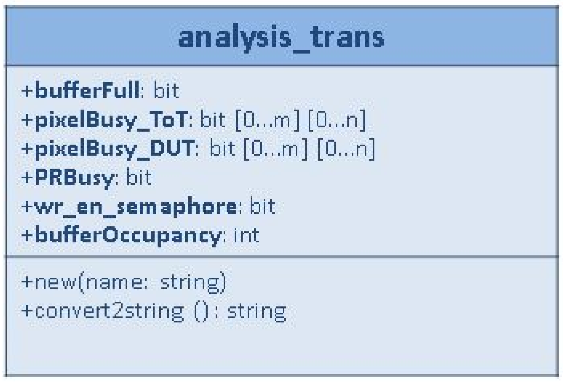


FIGURE 5.19: UML class diagram of the *analysis_trans* for the zero-suppressed FIFO architecture.

For the zero-suppressed FIFO architecture an additional flag is being probed since the PR buffer is shared: when one or more pixels are fired in a PR, it is blind for a certain number of clock cycles and such a dead time is monitored through the *PRbusy* flag. The *analysis_master_config* is referenced by the components wrapped in the master agent, while for the higher level components a pointer to the *analysis_master_config* is used. The UML class diagram of such an object can be seen in Figure 5.20. Beside the presence of

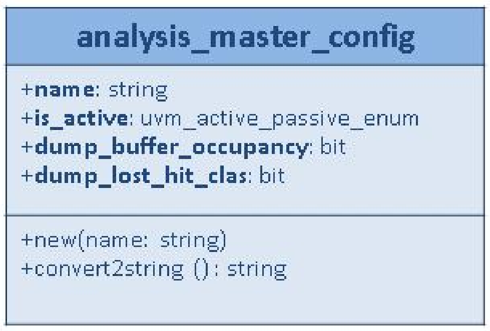

analysis_master_config
+name: string
+is_active: uvm_active_passive_enum
+dump_buffer_occupancy: bit
+dump_lost_hit_clas: bit
+new(name: string)
+convert2string (): string

Figure 5.20: UML class diagram of the *analysis_master_config*.

the usual fields (configurable name for the object and variable for active/passive agent), there are two additional fields that are actually responsible for the structure and behavior of the analysis environment. In particular, it has been said that such a component is capable of producing two output files with report on classification of lost hits and on buffer occupancy. It has been also shown as two dedicated components (subscribers) take care of generating them. In order to provide flexibility and configurability it has been left to the user to decide whether to print or not such output files. This can be achieved by setting accordingly the following switches in the test (as it will be shown in subsection 5.2.6):

- *dump_buffer_occupancy*, if active it causes the analysis buffer subscriber to be built into the analysis environment, and the corresponding output file to be generated;
- *dump_lost_hit_clas*, if asserted it causes the analysis lost hits subscriber to be built and the corresponding output file to be produced.

Such a strategy has been followed for all the text or data output files that generate reports or statistical analysis.

5.2.5 Top level environment

The top level environment represents the wrapper for the whole simulation framework, reported in the block diagram of Figure 5.2. All the other components that have been described are therein built during the UVM *build_phase()*. In addiction to those, a component which has a fundamental control role is defined: the *top virtual sequencer*, that runs virtual sequences contained in the correspondent library (*top virtual seq lib*). A virtual sequencer is in charge of coordinating the stimulus across different interfaces and the interactions between them. The use of the adjective "virtual" is related to the fact that such a sequencer is not directly connected to any driver linked to a DUT interface. It only contains pointer to the lower level sequencers. Once the pointers have been defined, it is possible for a virtual sequence to control which specific sequences (taken from the correspondent library) to run on each of the non-virtual sequencers and how to do it, e.g. in parallel or serially. Just to provide an example, the *top_virtual_sequence* present in the library launches in parallel (through a standard *fork...join* process) an instance of type *hit_sequence* and one *trigger_sequence* respectively in the *hit_master_sequencer* and in the *trigger_master_sequencer*. As regards the top level coordination of the environment it is probably also worth it to mention the way it is defined when the simulation ends. It could be potentially just defined a certain duration time or the Verilog system function *$finish* could be used, nevertheless when dealing with complex environments with multi-threading those classical approaches do not always ensure control on the actual moment in which the planned simulation comes to an end or they may not be flexible enough. UVM provides an objection mechanism to allow hierarchical status communication among components. There is a built-in objection for each phase, which provides a way for components and objects to synchronize their testing activity and indicate when it is safe to end the phase. In general, components or sequences have to raise a phase objection at the beginning of an activity that must be completed before the phase stops and to drop the objection at the end of that activity. Once all of the raised objections are dropped, the phase terminates. So end-of-test in the UVM is controlled by managing objections: for *uvm_component*s the objection mechanism is automatic while it is normally necessary to handle it for the sequences. Every time a root sequence (without any parent sequence) is launched an objection is raised, while when it ends its operations it is dropped, as suggested in [4]. For details on such a mechanism one can refer to UVM references like [27] and [28].

5.2.6 Top level tests

As it can be seen on top of the framework block diagram in Figure 5.2, UVM tests are separated from the top level environment itself. The *uvm_test* class defines a specific test scenario for the top environment specified in the test, enabling configuration of the verification components. A peculiarity of the *uvm_test* class is that it can be directly launched using command line options and it gets automatically instantiated. As recommended ([4]), a base test class that just instantiates and configures the top level environment has been defined, and then extended to obtain scenario-specific configurations of the environment. In the basic test it is also chosen to print in the log file the environment hierarchy and the components registered in the so-called UVM factory. Since this is the most important file to be modified by possible users, some details on how it is organized will be provided in this subsection. So far, three specific tests, all extended by the *top_base_test* one, have been written and they have been named with increasing numbers, i.e: *top_test1*, *top_test2* and *top_test3*. The correspondent files can be found in the *tests* folder in the *source/VerificationEnvironment/top* path, as shown in subsection 5.1.2. The *build_phase()* of such a class is crucial for the configuration of the environment. Indeed, during such a phase, for each one of the lower level environment for which a configuration object has been defined, it is created and specific values are specified for its fields. The way it is done in the *top_test1* for the trigger environment is shown as an example:

```
// Creating the configuration objects
m_trigger_cfg  = trigger_config  ::type_id::create("m_trigger_cfg");
m_hit_cfg      = hit_config      ::type_id::create("m_hit_cfg");
m_analysis_cfg = analysis_config ::type_id::create("m_analysis_cfg");

// Setting the trigger configuration
m_trigger_cfg.add_master("m_trig_agent",
                         UVM_ACTIVE,
                            ‘INIT_IDLE_CYCLES,
                            ‘PIXEL_CHIP_TRIGGER_LATENCY,
                            LIMITED,
                            3);
```

The fields of the trigger configuration object that were presented in subsection 5.2.2 can be recognized. Such objects are then *set*, going through the design hierarchy, in the pointers present in the correspondent environments, as shown in the following code:

```
    uvm_config_db#(trigger_config)  ::set(this,"*.m_trig_env",
                                     "cfg", m_trigger_cfg);
    uvm_config_db#(hit_config)      ::set(this,"*.m_hit_env",
                                     "cfg", m_hit_cfg);
    uvm_config_db#(analysis_config) ::set(this,"*.m_analysis_env",
                                     "cfg", m_analysis_cfg);
```

This is possible thanks to the UVM configuration mechanism. The *uvm_config_db* is a type-specific configuration mechanism, offering a robust facility for specifying hierarchical configuration values of desired parameters. It is built on top of the more general purpose *uvm_resource_db* which provides side-band (non-hierarchical) data sharing [4].

In the designed test files the final steps performed using once more the *uvm_config_db*, are the following:

- it is chosen which top level sequence of the library to run in the *top virtual sequencer*;
- the fields of specific lower level sequences (i.e. *hit_sequence* and *trigger_sequence*) are configured.

Here it is an example the detailed code from the *top_test1*:

```
// The top virtual sequence configuration
uvm_config_wrapper::set(this, "m_env.m_virt_seqr.run_phase" ,
                        "default_sequence",
                        top_pkg::top_virtual_sequence::type_id::get());
// Configuring the specific sequences run in the low level sequencers
// trigger sequence
uvm_config_db#(int)::set(this,"*.top_virtual_sequence.m_trig_seq",
                        "num_trans", 10000);
uvm_config_db#(int)::set(this,"*.top_virtual_sequence.m_trig_seq",
                        "trigger_rate_khz", `TRIGGER_RATE_KHZ);
// hit sequence
uvm_config_db#(hit_config)::set(this,"*.top_virtual_sequence.m_hit_seq",
                                "m_hit_cfg", m_hit_cfg);
uvm_config_db#(int)       ::set(this,"*.top_virtual_sequence.m_hit_seq"0,
                                "num_trans", 10000);
uvm_config_db#(real)      ::set(this,"*.top_virtual_sequence.m_hit_seq",
                                "is_hit_prob", 0.5);
uvm_config_db#(shortint)  ::set(this,"*.top_virtual_sequence.m_hit_seq",
                                "min_amplitude", 1);
uvm_config_db#(shortint)  ::set(this,"*.top_virtual_sequence.m_hit_seq",
                                "max_amplitude", 16);
```

Even in this case, the parameters of the specific sequences that have been described in the dedicated sections can be recognized. It should be at this point clear that what distinguishes one of the defined tests from the others is the way the environment is configured and which sequences are run. In detail:

- the *top_test1* has been used to run in parallel the basic *hit_sequence* and *trigger_sequence*, as already shown;
- in the *top_test2* instead the *hit_sequence* is substituted with the *hit_sequence_poisson*, taken from the same library;
- the *top_test3* is used to run, still in parallel to the *trigger_sequence*, the more complex *hit_sequence_cluster*, that will be presented in the next chapter.

5.3 User guide: scripts and UVM message facility

When approaching the use of UVM, the first step has been adopting a strategy for handling the design flow through scripts. Since the tool used for the code currently in repository has been Incisive Cadence, the design flow that is adopted is the already described MSIE design flow (see 3.2.1). As regards the actual files where the scripts are contained it has been followed the approach presented from a technical course [38] held at CERN. Two main files can be found in the *work* folder (previously shown in Figure 5.3):

1. the *Makefile* file, to which the user is most likely to refer, since it actually defines many variables that are then called by the compiling and running scripts that are contained in the subsequent file. The use of the *make* command is diffused in software development, above all under *Unix*, and it automatically builds executable programs and libraries from source code by reading files called *makefile*s which specify how to derive the target program. This means that from the terminal one can run the command *make* followed by the name of the specific script. In such a “higher level” file one can find:

    - a variable that points to the top level module of the project;
    - a pointer to all the source code related to the DUT;
    - a variable that points to all the packages of the different environments contained in the framework, included the top level one;
    - a pointer to the package that holds all the defined tests;
    - a variable that gathers the paths to the directories that need to be included when compiling the verification environment source code;
    - one for doing the same when compiling the DUT source code;
    - a variable that groups all the extra options that the user wants to specify to run the simulation (e.g. debug access to the signals, seed, verbosity level of messages, single/multi core,...);
    - a variable that holds the argument of the running command that specifies the test to be run: the user should modify such a variable for changing the specific test being run;

2. the *Makefile.defs* file, that the user is in principle not supposed to modify, since it actually contains scripts that simply refer to the previously listed variables. In details, here it follows what is contained in such a file:

    - variables related to the UVM version and that point to the UVM library location;
    - variable for pointing to the installation path;
    - a script (to launch it type *make plib*) for compiling the project library, that creates the *plib* library, where all the source files related to the DUT are contained (and to which the following commands refer);
    - a script (to start it use *make run1*) for generating the primary snapshot with the top level module of the project (called *PixelChip_tb*);
    - a script (to launch it type *make run2*) for generating the secondary snapshot and running the simulation (here all the packages of the verification environment and in particular where also the specific test are compiled);
    - an additional script (*make rungui*) that replicates the functions of the previous one, but also opens the Graphical User Interface (GUI), instead of simply showing the simulation results in the shell. In this case to actually start the simulation it is necessary to launch the start of the simulation from the GUI;
    - a script (*make clean*) for cleaning all the results from a previous simulation.

In the work directory one can also find a *run.sh* executable shell script that can be directly used to run in sequence: *make clean*, *make plib*, *make run1* and *make run2*. It is also highlighted that during the execution of the three main commands output *.log* files are produced (i.e. *compile.log*, *irun1.log*, *irun2.log*). Moreover, in the *work* folder there is also an additional directory where the output text and data files can be stored at the end of a simulation and before launching a new one. In order to move such files to a dedicated subdirectory of *stored_output_files*, named from the simulation date and time, one can execute the shell file *mv_hit_map_out.sh*. This also causes the current date and time to be appended to each output file name. To conclude this section, it is worth it to provide some details on the UVM approach used to handle the verbosity level of messages coming out from the simulation. It has been mentioned that a variable can be defined in the *Makefile* to specify such a verbosity as argument of the running commands. Such messages are the ones printed in the shell and stored in the *irun2.log* log file. Using Verilog's *$display* for printing messages, does not allow nonintrusive filtering and control of them. Changing the verbosity on the command line does not require to recompile and re-elaborate the design and enables the engineer to obtain more or less detailed information from the simulation (and respectively slower or faster engine). UVM reporting services are built into all components and dedicated macro APIs are defined for them to make their use easier. As regards their severity, four main levels can be distinguished:

- *'uvm_info(string id, string message, int verbosity)*;
- *'uvm_warning(string id, string message)*;
- *'uvm_error(string id, string message)*;
- *'uvm_fatal(string id, string message).*

The verbosity level is an integer value to indicate the relative importance of the message, specified just for information messages. When using the message macros, verbosity is instead ignored for warnings, errors and fatal errors in order to prevent verbosity modifications from hiding bugs. Specific information messages are issued only if the value is smaller or equal to the current verbosity level. An enumerated type provides several standard verbosity levels like: UVM_NONE=0, UVM_LOW=100, UVM_MEDIUM=200, UVM_HIGH=300, UVM_FULL=400, UVM_DEBUG=500. At the time of writing, to keep good simulation performance, the chosen verbosity level is UVM_LOW. With it, just messages related to the checkings done in the analysis scoreboard (matches/mismatches) are shown in the log file. For collecting other kinds of information on the simulation the text and data files are used. Lower level messages, intended to be used at the very debugging level or at an intermediate one have also been defined. For details on the specific level on each message one can simply look for *'uvm_info* in the source file of the different verification components or even modify them/their verbosity level if needed.

Chapter 6

Hit generation with constrained distribution within the framework

The hit environment has already been presented in the previous chapter as regards its basic blocks and structure. The sequence of transactions sent to the DUT through the simple *hit_sequence* and *hit_sequence_poisson* are random with settable and already described parameters, but the way the pixel is hit is not much related to actual physics inputs. In those cases, indeed, the randomization of the hits sent to each pixel is done independently from the others (they just have common constraints). For the RD53 collaboration program it is anyway mostly required to develop a highly flexible pixel hit generator that can emulate real pixel hits in phase 2 pixel detectors. Such a pixel hit generator should be capable of emulating pixel hits on a large number of pixels with appropriate correlations in order to be used to drive the critical optimization of a pixel ASIC design. The development of such a generator is subject of this work. It has to be anyway clearly highlighted that, even if it aims to reproduce physics input, such a hit generator is not meant to substitute inputs from Montecarlo or sensor simulations. It is just supposed to be a simple and easy-to-use tool (within the framework itself) that provides the flexibility of driving both realistic and extreme inputs to the chip, depending on the specific interest of the user that launches a test. Moreover, the hit generator should be made able to generate pixel hits based on detector Monte Carlo or sensor simulations, when such data will be made available. A collaboration has started on this with the LCD community.

6.1 Classes of hits

In order to develop a model for the hit generator easy to be simulated and not too sophisticated but able to run within the SystemVerilog UVM framework itself, some classes of inputs of interest have been identified. For each of them some basic parameters have also been defined, still aiming to obtain a simplified (but valuable as an additional tool for verification) model of physics inputs. It is herein highlighted that when mentioning a "hit" in this context we refer to a single pixel being hit:

- at a certain time (at the time of writing the generation is all synchronous and it is given through a BX cycle number, even if the possibility of adding an additional delay to it has been partially evaluated, just at the generation level)
- a certain amplitude, so far expressed as a TOT value (and that may be in the future modified into a charge value when developing more detailed and realistic models of the analog front end of the DUT)

The types of hits have been classified on the base of expected physics at LHC detectors and such information on particle generated after the proton-antiproton collision has been translated on how they are seen when detected from the ROC. The identified classes are herein listed and will be further described in the dedicated subsections:

1. single tracks, in general associated to energetic particle crossing the sensor at a certain angle;
2. loopers, soft charged particle that in the solenoidal magnetic field become curling tracks [39];
3. jets, collimated bunches of final state partons and hadrons coming from hard interactions [39];
4. monsters, not really a special phenomenon for physics, but a very extreme input to be used by engineers for extensive verification of the design;
5. noise hits, phenomena not directly associated with tracks, intended to add background noise.

In order to emulate such interactions using a geometrical model for the particle hitting the sensor, some generic parameters concerning the latter have been identified and can be set by the user. The main ones are:

- pixel pitch (intended as the dimension of the pixel in the same direction of the tracks, i.e. Z). For a square pixel size it is simply set to the edge value;
- pixel chip size of the full matrix;
- pixel chip area (specified separately, but should be calculated from the pixel size and total number of pixels in the chip);
- sensor thickness.

As regards the data type that has been used for them, a *real* has been used even where *int* would have been sufficient. This has been done because such values need to be used for performing mathematical calculations. When even only one operand is an *int*, in SystemVerilog the operation is evaluated as for integers. This can lead to undesired rounding off or, in particularly unlucky cases, to complete misleading results. For the same reason, it has been also avoided to perform calculations in the *ClassDefines.sv* file, since when dealing with compiler directives it is not clear with kind of format is being used for the operators. The assumption done on such parameters in order to obtain the graphical results shown in this section are based on values that are at the time of writing reasonable options for next generation sensors and ROCs, i.e.:

- pixel pitch equal to 50 μm (assuming a pixel size of 50x50 μm^2, as presented in subsection 1.2.1);
- pixel chip size equal to 512x512 pixels (leading to bigger ROC compared with previous generations);
- pixel chip area equal to 6.5536 cm^2;
- sensor thickness of 100 μm, a small value that seems to be at the limits of technological feasibility at the time.

Most of these parameters can be specified from the test and/or from *ClassDefines.sv* file, located in the *define* folder. As regards the test to choose to obtain the environment configuration presented in this chapter, the user should refer to *top_test3* and modify it according to needs. The setting of the various parameters is done both from configuring the UVM hit configuration object and from the specific *hit_sequence_cluster* used for generating transactions with the different classes of hits. Before going into details, it has to be highlighted that a basic distinction exists between the *top_test3* and the *top_test1-2*. While the latter simply generate inputs for the pixels of the actual ROC being simulated (whose dimension is so far quite small, correspondent to just one pixel region), the first is capable of generating inputs for a full matrix with the final expected pixel chip size. This means that so far just a subset of the generated inputs are driven to the actual (smaller) DUT being simulated. In order to keep the two ROC dimensions separate, a specific define has been used in *ClassDefines.sv* for the entire ROC size (i.e. PIXEL_CHIP_Z_FULL and PIXEL_CHIP_PHI_FULL) referring to the cylindrical coordinate space inside the detector. When configuring the environment through the tests, it has been chosen to use an enumerated parameter to decide whether to simulate the whole matrix (FULL_PIXEL_MATRIX) at the generation level or just the matrix correspondent to the actual DUT under simulation (CURRENT_DUT_PIXEL_MATRIX). Such a parameter can be set in the hit configuration object and clearly affects the behaviour of the hit master driver. Each of the defined tests has been designed for one of the two cases, as already said. Warnings are issued if the opposite choice is done. As concerns the hit configuration object that is created in the tests (that has not been shown in subsection 5.2.1), it is displayed in Figure 6.1. Many parameters can be therein noticed, starting from the very generic ones (name, definition of active or passive master agent, definition of the number of idle cycles at the beginning of the simulation before the driver starts sending hits to the DUT), going to more specific ones. In particular:

- *gen_matrix_size* is the enumerated type that chooses between the FULL_PIXEL_MATRIX or CURRENT_DUT_PIXEL_MATRIX simulation at the generation level;
- in case a FULL_PIXEL_MATRIX is being simulated with a smaller DUT, *submatrix_shift_z* and *submatrix_shift_phi* enable the user to define where to collocate the simulated chip into the bigger matrix;

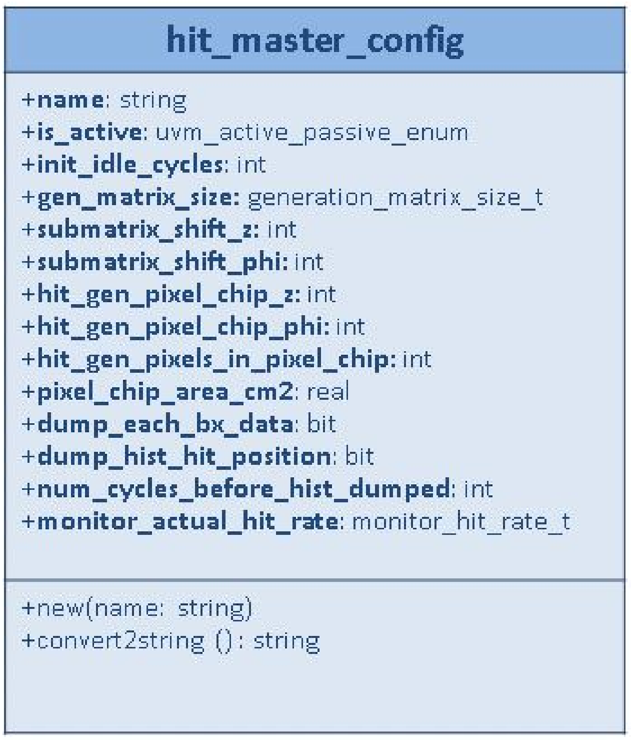

hit_master_config
+name: string
+is_active: uvm_active_passive_enum
+init_idle_cycles: int
+gen_matrix_size: generation_matrix_size_t
+submatrix_shift_z: int
+submatrix_shift_phi: int
+hit_gen_pixel_chip_z: int
+hit_gen_pixel_chip_phi: int
+hit_gen_pixels_in_pixel_chip: int
+pixel_chip_area_cm2: real
+dump_each_bx_data: bit
+dump_hist_hit_position: bit
+num_cycles_before_hist_dumped: int
+monitor_actual_hit_rate: monitor_hit_rate_t
+new(name: string)
+convert2string (): string

FIGURE 6.1: UML class diagram of the *hit_master_config* object.

- the parameters on the pixel matrix dimension (i.e. *hit_gen_pixel_chip_z*, *hit_gen_pixel_chip_phi*, *hit_gen_pixels_in_pixel_chip*) at the generation level are automatically set (when the configuration object is built) depending on the choice done on the previous parameter;
- the pixel chip area;
- the last four parameters are used for guiding the monitoring done for checking generated inputs and they will be better described in section 6.2.

. All the previously listed parameters are generic and used to configure the environment under any circumstances: they do not depend on the specific classes of hits being generated. It is instead the *hit_sequence_cluster* that takes care of generating a sequence of transactions that emulate such physics hits. This sequence is therefore the one run in the *hit_master_sequencer*, as defined in the top level sequence launched in the *top_test3*. It is also the one whose configuration enables the user to stimulate the ROC with different inputs, characterized by different parameters (e.g. rates, track angle, etc..).

6.1.1 Tracks

A simple representation of the track that a particle goes through after having been generated in the interaction point (or other secondary vertexes) is sketched in Figure 6.2. Each single track hits a variable number of pixels, generating a so-called cluster.

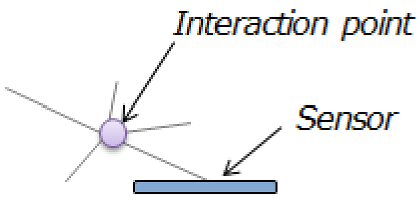


FIGURE 6.2: Sketch of the track of a particle crossing the sensor.

The dimension of such a cluster depends on the angle of the track itself with respect to the sensor surface. The less the particle path is close to being perpendicular to the sensor, the longer the cluster, as it can be noticed in Figure 6.3. In the Figure, the different types of small arrows denote the polarity of the charge carriers (the filled ones are electrons, the open ones holes) while the big arrows indicate the particle tracks. Such an angle is ultimately dependent on the position of the sensor itself in the whole

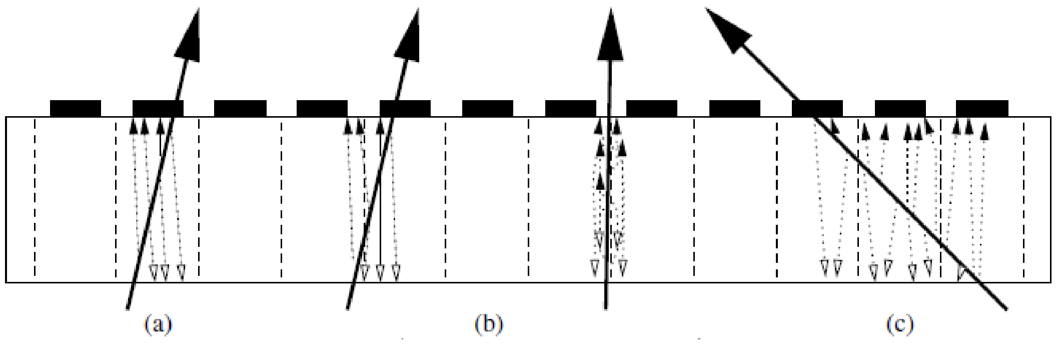


FIGURE 6.3: Formation of clusters of different size in silicon detectors: (a) size 1, (b) size 2, (c) size 3 ([1]).

pixel detector or on how it is tilted. The phenomenon of having multiple pixels hit from a single track is generally referred as charge sharing. Apart from the pure geometrical considerations, additional aspects that can influence the cluster dimension are ([1]):

- the presence of a magnetic field that makes the signal charge deflect by the Lorentz angle can increase the charge sharing as shown in Figure 6.4 for both perpendicular and tilted tracks;

- charge sharing can also happen in the perpendicular direction when perpendicular tracks hit the corner between multiple pixels (such a phenomenon can also cause the charge to be too much distributed and uncomfortably small);
- Operating in partial depletion, that can be necessary after irradiation, and trapping influence the charge sharing, since they decrease the collected signal charge;
- higher operation voltages needed to operate after irradiation, make the Lorenz angle decrease, still influencing the charge sharing.

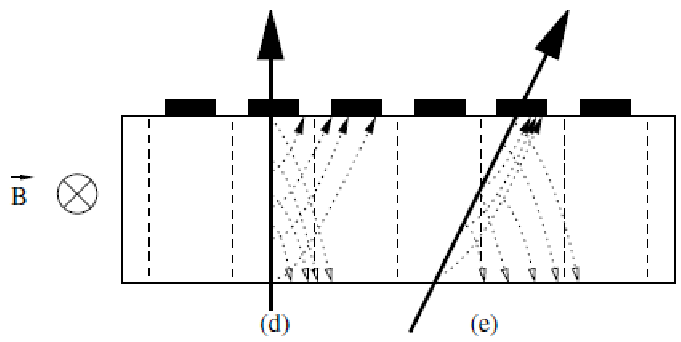


FIGURE 6.4: Formation of clusters in a silicon detector in presence of a magnetic field in two cases : (d) with the track perpendicular to the sensor surface and (e) tilted by the Lorentz angle ([1]).

It has been purpose of this work to model such (more complex) phenomena defining a subset of simple and easy parameters. Such parameters are:

a) track rate per cm^2 (can also be set to 0);

b) the track angle;

c) the charge sharing level;

d) the percentage of pixels surrounding the central cluster that are hit (used only if charge sharing is present).

As concerns the track angle, it is important to highlight the reference system used. It has been associated a a 90° angle to tracks perpendicular to the sensor surface. This value is used to geometrically evaluate the cluster length, as shown in Figure 6.5. The dimension of such a central cluster can therefore be from 1x1 to 1xN. As far as charge sharing is concerned, it has been described as many aspects (beside geometrical considerations)

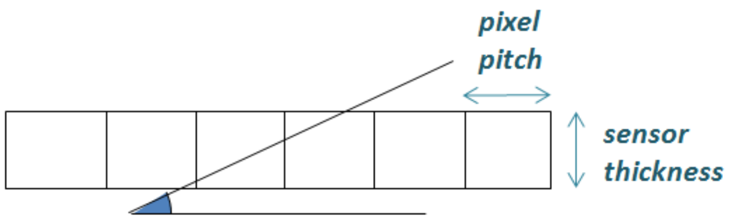


FIGURE 6.5: Geometrical representation of a track crossing the sensor with a certain angle.

can affect it. Trying to model all of them would clearly require a too sophisticated model for a SystemVerilog simulation framework (such inputs will be provided from actual sensor simulations). For this work, it has been chosen to represent the variability on the number pixels that are hit surrounding the central cluster with two parameters. A sketch of such a cluster is shown in Figure 6.6. In particular, the charge sharing level (either LEV_ZERO or LEV_ONE) defines whether or not to hit such adjacent pixels and, if so, their percentage. Even if such a model is clearly simplified, it makes it possible to obtain not only clusters of fixed dimension but with more realistic variable shape. It also provides some simple control knobs to generate them that can be set into the *hit_sequence_cluster* through the *top_test3*. Also some other generic parameters

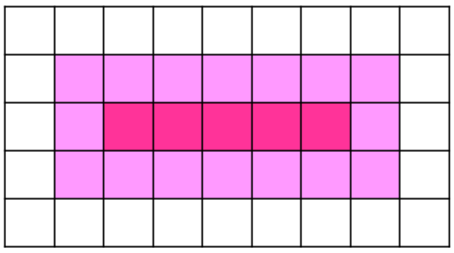


FIGURE 6.6: Sketch of a cluster with a central part and some adjacent pixels being hit as effect of charge sharing.

previously introduced, i.e. number of BX cycles to run, sensor thickness, pixel pitch and maximum and minimum amplitude of the incoming hits, can be specified in the same way. As concerns the algorithm used to generate such tracks, the following steps are performed:

- depending on the rate, for each BX cycle it is decided whether or not to generate tracks and, if so, how many clusters;

- for each cluster the starting address is randomly chosen in the matrix and an amplitude is randomized within the allowed range;
- the cluster length is evaluated from the pixel pitch, track angle and sensor thickness;
- the task *generate_single_track* is called in order to actually generate each cluster and the starting address, the amplitude, the length, as well as the details on the charge sharing parameters are passed to it. At the time of writing the same amplitude is used for the central cluster, while a slightly lower value (amplitude - 1) is used for the neighbours. Generated clusters are written (and eventually summed, if overlapping) in the *hit_trans* transaction that will be sent at the end from the sequencer to the driver.

In Figure 6.7 an example of a matrix where clusters have been generated in one BX cycle is shown. Details on the parameters set for the tracks are also provided in the caption. Such graphical results have been produced importing an output *.dat* file (that will be introduced in Section 6.2) in MATLAB. More detailed examples on the generated clusters are provided in order to show the results produced by the model used for the charge sharing. In Figure 6.8, two clusters are shown without charge sharing and with two different track angles, respectively 90° and 9°, that give rise to square and elongated clusters. The same types of clusters, both square and elongated, are also shown in Figures 6.9 and 6.10 when charge sharing is turned on (LEV_ONE) and with an increasing percentage of surrounding pixels being hit.

6.1.2 Loopers

A simple representation of a low energy particle going through a curling track is sketched in Figure 6.11. It can be noticed as such particles are likely to hit the sensor in a direction that is close to being perpendicular to its surface. For this reason, in this work it has been chosen to model them as single tracks crossing the sensor at 90°, as shown in Figure 6.12. This means that the dimension of the central cluster is fixed to 1x1. It is anyway not excluded that in the future a range of angles might be used, instead of a single value. Therefore, the parameter that has been defined for the generation of loopers is just the rate normalized by the area (*looper_rate_per_cm2*), that can also

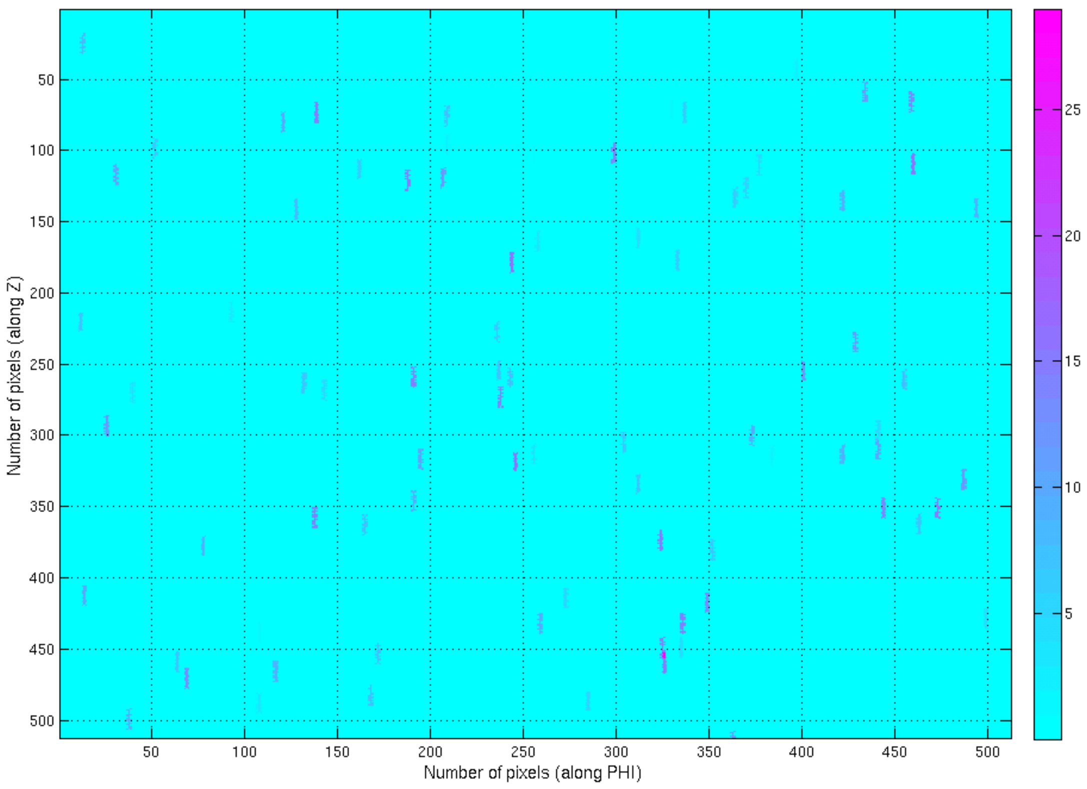


FIGURE 6.7: Example of tracks generated in one BX cycle. Track rate per cm^2= 500 MHz, track angle= 9°, charge sharing = LEV_ONE, percentage of surrounding pixel hit = 50%.

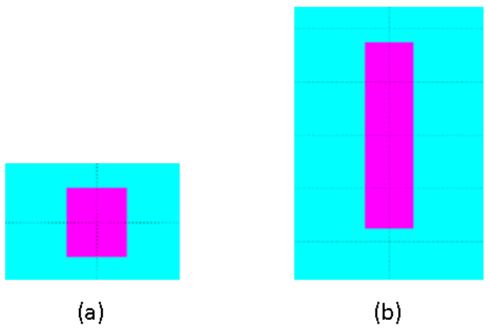


FIGURE 6.8: Example of square and elongated clusters generated with different track angles, i.e. 90° (a) and 9° (b) without charge sharing with adjacent pixels.

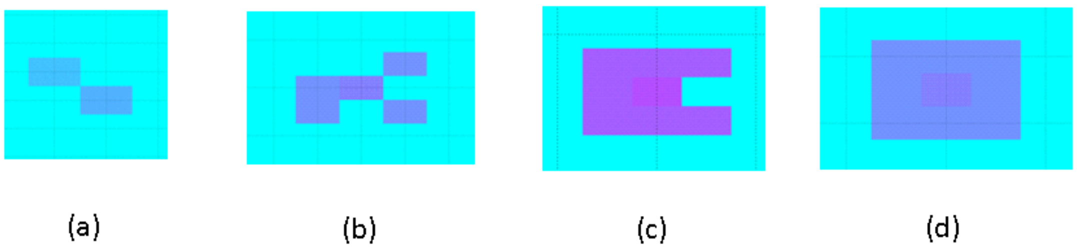


FIGURE 6.9: Examples of square clusters with charge sharing. The % hit pixels are: 10% (a), 50% (b), 80% (c), 100% (d).

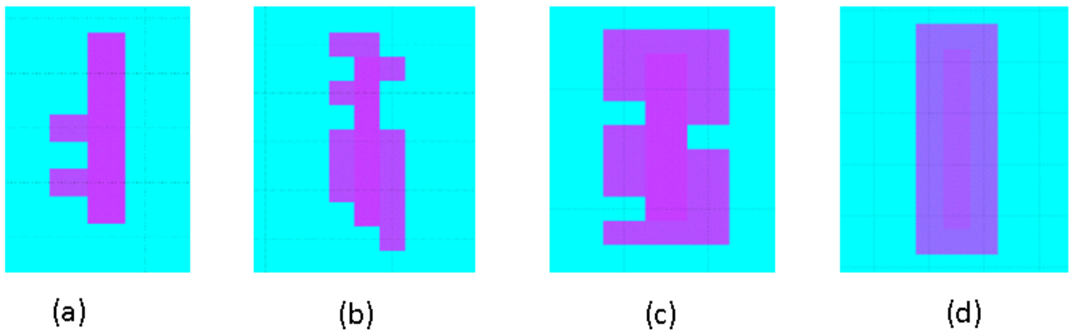


FIGURE 6.10: Examples of elongated clusters with charge sharing. The % hit pixels are: 10% (a), 50% (b), 80% (c), 100% (d).

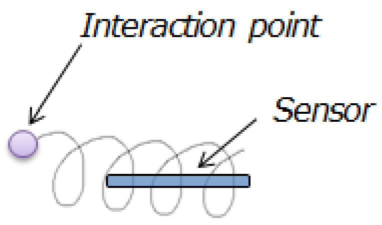


FIGURE 6.11: Sketch of a looper crossing the sensor surface.

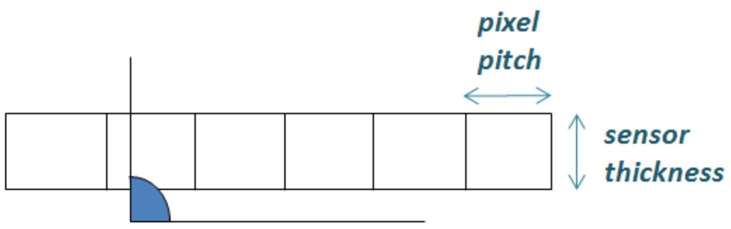


FIGURE 6.12: Geometrical representation of a track crossing the sensor with a certain angle.

be set to 0. Furthermore, the settings done for the charge sharing of the tracks are also used for the generation of loopers. In practice indeed, the generation of loopers is performed using the same approach as for the tracks, and in particular for each looper the *generate_single_track* task is called specifying cluster length equal to 1. They are all summed in the same transaction that is meant to be sent to the driver. An example of a whole matrix to which only loopers are sent is shown in Figure 6.13. Also in this case the charge sharing has been turned on with 50% of surrounding pixels hit. Since they

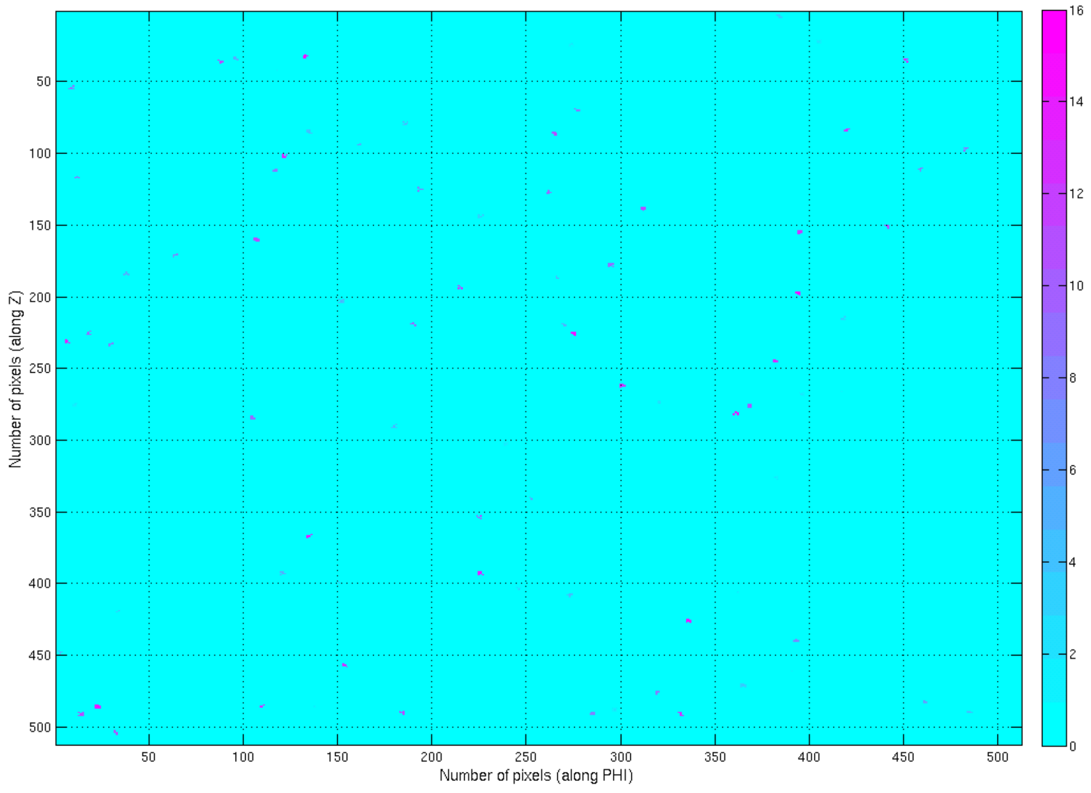


FIGURE 6.13: Example of (only) loopers generated in one BX cycle in the whole 512x512 matrix. Looper rate per cm^2 = 500 MHz, charge sharing = LEV_ONE, percentage of surrounding pixel hit = 50%.

are quite small clusters in a big 512x512 matrix, also a zoom of a subset is reported in Figure 6.14.

6.1.3 Jets

A jet can be seen as a narrow cone of particles concentrated in a given area, as shown in Figure 6.15. Such a phenomenon has been modeled, per each BX, as a combination of multiple and close tracks that hit the sensor, giving rise to a group of clusters localized

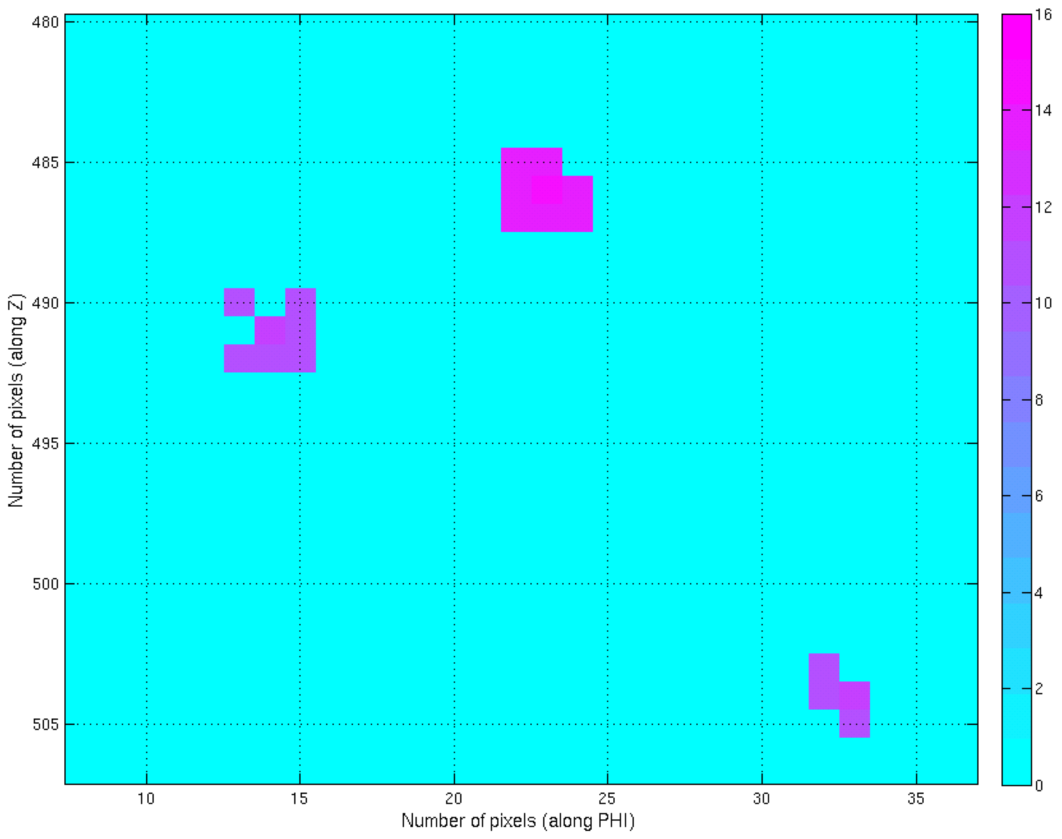


FIGURE 6.14: Zoom on few loopers generated in one BX cycle. Charge sharing = LEV_ONE, percentage of surrounding pixel hit = 50%.

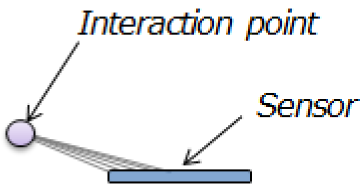


FIGURE 6.15: Sketch a jet (bunch of particles) crossing the sensor.

in a certain area of the matrix. In order to achieve such a model, the definition of a set of additional parameters is required, i.e.:

a) jet rate per cm^2 (can also be set to 0);

b) average number of tracks per jet (a poisson distribution with such an average has been used);

c) size of the jet area, given in number of pixels for each edge.

In order to generate a jet, the first step is to randomize the central address of the jet area, whose limits are specified. At that point each track in the jet is created using the routine for single track generation. The only peculiarity is that the starting address of each cluster (passed as an argument to the task) is randomized in the obtained jet area. The parameters set for the track angle (related to the position of the sensor in the detector) and for the charge sharing are also used. An example of a whole matrix where only jets have been generated is shown in Figure 6.16. The detailed parameters set are listed in the caption. In order to visualize it more in detail, a zoom on a single

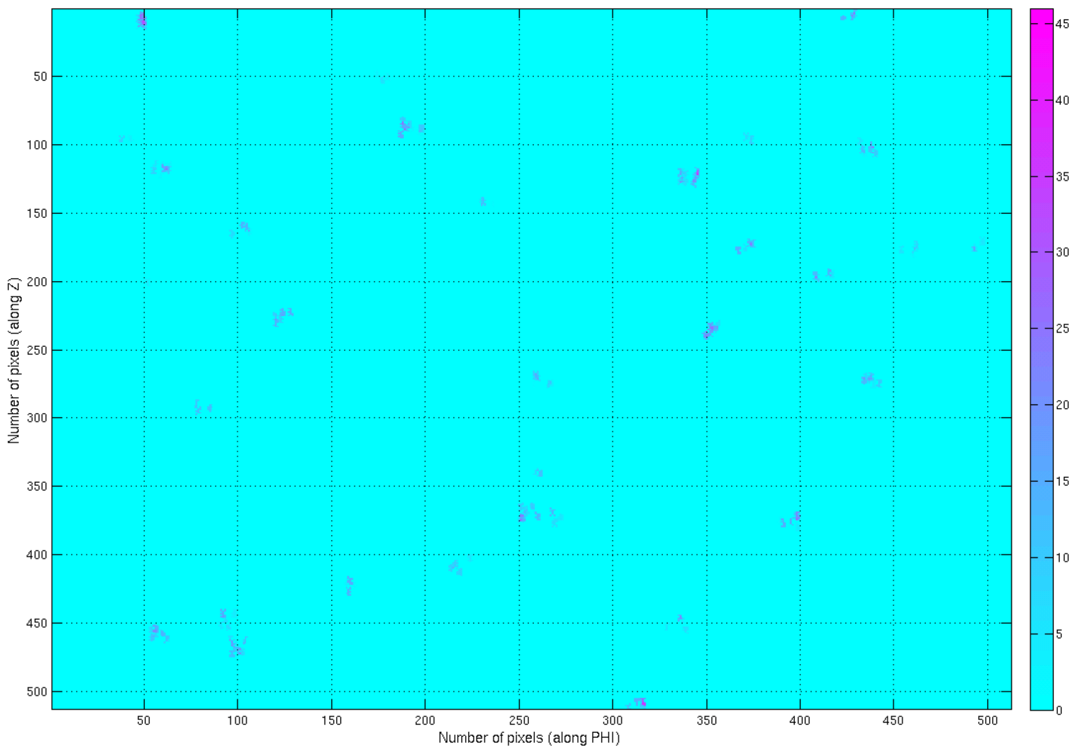


FIGURE 6.16: Example of tracks generated in one BX cycle. Jet rate per cm^2= 200 MHz, average number of tracks per jet = 10, jet area = 10x10 pixels, track angle= 30°, charge sharing = LEV_ONE, percentage of surrounding pixel hit = 50%.

jet obtained from the previous simulation is presented in Figure 6.17.

6.1.4 Monsters

As introduced at the beginning of the section, monsters are not a special phenomenon expected from physics. The possibility of sending extreme inputs to the ROC is anyway of interest for designers. It is indeed important to be sure that it is able to recover under every kind of stimuli. In the context of ROC design, it has become common to refer to

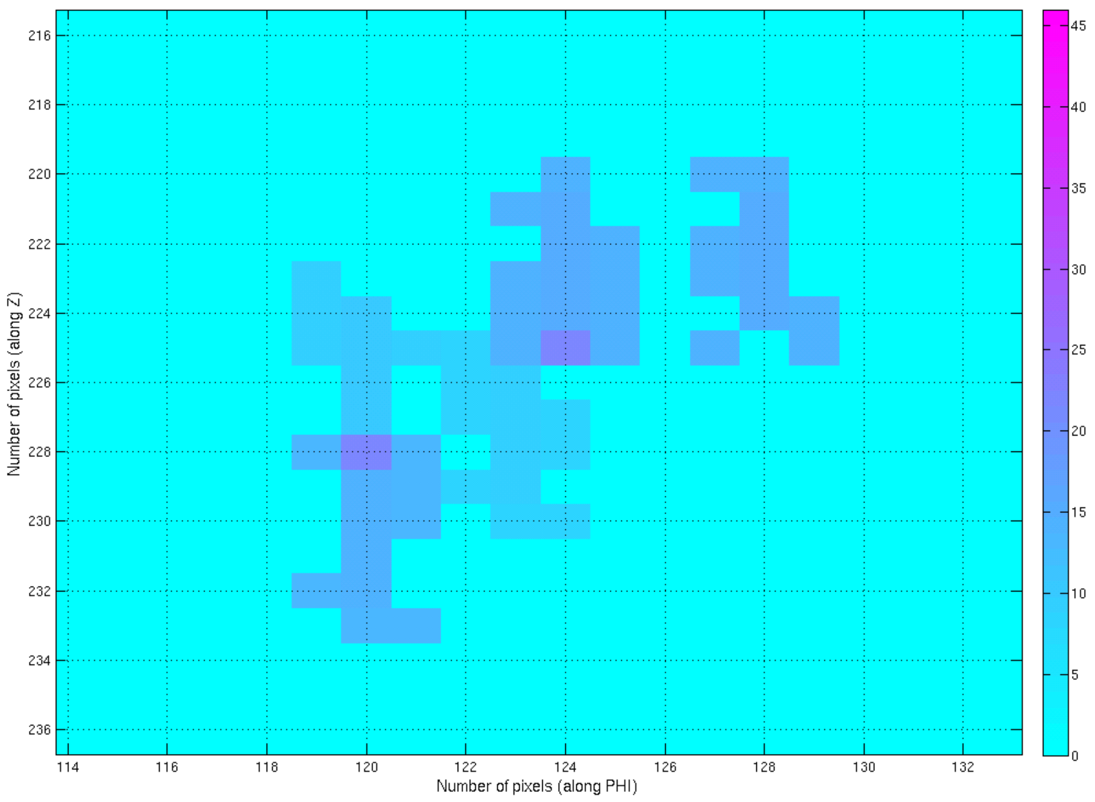


FIGURE 6.17: Zoom on a single jet. Parameters: average number of tracks per jet = 10, jet area = 10x10 pixels, track angle= 30°, charge sharing = LEV_ONE, percentage of surrounding pixel hit = 50%.

extreme inputs (that stimulate a whole row/column of the chip) as "monsters". They can be seen as tracks very close to be parallel to the sensor surface , as shown in Figure 6.18.For this reason they have been considered as a separate class of hits. Some basic

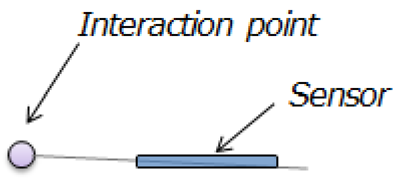


FIGURE 6.18: Sketch a track coming almost parallel to the sensor (so-called monster).

parameters have been defined to drive them:

a) monster rate per cm^2 (can also be set to 0);

b) monster direction (either PHI or Z).

Also in this case the parameter set for the charge sharing is used. Examples of single monsters generated in the usual 512x512 ROC matrix in the the two possible directions are shown in Figure 6.19 and 6.20.

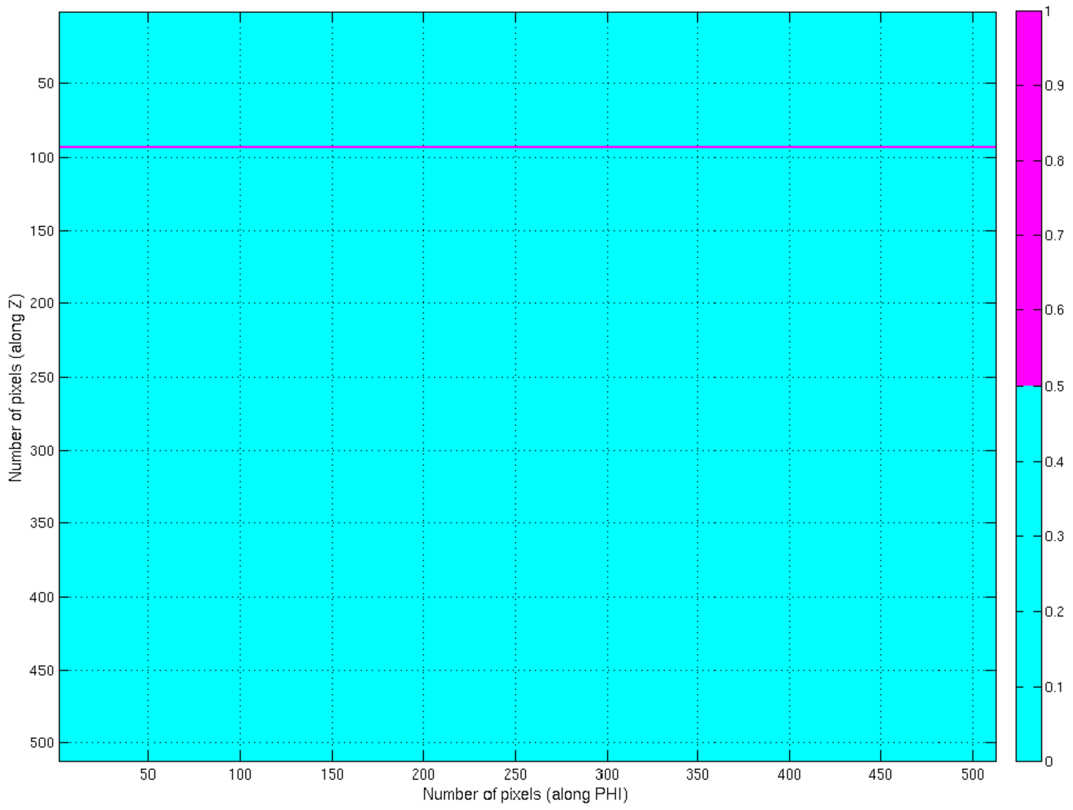


FIGURE 6.19: Example of a BX cycle in with only one monster has been generated in the PHI direction.

6.1.5 Noise Hits

In order to provide the possibility of also generating another class of hits, that are “unexpected”, in the sense that they have no correlation with tracks, a control knob for generating noise hits has also been provided. They have been simply modeled as 1x1 clusters randomly hit. The only parameter that can be set is the noise hit rate normalized by the ROC area, whose value can also be zero. Such minimum size clusters can be seen in 6.21 in the whole matrix, or more clearly in a subset of it that has been highlighted in Figure 6.22.

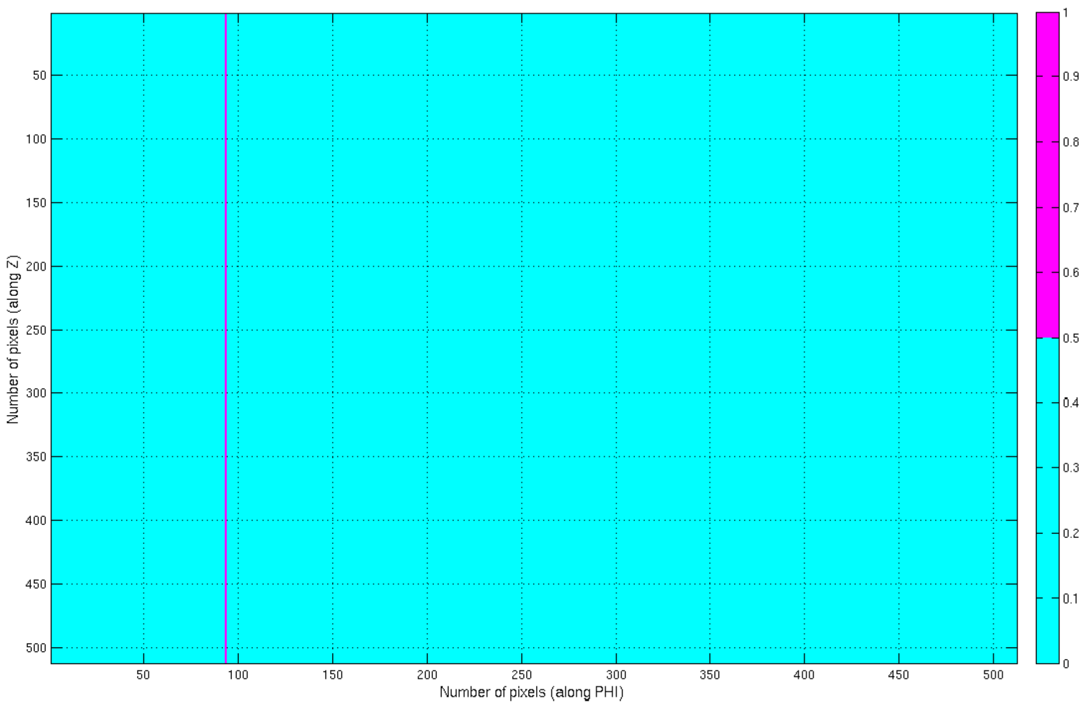
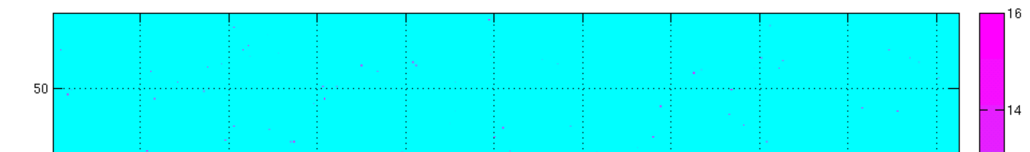


FIGURE 6.20: Example of a BX cycle in with just one monster has been generated in the Z direction.



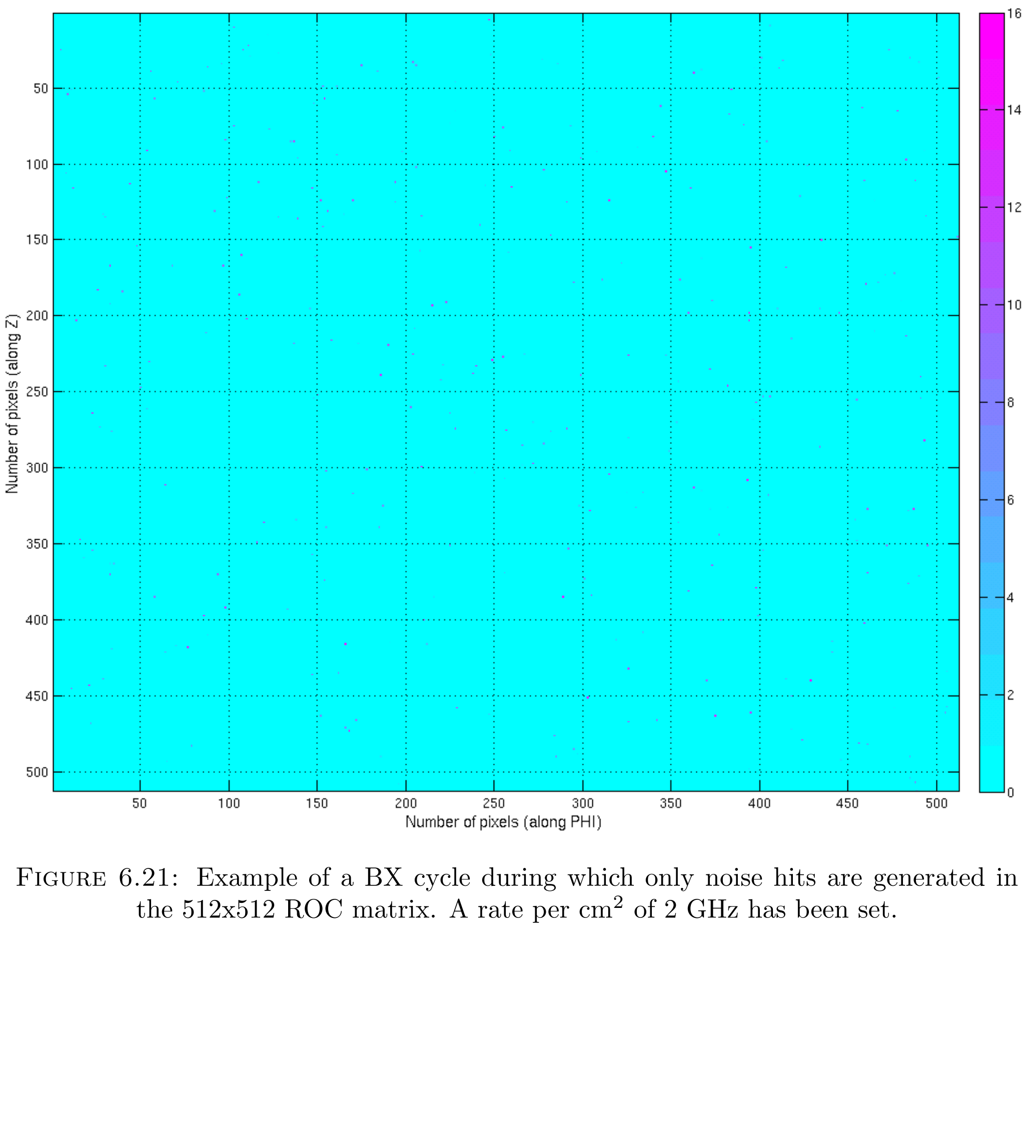


FIGURE 6.21: Example of a BX cycle during which only noise hits are generated in the 512x512 ROC matrix. A rate per cm^2 of 2 GHz has been set.

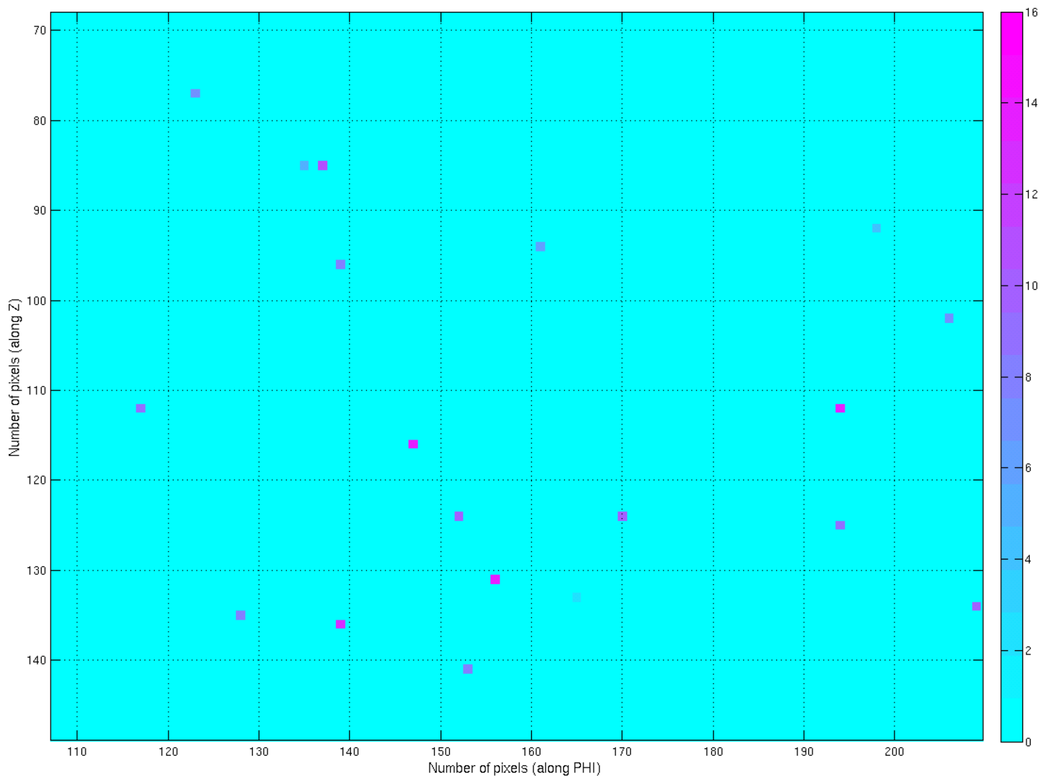


FIGURE 6.22: Zoom on generated noise hits. 1x1 clusters are clearly visible.

6.2 Monitoring and statistics collection

Some additional work was needed in order to obtain a straightforward way of checking the actually produced stimuli, necessary to guide the development of the hit generator, to debug it and (last but not least) to obtain graphical representation to be shown (as the ones presented in the previous and in this section). For this purpose some different approaches have been followed:

1. it has been decided to write (per each BX) to a *.dat* file all generated hits specifying the position in the matrix and the amplitude (mainly for debugging and also for obtaining graphics on clusters);
2. some statistical analysis has been partially performed within the framework (i.e. histograms on generated hits are accumulated):
   - they can be then printed every m BX cycles into a *.dat* output file for further elaboration or graphical visualization;
   - they can be used in the framework itself to evaluate observed hit rate all over the simulation (and finally print it to a text file);

3. a third possible approach would be to develop a direct interface between SystemVerilog and C/C++ to directly call C++ routines from the SV framework and to immediately obtain graphical visualization without the need to import output files. This solution has not been taken into account in this thesis.

It is highlighted that, in order to keep track in each output file of the type of configuration set by the user for the simulation, many parameters are printed on top of the files before writing there the simulation results. In particular on each of the files that are going to be described, one can always check settings done on:

- hit configuration object parameters (generation matrix size, pixel chip area, etc..);
- hit sequence parameters (number of BXs, sensor thickness, track angle, charge sharing level, track/looper/jet/monster/noise hit rates per cm^2, etc..).

Performing monitoring, statistics collection and above all writing to files, can clearly affect simulation time. For this reason it is important to assure that the environment is capable of deactivating such operations if not needed. It has been therefore decided to:

- mainly use a dedicated component for taking care of such operations and, if not needed, to even avoid building it;
- define switches in the hit configuration object (see Figure 6.1) to enable the user to activate/deactivate such features.

A block diagram of the hit environment changes when at least one of such switches is activated, since it causes the *hit_subscriber* to be built, as shown in 6.23. In this block diagram it is assumed that just information on total observed hit rate is being collected. Details on the level of monitoring will be presented below. It can be therein noticed as the subscribers receives from the sequencer the same transactions that are sent to the driver, so it is capable of performing statistical analysis on generated stimuli. Moreover, it is capable of printing each of them into the *hit_map.dat* file and/or to accumulate info on hit pixels in an array and write the histograms in the *hit_position_histogram.dat* file. Such information can also be used to calculate the observed hit rate per cm^2 during the whole simulation. This is the kind of statistical analysis which can be printed in *statistics_file.txt*.

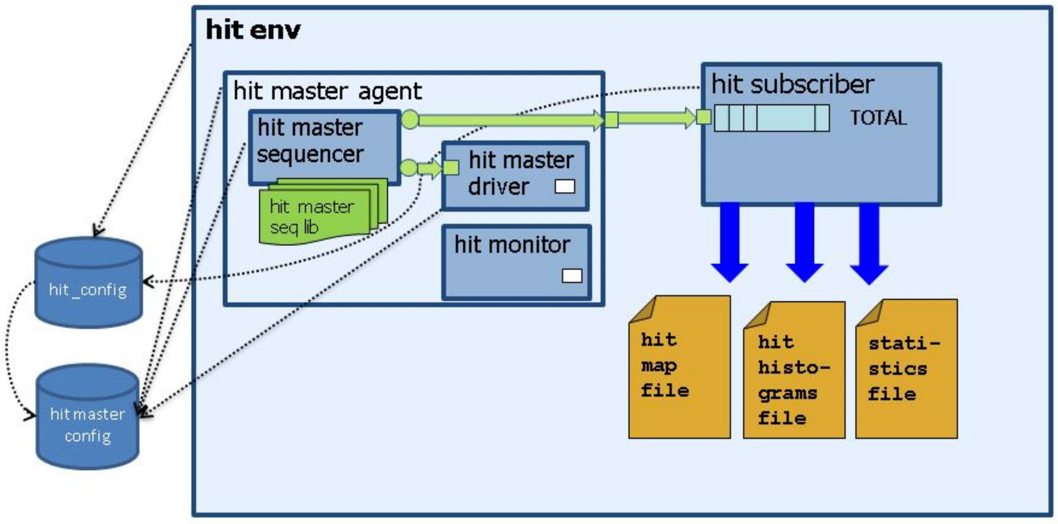


FIGURE 6.23: Hit environment block diagram with the hit subscriber built (and hit monitoring only on total hits).

6.2.1 Graphics by MATLAB

In the context of this work, the file containing data have been imported in MATLAB for graphical visualization. The results obtained for the cluster shapes and distribution in the matrix have been extensively presented in the previous section in different situations. As regards the histograms on the positions of the hits, accumulated during a certain settable number of BX cycles, they have also played an important role when debugging the developed tool. Since the way pixel coordinates (where tracks, loopers, jets, monsters and noise hits are sent) are chosen is random, it is expected to obtain uniformly distributed histograms all over the matrix. Such a checking has been constantly done while developing the different classes of hits. An example of the bi-dimensional histogram is shown in Figure 6.24, represented as a 2D matrix. It can be noticed that the values are very low, since they are normalized to the total number of hit pixels during the considered period. The sum of all the values is also performed in the framework, in order to check that it equals 1. Otherwise an error is issued.

6.2.2 Monitoring actual hit rates per cm^2

It has been mentioned how accumulated histograms are also used to calculate (and output to a dedicated file) information on observed hit rates. Some more details on

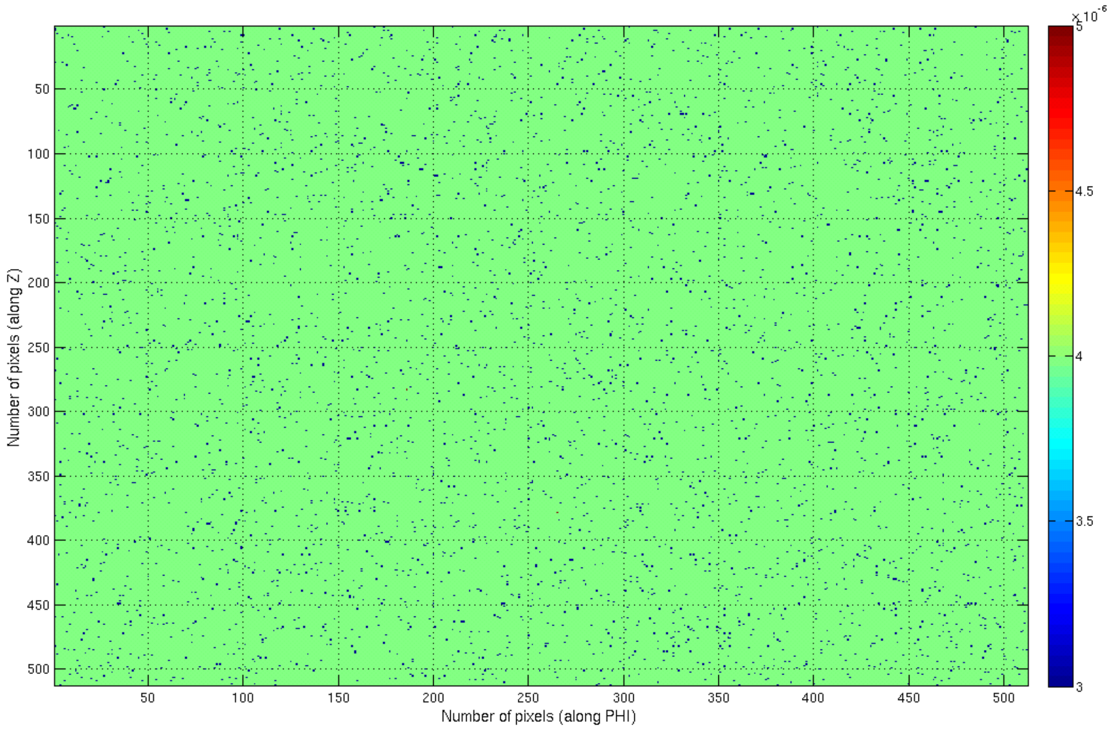


FIGURE 6.24: Histogram accumulated in 500000 BX cycles while generating only tracks.

this feature are presented in this section. First, it is reminded that so far a difference exists between the big pixel matrix considered at the generation level and the subset of it being actually linked to the ROC. When simulating such a small DUT, one also wants to check that the hit rate observed in such a subset of pixels is as expected. For this reason, statistics are printed for both:

- the whole matrix used at the generation level;
- the DUT matrix (that can be a subset, depending on configuration).

To provide with some additional control knobs for the hit monitoring, it has been also chosen to define different levels of details for it. A field of the configuration object, defined as an enumerated type, is used to set it. The possible options are the following:

- NONE, which means that the *statistics.txt* file is not generated at all;
- ONLY_TOTAL, that causes the file to contain only info on observed total hit rate per cm^2 (possibly originating from different sources);

- DETAILED, with which one can obtain both information on total hit rate per cm^2 and classified hit rate per cm^2, depending on the source (i.e. tracks, loopers, jets, monsters, noise hits). It is reasonable to use such a level only when running the *hit_sequence_cluster*, i.e. with the *top_test3*. For the other tests, a warning is generated if such a level is chosen. It would anyway simply cause zero values to be obtained for each specific class of hits.

In order to obtain a detailed monitoring of different sources, sending just a unique *hit_trans* (where all the contributions are already summed) to the *hit_subscriber* is clearly not sufficient. Indeed, after the transaction has been written, there is no more any way of distinguishing to which class of hits it belongs. For this reason, in this particular case a new structure for the hit environment has been defined. It is presented in Figure 6.25.

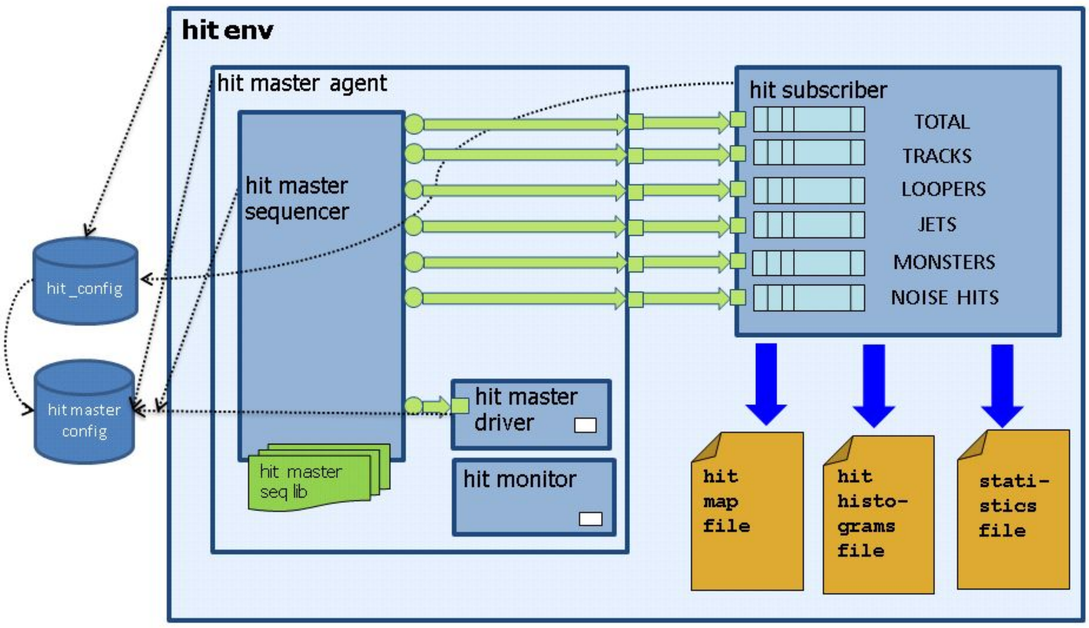
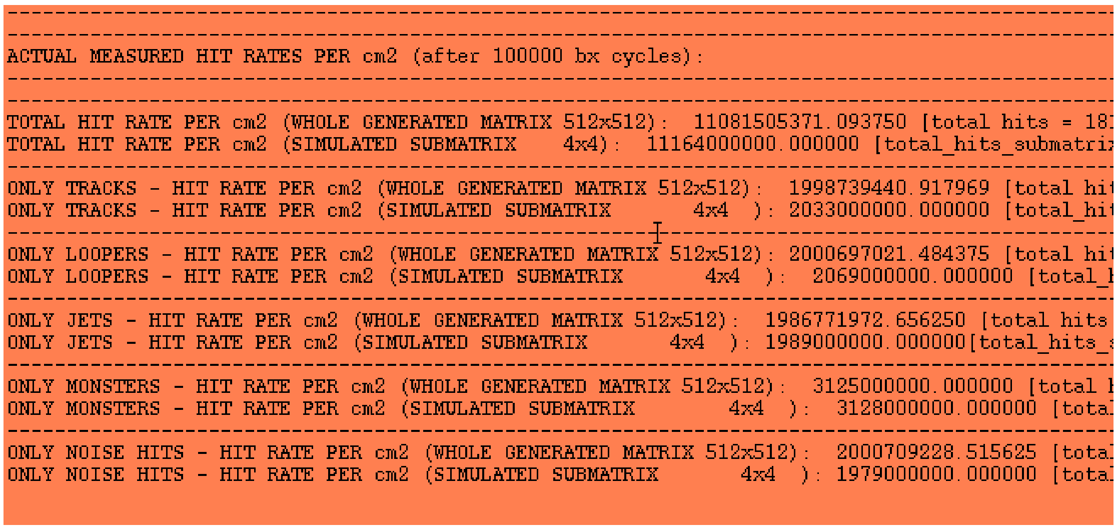


FIGURE 6.25: Hit environment block diagram with the hit subscriber built and detailed level of hit monitoring provided.

The *hit_sequence_cluster* run by the sequencer not only writes into the final transaction to be sent to the driver (that contains all the generated hits) but also creates five more transactions per each BX cycles, each one just hosting one class. In this way it is possible to the sequencer to send such separate information to the subscriber. As it can be seen in the block diagram, it accumulates a different histogram for each class of hits plus.

This enables the *statistics_file.txt* to contain a classification of the hit rates per cm^2. An example of the obtained results is shown in Figure 6.26. The main settings of interest are listed in the caption. At a first glance it can be crosschecked as:



```
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
ACTUAL MEASURED HIT RATES PER cm2 (after 100000 bx cycles):
----------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------
TOTAL HIT RATE PER cm2 (WHOLE GENERATED MATRIX 512x512):  11081505371.093750 [total hits = 18
TOTAL HIT RATE PER cm2 (SIMULATED SUBMATRIX    4x4):  11164000000.000000 [total_hits_submatri
----------------------------------------------------------------------------------------------
ONLY TRACKS - HIT RATE PER cm2 (WHOLE GENERATED MATRIX 512x512):  1998739440.917969 [total hi
ONLY TRACKS - HIT RATE PER cm2 (SIMULATED SUBMATRIX       4x4  ): 2033000000.000000 [total_hi
----------------------------------------------------------------------------------------------
ONLY LOOPERS - HIT RATE PER cm2 (WHOLE GENERATED MATRIX 512x512): 2000697021.484375 [total hi
ONLY LOOPERS - HIT RATE PER cm2 (SIMULATED SUBMATRIX       4x4  ):  2069000000.000000 [total_
----------------------------------------------------------------------------------------------
ONLY JETS - HIT RATE PER cm2 (WHOLE GENERATED MATRIX 512x512):  1986771972.656250 [total hits
ONLY JETS - HIT RATE PER cm2 (SIMULATED SUBMATRIX       4x4  ): 1989000000.000000[total_hits_
----------------------------------------------------------------------------------------------
ONLY MONSTERS - HIT RATE PER cm2 (WHOLE GENERATED MATRIX 512x512):  3125000000.000000 [total 
ONLY MONSTERS - HIT RATE PER cm2 (SIMULATED SUBMATRIX        4x4  ):  3128000000.000000 [tota
----------------------------------------------------------------------------------------------
ONLY NOISE HITS - HIT RATE PER cm2 (WHOLE GENERATED MATRIX 512x512):  2000709228.515625 [tota
ONLY NOISE HITS - HIT RATE PER cm2 (SIMULATED SUBMATRIX       4x4  ): 1979000000.000000 [tota
```

FIGURE 6.26: Example results on detailed hit monitoring. Settings: track rate: 1 GHz/cm^2 (with 45° angle, causing 2 pixel clusters), looper rate: 2 GHz/cm^2, jet rate: 500 MHz/cm^2, 2 tracks per jet), monster rate:7 MHz/cm^2, noise hit rate: 2 GHz/cm^2, no charge sharing.

- the sum of the classified hit rates corresponds to the total monitored;
- the observed hit rate per cm^2 coming from tracks (close to 2 GHz/cm^2) corresponds to the expected one, since the track rate set is 1 GHz/cm^2 and each cluster is formed by two pixels;
- the observed hit rate per cm^2 coming from loopers (close to 2 GHz/cm^2) corresponds to the expected one, since loopers are 1x1 clusters and no charge sharing is present;
- the observed hit rate per cm^2 coming from jets corresponds to the expected one equal to 2 GHz/cm^2. It can be calculated multiplying the jet rate set (500 MHz/cm^2), by the average number of tracks per jet (2 clusters) and by the number of pixel of each cluster (two pixels);
- the observed monster rate is close to the expected one (that can be obtained multiplying the set rate by the number of pixels in a monster, i.e. 512);
- the observed hit rate per cm^2 coming from noise hits (close to 2 GHz/cm^2) corresponds to the expected one, since they are 1x1 clusters;

- the observed hit rates match pretty well between the whole matrix and the smaller simulated submatrix, showing once more a uniform distribution of generated hits.

In conclusion, it has anyway to be highlighted as such a model has been designed in order to be used with ROC matrices whose sizes are not too small. First of all, indeed, addresses are randomized in the existing matrix, so when very small ones are used at the generation level it does not make much sense to use very complex stimuli (like jets). On the other hand, when a very small ROC is used as a DUT (e.g. 2x2), the hit generator model can show some apparently unexpected results. In particular when generating long elongated tracks (longer than the DUT edge) it has been seen that the hit rate observed on the DUT sub-matrix is smaller than the one observed in the whole matrix. This can be understood since no long cluster actually happens to be contained in the smaller simulated ROC (meaning that for each cluster less pixels are hit with respect to the bigger matrix). Since it is not really of interest to use the model with so small ROCs, it has be chosen to avoid going into complicate modification to correct such a behaviour.

6.3 The configuration of the hit generator

Even if many concepts have been already presented in the previous sections, this one is meant to be a summary and above all a practical guide for correct configuration of the hit generator. The user basically needs to access two files, i.e. the *ClassDefines.sv* one (path: *trunk/source/define*) and the *top_test3* (path: *trunk/source/VerificationEnvironment/top/tests*). The idea for setting the test is the same as in subsection 5.2.6, where it has been presented how to set the trigger and analysis configuration objects. It is shown the code from *top_test3* to be modified to configure the hit configuration object.

```
m_hit_cfg.add_master("m_hit_agent",
                       UVM_ACTIVE,
                       ‘INIT_IDLE_CYCLES,
                       FULL_PIXEL_MATRIX,
                       (‘PIXEL_CHIP_Z_FULL/2),
                       (‘PIXEL_CHIP_PHI_FULL/2),
                       ‘PIXEL_CHIP_AREA_cm2,
                       0,
                       0,
                       999,
                       DETAILED
                       );
```

There one can recognize all the fields of the UML class diagram. In particular, in this example many defines are used to set the different parameters like the initial number of idle cycle of the driver and the pixel chip area. Moreover, it is chosen to simulate a full matrix at the generation level and it is defined to move the simulated DUT at the center of it. As regards the fields related to the hit monitoring: it is chosen not to dump each BX cycle (switches set to 0), to dump histograms into the output file every 999 BX cycles and to activate a detailed level of monitoring on the hit rates. Other important parameters are then set into the hit sequence:

```
// Common parameters for cluster generation
uvm_config_db#(hit_config)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "m_hit_cfg",m_hit_cfg);
uvm_config_db#(int) ::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "num_bx_cycles", 50000);
uvm_config_db#(int) ::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq,
                        "track_angle_deg", ‘TRACK_ANGLE_DEG);
uvm_config_db#(real)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "sensor_thickness", ‘SENSOR_THICKNESS_um);
uvm_config_db#(real)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "pixel_pitch", ‘PIXEL_DIM_ANGLE_DIRECTION_um);
uvm_config_db#(csharing_level_t)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "csharing_level", LEV_ZERO);
uvm_config_db#(int) ::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "percentage_hit_pix_env",
                        `PERCENTAGE_HIT_PIXEL_ENVELOPE);
uvm_config_db#(shortint)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "min_amplitude, 1);
uvm_config_db#(shortint)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "max_amplitude", 16);
// 1. Single tracks
uvm_config_db#(real)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "track_rate_per_cm2", `TRACK_RATE_PER_CM2);
// 2. Loopers
uvm_config_db#(real)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
uvm_config_db#(real)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "jet_rate_per_cm2",`JET_RATE_PER_CM2);
                        "looper_rate_per_cm2", `LOOPER_RATE_PER_CM2);
// 3. Jets
uvm_config_db#(int)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "mean_tracks_per_jet",`MEAN_TRACKS_PER_JET);
uvm_config_db#(int)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "jet_area_z_edge_size", `JET_AREA_Z_EDGE_SIZE);
uvm_config_db#(int) ::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "jet_area_phi_edge_size",
                        `JET_AREA_PHI_EDGE_SIZE);
                        codeComment// 4. Monsters
```

```
uvm_config_db#(real)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "monster_rate_per_cm2", `MONSTER_RATE_PER_CM2);
uvm_config_db#(monster_direction_t) ::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "monster_dir", PHI);
// 5. Noise hits
uvm_config_db#(real)::set(this,
                        "*.top_cluster_virtual_sequence.m_hit_seq",
                        "noise_hit_rate_per_cm2",
                        `NOISE_HIT_RATE_PER_CM2);
```

Therein one can notice the initial settings done on generic parameters like the number of BX cycles to be run (that has to agree with the one set for the trigger environment), the track angle, the sensor thickness, the pixel pitch, charge sharing fields and the hit amplitude range. Moreover, other specific parameters used only for the single classes of hits are set afterwards. Also in this case, for most of them it has been preferred to use defined variables. This does not mean anyway that the user cannot directly write desired numbers into the test file. Nevertheless, so far such definitions are done in the *ClassDefines.sv*, as shown in the code below:

```
`define PIXEL_CHIP_Z_FULL              512
`define PIXEL_CHIP_PHI_FULL            512
`define PIXEL_CHIP_AREA_cm2            6.5536
`define SENSOR_THICKNESS_um            100
`define PIXEL_DIM_ANGLE_DIRECTION_um   50

//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// 1. SINGLE TRACKS
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
`define TRACK_RATE_PER_CM2             1000000000
`define PERCENTAGE_HIT_PIXEL_ENVELOPE  0
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// 2. LOOPERS
```

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
‘define LOOPER_RATE_PER_CM2              0
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// 3. JETS
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
‘define JET_RATE_PER_CM2                 0
‘define MEAN_TRACKS_PER_JET              10
‘define JET_AREA_Z_EDGE_SIZE             20
‘define JET_AREA_PHI_EDGE_SIZE           20
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// 4. MONSTERS
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
‘define MONSTER_RATE_PER_CM2             0
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// 5. NOISE HITS
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
‘define NOISE_HIT_RATE_PER_CM2           0
```

It has been highlighted the definition of the full matrix dimension, the settings of some generic parameters for the chip and finally the ones related to the generation of the different classes of hits. Once the settings are done, the user can always check on top of all the files that the hit subscriber outputs, that they are correct. Moreover the possibility of using graphical visualization of clusters and histograms, as much as the capability of checking the hit rate offers a way of crosschecking what has been generated.

Chapter 7

Generation of input stimuli for a study of buffering architectures

The developed hit generator has been used for architectural studies, in particular for a the critical optimization of buffering requirements of the ROCs. It has been already introduced how the simultaneous increase in trigger latency and hit rate brings to ∼100 times higher buffer requirements. It has also been mentioned how significant reduction of storage resources could be obtained using shared buffering resources at the pixel region level. An analytical study [33] has been carried out on this topic with simple assumptions on the shapes of clustered hits and it will be introduced in the first subsection of this chapter.

7.1 Statistical/analytical cluster and PR buffer models

In the previous chapter it has been seen how cluster shape varies depending on the position in the pixel detector (or track angle), pixel size, sensor thickness, plus on additional factors that influence charge sharing (e.g. magnetic field component, radiation damage) and how to instruct the framework to generate such clusters. Previously, such a tool for intensive simulation was still to be developed and an initial statistical/analytical study had been carried out starting from evaluating cluster shapes only with dependence to one factor, i.e. the position of the chip in the pixel detector. In particular, when considering pixel chips located:

a) in the center of the barrel where tracks are expected to come perpendicular to the sensor: a cluster is seen as a central hit pixel (HP) with some pixels fired in the periphery;

b) at the edges of the barrel where track angles are such that signal is generated on several pixels and the shape is elongated in one preferential direction.

The exact shapes considered in the study are reported in Figure 7.1.

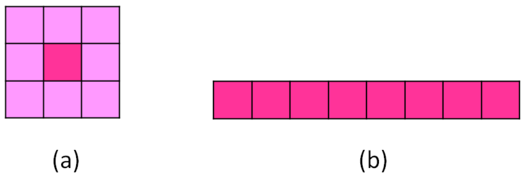


FIGURE 7.1: Symmetrical cluster model; (b) elongated cluster model. Hit pixels are highlighted in dark pink.

The first one (a) is a fixed size square envelope made of 3x3 pixels (to be referred as symmetrical model), inside which it is possible to have from 1 to 9 HPs, where the central one is always hit. As concerns the second (b), that can be referred as elongated model , a rectangular shape made of 1xn_p has been considered, where n_p, contained in the range [1,16], is the number of HPs in the cluster. It can be observed as they are quite similar to the ones that have been obtained with the hit generator respectively in Figure 6.9 using 90° track angle tracks and in Figure 6.8 (b) with lower angles. Moreover, each cluster model has then been detailed by adding statistical information on number of HPs inside them. Four typical statistical distributions of n_p, shown in Figure 7.2 have been made up in order to describe the extreme and intermediate cases of having clusters made from just one up to the maximum number of HPs. Moreover, such assumption on the n_p probability distribution have been validated by comparing them with real physics data obtained from data coming from a CMS run [40].

The two cluster models have been used to determine the average number of pixel regions that are occupied by a cluster, $\overline{PR}$. Such quantity depends on the PR configuration and the cluster model taken into account. A dedicated procedure, that also takes into account

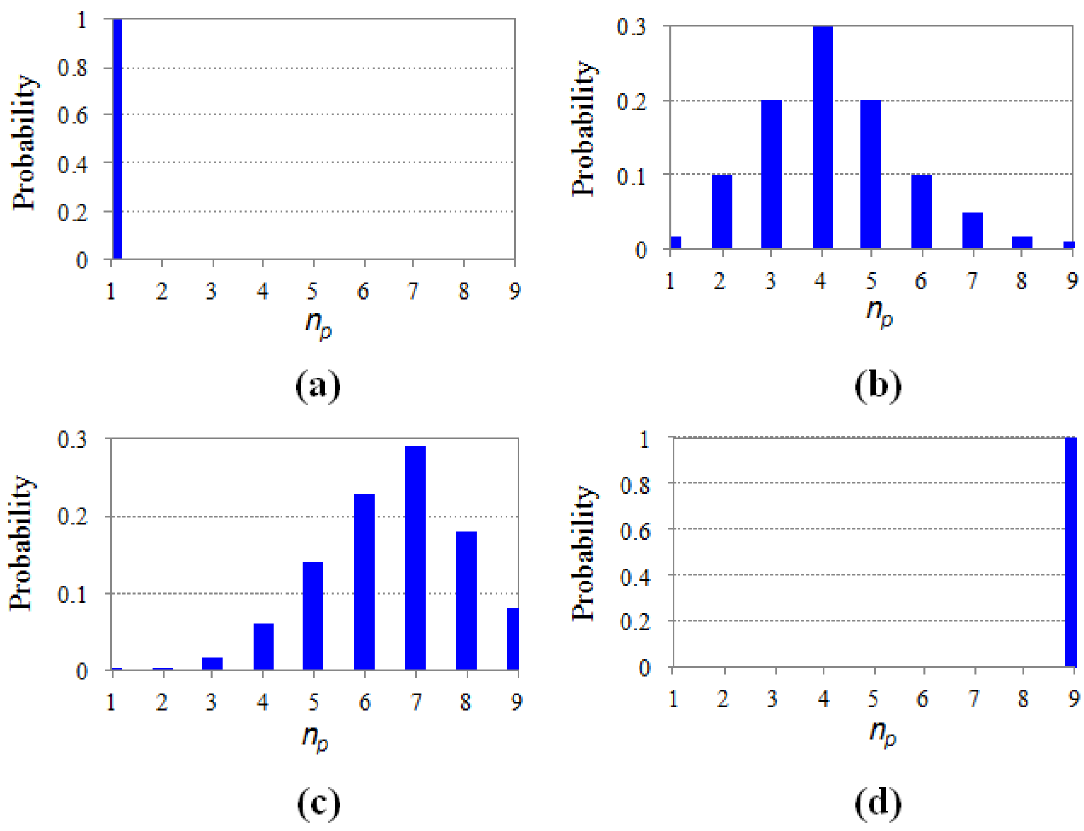


FIGURE 7.2: Examples of statistical distributions of n_p in a cluster: (a) "Single"; (b) "Average 4.22"; (c) "Average 6.57"; (d) "Maximum".

the n_p probability distribution, has been elaborated for each of them but is not herein reported, details can be found in [33]. The average number of pixel regions that are occupied by a cluster obtained have been used to evaluate performance of square and rectangular PR configurations. In particular, in order to critically design the PR logic, given a certain latency time, it is of interest to understand how many buffer locations are (at least) necessary to assure a buffer overflow probability lower than a target value, since buffer overflow translates in hit losses. For HEP applications an acceptable one is 10^{-2}, a more conservative approach would be to keep it below 10^{-3}. In order to carry out the overflow probability, the PR buffer has been mod-eled as an ideal array of finite memory locations where hit packets arrive at a rate that depends on the track rate in the detector and they receive a trigger signal after a fixed latency time. If the number of memory locations is h, assuming uncorrelated interactions taking place at each bunch crossing, the probability of arrival of more than h hit packets (i.e.probability of buffer overflow) during latency L can be obtained from the binomial distribution formula:

$$P_{overflow} = 1 - \sum_{i=0}^{h} \begin{pmatrix} i \\ \frac{L}{BX} \end{pmatrix} p^i (1-p)^{\frac{L}{BX}-1}$$

Where BX is the bunch crossing period of 25 ns and p is the probability of arrival of a hit packet during a bunch crossing period, that can be obtained once the track rate and the pixel size (and therefore the pixel region area) are fixed the certain values. Graphs can be therefore plot showing how the overflow probability varies depending on the number of buffer locations for a fixed pixel region configuration. From such plots, given a target value for the overflow probability, the required number of locations (h) necessary to avoid buffer overflow can be obtained. The calculations can be repeated for different cases of study of interest. In [33] different PR configurations have been compared for each one of the two cluster models and with the various n_p probability distribution. The comparison has been actually performed in terms of memory bits per pixel. The main steps are summarized in the following:

1. as mentioned, given the track rate, trigger latency, pixel size, pixel region configuration and finally the target value for the overflow probability, one can obtain the required number of locations (h);

2. the quantity that is most appropriate to study is actually the total number of memory bits in the PR buffer, as it contains information related to both the buffer locations and the memory organization, that therefore needs to be defined;

3. the total number of memory bits in a PR can be obtained from the product between the required number of buffer locations and the number of bits of a single location;

4. normalizing the obtained total number of memory bits per PR by the number of pixels in the region gives a quantity that enables the results between different configurations to be compared.

In [33] a case study starting from assumptions related to next generation pixel chips (on track rate, trigger latency, etc...) was presented and conclusions on the total number of required memory bits for the symmetrical and elongated cluster models had been obtained for both square and rectangular pixel region configurations. The conclusions of the study have been in support of square pixel regions, going from 2x2 to 4x4. This last option is particularly of interest if one starts thinking of layout and area related issues. Indeed, when considering dedicated SRAM blocks, a few "large" memory blocks are preferable than many "small" memory blocks and this means that relatively bigger configurations could be more attractive even if analytically they do not look like being the optimum.

7.2 Simulation results from VEPIX53

It has been shown in the previous section how for this study, that aims to optimize the pixel chip architecture comparing different suitable PR configurations, a key quantity of interest is the buffer overflow probability. Preliminary simulations have been performed with the VEPIX53 framework to obtain results on such a quantity coming from simulations to compare them with the analytical ones (they have been presented in [41]). The parameters that have to be defined for the statistical model are herein summarized:

a) cluster model (square or elongated);

b) pixel region configuration;

c) n_p probability distribution;

d) track rate per cm^2;

e) trigger latency;

f) pixel size.

Such quantities need to be accordingly chosen both when obtaining graphs on buffer overflow probability analytically and when generating them from simulation results. For this purpose, the developed hit generator has to be appropriately tuned in order to create cluster models that are quite similar, or even (on purpose) slightly different, from the ones assumed analytically. In those initial simulations, only the square cluster model (a) has been taken into account. The preliminary results will be reported in the following subsections using two different configurations for the hit generator and the respective n_p probability distribution (c) will be specified, as well as the track rate (d). The remaining parameters have been kept the same in both the simulations. It has been chosen to start using the proposed extended CMS trigger latency of 10μs (e) that is the baseline value. As regards the sensor, the usual 50x50μm^2 pixel size (e) and 100 μm sensor thickness have been used (this last parameter is not an input of the analytical model). In particular, it has been chosen to first take into account a 4x4 PR configuration (b) for the DUT (since it is particularly attractive from a layout point of view). Furthermore, as far as the simulation framework is concerned, it has been chosen to simulate a bigger 512x512 pixel chip matrix at the generation level. In order to obtain sufficient statistics, simulations have been run for 500,000 BX cycles. In this way the 4x4 sub-matrix of the pixel chip being simulated has seen around 10,000 incoming hits. In conclusion, as regards the DUT, the results presented have been obtained simulating the zero-suppressed FIFO archicture.

7.2.1 Clusters with fixed size

Since a single parameter was used in the model to describe the rate of incoming hits (i.e. track rate normalized by the pixel area), for the first simulations the hit generator has been configured in order to generate only tracks. In this first example it has been chosen to compare cluster models in principle quite different (but on average quite similar to each other), keeping all the other parameters as the same. In particular the track rate has been fixed to 1 GHz/cm^2. For the simulations, fixed size clusters have been used. In order to instruct the hit generator to produce them no charge sharing has been allowed. Setting the track angle to 45° (with the set sensor thickness and pixel pitch) has brought to generation of clusters made of 2 pixels. With the set track rate it translates into an expected (and actually observed) hit rate close to 2 GHz/cm^2. Such a simple distribution on cluster size is shown in Figure 7.3. Considering the square cluster model (related to a position of the detector in the center of the barrel) described in the previous section, some examples of typical statistical distributions of n_p have been previously reported. Other ones can anyway also be considered to repeat the study and in this case one with a lower average number of hit pixels, equal to 2, has been taken into account (Figure 7.4). Moreover, a different distribution for n_p coming from physics data (acquired in the center of the barrel) with average 2.46 has also been taken into account. It is shown in Figure 7.5.

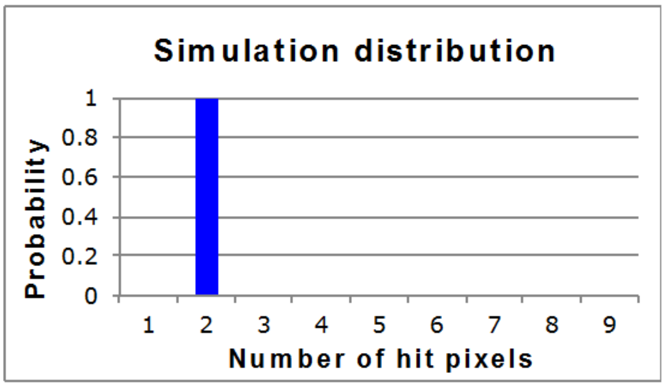


FIGURE 7.3: Distributions of number of hit pixels in the simulation

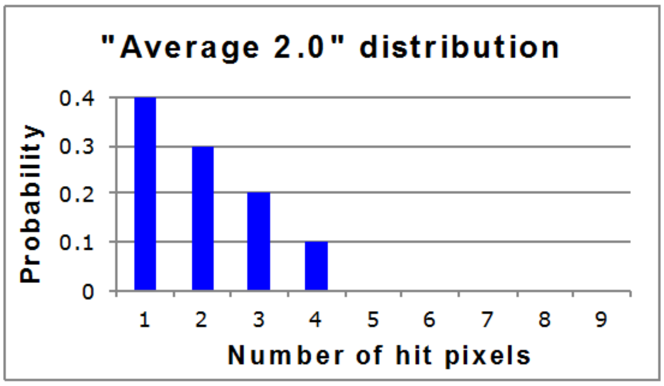


FIGURE 7.4: Typical distribution of number of hit pixels (i.e. n_p) for the square cluster model

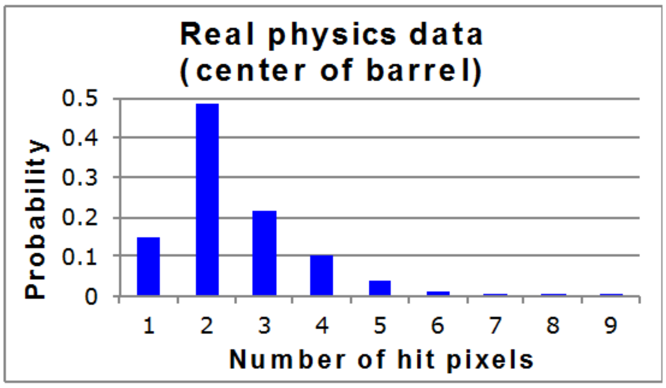


FIGURE 7.5: Distribution of number of hit pixels (i.e. n_p) for the square cluster model (obtained from real physics data)

For the analytical model, a formula has been presented for calculating the overflow probability. From the binomial distribution and with the same assumption (and symbols), another quantity that can be considered is the buffer occupancy. The probability of having a number i of buffer locations occupied can indeed been obtained from:

$$P_{occupancy}(i) = \begin{pmatrix} i \\ \frac{L}{BX} \end{pmatrix} p^i (1-p)^{\frac{L}{BX}-1}$$

The same symbols have also been used. On the other hand, such a quantity can be easily obtained with appropriate settings of the simulation and verification framework. It particular, it is necessary to instruct the devoted subscriber in the analysis environment to dump the output file containing information on buffer occupancy (that need to be normalized to the total number of entries for 1:1 comparison with the analytical one). The comparison of the plot obtained analytically (using the two distributions for n_p) and from VEPIX53 simulations are reported in Figure 7.6. Despite the cluster models

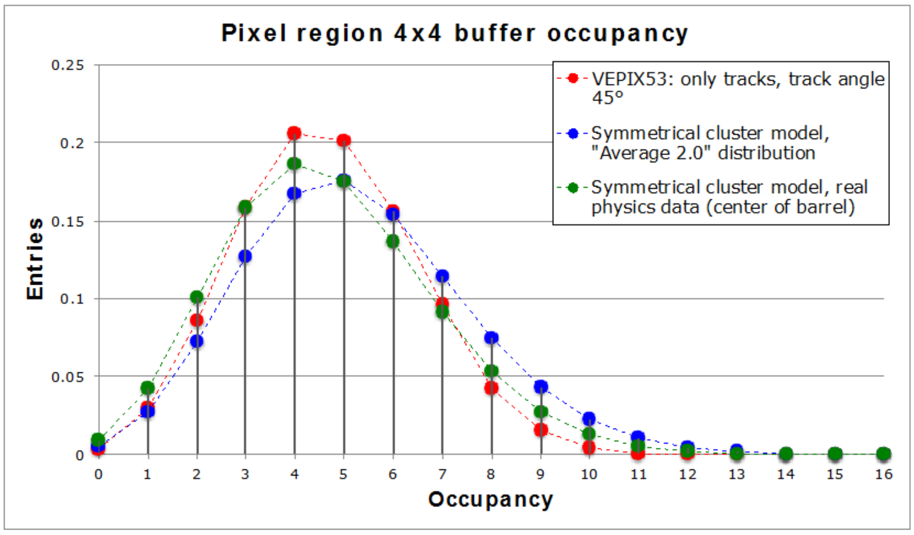


FIGURE 7.6: Comparison between VEPIX53 simulations (with generation of fixed size clusters) and the symmetrical cluster model based on buffer occupancy.

used in the simulations (tracks at 45°) is in principle different from the one assumed analytically (symmetrical cluster, center of the barrel), using the same track rate, trigger latency and just similar average cluster sizes brings to a good consistency between the plots can be seen. A comparison can be also done in terms of buffer overflow probability and it is straightforward to calculate by using the simulation results. Given a certain buffer depth h, it is indeed sufficient to sum all the probabilities of having an occupancy greater than such a value:

$$P_{overflow} = 1 - \sum_{i=0}^{h} P_{occupancy}(i).$$

The comparative graphs are shown in Figure 7.7 and are once more quite consistent, if one considers the different statistical assumptions done in the 3 different cases. Simulation results look more optimistic than the analytical ones. It can be remarked

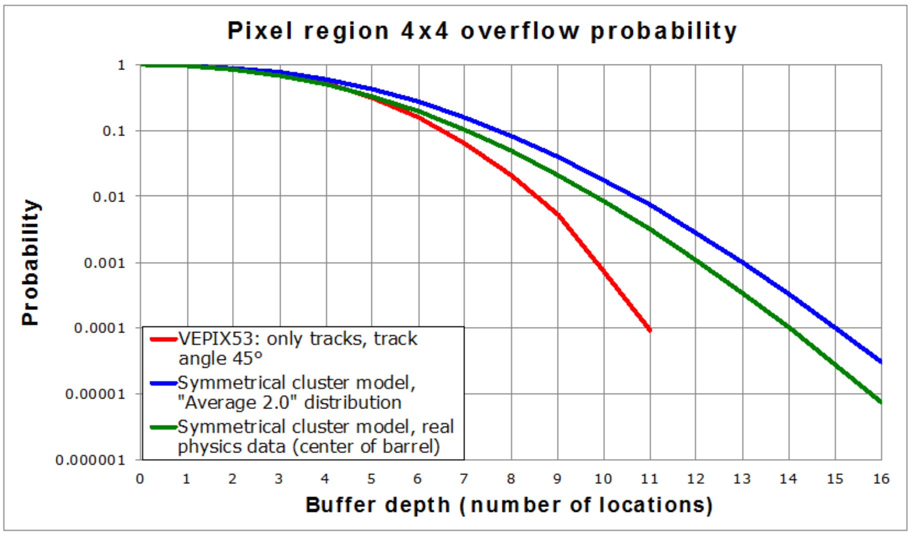


FIGURE 7.7: Comparison between VEPIX53 simulations (with generation of fixed size clusters) and the symmetrical cluster model in terms of buffer overflow probability.

how the fixed cluster size used in the first case has probably played a positive role in this sense. It can be noticed how for the less strict requirement (10^{-2}) a buffer depth between 8 and 11 is necessary, while for the more conservative one (10^{-3}) goes from 11 to 15 for the different models.

7.2.2 Clusters with variable size

A second simulation has been performed with the aim of using cluster models as similar as possible to the analitycal ones. For this reason, it has been chosen to generate only tracks coming at 90° (as happens in the center of the barrel) and to introduce charge sharing. Such a choice instructs the generator to produce clusters with a central hit pixel and some fired in the periphery. The specific typical n_p distribution herein taken into account for the statistical cluster model presents an average value equal to 4.22, as shown in Figure 7.8. In order to reproduce a similar distribution, with average cluster

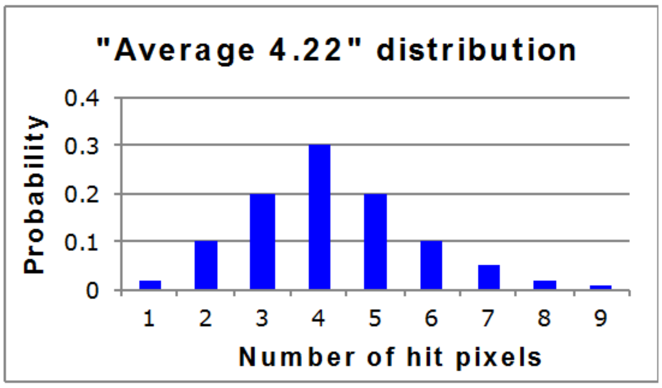


FIGURE 7.8: Typical distribution of number of hit pixels (i.e. n_p) for the square cluster model with Average 4.22

size around 4, through the hit generator an appropriate value for the percentage of pixels to be hit in the envelope that surrounds the central one has been chosen. For each cluster it is clearly know that at least one pixel is hit (in the center) and that the 8 peripheral ones may be fired depending on the set parameter. In order to hit the 3 additional pixels needed (out of 8), the probability of hitting each is obtained: $\frac{3hitpixels}{8totalpixelsintheenvelope} * 100 =$ 37.5. Through some additional coding, it has also been possible to observe the actual distribution of the number of hit pixels per cluster produced by the hit generator (7.9). Even if not identical, both distributions show some variance around a close average value. In order to evaluate the impact of a bigger cluster size (with respect to the one

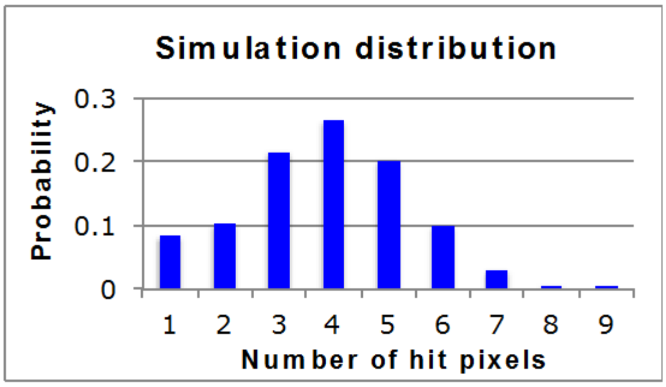


FIGURE 7.9: Observed distributions of number of hit pixels in the simulation. The measured average is slightly lower than expected: 3.84

in the previous subsection), it has been chosen to keep a constant hit rate (2 GHz/cm^2). For this reason a track rate of 500 MHz/cm^2 has been used for the simulations, while a slightly lower one has been used in the statistical model. The obtained results are both presented in terms of buffer occupancy and overflow probability in Figure 7.10 and in Figure 7.11. It can be noticed as in both cases comparative graphs are quite consistent.

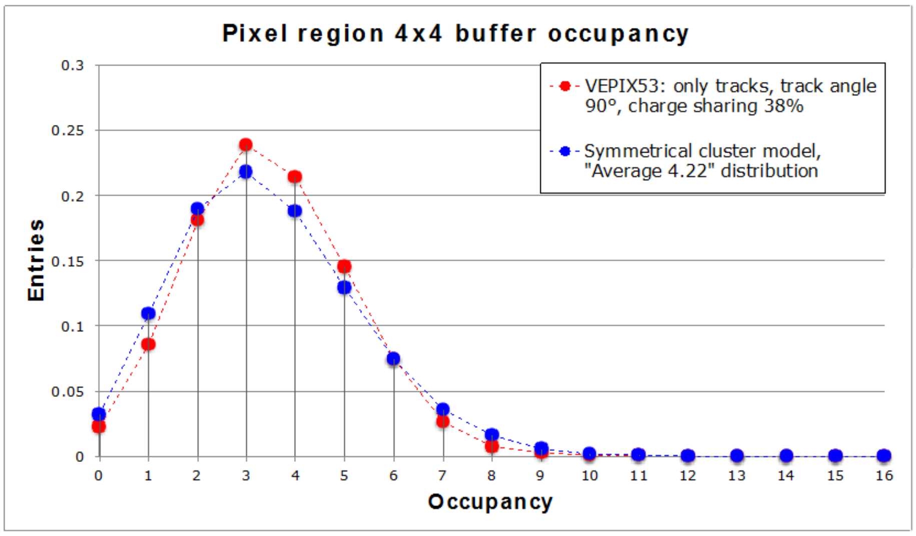


FIGURE 7.10: Comparison between VEPIX53 simulations (with generation of variable size clusters) and the symmetrical cluster model based on buffer occupancy.

From the overflow probability plot, it can be observed how with the assumptions made on

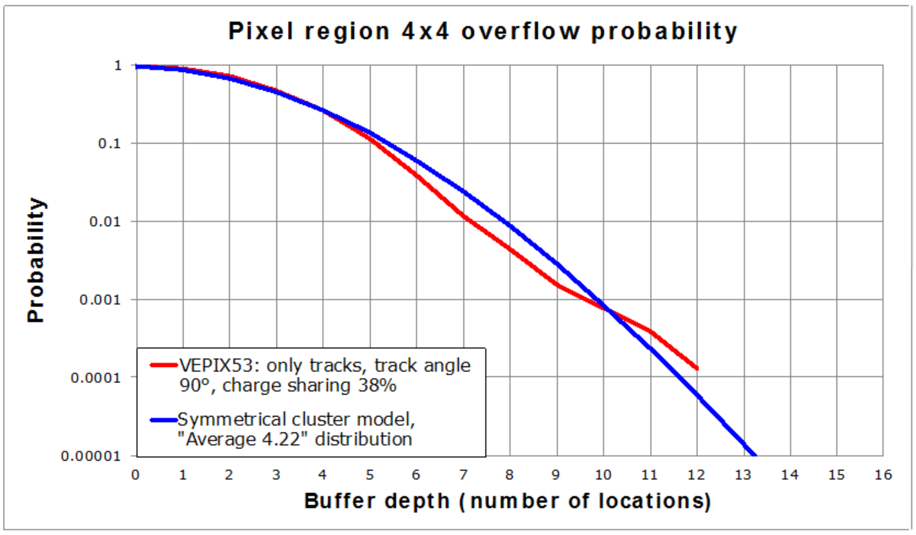


FIGURE 7.11: Comparison between VEPIX53 simulations (with generation of fixed size clusters) and the symmetrical cluster model based on buffer overflow probability.

track rate, average cluster size and trigger latency a buffer depth around 10 is sufficient to satisfy the stricter requirement (10^{-3}) on overflow probability.

Conclusions and future work

The development of a pixel simulation and verification platform capable of simulating alternative pixel chip architectures of the high rate pixel detectors for the ATLAS/CMS upgrades pixel chip has been the main focus of this thesis. First, the evaluation of performances of commercial simulation software tools has been performed. It has shown specific bottlenecks that have been further investigated through the profiler and choices have been taken in order to improve simulation time. Also the design flow used has been optimized for minimal re-elaboration of code. Results have anyway shown that the software used assures good simulation performance. Secondly, the author has collaborated in the development of a behavioural model for the system, working on the definition of a simple and basic module of a single pixel unit cell. Thirdly, the set up of the framework based on UVM classes has been done, defining the project organization, the design flow and the running scripts. The main contribution on the development of the verification components has been on the stimuli generation, both for the hits and the trigger and for their coordination from the top level. Different and configurable tests have also been defined, and a short guide for the user has been provided for setting and running them. Moreover, configuration objects for re-configurability of the different verification components and for controlling generation of output file have been defined. Special attention has been put on the development of a hit generator capable of emulating pixel hits on a large number of pixels with appropriate correlations: based on a geometrical model of the sensor, it has been made capable of generating tracks, loopers, jets, monsters and noise hits with parameterized rates and other parameters characteristic of the specific hit class. Furthermore, generated inputs have been monitored through graphical visualization (performed with MATLAB) and through

statistics collection in terms of total and specific hit rate. Results on generated clusters and observed rates have shown good consistence with the parameters set for the hit generator. Lastly, the author has collaborated in using the developed engine for architectural studies, in particular for a the critical optimization of buffering requirements. Simulation results have been obtained for buffer occupancy and overflow probability for two preliminary cluster models that were interesting to be compared with the ones of a previous statistical/analytical study. The reported plots have shown good agreement between the models and they show buffering requirements to be compatible with the proposed extended CMS trigger latency of 10 μs and for the absolute highest hit rate of 2 GHz/cm^2.
Further investigations and simulations will need to be performed in order to evaluate different classes of stimuli (e.g. jets) and to compare different PR configurations for architecture optimization. A deep simulation analysis will be performed to evaluate the impact of the optional 20 μs trigger latency on the buffering requirements and to compare the obtained results with the analytical ones. Future work will be needed in the context of the working group both on the DUT and on the simulation and verification framework. As regards the first, beside the independent PUCs and the zero-suppressed FIFO one, more buffering architectures will be implemented and evaluated. Moreover it will be needed to replicate PRs in the pixel chip for further buffering study and column arbitration study. Several description of the chip are also being considered, at different level of abstraction. As concerns the framework, a further refinement of the hit generator and on interfacing it with hit patterns coming from external full detector/experiment Monte Carlo simulations and detailed sensor simulations will be required. It should be also improved the generality and flexibility of the whole environment and of the reference model of the ROC, in order to simulate chips with different functionality. More quantities will also need to be monitored in order to provide results on significant performance indicators. In conclusion, integration of graphical analysis in the framework itself could also be evaluated, considering interfacing SystemVerilog with C/C++.

References

[1] L. Rossi. *Pixel Detectors: From Fundamentals to Applications*. Particle Acceleration and Detection. Springer, 2006. ISBN 9783540283324. URL `http://books.google.fr/books?id=Jbp73yTz-LYC`.

[2] Hemperek. Hybrid or monolithic? pixel detectors for future lhc experiments, December 2013. URL `https://indico.cern.ch/event/273886/`.

[3] Jorgen Christiansen. Tdc's architectures in asics, November 2011. URL `https://indico.cern.ch/event/122027/session/11/contribution/24/material/slides/1.pdf`.

[4] Accellera. Universal verification methodology (uvm) 1.1 user's guide, May 2011. URL `https://www.cadence.com/rl/resources/white_papers/max_metric_driven_ver_wp.pdf`.

[5] Maurice Garcia-Sciveres. The new atlas pixel chip: Fei4, 2011.

[6] J (CERN) Chistiansen and M (LBNL) Garcia-Sciveres. RD Collaboration Proposal: Development of pixel readout integrated circuits for extreme rate and radiation. Technical Report CERN-LHCC-2013-008. LHCC-P-006, CERN, Geneva, Jun 2013. The authors are editors on behalf of the participating institutes. the participating institutes are listed in the proposal.

[7] Jorgen Christiansen. Development of pixel readout integrated circuits for extreme rate and radiation, 2014. URL `https://indico.cern.ch/event/306848/`.

[8] Victor Berman. A tale of two languages: Systemc and systemverilog. URL `http://chipdesignmag.com/display.php?articleId=116`.

[9] Viktor Veszpremi. Operation and performance of the CMS tracker. Technical Report arXiv:1402.0675, Feb 2014. Comments: accepted for publication in Journal of Instrumentation.

[10] Roland Horisberger. Readout architectures for pixel detectors. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 465(1):148 – 152, 2001. ISSN 0168-9002. doi: http://dx.doi.org/10.1016/S0168-9002(01)00378-3. URL `http://www.sciencedirect.com/science/article/pii/S0168900201003783`. {SPD2000}.

[11] Wermes. Current status and future prospects of pixel detectors, November 2013. URL `http://indico.cern.ch/event/279759/`.

[12] Lucio Rossi. Lhc upgrade plans: options and strategy. *Proceedings of IPAC2011*, pages 908–912, 2011. URL `http://accelconf.web.cern.ch/accelconf/IPAC2011/papers/tuya02.pdf`.

[13] Brezina et al. The timepix3 chip, February 2014. URL `https://indico.cern.ch/event/267425/`.

[14] G. Aad, M. Ackers, F.A. Alberti, M. Aleppo, G. Alimonti, et al. ATLAS pixel detector electronics and sensors. *JINST*, 3:P07007, 2008. doi: 10.1088/1748-0221/3/07/P07007.

[15] M. Barbero, W. Bertl, G. Dietrich, A. Dorokhov, W. Erdmann, K. Gabathuler, St. Heising, Ch. Hörmann, R. Horisberger, H.Chr. Kästli, D. Kotlinski, B. Meier, and R. Weber. Design and test of the {CMS} pixel readout chip. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 517(1–3):349 – 359, 2004. ISSN 0168-9002. doi: http://dx.doi.org/10.1016/j.nima.2003.09.043. URL `http://www.sciencedirect.com/science/article/pii/S0168900203026391`.

[16] R Ballabriga, J Alozy, G Blaj, M Campbell, M Fiederle, E Frojdh, E H M Heijne, X Llopart, M Pichotka, S Procz, L Tlustos, and W Wong. The medipix3rx: a high resolution, zero dead-time pixel detector readout chip allowing spectroscopic imaging. *Journal of Instrumentation*, 8(02):C02016, 2013. URL `http://stacks.iop.org/1748-0221/8/i=02/a=C02016`.

[17] X. Llopart, R. Ballabriga, M. Campbell, L. Tlustos, and W. Wong. Timepix, a 65k programmable pixel readout chip for arrival time, energy and/or photon counting measurements. *Nuclear Instruments and Methods in Physics Research A*, 581:485–494, October 2007. doi: 10.1016/j.nima.2007.08.079.

[18] CMS Collaboration. Technical proposal for the upgrade of the CMS detector through 2020. Technical Report CERN-LHCC-2011-006. LHCC-P-004, CERN, Geneva, Jun 2011.

[19] M. Garcia-Sciveres, D. Arutinov, M. Barbero, R. Beccherle, S. Dube, D. Elledge, J. Fleury, D. Fougeron, F. Gensolen, D. Gnani, V. Gromov, T. Hemperek, M. Karagounis, R. Kluit, A. Kruth, A. Mekkaoui, M. Menouni, and J. D. Schipper. The fe-i4 pixel readout integrated circuit. volume 636, pages S155 – S159, Amsterdam, 29082009 - 01092009 2011. 7th International ””Hiroshima”” Symposium on the Development and Application of Semiconductor Tracking Detectors, Hiroshima(Japan), North-Holland Publ. Co. doi: 10.1016/j.nima.2010.04.101. URL `http://juser.fz-juelich.de/record/136446`.

[20] W. Fornaciari and C. Brandolese. *Sistemi embedded. Sviluppo hardware e software per sistemi dedicati.* Pearson, 2007. ISBN 9788871923420. URL `http://books.google.it/books?id=3FfZYF9vLY0C`.

[21] *1666-2005 - IEEE Standard SystemC(R) Language Reference Manual.*

[22] Lukai Cai and Daniel Gajski. Transaction level modeling: An overview. In *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS ’03, pages 19–24, New York, NY, USA, 2003. ACM. ISBN 1-58113-742-7. doi: 10.1145/944645.944651. URL `http://doi.acm.org/10.1145/944645.944651`.

[23] *1364-2005 - IEEE Standard for Verilog Hardware Description Language.*

[24] Stuart Sutherland, Simon Davidmann, and Peter Flake. *SystemVerilog for Design: A Guide to Using SystemVerilog for Hardware Design and Modeling.* Springer Publishing Company, Incorporated, 2nd edition, 2010. ISBN 1441941258, 9781441941251.

[25] *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language.*

[26] Chris Spear. *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features.* Springer Publishing Company, Incorporated, 2nd edition, 2008. ISBN 0387765298, 9780387765297.

[27] Verification Academy. Uvm cookbook. URL `https://verificationacademy.com/cookbook/uvm`.

[28] S. Rosenberg and K.A. Meade. *A Practical Guide to Adopting the Universal Verification Methodology (UVM).* Cadence Design Systems, 2010. ISBN 9780578059556. URL `http://books.google.fr/books?id=5p7pZwEACAAJ`.

[29] Accelera. Universal verification methodology (uvm) 1.1 class reference. URL `http://www.accellera.org/downloads/standards/uvm/UVM_1.1_Class_Reference_Final_06062011.pdf`.

[30] Cadence. Cadence functional verification.

[31] Mentor Graphics. Mentor graphics functional verification.

[32] Synopsys. Synopsys functional verification.

[33] E Conti, J Christiansen, P Placidi, and S Marconi. Pixel chip architecture optimization based on a simplified statistical and analytical model. *Journal of Instrumentation*, 9(03):C03011, 2014. URL `http://stacks.iop.org/1748-0221/9/i=03/a=C03011`.

[34] Cadence Richard Goering. Webinar: Is systemverilog the future of mixed-signal modeling?, October 2012. URL `http://www.cadence.com/Community/blogs/ii/archive/2012/10/04/webinar-is-systemverilog-the-future-of-mixed-signal-modeling.aspx`.

[35] Hemperek. Rd53 - wg3 simulation testbench.

[36] www.testbench.it. Array randomization.

[37] Cadence Timothy Pylant. Create a sine wave generator using systemverilog, June 2009. URL `http://www.cadence.com/Community/blogs/fv/archive/2009/06/30/create-a-sine-wave-generator-using-systemverilog.aspx`.

[38] Doulos. Systemverilog and uvm adopter class.

[39] T. Binoth, C. Buttar, P.J. Clark, and E.W.N. Glover. *LHC Physics*. Scottish Graduate Series. Taylor & Francis, 2012. ISBN 9781439837702. URL `http://books.google.it/books?id=sMZDMZKLLx0C`.

[40] Cms lhc run 200091. Data provided by M. Swartz (Johns Hopkins University).

[41] RD53 General Meeting. Simulation framework in system verilog & uvm, 2014. URL `https://indico.cern.ch/event/296570/other-view?view=standard`.