FINAL REPORT

A CMOS Imager Integrating Compression Sampling

Name of the student: Alexandre Bibet

Name of supervisor: Alexandre Schmid

Name of INPG Tutor: Gilles Sicard

Copyright 2008

By

Alexandre Bibet
Acknowledgements

I would like to express gratitude to my tutor, Dr. Alexandre Schmid, for his guidance and wealth of knowledge, which has contributed immensely towards my thesis. My sincere thanks to Ph.D. Vahid Majizadeh (my assistant) and Laurent Jacques (working in the signal processing laboratory) for their support and advice throughout my internship. I am thankful to my committee members for sparing their valuable time for reviewing my thesis. Last, but not least, I would also like to thank my mother, brother and sister and all my friends in Taiwan and Switzerland for their support throughout the course of my stay at the EPFL.
Table Of Contents

Project information........................................................................................................................................1

Copyright....................................................................................................................................................2

Acknowledgements........................................................................................................................................3

Abstract.......................................................................................................................................................9

Chapter 1: Introduction..................................................................................................................................10

I. Specifications...........................................................................................................................................10
II. Project development and foreseen milestones.........................................................................................10

Chapter 2: State of the art: photosensor and compression type choice.....................................................12

Part 1: State of the art of photosensors and image sensors.........................................................................12

I. Photodetectors.........................................................................................................................................12
II. Principle..................................................................................................................................................12
III. Charge coupled devices...........................................................................................................................13
   III.1. History of charged coupled devices..................................................................................................13
   III.2. The used materials...........................................................................................................................14
   III.3. Working principle of CCD devices..................................................................................................14
   III.4. CCD arrays.....................................................................................................................................14
   III.5. Development of CCDs....................................................................................................................14
IV. Photodiodes............................................................................................................................................16
   IV.1. Photodiode materials, structure and operation...............................................................................16
   IV.2. Photodiode model............................................................................................................................17
V. CMOS image sensors.................................................................................................................................18
   V.1. Introduction.......................................................................................................................................18
   V.2. Passive pixel sensors.........................................................................................................................19
   V.3. Active pixel sensors..........................................................................................................................20
VI. Summary................................................................................................................................................21
Part 2: State of the art of the compression of images from sensor arrays

I. Introduction
II. Various techniques for compression
   II.1. Compressing the information from a single pixel or a block of pixels
   II.2. Performing an operation on all pixels and storing the sum of all results
III. Previously encountered problems
IV. Conclusion

References for chapter 2

Chapter 3: Compression and decompression

I. Compression principle
   I.1. The compression type
   I.2. What was tried to be done
   I.3. What was finally chosen and why
II. Decompression principle
III. Compression and decompression algorithm main blocks

Chapter 4: System overview

I. The array
II. The digital unit
III. The analog signal processing unit

Chapter 5: The digital unit design: a first idea

I. Introduction
II. Control system first part: clock divider
III. Control system second part: seed and SLF signal generation system
   III.1. Structure description
   III.2. Simulation results with Modelsim
   III.3. Synthesis results
IV. Control system third part: control unit

IV.1. Structure description

IV.2. Simulation results with Modelsim

IV.3. Synthesis results

V. Control system fourth part: LFSR and control unit clock generation circuit

V.1. Structure description

V.2. Simulation results with Modelsim

V.3. Synthesis results

VI. Control system fifth part: 12 bit decoder

VI.1. Structure description

VI.2. Simulation results with Modelsim

VI.3. Synthesis results

VII. Control system sixth part: The LFSR or PRNG

VII.1. The PRNG structure search

VII.1.1. Dividing the array into P parts

VII.1.1.1. Principle

VII.1.1.2. Number of parts determination

VII.1.1.3. Results

VII.1.1.4. Another number of bits for the LFSR

VII.1.1.5. Conclusions

VII.1.2. A safer solution

VII.2. The LFSR design

VII.2.1. Framework

VII.2.2. LFSR modelling

VII.2.3. Simulation results with Modelsim

VII.2.4. Synthesis results

VIII. The 1 bit memory

VIII.1. Structure description

VIII.2. Simulation results with Modelsim

VIII.3. Synthesis results

IX. The whole system

IX.1. Structure description

IX.2. Simulation results with Modelsim
Chapter 6: Analog signal processing .................................................................55

Part 1: Multiplication by the coefficient ..........................................................55

I. A first solution ..........................................................................................55
II. A better solution ......................................................................................56

Part 2: The operational amplifier: the started job .........................................57

I. The OP AMP chosen structure .................................................................57
II. Determining the transfer function of the fully differential OP AMP .......57
   II.1. With a third order transfer function ....................................................57
      II.1.1. Determining the output load capacitance ..................................57
      II.1.2. Determining the value of the input capacitor .........................60
      II.1.3. Determining the values of the resistors ..................................60
      II.1.4. Determining the transfer function of the OP AMP .....................60
   II.2. With a second order transfer function ..............................................64
III. The OP AMP complete structure ............................................................68
IV. What remains to be done ........................................................................69

References for chapter 6 ............................................................................70

Chapter 7: The design of the photodiode array ............................................72

I. The photodiode .........................................................................................72
   I.1. The dimensions of the photodiode .....................................................72
   I.2. The photodiode layout ....................................................................72
II. Our first idea for the floor plan ...............................................................73
   II.1. For the pixel ..................................................................................73
   II.2. For the whole system ....................................................................73
   II.3. Estimation of the size of the system ................................................74
III. Our final idea for the floor plan ..............................................................74
   III.1. Storing the coefficient within the pixel ...........................................74
   III.2. The new floor plan of the pixel .......................................................76
   III.3. Ideas for the new floor plan of the whole system .........................77
   III.4. Estimation of the array area ..........................................................79
IV. Why avoiding using too many metal layers .........................................79
V. The schematics and layouts........................................................................79

V.1. The photodiode.................................................................................79
  V.1.1. The schematic...........................................................................79
  V.1.2. The layout...............................................................................80

V.2. The logic.........................................................................................80
  V.2.1. The schematic........................................................................80
  V.2.2. The layout.............................................................................81

V.3. The flip-flop....................................................................................82
  V.3.1. The schematic........................................................................82
  V.3.2. The layout.............................................................................82

V.4. The pixel........................................................................................83
  V.4.1. The schematic........................................................................83
  V.4.2. The layout.............................................................................84

V.5. The array.........................................................................................85
  V.5.1. The schematic........................................................................85
  V.5.2. The layout.............................................................................85

V.6. The pad..........................................................................................87
  V.6.1. The Schematic........................................................................87
  V.6.2. The layout.............................................................................88

V.7. The floor plan of the whole system: a more precise insight..........89

Conclusions and future work.................................................................91

Appendixes..........................................................................................93
Abstract

Electronic devices dealing with images are numerous. One can name cameras, video cameras, cell phones, webcams… These devices all have to acquire images and stores them shortly after.

The size of the data taken in charge by these machines is getting larger and larger as time goes by and now it is not uncommon to have a camera acquiring great quality images of 10MB each!

The problem is that memory does not increase as fast as the data does. Therefore, the only way to counter this fact is to compress this data. A compression algorithm will thus have to be embedded within the device, and will be located just after the array of photosensors (CCD or photodiodes). What will be stored is the compressed image. The latter will be unzip on the computer thanks to a reverse algorithm to the one used to zip the image.

This idea has thus many applications and the market to sell photosensor arrays associated to a compression circuit is huge.

During this internship, we began to develop one chip with a 64x64 (size determined thanks to simulations) photodiode array associated to a compression circuit in the 0.35μm CMOS technology. This chip and a testing chip for this system will be, in the future, soldered onto a PCB to create a complete device with its working part and its testing part.
CHAPTER 1: INTRODUCTION

I. Specifications

In this project, we will have to develop an intelligent imager, embedding compression sampling on-chip. It should be able to acquire completely fixed images and compress them in a reasonable time (from hundredths of milliseconds to few seconds per acquired image).

A dedicated pixel sensor will be developed in a CMOS 0.35μm Optoelectronic-compatible fabrication technology, and used to form a two-dimensional array. Analog signal processing as well as digital system control will be developed and integrated. Full-custom analog and digital, as well as semi-custom development methodologies will be involved, and a tape-out is targeted for the June 2, 2008 deadline.

II. Project development and foreseen milestones

Following project schedule sketches the main stages. The dates are only approximate, and some sections will necessarily overlap.

1) February 18 – March 7
- Acquisition of competences in compressive sampling, as well as CMOS imager systems and development. A Matlab model will be used to simulate compressive sampling algorithms.
- Support is provided by Professor P. Vandergheynst at this level.

2) March 10 – April 18
- Study of the system architecture.
- Development of a pixel, based upon the general theory of optoelectronics, as well as the specific constraints of the project, which have been evidenced in the first phase.
- Adaptation of the compressive sampling algorithms to VLSI.

3) April 21 – May 9
- Development of the imager array, of the analog signal processing and digital control.

4) May 13 – June 2
5) **June 2 – June 20**
- Project documentation.
- Project report delivery on June 20, 2008 at 12:00.

6) **June 20 – August 18**
- Preparation of chip testing. Development of a test PCB, and measurement interface.
- Preparation of the full technical documentation.
- Formal project presentation.
CHAPTER 2: STATE OF THE ART: PHOTOSENSOR AND COMPRESSION TYPE CHOICE

PART1: STATE OF THE ART OF PHOTODETECTORS AND IMAGE SENSORS

Photodetectors detect light and are thus important components of image sensors. Before studying image sensors, the basic types of photodetectors will be reviewed. They are of two types, generally using silicon as substrate: charge coupled devices (CCD) and photodiodes. A photodiode is usually used in active or passive pixel sensors. CCD sensors, being the basic technologies used for image sensing, have to be studied before seeing the complementary metal oxide semiconductor (CMOS) image sensors.

I. Photodetectors

Photodetectors are able to absorb and convert light into electricity. They are thus working as transducers. They are capable of detecting optical signals of different wavelengths (for example wavelengths corresponding to the visible light) using various semiconductor materials. Photodetectors are widely used for light sensing in image sensor arrays, security sensing, scientific and industrial applications.

II. Principle

The operation of photodetectors is based on the light induced electron-hole pair generation principle. When the light enters the material, it has a given photon energy \( (h\nu) \). Photons then destroy the covalent bond, creating such a pair. In other words, photons excite carriers across the conduction band, reducing the material resistance [1]. The energy of the photon being larger than the silicon energy band gap (-1.123 eV), they are absorbed by the silicon leading to electrons excitement from the valence band to the conduction band, thus supplying an electron-hole pair (EHP) [2]. The energy bands for visible light wavelengths are indicated in table 1. Optical signals in the visible light spectrum having the largest wavelengths supply a sufficient energy to excite electrons from the valence band to the conduction band in silicon. The generated EHP in the depletion region is then swept by the junction field. The measured current is the so-called photocurrent. The quantum efficiency, giving the rate at which the EHPs are generated per incident photon, is given by \( \eta \) [2]:

\[
\eta = \frac{I}{e} \frac{1}{P/h\nu}
\]
where, \( P \) is the photodiode illumination power, \( I \) is the produced current, \( e \) is the electron charge, \( h \) is the Planck’s constant, and \( \nu \) is the angular velocity. As the energy gap is lower at larger wavelengths, \( \eta \) becomes lower. At low wavelengths, the photon is absorbed before reaching the depletion layer, creating an EHP that can recombine. At maximum, \( \eta \) is equal to 1 and for absolute silicon this maximum occurs at a wavelength of 800nm. Fig.1 shows the EHP generation and recombination, where \( E_v \) and \( E_c \) are the energies in valence and covalent bands.

In what follows, CCD and photodiode models for sensing light using this EHP generation and recombination principle will be studied.

<table>
<thead>
<tr>
<th>Visible Lights</th>
<th>Wavelengths</th>
<th>Energy bands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>510nm</td>
<td>2.43eV</td>
</tr>
<tr>
<td>Red</td>
<td>650nm</td>
<td>2.43eV</td>
</tr>
<tr>
<td>Green</td>
<td>475nm</td>
<td>2.610eV</td>
</tr>
</tbody>
</table>

Table 1. Energy bands for visible light.

Fig.1. Electron hole pair generation and recombination [3].

III. Charge coupled devices

III.1. History of charged coupled devices

CCDs were invented by Boyle and Smith in 1969 at Bell laboratories [4]. Using the novel technology of metal-insulator semiconductor (MIS), a new device was invented for computers, called charged couple device [5] [6]. The first image sensor was fabricated and used in a television of a 64 x 106 resolution which is like using 13000 CCDs [7]. Later, many applications for CCDs appeared and, in 1974, the first astronomical CCD image was constructed using 100 x 100 CCD array [8] [9]. Later, traditional chemical based imagers in cameras were replaced by CCDs.
III.2. The used materials

The first CCD was built on a metal oxide semiconductor (MOS) capacitor. The MOS physical structure is shown in Fig.2. Initially Cr-Au metal was used as a metal layer and silicon dioxide as an oxide, thus building a MOS capacitor [10]. Polysilicon is also used in CCD arrays, where MOS transistors are closely packed.

III.3. Working principle of CCD Devices

The concept of CCDs revolves around the MOS capacitor operation. A CCD pixel accumulates the photons-induced charge on the potential well. The latter is created when the positive charge or the light falls on the gate of the MOS. Then, carriers form a potential well so that it can store the charge formed during the EHPs creation due to photons hitting the gate of the MOS. This potential well collects electrons before their recombination and holes are separated by the substrate.

III.4. CCD arrays

A single MOS transistor builds a pixel of the CCD array. The well depth depends on the applied gate voltage set at the staring point to keep charges below the electrode. The potential on that line of electrode drops with the increase of the potential in the neighbour cells thus creating a large well that can store all charges. At the end of the cycle, a first well is decreased while a second one is increased thus transferring the charge by one pixel. This cycle is repeated till the charge on each pixel is read out sequentially at the end of the row. The above scan lines drive the active charge from the periphery of the chip, with each pixel transferring the charge to its neighbour. The top layer along with metal in the MOS forms the gate [11, 12]. Shorter wavelengths lead to poorer detection by photodetectors. This is true for CCDs, as most of the power is absorbed by the gate. This is due to the penetration depths of the light hitting the gate. Table 2 shows the penetration depth for various light wavelengths [13]. From this table, we can observe that CCDs detect 90% of photons hitting the MOS.

III.5. Development of CCDs

CCDs fabrication started 30 years ago. Today, CCDs provide low noise, high speed, high resolution and high fill factor [14]. Some special CCDs like Fujifilm's super CCD, super CCD HR, and super CCD SR have been developed [15]. Latest CCD developments include increased sensitivity, dynamic range, signal-to-noise ratio, and reduced pixel pitch.
CCDs are widely used in point and shoot-type digital cameras. Novel CCDs are cost effective and have higher yield. CMOS active pixel sensors are more compatible with digital cameras as they are integrated in a single chip which is not possible for CCDs. Besides, CCDs do not contribute much towards the lower end cameras like PDAs, webcams, and cell phone cameras due to power inefficiency and as a result are driving researchers to develop low-power sensors like photodiodes [16]. Thus, CCDs can not be used in our project since we want to integrate the array and the rest of the system on a same chip.

![Working principle of charged couple devices](image)

**Fig. 2. Working principle of charged couple devices [2].**

<table>
<thead>
<tr>
<th>Wavelengths (Nanometers)</th>
<th>Penetration Depths (Micrometer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>0.19</td>
</tr>
<tr>
<td>450</td>
<td>1.0</td>
</tr>
<tr>
<td>500</td>
<td>2.3</td>
</tr>
<tr>
<td>550</td>
<td>3.3</td>
</tr>
<tr>
<td>600</td>
<td>5.0</td>
</tr>
<tr>
<td>650</td>
<td>7.6</td>
</tr>
<tr>
<td>700</td>
<td>8.5</td>
</tr>
<tr>
<td>750</td>
<td>16</td>
</tr>
<tr>
<td>800</td>
<td>16</td>
</tr>
<tr>
<td>850</td>
<td>46</td>
</tr>
<tr>
<td>900</td>
<td>62</td>
</tr>
<tr>
<td>950</td>
<td>150</td>
</tr>
<tr>
<td>1000</td>
<td>470</td>
</tr>
<tr>
<td>1050</td>
<td>1500</td>
</tr>
<tr>
<td>1100</td>
<td>7600</td>
</tr>
</tbody>
</table>

**Table 2. Penetration depth for various wavelengths.**
IV. Photodiodes

IV.1. Photodiode materials, structure and operation

Photodiodes are fabricated in semiconductor materials, and in general, in silicon. Silicon photodiodes are preferred to materials like gallium arsenide (GaAs), indium antimonide (InSb) or indium arsenide (InAs). Silicon absorbs light of wavelengths from 260nm up to 1100nm.

Silicon photodiodes are generally formed by joining some p-type and n-type silicon. This junction forms a depletion region. When some light is absorbed within this area, an EHP is formed there (see Fig.4.). When a voltage is applied to the junction, holes drift towards the p-side, and then reverse biased conditions arise. Carriers are therefore separated by the electric field and travel in opposite directions. Current thus passes through the semiconductor. This current is called diffusion current (shown in Fig.3.). There will be also a net charge formed due to the depletion layer of un-neutralized silicon ions. This net charge creates an electric field which causes another current called drift current. Photon-induced carriers are finally collected as current or voltage at electrodes before their recombination.

An increase in the depletion layer will allow photodiodes to collect more photons or would result in the increase in the storage of electrons. An increase of the depletion layer may also result in the increase of the transit time and the response of the photodiode model may be reduced. Desirable performance characteristics are essential with respect to response speed, quantum efficiency, and dark noise of a photodiode. Optimization of its size should be considered due to its importance in APS.

Fig.3. Photodiode depletion layer [17].
IV.2. Photodiode Model

Accurate models for photodiodes in CMOS image sensors, with high frequency and optimal size are discussed in [18]. The model is proposed by Swe and Yeo [18]. A photodiode can be modelled as a diode, a current source, a capacitor and two resistors. This model is shown in Fig.5. below. $R_s$ is the series resistance coming from the contacts and from the undepleted silicon resistance. $R_j$ is called the junction resistance which varies with the current crossing the photodiode.

Using symbols defined in table 3, the photocurrent is calculated thanks to the expression [19]:

$$I_p = \frac{q\eta P_{opt}}{h \nu} = \frac{q\eta P_{opt}}{h \nu \lambda}$$

The depletion or junction capacitance is calculated using the following expression [20]:

$$C_{j,dep} = \frac{c_j A_D}{(1 + \frac{V_{BB}}{p_h}) m_j} + \frac{c_j s w P_D}{(1 + \frac{V_{BB}}{p_s w}) m j s w}$$

The typical capacitance value taken for active pixel sensor simulation is 1pF, for AMI 0.6µm n-well CMOS transistors. Series resistors and junction capacitance can be calculated [19, 21].
Table 3. Parameters used in the photodiode model description.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\eta$</td>
<td>quantum efficiency of the photodiode</td>
</tr>
<tr>
<td>$q$</td>
<td>Charge of Electron=$1.602 \times 10^{-19}$ in Js</td>
</tr>
<tr>
<td>$v$</td>
<td>Frequency</td>
</tr>
<tr>
<td>$P_{opt}$</td>
<td>Optical Power</td>
</tr>
<tr>
<td>$c$</td>
<td>Speed of Light=$30 \times 10^8$ m/s</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Wavelength</td>
</tr>
<tr>
<td>$h$</td>
<td>Planck's constant in Joule-See</td>
</tr>
<tr>
<td>$c_{f}$</td>
<td>Zero-bias depletion capacitance</td>
</tr>
<tr>
<td>$c_{jsw}$</td>
<td>Sidewall zero-bias depletion capacitance</td>
</tr>
<tr>
<td>$A_D$</td>
<td>Area of diode</td>
</tr>
<tr>
<td>$P_D$</td>
<td>Periphery of diode</td>
</tr>
<tr>
<td>$V_{DB}$</td>
<td>Voltage across the diode</td>
</tr>
<tr>
<td>$p_b$</td>
<td>built-in potential</td>
</tr>
<tr>
<td>$m_j$</td>
<td>grading coefficient</td>
</tr>
<tr>
<td>$p_{bsw}$</td>
<td>built-in potential of the sidewall</td>
</tr>
<tr>
<td>$m_{jsw}$</td>
<td>grading coefficient of the sidewall</td>
</tr>
</tbody>
</table>

V. CMOS image sensors

V.1. Introduction

CMOS image sensors consume far much less power and are much quicker than CCDs. They are of two types: passive pixel sensors (PPS) and active pixel sensor (APS).

Active pixel sensors are more widely used than passive pixel sensors, as the latter have numerous drawbacks. The main area of research is now CMOS APS.

The basic block diagram of a CMOS image sensor is shown below in Fig.6. [22]. The CMOS image sensor block diagram has key components like an array of pixels which can be made of any CMOS image sensor according to the size of the array such as 1,024 x 1,024 photodiode based active pixel sensor (APS) array or photogate APS. The row decoder reads the rows of the array one by one using a timing control. The noise of the pixel is reset by the photodiode sensing node while the row decoder accesses one particular row. Column amplifiers provide signal strength to the output of the pixel before it is being read by the column decoder. The analog multiplexer, sample and hold circuit and correlated double sampling circuit (CDS) are provided on one block for temporary memory storage and to remove any unwanted signal.
V.2. Passive pixel sensors

The passive pixel sensor was introduced by Weckler in 1967 [23]. The functioning of a PPS is similar to the one of a single-transistor DRAM cell [24]. Though PPS is not as efficient as APS and CCD, PPS has one important advantage: a high fill factor obtained with a lower area, leading to high quantum efficiency. The schematic of a PPS is shown in Fig.7. PPS contains a photodiode and a transistor (called thin film transistor or TFT). The TFT enables us to select one particular pixel in a row and also to help to perform a reset of this pixel. There is a charge integration amplifier (CID) for each column of the array used to keep the voltage constant on each bus of the array in the PPS. PPS has three operation modes:

1: The reset mode.
2: The integration mode.
3: The readout mode.

In the reset mode, the photodiode is simply emptied of its information. This is done mainly before the integration phase. In the integration mode, the TFT is switched off and the charge is accumulated in the photodiode. In the readout mode, the accumulated charge is read out. The PPS has a major drawback due to its large capacitive loads, as the larger bus is directly connected to each pixel during its readout. PPS cannot be used for large array sizes or for fast pixel readout. Indeed, it causes fixed pattern noise (FPN). Researchers are still working on the implementation of PPS arrays with FPN reduction techniques [26]. PPS is also used in medical imaging [27]. PPS is therefore more useful when using pixels of small size and their fabrication is less costly than for APS.
V.3. Active pixel sensors

Active pixel sensors (APS) appeared when Fossum introduced them in 1992 [28] in the NASA’s Jet laboratory. APS overcomes the problems of PPS. APS has one or more active transistors used to buffer, amplify and read out the signal going out from the photodiode. APS is advantageous since it consumes less power and costs less than CCD and has attracted researchers to work on its numerous applications like webcams, automobiles, digital cameras, toys, computer based video games, etc. Power optimization, reduction of dark current [29, 30, 31, 32, 33] are some attributes that enticed researchers in the present portable appliance scenario.

APS also has some other advantages over CCD such as:

1: High sensitivity.
2: Typical fill factor from 50 up to 70%.
3: Compatibility with CMOS technology leading to system-on-a-chip (SoC) solutions.
4: Lower cost.
5: High readout speed.
6: Lower power consumption.

The basic framework of an array of APS is shown in Fig.8. The core part of the APS architecture is the APS array connected to a row decoder. The column of the array is selected thanks to a column decoder. The signal coming out from the selected pixel is amplified before being converted from an analog signal to a digital signal.
There are three types of APS:

1: Photodiode APS.
2: Photogate APS.
3: Pinned photodiode APS.

Photodiode APS and photogate APS are the most often used of these three APS types.

VI. Summary

We have here discussed the basic types of silicon photodetectors present in the digital cameras in the consumer electronics. We saw that CCD does not fit our project. Below (table 4), are shown the main differences between APS and PPS. For our application, i.e. a small array (64 x 64 or 128 x 128, the two sizes we hesitated between) of pixels being as cheap as possible, having a quite good quantum efficiency, and whose pixels will be read at a reasonably low speed, the passive pixel sensor is the solution. In our case, as we will see later in our final solution, the photodiode and its electronics (compression algorithm circuit non-included within this electronics) will be extremely simple since we do not require any addressing to select photodiodes one by one. It will thus only include one transistor to reset the photodiode (please refer to chapter 7, § IV.2.1., Fig.9.).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>APS Sensor</th>
<th>PPS Sensor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Larger</td>
<td>Smaller</td>
</tr>
<tr>
<td>Fill Factor</td>
<td></td>
<td>Good</td>
</tr>
<tr>
<td>Quantum Efficiency</td>
<td></td>
<td>Good</td>
</tr>
<tr>
<td>Fabrication Cost</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Larger Pixel Array</td>
<td>Good</td>
<td></td>
</tr>
<tr>
<td>Capacitance Load</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Readout Modes</td>
<td>Fast</td>
<td>Slow</td>
</tr>
</tbody>
</table>

Table 4. Main differences between APS and PPS.
I. Introduction

The success of 2D sensors development, as we have seen in part 1 of this chapter, has permitted relatively high levels of integration, so that several blocks, such as timing control, PGA and ADC can be provided together with a 2D sensor array (for example and in our case a CMOS array) on a single integrated circuit chip.

Nowadays, many devices are acquiring images. These images being larger and larger, and the memory size increasing more slowly than the images size, researchers have thus been pushed to develop image compression algorithms which were then implemented at the hardware level.

All this enables to have the sensor array and the compression circuit on the same chip.

Image compression is finally a really young science since it born roughly twenty years ago. Most of papers on the subject are thus very recent. Still, many compression techniques have been invented since that time.

We are now going to see the state of the art for the compression of images coming from arrays of sensors.

II. Various techniques for compression

II.1. Compressing the information from a single pixel or a block of pixels

Merrill and al. developed a photodetector array in which each photosensor is coupled to a nonlinear capacitor arranged to have a compressive photocharge-to-voltage gain function. An amplifier having an input and an output is coupled to the nonlinear capacitor [34]. Therefore, this compression solution performs the image compression at the source, i.e. at the pixel level.

Chen et al. developed an image sensor array whose pixels can be read out in $M \times N$ blocks that are compatible with the operation of a desired image compression algorithm, thereby reducing the amount of memory required by the image compression algorithm [35]. With this technique, we can thus compress the information coming from several pixels instead of only one.
II.2. Performing an operation on all pixels and storing the sum of all results

Researchers in the Rice University built a prototype of a new camera architecture based on a digital micromirror device (DMD) with the new mathematical theory and algorithms of compressive sampling (CS) which are quite simple but hugely efficient compression algorithms. CS combines sampling and compression into a single nonadaptive linear measurement process. Rather than measuring pixel samples of the scene under view, inner products between the scene and a set of test functions are measured. Interestingly, random test functions play a key role, making each measurement a random sum of pixel values taken across the entire image. When the scene under view is compressible by an algorithm like JPEG or JPEG2000, the CS theory enables to stably reconstruct an image of the scene from fewer measurements than the number of reconstructed pixels. In this manner they achieve sub-Nyquist image acquisition. Their “single-pixel” CS camera architecture is basically an optical computer (comprising a DMD, two lenses, a single photon detector, and an analog-to-digital converter) that computes random linear measurements of the scene under view. The image is then recovered or processed from the measurements by a digital computer [36].

When they speak about random sum of pixel values taken across the entire image, they mean that to make a so-called measurement they multiply each pixel of the image by a pseudo random coefficient being 1 or -1 (there is another solution using coefficients -1, 0 and 1). They use a random number generator which is implemented by a 10 bit maximal-length linear feedback shift register (MLFSR) to generate these coefficients. The MLFSR has the benefit of providing a random sequence of ±1 with zero average, while offering the possibility of regenerating the same sequence again given the initial seed. This feature allows the decoder to re-generate the pseudorandom sequence in the reconstruction algorithm. The MLFSR is reset to its initial state every time frame, which is the period of time that is captured from the simulations and fed to the frame-based reconstruction algorithm. The time-frame based operation imposes synchronization between the encoder and the decoder for proper signal reconstruction. To identify the beginning of each frame, header bits can be added in the beginning of each data frame in order to synchronize the decoder [37].

III. Previously encountered problems

The previously developed devices including arrays of photodetectors and a circuit for acquisition and compression of the images encountered many problems.
The most important problem (occurring in the solution proposed in § II.2.) is the correlation between the coefficients used to obtain a first measurement and the ones used to get the following measurement (in the time). It should be minimized as much as possible to improve the image quality of the rebuilt image. A solution to reduce it is to divide the array into several parts that all use a different PRNG. This idea was found during a meeting with the signal processing laboratory and will be taken into account in chapter 5 (§ VII.).

Another problem is the limitation on the dynamic range of images that can be captured by the array. Images that contain both low-light-level pixels and high-light-level pixels could be improved if the dynamic range of the imager could be increased [34].

Another non negligible problem is the interferences between neighbouring photodiodes that must be minimized. In our system, they will be limited by insulating photodiodes from one another thanks to rings of metal and diffusion layers surrounding each photosensor of the array (see chapter 7, § I.2.). These rings will just be slightly opened to be able to connect the anode and cathode of the photosensors to the rest of the circuit.

IV. Conclusion

For us, the best compression technique is the third one (§ II.2.).

Indeed, the first technique uses a capacitor. Such components are quite large in area and their value, in Farad, is always known with a more or less good tolerance which is not acceptable for us since we require a maximal precision for the compressed image to be as close as possible to the original one.

The second technique is telling that we can gather the pixels of the array and perform the compression of the information contained in these blocks. For us, gathering pixels in blocks presents advantages like reducing the correlation between coefficients. Therefore, as we told before, we will cut the array into P parts (P is determined in chapter 5, § VII.1.1.2.) and we will associate a different PRNG to each of these P parts.

The last technique is the most attractive one. It is one of the most recent and efficient. It has never been used to compress an image coming from a photodiode array and therefore there is no chip with a sensor array plus a circuit implementing this compression algorithm.

In this project, we will thus be the first ones putting these two elements on the same chip. As we will see in chapter 3, we tried to perform the compression with two and three different random coefficients (-1 and 1 and -1, 0, 1 respectively) using LFSRs. We have even tried (see appendix 6, § A6.3.) to store all needed coefficients in a large memory. This was working but given up because requiring too many measurements to achieve an acceptable image quality. We will see why we finally chose the solution with only two coefficients.
References for chapter 2

[3] Carrier Generation and Recombination:
[16] CMOS vs CCD and the Future of Imaging:
[21] Photodiode Characteristics and Applications:


[37] Sami Kirolos, Jason Laska, Michael Wakin, Marco Duarte, Dror Baron Tamer Ragheb, Yehia Massoud, Richard Baraniuk, Analog-to Information Conversion via Random Demodulation, Dept. of Electrical and Computer Engineering Rice University Houston, TX, 2006.
CHAPTER 3: COMPRESSION AND DECOMPRESSION

We will study in details the functioning of the compression and decompression algorithms. A random number generation unit is required. A linear feedback shift register (definition in appendix 4 called LFSR) has been used (choice explained in chapter 2, part 2, § II.2. and IV.), which is the simplest, smallest and quite fast pseudo random number generator (PRNG). A 64x64 photodiode array has been chosen after many simulations for area considerations. Nevertheless, in appendixes 5 and 6, simulations concerning 64x64 and 128x128 (the two sizes between which we hesitated) photodiodes arrays are present.

I. Compression principle

I.1. The compression type

Compressive sampling (CS) has recently emerged as a very efficient data compression technique, exploiting relaxation of the Shannon sampling criterion. Sub-Nyquist rate sampling requires complex reconstruction at the receiver, but is hugely reducing the complexity in data acquisition, as well as the communication throughput. These are reasons why we used this compression type in our project.

I.2. What was tried to be done

Each pixel of the array was multiplied by its own pseudo random coefficient -1, 0 or 1 having respectively probabilities of 1/6, 2/3 and 1/6 to appear. The results of these multiplications were summed together. This sum and the LFSR seed where stored: these two gathered elements are called a measure.

The latter coefficients with the latter probabilities were generated by comparing the output of the LFSR with two thresholds: when this output was in a given interval, the generated coefficient was 0, when in another one it was 1 and when in a last one it was -1 (see appendix 5). It was found that increasing the number of thresholds (and thus the number of intervals) was increasing the PRNG randomness (see appendix 6). This solution was given up because too hard to implement at the hardware level.

Another way to obtain these coefficients with these probabilities would have been to add some logic whose inputs would have been the N outputs of the N bit LFSR and to build this logic to obtain a probability of 1/6 for YZ=10 (1 coefficient) and YZ=01 (-1 coefficient) and 4/6 for YZ=11 (0 coefficient) and 0 for YZ=00 (forbidden state), where Y and Z are the
outputs of this logic. This would have implied to build an algorithm being able to determine this logic by a trial-correction method. For example, all the N outputs of the LFSR are used, and a random logic circuit (with the above characteristics) with n stages is generated. Then, the probabilities of occurrence of YZ output patterns (being 00, 01, 10 or 11) are computed. If the obtained probabilities are close enough to the expected ones, then the circuit is accepted, otherwise it is slightly modified and the process starts again till probabilities specifications are met. This process would have been hard and time consuming to implement.

It was given up since we found a much simpler method (see I.3.).

I.3. What was finally chosen and why

What we finally did for the PRNG is the same thing as what we did in § I.2. but with only two coefficients (-1 and 1) with equal probabilities (1/2). A 0/1 at the output of the LFSR generates a -1/+1 coefficient.

II. Decompression principle

The decompression algorithm starts with a set of nb_meas measures and does the compression reverse operation to recover the initial image. The higher the number of measures is, the higher the compressed and then decompressed image quality is. The aim, in this compression technique, is to keep the number of measures much lower than the number of pixels. The optimum number of measures is computed in Appendix 5 (35% of the number of pixels). The simplest of this decompression algorithm will be used. But, normally, a very complex algorithm should be used to really efficiently recover the acquired image.

III. Compression and decompression algorithm main blocks

The algorithm is composed of the following code blocks:

1. Matlab Master code block:

   It specifies a 64x64 (8 bits, ASCII) image name, a LFSR-4 structure (specifying the LFSR number of bit N and LFSR taps locations), a N bit binary seed, and the number of measures (nb_meas).

   It returns the PSNR between the source image and the compressed and then decompressed image, the image number of pixels (nb_dim) and many of its inputs.
2. Matlab pgma_read code block:

This block converts the raw PGM image into a column vector X with nb_dim rows.

The two following blocks were written in C, C which is able to perform loops much faster than a Matlab code. These C files were turned into mex files able to communicate with Matlab typical M-files.

3. C code multlogiLFSR block:

This block takes as inputs X, the seed, the structure of the LFSR and nb_meas.

The PRNG is integrated within this block. It is a LFSR-4 starting to work with the above defined seed and then works as a normal LFSR generating a binary number sequence. The coefficients were generated as explained in the § I.3.

For each integer value i between 1 and nb meas, a series of nb_dim coefficients are generated.

The j\textsuperscript{th} coefficient of a series multiplies the one in the j\textsuperscript{th} row of X. This is done for all j in \([1:\text{nb_dim}]\). The sum of all these multiplications is stored in the i\textsuperscript{th} row of a column vector Y of dimension nb_meas. This is thus done for all i in \([1:\text{nb_meas}]\).

If all of the nb_meas coefficient row vectors of dimension nb_dim where tidied in a matrix A (thus of dimensions nb_meas \times nb_dim) then we would simply have: Y=AX.

This block then returns the value of Y.

4. C code multadlogiLFSR block:

This block takes as input Y, the seed, nb_dim and the structure of the LFSR.

From the seed, the C file multadlogiLFSR regenerates the coefficients and builds the matrix \(^1A\) and computes the matrix product \(^1AY\) to obtain X_new.

This block returns X_new (compressed and then decompressed image or rebuilt image).

Finally, the outputs are the plot of the reshaped X_new matrix into a matrix having the same dimensions as the image and the PSNR between X and X_new to evaluate the quality of the compression and decompression. Below (Fig.1.) is the obtained block diagram of the algorithm presented here.
Fig.1. The algorithm main blocks

The Matlab and C codes for each of these blocks (pgma_read, multlogicLFSR, multadlogicLFSR and Matlab master) are all given in appendix 7. Please refer to the appendix part at the end of this report (page 93).
CHAPTER 4: SYSTEM OVERVIEW

The system is basically composed of three parts: a 64 x 64 photodiode array, a digital control circuit and an analog signal processing unit. It also includes some pads whose aim is to enable us to access the inputs and outputs of our system.

I. The array

The role of this array is to sense the image arriving on its surface. In our electronic system, the array plays the role of the 64 x 64 image in the algorithm explained in chapter 3. In our final solution, as we will see in chapter 7, the array is more than that: it is a juxtaposition of 4096 0.35µm CMOS technology photodiodes, each of them being accompanied by a logic circuit helping to perform the multiplication of the content of the photosensor by its coefficient (-1 or +1). This logic will be associated to an operational amplifier located outside the array in a way that this logic with this operational amplifier will perform the complete multiplication and then summation operation (see chapter 6, part 1, § II.). This logic contains one memory point able to store one bit. The image compression is thus done within the array. The array has some inputs: two for power supplies (GND and VDD), two for reset signals (one for the photodiodes and one for the memory points) and one for the clock (needed by memory points). It has two outputs per column. Each of these two outputs goes to the corresponding input of the operational amplifier associated to this column and located in the analog signal processing unit block.

II. The digital unit.

The role of this digital unit is to master the image compression. It commands the scanning of the array, the generation of pseudo random coefficients and the association of a coefficient to each pixel of the array. This scanning of the pixels of the array and the association of a coefficient to each of them is done nb_meas times (to get nb_meas measures). This finally gives an image. The number of times all this is done is equal to the number of images we desire.

This unit, as we will see in chapter 5 where we present our first solution for this block, is composed of many parts: a clock divider, a seed generation circuit, a control unit, a LFSR and control unit clock generation block, a 12 bit decoder, a LFSR, and 4096 memory points. This digital unit still has, as a future work, to be modified to go along with our final solution for the framework integrating a memory point and switches within each pixel of the array.
III. The analog signal processing unit.

The role of this block is to perform the multiplication of the pixels by their coefficients (to do that, we also require and use two switches located within each pixel), to amplify the current from one column (see why in chapter 6, part 2, § I) and to convert this current into a voltage, voltage that will then be digitalized thanks to an ADC (not built in this project since we will just choose one meeting our specifications in a catalogue). This analog unit contains 64 fully differential operational amplifiers (one per column of the array) and each operational amplifier is associated to some resistances and some capacitors as we will see in chapter 6. Such an operational amplifier has two inputs and two outputs called inverting and non-inverting inputs and outputs. We have just begun to design the operational amplifier. We have built some models but have not sized its components yet. We will also need to build a digital block to multiplex the 128 outputs of these 64 devices to only two outputs.

The system described in this chapter will be integrated on a single chip. An overview of it is shown below in Fig.1. All its various blocks will be described more in detail in the corresponding chapters located in this report.

Fig.1. System overview.
CHAPTER 5: THE DIGITAL UNIT DESIGN: A FIRST IDEA

I. Introduction

We will here present our first idea for the digital part corresponding to the case were coefficients are stored outside the array within memory points. Another digital part, that will replace this one, has to be designed since we finally decided to store coefficients within the pixels (see chapter 7, § III.1.) given that it tremendously simplifies the system. This new digital unit will not be shown here but in the next report. Anyway, It has many similarities with the one shown here as briefly explained in chapter 7.

The fully digital control system (whose aim is to master the compression of the acquired data from the array) elaboration is made of seven main parts: a clock divider block, a SLF and seed signal generation circuit, a control unit, a 12 bit decoder, a LFSR and memory and control unit clock generation block, a LFSR, a 1 bit memory and many additional components (see appendix 8). All the VHDL codes for each part are shown in appendix 9.

For each part, structure, simulation results, and synthesis results will be detailed.

II. Control system first part: clock divider

This block has, as input, the clock signal Clk_fast and has, as outputs, the master clock Clk of the system and the refresh clock signal for the finite state machines (FSM). Clk_refresh and Clk have a period of Clk_per/2 and Clk_per/4 respectively, Clk_per being the clock period of Clk_fast.

This block, described in VHDL, was simulated, synthesized and post-synthesis simulated successfully.

III. Control system second part: seed and SLF signal generation system

III.1. Structure description

The aim of this system is firstly to generate the LFSR seed. This seed is alternatively equal to 0 and 1 (value changing after each period of the clock Clk). Secondly, its aim is to tell to other blocks when the seed has been loaded into the LFSR.
The SLF generation system has:
- 5 inputs:
  - Bit_Nb_SEL: signal used to select the LFSR length.
  - Load_seed: externally defined signal telling the seed sending has to begin (when at 1) or telling the LFSR is used in its normal operation mode (when at 0).
  - Clk: master clock.
  - Reset: active low, resets the FSM to its IDLE state.
- 2 outputs:
  - SLF: signal telling the seed was loaded when at 1.
  - MEM_out: the generated seed.

The I/O (input/output) diagram is thus given in Fig.1 below.

![Fig.1. System I/O diagram.](image)

This system gathers a system determining a signal Threshold from Bit_Nb_SEL, a Moore FSM, a counter, and a comparator (see Fig.3.).

The threshold is equal to the LFSR length (in binary).

The FSM (Fig.2.) uses Load_seed, Reset and Equal to generate transition conditions. It has 3 states. In the IDLE state, some initialisations are done (SLF and MEM_out are 0 and the counter is reset). In the State1 state, SLF and MEM_out are at 1 and the FSM sends a signal at logic 1 (SLF) to increment the counter. In the State2 state it is the same thing except that MEM_out is at 0. The system will oscillate between State1 and State2 and MEM_out will be alternatively 0 and 1. The FSM goes to the IDLE state when Equal is at 1.

The counter (Count) is incremented each time SLF is at 1 and reset when at 0.

The comparator compares Threshold and Count. It supplies a signal Equal at logic 1 when they are the same and 0 otherwise.
III.2. Simulation results with Modelsim

In Fig.4, only simulation results for a 12 bit LFSR are shown (see why in § VII.).

The reset is initially active for two clock periods. The signal Bit_Nb_SEL is set to have a 12 bit LFSR (equal to 101, see why in § VII.2.1., Fig.29.). Threshold is deduced from Bit_Nb_SEL. The Load_seed signal is active for three clock periods. The system responds by a SLF at 1 during the seed sending and at 0 elsewhere. Count is incremented till it reaches the number of bits of the LFSR. The FSM correctly runs across the various states. The block correct functioning has thus been checked. The next step is synthesis.

III.3. Synthesis results

The SLF and seed generation system synthesis inferred memory devices are shown in Fig.5. No latches were inferred, thus giving a safe and fast design. Five flip-flops are inferred for the counter and five others for the threshold. Two are created for the states since the FSM contains three of them. The obtained area is of about 8260µm² and the power consumption of 3.77mW.
IV. Control system third part: control unit

IV.1. Structure description

The control unit (CU) goal is to count the number of rising edge in its clock Clk_memory. At each rising edge, the CU selection signal selects the next pixel in the array. As soon as $64^2$ rising edges have been detected, all pixels have been scanned and one measure has been done, and as soon as a given percentage of the number of pixels measures have been done (for us 35%), one image is generated.

The CU has:
- 4 inputs:
  - Reset: active low signal, resets to the IDLE state.
  - Clk_CU: Control unit clock.
  - SLF.
  - nb_meas: an externally applied 12 bit word to specify the number of measures needed to build one image.
- 3 outputs:
  - Pixel_selection: a 12 bit word used to know which pixel of the array is concerned by the coefficient currently generated by the LFSR.
  - Meas_done: tells that one measure has been done when going to logic 1.
  - Im_done: tells that one image has been completed when going to logic 1.

Thus, the CU I/O diagram is given in Fig.6.
The CU is composed of a Mealy FSM, two counters and two comparators (see Fig.7.). The FSM (Fig.8.) uses signals Reset, SLF, Im_done and Meas_done to generate the transition conditions. In the IDLE state (initial state), both counters are reset. In the Select_pix state, if both Im_done and Meas_done are at 0, the pixel counter is incremented. If both at 1, it is the same thing but the measure counter is reset. If only Meas_done is at 1, both counters are incremented.

The role of the pixel counter (Count_pix) is to generate a signal telling which is the pixel currently concerned. The measure counter (Count_meas) counts the number of times the array has been scanned.

The first comparator tells us if Count_pix reached 4,096. If yes, Meas_done is at one (a measure has been done). The second comparator says if Count_meas is equal to Number_meas. If yes, Im_done goes to 1 (an image has been done).

**Fig.7. System block diagram.**

**Fig.8. FSM of the control unit.**

**IV.2. Simulation results with Modelsim**

- Below (Fig.9.) is a zoom on what we have at the beginning of the simulation. SLF has been set to 1 (load seed) and then 0 (seed loaded). The reset has been set to 1 for it to be inactive. Count_pix is incremented. The Control_Unit is thus selecting one pixel, and the next clock period, the following one till reaching the end of the array. The number of measure and of image is still 0 since the end of the array has not been reached yet.

**Fig.9. Simulation of the control unit: zoom on initial waveform.**
- Below (Fig.10.) is what happens when a measure is done. Count_pix reaches 4,096 and thus the last pixel is selected. Meas_done goes to 1 for one clock cycle.

![Fig.10. Simulation of the control unit: zoom on Meas_done='1'.](image1)

- Below (Fig.11.), the simulation shows the presence of four measurements as indicated in the simulation with nb_meas=(000000000100)₂. After four (but normally 1,434) measurements, an image is ready (peak in the Im_done signal).

![Fig.11. Simulation of the control unit.](image2)

- Below is a zoom on the peak in the signal Im_done. One measure ends at the same time.

![Fig.12. Simulation of the control unit file zoom on Im_done='1'.](image3)

We thus verified that, regularly, Meas_done and Im_done go to logic 1. We have verified the proper functioning of the control unit. A system to generate Clk_LFSR and Clk_CU is required.
IV.3. Synthesis results

During the optimization and synthesis, no latches were inferred, thus giving a safe design. 12 flip-flops for each counter are inferred and one for the states since the FSM contains two states.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>RB</th>
<th>AR</th>
<th>RS</th>
<th>BS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter_reg</td>
<td>FLPF</td>
<td>12</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>state_reg</td>
<td>FLPF</td>
<td>1</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Pixel_sel_reg</td>
<td>FLPF</td>
<td>12</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Fig.13. Inferred memory devices.

V. Control system fourth part: LFSR and control unit clock generation circuit

V.1. Structure description

The aim of this system is to generate the clock signal for the LFSR (Clk_LFSR) and the one for the control unit and memory (Clk_Memory) from the refresh clock (Clk_refresh) and the master clock (Clk). Clk_LFSR and Clk_Memory will not have to run at the very beginning (when we do the reset). Then, when the reset is done, the seed is sent. At each new bit of the seed, the LFSR must shift its content towards its output. Thus, Clk_LFSR must have one rising edge each time a bit of the seed is sent into the LFSR. But, the coefficients should not be loaded into the memory during this phase. So Clk_Memory should not run. Afterwards, when the seed has been sent, Clk_LFSR and Clk_Memory run normally without interruption for the LFSR to generate one by one the coefficients for the pixels.

The system has:

- 4 inputs: Clk, Clk_refresh, SLF and Reset (signal to reset the circuit to its IDLE state).
- 2 outputs: Clk_LFSR, Clk_Memory.

Fig.14. System I/O diagram.
The system is a simple FSM (Fig.15.) with three states. It uses signals Reset and SLF to generate the transition conditions. In the IDLE state, Clk_memory and Clk_LFSR are set to 0. In the IDLE1 state, Clk_memory is at 0 and Clk_LFSR identical to Clk. Finally, in the STATE1 state, both clocks are identical to Clk.

V.2. Simulation results with Modelsim

The FSM machine was simply written in VHDL and simulated with Modelsim.

- Below (Fig.16.) is what we obtained at the beginning of the simulation for a 12 bit LFSR. While the reset is active, both clocks are at 0. Then, when SLF is at 1 (load the seed), we notice 12 rising edges in Clk_LFSR to make the incoming bits of the seed enter the LFSR. In the meantime, Clk_Memory is inactive. Afterwards, both clocks have one glitch due to the change of state (going from IDLE1 to STATE1) and therefore due to a sudden change of signals assigned to both clocks. Finally, both clocks are identical to Clk to generate and load the coefficients into the memories.

The LFSR and memory clock generation system works as we wanted it to work. Its functioning was therefore checked.
V.3. Synthesis results

During the synthesis, only flip-flops were inferred and their number is minimal (only two were inferred since the FSM has only three states). As a consequence, a safe, stable and controllable system was obtained. The area was of about 1550µm² and the power consumption of about 1.59mW.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>AR</th>
<th>RS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>state_reg_reg</td>
<td>flip-flop</td>
<td>2</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Fig.17. Inferred memory elements.

VI. Control system fifth part: 12 bit decoder

VI.1. Structure description

To be able to choose among $64^2 = 2^{12}$ pixels, we need a 12 bit decoder. Sending $(00000001001)_2$ at the input of this decoder, we will select the 9th pixel of the array i.e. the 9th output of the decoder (there are 4096 outputs in total) will be at logic 1 and all the others at logic 0. The VHDL code of such a component was written with Modelsim. The 4,096 required lines of code were written thanks to a Matlab routine.

The VHDL code of this 12 bit decoder will not be shown because even the simulation on Modelsim was successful, the synthesis failed because the number of gates (increasing exponentially with the decoder number of bits) was too high.

The solution to get round this obstacle was thus to divide this decoder into several (65 in total) 6 bit decoders. The scheme of the finally used circuit is shown in Fig.18.

The signal Pixel_selection_12bits (11 down to 0) is used to select the pixel. The Decoder_12bits_out (4,095 down to 0) signal will be used to control each of the 4,096 pixels.

To model this system in VHDL, we first modelled the Decoder_6bits component as a simple 6 bit decoder. This component was instantiated to create a second component composed of one 6 bit decoder and 64 AND21 gates. The inputs of this component were Pixel_selection_12bits (5 down to 0) and one Decoder_6bits_out signal. Its outputs were decoder_12_bits_out (i+64 down to i, where i goes from 1 to 64 and is the considered decoder_6bit number). Finally, these two components were compiled, simulated and instantiated and a generate statement was used to build the entire 12 bit decoder.
VI.2. Simulation results with Modelsim

- Below is the simulation of the 12 bit decoder. We see that the decoder outputs are at one the
ones after the others and this from pixel n° 0 to pixel n° 4,095 as shown in Fig.19. and 20.
When the last pixel (n° 4095) is reached, the process starts again from the beginning and this
as long as the clock runs. The selection of 4,096 pixels one by one with a clock period of 3ns
lasts about 12.288µs. It is also the length of one measure. To make one image with 1,434
measures one requires about 17.62ms. Thus, we get 56.8 images per second. The clock period
can be increased (to about 43ns) to obtain, for example, only four images per second.

We have thus checked the proper functioning of the 12 bit decoder. The next step is the synthesis.
VI.3. Synthesis results

We noticed that the 12 bit decoder contains very few cells (162 gates like NOR, INV, NAND…) and a lot of wiring which is logical since we want to select one of 4,096 outputs with a 12 bit number. The area is of about 450,000µm² and the dynamic power of 377mW.

VII. Control system sixth part: The LFSR or PRNG

VII.1. The PRNG structure search

We have to use only two coefficients (-1 and +1) and we must try to cut the array into P parts associated to different PRNGs to remove the correlation between coefficients in successive measures. These were above simulations and meetings results.

VII.1.1. Dividing the array into P parts

VII.1.1.1. Principle

Below (Fig.21.), the array of pixels is shown in four successive situations. In this example, we decided to cut the LFSR into four parts. Four different orthogonal LFSR are used. In this technique, each LFSR works for a different part of the array at each measure enabling to completely de-correlate coefficients between successive measures.

![Fig.21. Using a LFSR for each part of the array.](image)

VII.1.1.2. Number of parts determination

It is a bit tricky to determine the number of parts in which the array should be cut into and thus the number of required LFSRs. To do that, Matlab simulations previously written were reused and modified, and the following steps were followed:

- Firstly, a unique N bit LFSR was used to generate coefficients of all pixels during one measure. Having an image of 64 x 64, requiring 35% of the number of pixel measures (1,434) to make one image and requiring $2^N > \text{nb\_meas}$ to make the compression successful, a 11 bit LFSR was required (results obtained from the simulation job shown in appendix 5).
- Secondly, the master Matlab code and the two mex files were modified to work with a 11 bit LFSR (Fibonacci implementation with taps in 11th and 9th positions), to only use the bit going out from the 11th output of the LFSR to work with only two different coefficients (logic 0 gives the coefficient -1 and logic 1 gives the coefficient +1).

- Thirdly, the probabilities to obtain either 0 or 1 at the 11th output of the LFSR were computed. They were respectively 0.4998 and 0.5002. Thus, logic 0 and 1 can be considered as equiprobable, important fact to be checked for a good functioning of the algorithm.

- Fourthly, a column matrix X with 64 rows filled with logic 0 except in the kth row (k being equal to 32 here) was the input of the compression algorithm. This matrix was compressed and decompressed and X_new was obtained. Matrixes X and X_new were plotted (see Fig.22.). This enables us to have a first glimpse of the correlation between coefficients. As X and X_new are almost perfectly superimposed and this much better than in previously done experiments (with three coefficients), the correlation between coefficients of two successive measures seems to be low at first view.

- Fifthly, this low correlation was checked. The following steps were followed:

1. For k varying from 1 to 64, X_new is computed. Each obtained X_new is stored inside a column of a matrix A of dimensions 64 x 64.
2. The diagonals of A (Fig.23.) are extracted. The central one is called D_0. The ones located above D_0 are called D_1 to D_{63}, and the one below D_0 are called D_{-1} to D_{-63}.
3. The mean values of absolute values of each of these diagonals (i.e. the M_p=mean(abs(D_p)) with p in [-63;63]) are computed and stored into a column matrix D with 127 rows.
4. A gathering of diagonals around $D_0$ is chosen. We gather diagonals $D_{-10}$ to $D_{10}$.
5. The mean value $\mu$ and standard deviation $\sigma$ of other diagonals are computed.
6. $T=\mu+3\sigma$ is computed.
7. All $M_p$ (with $p$ in $[-10;10]$) with values above the one of $T$ are counted.

**VII.1.1.3. Results**

The process in the fifth step of § VII.1.1.2. was described in Matlab. Only one $M_p$ ($M_0$) was above $T$, meaning it is not necessary to divide the array and that one LFSR should be used to determine coefficients 1 and -1 for all pixels, for all measures and for all images.

A 64 x 64 image was compressed and decompressed to check if the compression was operating correctly. Above (Fig.24.) is what was obtained for 1,434 measures.

We therefore see some periodicity in the image even if we fulfil the specifications established for the PRNG generating three coefficients. But here, when using only two coefficients, the functioning seems different and we have to study it further. Thus, we need another number of bits for the LFSR as we will see in the next paragraph.

**VII.1.1.4. Another number of bits for the LFSR**

As with the 11 bit LFSR the compression does not work, a code was written and simulation plots were obtained to see for which LFSR number of bits the compression works. The image was compressed and decompressed for 16 different LFSR number of bits (3 to 21 by steps of 3) and for a worst case of 3300 measures. Results are shown below (Fig.26-28).

The best image was obtained with a 12 bit LFSR. Indeed, the PSNR reaches 7.6dB and much lower values for other LFSRs (see Fig.27.).

Then, the probabilities to get 0 or 1 at the LFSR output were plotted as a function of the LFSR number of bits (see Fig.25.). With the increase of this number, these probabilities tend to 0.5 as expected. They are close to 0.5 for 12 bits (see cursors in Fig.25.).
Finally, $X_{\text{new}}$ was plotted as a function of $X$ to see if the noise was not too high for 3300 measures (see Fig.28.). We see that $X_{\text{new}}$ and $X$ superimpose correctly in the 12 bits case. The number of $M_p$ above $T$ was checked too. It was of 1.

VII.1.1.5. Conclusions

It is thus required to build a unique 12 bit LFSR and a seed generation system. A multiplexer will be created to choose either this generated seed ($\text{MEM}_{\text{out}}$) or an externally supplied one ($\text{Seed}_{\text{applied}}$).
Fig. 27. The PSNR between $X$ and $X_{\text{new}}$.

Fig. 28. $X_{\text{new}}$ plotted as a function of $X$. 
VII.1.2. A safer solution

If we want, in the future, for a given reason, to modify the LFSR number of bits, because the compression fails completely in reality with a 12 bit LFSR, then a variable length LFSR has to be built.

VII.2. The LFSR design

VII.2.1. Framework

The LFSR length can be of 3, 6, 9, 12, 15, 18 or 21 bits. The length was varied by steps of 3 and not 4, because with 4, the obtained LFSRs (4, 8 …, 24 bit LFSRs) most required the use of four taps thus leading to more connections for a lower degree of flexibility.

21 D-flip-flops connected in series were used. Then, 7 XOR gates were used to supply feedbacks needed by each LFSR. These feedbacks are the inputs of an 8 to 1 multiplexer with a 3 bit selection signal Bit_Nb_Sel (selection table in Fig.29.) permitting to select the LFSR length. The 8 to 1 multiplexer output is one input of a 2 to 1 multiplexer. The other input of the latter, is the seed. This 2 to 1 multiplexer output is the LFSR input (i.e. taps 0). The LFSR output is a multiplexed signal of LFSRs of each length (signals in taps 3, 6, 9, 11, 12, 15, 18 and 21) and is controlled by Bit_Nb_Sel.

A Fibonacci implementation with 2 or 4 tap LFSRs was chosen. Indeed, this implementation used only one XOR gate (2 to 1 or 4 to 1 XOR). Thus, the system will be faster than a Galois implementation.

Below (Fig.30.) is the schematic of this variable length LFSR:

<table>
<thead>
<tr>
<th>Bit_Nb_Sel(2)</th>
<th>Bit_Nb_Sel(1)</th>
<th>Bit_Nb_Sel(0)</th>
<th>Number of bits of the LFSR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Undefined</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Fig.29. LFSR number of bits according to Bit_Nb_Sel value.
VII.2.2. LFSR modelling

The framework was described in VHDL, simulated and synthesized. Some systems were added to turn off the non useful registers for the chosen LFSR length, thus saving power.

VII.2.3. Simulation results with Modelsim

Simulations were done for each LFSR length. Only the 12 bit LFSR results are shown here since it is the LFSR length with which we get the best results.

As seen in Fig.31, the seed \((111111111111)_2\) is loaded into the LFSR. When the loading is done, the LFSR generates one coefficient at each clock rising edge. We finally notice that the non useful registers (whose outputs are called input_13 to input_21) are turned off.

Fig.31. Simulation of the 12 bit LFSR with Modelsim.
VII.2.4. Synthesis results

The synthesis result is shown below (Fig.32.). The structure corresponds to the one in Fig.10. The area is of about 10,000µm².

Fig.32. Synthesis of the variable length LFSR with Design_Vision.

VIII. The 1 bit memory

VIII.1. Structure description

The one bit memory is the system that will store a LFSR coefficient (0 or 1) used latter to tell to the operational amplifier if the current from the pixel is multiplied by -1 or 1. Thus, such a system will be needed for each pixel. So, its area must be minimized.

Fig.33. Schematic of the one bit memory.

Fig.34. System I/0 diagram.

This memory is a simple flip-flop with one MUX21 and one transmission gate as shown in Fig.33. (when the selection signal is 0, then in the MUX21 Q=A, and in the transmission gate Z=A).
Its inputs are the Decoder_output(i) (one bit of the Decoder_output signal), LFSR_output, Clk_LFSR and Reset. Its output is a one bit signal Q supplying the coefficient (0 for –1 and 1 for +1).

VIII.2. Simulation results with Modelsim

The simulation results are shown in Fig.35. We verify that, when the Decoder_output signal is at one, then the coefficient generated by the LFSR and arriving on the LFSR_output input of this block is stored into the memory. If at 0, then we simply have a memory effect. Thus, the memory block works properly.

VIII.3. Synthesis results

The synthesis result is shown in Fig.37. The schematic is equivalent to the one in Fig.33. Thus, only one flip-flop was inferred. The obtained area was of about 680µm².

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>RB</th>
<th>AS</th>
<th>SP</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_reg</td>
<td>Flip-flop</td>
<td>1</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

IX. The whole system

IX.1. Structure description

As a final step, all the blocks above were gathered. The block diagram of the whole system is shown in Fig.38. Block diagrams of the various blocks are shown above.
IX.2. Simulation results with Modelsim

Afterwards, the system was simulated with a test bench gathering all the previous test benches. What we obtained below is simply a gathering of all the simulations obtained above.
Fig. 40. Simulation of the whole system with Modelsim: measure done.

Fig. 41. Simulation of the whole system with Modelsim: image done.
Fig. 42. Simulation of the whole system with Modelsim: whole simulation.

The only difference with the simulations seen in the previous paragraphs is a very fast scanning of the first two pixels during the first measure as seen in Fig. 39. by observing signals Clk_LFSR and decoder_12bits_out. Thus, the first measure should be discarded and we should only consider the next ones.

In this simulation, only four measures were asked to make an image for simulation length reasons. We can therefore see in Fig. 42, four peaks in the Meas_done signal and one in the Im_done signal.

Therefore, the whole system works fine as we checked the functioning of all the various blocks.
CHAPTER 6: ANALOG SIGNAL PROCESSING

PART 1: MULTIPLICATION BY THE COEFFICIENT

I. A first solution

A first solution using two current mirrors was used to multiply the photodiode current by 1, 0 or -1 (see Fig.1.). If the signals Control_1 and Control_2 are at 1 and 0 respectively, the current is multiplied by -1. Indeed, the six bottom NMOS transistors copy the current to the output and the transistor with the gate signal Control_1 authorizes the current to go to the output mult_cur. If Control_1 and Control_2 are at 0 and 1 respectively, then the input current is copied to the second branch and is allowed to go to the upper branch composed of PMOS. The input current is finally unchanged. Then, to sum the multiplication results of the pixel’s currents by their own coefficients we connected all mult_cur nets of all pixels together. This solution was given up because too area consuming in the pixel and also because the output resistance of the whole array was too low (several Ohms) and therefore the array output current was going into the array output resistance and not into the one in parallel with the OP AMP (several hundredth of Ohms) which is performing the current to voltage conversion (see Fig.2. but imagine that the array output current goes to the OP AMP + input and that the – input is plugged on a resistor in parallel with a capacitor, components equal to the array output resistance and capacitance to equilibrate the OP AMP + and - inputs). Thus the system was failing at the I-V conversion level.

Fig.1. Multiplication method with two current mirrors.
II. A better solution

A second solution is shown in Fig.2. The Q signal going out from the digital unit (see chapter 5, § VIII.1., Fig.33.) is used as a control signal of two transmission gates. If at one, the photodiode current goes to the OP AMP + input. Thus, the Mult&Sum+ voltage goes high while Mult&Sum- remains constant. Thus, the difference of the two previous voltages goes high. The current is so multiplied by +1 and converted into a voltage at the same time. If the Q signal is 0, the photodiode current goes to the - input of the OP AMP. Mult&Sum+ is constant and Mult&Sum- goes high and the difference goes down. The current is multiplied by -1. The sum is here performed by linking all nets controlled by Q and by linking all nets controlled by Q\ as shown in Fig.2. below. Here, we have no problem of too low output resistance since the output resistance of the pixel is the photodiode reverse resistance being of the order of TOhms. Consequently, this solution is successful.

Fig.2. Multiplication method with an OP AMP.
PART 2: THE OPERATIONAL AMPLIFIER: THE STARTED JOB

I. The OP AMP chosen structure

The OP AMP structure is from [1] and is shown below (Fig.3.). It is a two stage OP AMP composed of a telescopic cascode and of a common source circuit. With such a framework, very high gains and bandwidths can be reached. It can also achieve large output swing. And this is what we need since we require an as high accuracy as possible and this all along our circuit. Indeed, the accuracy is inversely proportional to $\beta A$ where $\beta$ is the feedback factor and $A$ the gain of the amplifier: we increase the gain, we increase the precision. A more complete structure from the same source [1] is shown in Fig.7. It includes another additional circuit called the CMFB (Common Mode Feedback circuit) used to make the design complete and to reduce the sensitivity of low frequency gain to common mode input voltage. Another advantage is that it gives us the flexibility of setting the common mode output voltage $V_{o,cm}$ and guarantee the maximum output swing with process variations [1].

We will use one OP AMP of this type per column of the array since a Matlab simulation showed us that the maximum current arriving at each input of the OP AMP is of 218mA when considering the whole array and only 6mA when considering only one column.

II. Determining the transfer function of the fully differential OP AMP

II.1. With a third order transfer function

II.1.1. Determining the output load capacitance

Determining the output load capacitance $C_{load}$ of the OP AMP, and thus the input capacitance of the sample and hold circuit of the ADC, will already give us the second pole position which is $P_2 = \frac{g_{m2}}{C_{load}}$ where $g_{m2}$ is the transconductance of the transistor M2. To find $C_{load}$ we did two things:

- We computed the minimum load capacitance to have a noise power less than -80dB and thus we solved this inequality:

$$P_{noise} < -80dB \iff 10 \log(4k_BT/C_{load}) < -80dB$$

where $k_B$ is the Boltzmann constant and $T$ is the temperature in Kelvin (taken equal to 300K).

Thus, we obtained that the load capacitance should be at least of 1.656pF at 300K.
We computed the minimum load capacitance to have an acceptable capacitor mismatch that is important for having a good precision for the ADC. To determine this, we first had to determine the required ADC resolution. A Matlab simulation was therefore built. We took the one already made before to compress and decompress an image of 64 x 64 pixels with a 12 bit LFSR as PRNG. Then, the only thing that was changed is the master Matlab code in which the obtained Y matrix (from the column image matrix X, with nb_meas) was taken and the matrix $Y_M$ was computed such that:

$$ Y_M = \frac{\text{Round}(Y.2^M)}{2^M} $$

Fig.3. Scheme of the OP AMP without the common mode feedback circuit.
where $M$ is the number of bits of the ADC and Round the rounding function of Matlab (rounding to the closest integer). This models the fact that we will not store the $\text{nb\_meas}$ analogic values in $Y$ but a series of $\text{nb\_meas}$ $M$ bit words. A loop on $M$ in the Matlab master code is then written and for $M$ varying from 7 to 16 the $Y$ matrix is computed, then $X\_new$ and finally the PSNR of $X$ and $X\_new$. At the end of this loop, 10 PSNR for the 10 different $M$ are obtained. The corresponding plot is the one in Fig.4. It shows the PSNR as a function of $M$ minus 6.

![Fig.4. PSNR of X and X_new as a function of M-6.](image)

Here we thus see a curve having a similar shape to $1-e^{-t/a}$ with $t$ increasing. The PSNR is thus better and better as the ADC resolution is increased. But, the latter should not be increased too much. Indeed, when the curve begins to be flat (on the right), the gain in precision by adding one more bit to the ADC will be ridiculous but the power consumption will tremendously go up. We thus choose a reasonable number of bits. Besides, in the 0.35$\mu$m technology, it is hard to build ADCs of resolutions above 10 bits. Thus, the best trade off according to the curve above is $M=10$. Therefore, for nice precision and low power consumption, the ADC will have 10 bits.

Then, with this, the mismatch $\delta$ should satisfy $3\delta < 1/2^M$, since the mismatch curve is assumed to be a Gaussian which is non null between $-3\delta$ and $3\delta$. Thus, $\delta$ must be below 0.004069%. Then, according to the technology paper from Austria Micro Systems (AMS)
called Matching parameters for 0.35µm technology, and choosing the CMIM (Metal-Insulator-Metal Capacitor) technology to realize capacitors (since CMIM capacitors are linear and not the ones built with the CPOLY technology), we have a linear relation between the mismatch δ in percents and $1/\sqrt{A}$, where A is the area of the capacitor. The slope is of about 0.95%µm. Thus, we can obtain A which is of about 54509µm² corresponding to a capacitor of 68.14pF (1µm² corresponding to 1.25fF according to the document ENG-182 for 0.35µm AMS technology).

Finally, the load capacitance is equal to the largest of the two capacitances computed above and is thus equal to the mismatched deduced capacitance: $C_{\text{load}}=68.14\text{pF}$.

**II.1.2. Determining the value of the input capacitor**

The capacitor seen by the OP AMP at its input is the input capacitance of the OP AMP (196fF) plus the sum of all the output capacitances of all the pixels (64² identical capacitors of 600aF in parallel). So, it is equal to about 2.65pF.

**II.1.3. Determining the values of the resistors**

First, the value of $R_1$ is the reverse resistance of the photodiode and is therefore huge (about 1TΩ).

Secondly, the value of $R_2$ is the ratio of the maximum output voltage of the OP AMP (let’s say 3V) and of the maximum input current (about 6mA according to a Matlab simulation that simulated 1,000 measures and took the maximum result). Thus, $R_2=500\Omega$.

**II.1.4. Determining the transfer function of the OP AMP**

This is done by considering the schematic in Fig.3.

First, the gain is determined. It is $A_1A_2$ where $A_1$ and $A_2$ are the gain of the first and second stage respectively. Their expressions are the followings:

$$A_1 = \frac{gm_9}{gds_{10}^2 + gds_{14}^2 \over gds_{10} + gm_{12} \over gds_{14} + gm_{16}}$$
\[ A_2 = \frac{\text{gm}_1}{\text{gds}_2 + \text{gds}_4} \]

Thus, the total gain \( A_0 \) is:

\[ A_0 = \frac{\text{gm}_1 \cdot \text{gm}_9}{(\text{gds}_2 + \text{gds}_4) \left( \frac{\text{gds}_{10}^2}{\text{gds}_{10} + \text{gm}_{12}} + \frac{\text{gds}_{14}^2}{\text{gds}_{14} + \text{gm}_{16}} \right)} \]

Secondly, the capacitances at the output of the first stage (\( C_1 \)) and of the second stage (\( C_2 \)) are computed. Here are their expressions:

\[
C_1 = C_{db12} + C_{db16} + C_{gs2} = C_{gs2} = 196\text{fF}
\]
\[
C_2 = C_{db2} + C_{db4} + C_{load} = C_{load} = 68.14\text{pF}
\]

where \( C_{gsi} \) and \( C_{dbi} \) are gate-source and drain-bulk capacitances of the transistor \( i \), and \( C_{load} \) the capacitance computed in paragraph II.1.1.

Thirdly, the poles were determined. There are two of them: one at each stage output. They are given by:

\[
P_1 = \frac{\left( \frac{\text{gds}_{10} \cdot \text{gds}_{12}}{\text{gm}_{12}} + \frac{\text{gds}_{14} \cdot \text{gds}_{16}}{\text{gm}_{16}} \right)}{C_{gs2}}
\]
\[
P_2 = \frac{\text{gm}_2}{C_{load}}
\]

Then, the transfer function of the OP AMP was computed. The obtained transfer function is the following:

\[
A_0 \quad H_1(s) = \frac{s}{\left( \frac{s}{p_1} \right) + \left( \frac{s}{p_2} \right) + \left( \frac{1}{p_1} \right) + \left( \frac{1}{p_2} \right)}
\]
Afterwards, the transfer function of the whole circuit can easily be computed and put in the form below:

\[
H_2(s) = \frac{B_0}{a_3s^3 + a_2s^2 + a_1s - 1}
\]

where:

\[
R_2
B_0 = \frac{R_1(A_0 - 1) - R_2}{a_0}
\]

\[
a_0 = \frac{R_1(A_0 - 1) - R_2}{A_0R_1}
\]

\[
a_1 = \left( \frac{1}{P_1} + \frac{1}{P_2} \right) \frac{1}{A_0a_0} \left( \frac{R_2}{1 + \frac{R_2}{R_1}} \right) + \frac{R_2C_p}{A_0a_0}
\]

\[
a_2 = \frac{1}{a_0} \left( \frac{1}{A_0} \left( \frac{1}{R_1} \left( \frac{1}{P_1P_2} \right) \frac{R_2}{1 + \frac{R_2}{R_1}} \right) + \frac{R_2C_p}{A_0} \left( \frac{1}{P_1} + \frac{1}{P_2} \right) \right)
\]

\[
a_3 = \frac{R_2C_p}{A_0\cdot P_1\cdot P_2\cdot a_0}
\]

We will easily notice here that \(a_0\) is very close to one. \(H_2\) can be easily written in the following form:

\[
H_3(s) = \frac{B_0}{(1 + \tau_1s) \left( \frac{2\xi s}{\omega_n} \right) + \frac{s^2}{\omega_n^2}}
\]

In the formula above, the first part of the denominator with the pole \(\tau_1\) comes from the capacitor at the input of the OP AMP, and the second part comes from the OP AMP itself.

The denominator of the transfer function \(H_3\) can be developed and then coefficients \(a_3, a_2, a_1, a_0\) and \(B_0\) can be determined as a function of \(\omega_n, \xi\) and \(\tau_1\) and the latter as a function of the circuit parameters. The calculus is shown below:
\[
(1 + \frac{2 \xi s}{\omega_n} + \frac{s^2}{\omega_n^2}) = 1 + \left( \frac{2 \xi}{\omega_n} \right) s + \left( \frac{2 \xi \tau_1}{\omega_n^2} \right) s^2 + \frac{\tau_1}{\omega_n^2} s^3
\]

We know that, \( \omega_n \) is large with respect to \( 1/\tau_1 \). So, \( \tau_1 \) is the dominant term of the \( s \) coefficient. Therefore:

\[
\tau_1 = a_1 = \frac{1}{A_0} \left( R_2 C_p + \left( \frac{1}{P_1} + \frac{1}{P_2} \right) \right)
\]

Then, for the same reason and because \( \xi \) is between 0 and 1, if we look at the coefficient of \( s^2 \) we have:

\[
a_2 = \frac{2 \xi \tau_1}{\omega_n}
\]

Finally, and obviously by observing the coefficient of \( s^3 \):

\[
\sqrt{\frac{\tau_1}{a_3}} = \frac{\omega_n}{\sqrt{\left( \frac{P_1 P_2 R_2 C_p + P_1 + P_2}{R_2 C_p} \right)}}
\]

Having the transfer function in the form of \( H_3 \), we can see that the second order denominator of \( H_3 \) has two poles. We therefore have:

\[
\frac{2 \xi s}{\omega_n} + \frac{s^2}{\omega_n^2} = \frac{[s + \omega_n(\xi + j\sqrt{1 - \xi^2})][s + \omega_n(\xi - j\sqrt{1 - \xi^2})]}{\omega_n^2}
\]

Thus, we get:

\[
H_4(s) = \frac{B_0 \omega_n^2}{(1 + \tau_1 s)[s + \omega_n(\xi + j\sqrt{1 - \xi^2})][s + \omega_n(\xi - j\sqrt{1 - \xi^2})]} = \frac{V_{out}}{I_i}
\]

Then, the impulse response of the whole circuit was computed, which is very easy since we are using Laplace variable. We do so to obtain it:

\[
H_4(s) \quad V_{out} = \frac{H_4(s)}{s}
\]
Afterwards, to go to the time space more easily, we should cut the ratio in the expression of \( V_{out} \) in four ratios (one for each term in the denominator) and thus in the following form:

\[
V_{out} = \frac{a}{s} + \frac{b}{1 + \tau_1 s} + \frac{c}{[s + \omega_n (\xi + j\sqrt{1 - \xi^2})]} + \frac{d}{[s + \omega_n (\xi - j\sqrt{1 - \xi^2})]}
\]

where \( a, b, c \) and \( d \) are coefficients to be determined. To obtain the coefficient \( I \) (\( I \) being \( a, b, c \) or \( d \)) we use a very common trick in mathematics. We take the two expressions of \( V_{out} \) above and we multiply them by the divisor of \( I \) (\( s \) for \( a \), \( 1 + \tau_1 s \) for \( b \)... ) and then we take the limit of these two expressions when \( s \) tends to the zero of the divisor of \( I \). We then get the final expression of \( V_{out} \).

Having the values of \( a, b, c, d \) and \( \xi \) (the damping), we can then make currents \( I_1 \) and \( I_2 \) (bias current of the first and second stage respectively) vary as well as the undetermined component values and this to obtain the wanted positions for the poles and for the gain. To achieve these positions, we required very high currents and large components. So, the modelling of the OP AMP by a third order transfer function failed.

**II.2. With a second order transfer function**

We have seen that modelling the OP AMP by a third order transfer function was giving rise to too high necessary currents. In fact, in the circuit used to convert the current into a voltage, the capacity \( C_p \) and resistor \( R_1 \) can be neglected because they do not limit the position of the poles to low frequencies but to high frequencies.

Indeed we know that the transfer function of the I-V conversion circuit with the OP AMP is equal to:

\[
H(s) = \frac{R_2.A}{R_2 + 1 + R_2.C_p.s}
\]

where \( A \) is the transfer function of the OP AMP. Here, we assume it to be simply \( A_0 \). Since \( R_1 \) is very high with respect to \( R_2 \) we thus have:
The numerator of the transfer function is simply $R_2$ since $A_0$ is high. As a consequence, we can see that there is a pole $P_1$ given by:

$$P_1 = \frac{A_0}{R_2C_p}$$

$A_0$, for a 10 bit resolution, is of the order of 1000. $R_2$ is of about 500Ω (ratio of $V_{\text{ref}}$ and maximum input current) and $C_p$ about 2pF. Thus, $P_1$ is of the order of 1THz.

The I-V conversion circuit transfer function can thus be the one of the OP AMP and thus a second order transfer function with two real poles. The damping $\xi$ is therefore above 1. The transfer function of the OP AMP can be written as:

$$H(s) = \frac{N}{D} = \frac{A_0}{\left(1 + \frac{s}{P_1}\right)\left(1 + \frac{s}{P_2}\right)}$$

where the denominator can be written as:

$$\frac{2 \cdot \xi}{\omega_n} \frac{1}{1 + \frac{\omega_n}{\omega_n} s + \frac{\omega_n^2}{\omega_n^2} s^2}$$

where:

$$\xi = \frac{P_1 + P_2}{2 \cdot \sqrt{(P_1 \cdot P_2)}}$$
\[ \omega_n = \sqrt{P_1 P_2} \]

The two poles can be written as:

\[ P_1 = \frac{\omega_n}{\xi \pm \sqrt{\xi^2 - 1}} \]

\[ P_2 = \omega_n \left( \xi \pm \sqrt{\xi^2 - 1} \right) \]

where \( \xi > 1 \).

Then, the response of the OP AMP to a pulse can be computed just by dividing the transfer function above by \( s \) and then by changing the Laplace variable into the time variable. The response to a pulse of amplitude 1 is given by:

\[ V_{out}(t) = A_0 \left\{ 1 - 0.5 \left[ (1-a) e^{(\xi + \sqrt{\xi^2 - 1}) t s} + (1+a) e^{-(\xi - \sqrt{\xi^2 - 1}) t s} \right] \right\} \]

where:

\[ a = \frac{\xi}{\sqrt{\xi^2 - 1}} \]

The static error of the circuit is thus:

\[ \varepsilon_s = \frac{A_0 - V_{out}(t_s)}{A_0} \]

Afterwards, a Matlab code was written to plot the static error for a given \( \xi \) as a function of \( \omega_n t_s \). We made the \( \xi \) vary till the \( \omega_n t_s \) (for which the static error was below \( 1/2^{10} \)) had a reasonable value. Indeed, increasing \( \omega_n t_s \) value gives a robust design and decreasing it gives an optimal settling time [1]. A value close to 12 was chosen as done in [1]. In Fig.5. is what we obtained for the best \( \xi = 1.125 \).
Fig. 5. Static error as a function of $\omega_n t_s$ for $\zeta = 1.125$.

The required resolution is obtained for $\omega_n t_s$ above 12.13. We notice that here the curve is almost linear in semi logarithmic scale which shows that we have an aperiodic behaviour and not a pseudo periodic behaviour as with the third order system.

This is then what we obtain for $\zeta$ ranging from 1.025 to 1.275 with steps of 0.025. The top curve corresponds to a $\zeta$ of 1.275.

Fig. 6. Static error as a function of $\omega_n t_s$ for various $\zeta$ values.

We notice that, as the $\zeta$ increases, for a given $\omega_n t_s$, the static error increases. This is why a small enough $\zeta$ was chosen.
After that, the poles $P_1$ and $P_2$ were determined from the values of $\zeta$ and $\omega_n$. Here $t_s$ (which is $1/3.(T/2 - \mu)$ where $T$ is the period of the signal at the input of the OP AMP which is here $1/(1434\times4)$ and $\mu$ is a security amount taken equal to $1.5\mu$s) was $28.556\mu$s accounting for 4 images per second and 1,434 measures per image for this OP AMP to treat data coming from the pixel array used as a camera to take pictures. The value of $\omega_n$ is of $424.78 \text{ krad.s}^{-1}$ since $\omega_n \cdot t_s = 12.13$.

The obtained poles values were thus of about $258.95 \text{ kHz}$ and $696.80 \text{ kHz}$. Given that a plus or minus solution is possible for each pole, we are able to choose $P_1$ equal to the minimum or maximum value and $P_2$ equal to the maximum or minimum value. We saw with simulations that if we choose a lower pole $P_1$ then the OP AMP gain will increase. And if a high $P_2$ is chosen, then $gm_2$ is high and the gain increases too. Thus, it is better to have:

$$P_1 = 258.95 \text{ kHz}$$
$$P_2 = 696.80 \text{ kHz}$$

The next step was to properly choose the biases in the circuit.

After choosing the poles positions and proper biasing, another Matlab code was written to be able to obtain the sizes of all transistors, capacitors and resistors in the circuit from the values of these poles and bias. However, a good sizing has not been found yet and will be shown in the next report.

### III. The OP AMP complete structure

Even if the OP AMP satisfies the CMRR constrain, we will choose to implement a common mode feedback (CMFB) of both stages to make the design more complete, and it will also help reducing the sensitivity of low frequency gain to common mode input voltage. Another advantage of CMFB is that it gives us the flexibility of setting the common mode output voltage $V_{o,cm}$, and guarantee the maximum output swing with process variations. We will use two large resistors to take out the CM output voltage. Although the absolute value of a CMOS resistor is not accurate, we can usually guarantee that the two resistors have the same value by properly laying them out. The design flow of our CMFB will be summarized in the following steps:

1) Set the optimum operating point by changing $V_{\text{in,cm}}$ and determine the best $V_{o,cm}$.
2) Put a test current source in and test the relationship between the tail current and $V_{o,cm}$. 
3) Referred to this relationship, cut the original tail current so that the variation of the amount of current drain by this test source can adjust the $V_{o,cm}$ over a large enough range, and include the optimum output CM level in this range.

4) Replace this test current source by $V_{cmc}$ (common mode control voltage) biased NMOS find out the relationship between this $V_{cmc}$ and $V_{o,cm}$.

5) According to step 3, find out the $V_{cmc}$ that can bias $V_{o,cm}$ to the desired level.

6) Make a simple OP AMP, set its common mode input level ($V_{cm}=V_{in+}=V_{in-}$) as $V_{o,cm}$. Size this OP AMP so that its common mode output ($V_{cmc+}$ and $V_{cmc-}$, as shown in Fig. 7.) is $V_{cmc}$ acquired from step 4. Another choice is to size the additional tail NMOS to achieve the same thing. Make sure the feedback is negative.

7) Use two 100k $\Omega$ resistors to take $V_{o,cm}$ and feed into the CMFB OP AMP. And connect $V_{cmc-}$ to the gate of the additional tail NMOS for CMFB.

[1] shows the effect of the CMFB on the sensitivity of DC gain vs. $V_{in,cm}$. With no CMFB, the response curve is quite sharp, indicating that the DC gain depends a lot on input CM voltage. With one CMFB to the first stage, we can see a much flatter response. If CMFB is applied to both stages of their OP AMP, the characteristic is further improved and the OP AMP is useful almost over the entire input common mode range (ICMR depends on biasing).

In terms of controlling $V_{o,cm}$, they could change $V_{o,cm}$ linearly from 500 mV to 1.2 V (for 0.18$\mu$m technology). This makes the adjustments of DC operating point at the output end (to the middle of output swing) much easier and unsusceptible to process variations.

Finally, CMFB also improves the common mode rejection ration (CMRR) by introducing negative feed back for the common mode signals. Figure 6(f) in [1] is the common mode gain after adding CMFB, which drops below zero dB.

IV. What remains to be done

We should obtain the component values in the OP AMP, enter these values in a Cadence schematic window and simulate the circuit to get the DC analysis with DC operating points, the AC analysis giving gain and phase of the OP AMP. We will then check that the bandwidth, the phase and the poles are correctly positioned. We should also check the FPN value is not too high. And finally, we should do the layout of this OP AMP.

Afterwards, the schematics of the I-V conversion circuit (the one presented in chapter 6, part 1, § II.) should be drawn in Cadence and simulations should be done to get the DC and AC analysis results. We should then proceed with its layout and post-layout simulation.
References for chapter 6

Fig. 7. Complete schematic of the OP AMP with the CMRR circuit implemented.
CHAPTER 7: THE DESIGN OF THE PHOTODIODE ARRAY

All layouts have been done in the 0.35µm CMOS front-back technology (corresponding to Cadence’s c35b4 technology and fb mode) that uses four metal layers. Two solutions that have been considered during this project will be presented in this chapter. Firstly, we will study the common part of these two solutions (the photodiode) and then we will present the parts (pixel, array, whole floor plan) that differ in these two solutions. The solution that we have finally chosen will be explained more in detail.

I. The photodiode

I.1. The dimensions of the photodiode

The photodiode size was chosen by considering the work previously done in the LSM by a Ph.D. student and by considering the estimated size of the array obtained with this photodiode size. The total width of the photodiode was chosen equal to 30µm. We obtained this value by removing to 5mm (our absolute width limit for the chip) the widths of the digital part and of the analog part (each of them assumed to be of 0.6mm, which really is a worst case) and twice the width of the pads for accessing the inputs and outputs of our system (so two times 400µm). We then divided the result by 64 (being the number of pixels in one row of the array) and we got something slightly above 30µm. This photodiode size will therefore leave enough space for other parts (digital control unit, OP AMP bank, pads…) whose sizes are quite small with respect to the one of the array as we will see later.

I.2. The photodiode layout

The photodiode layout was previously realized by a Ph.D. student of the LSM and was redone and improved following his explanations and warnings.

The photodiode is a simple reversely biased CMOS diode. Its structure is the usual one and is shown below (Fig.1.). It is composed of a p+ well enclosed into a n-well. Both of these wells are associated to a contact having a ring shape (see the layout in Fig.8.) to increase the photodiode efficiency. Below each contact, the doping is set higher than in the rest of the well to improve the quality of the contact and therefore to reduce the contact resistance. Finally, there is one (or two rings as shown in Fig.1. below) to isolate each photodiode from its neighbours by stopping the currents travelling close to the surface (currents shown in red in Fig.1.). Adding a second ring just improves this insulation.
II. Our first idea for the floor plan

II.1. For the pixel

This first idea consisted in integrating only the photodiode and its biasing (and eventually amplifying) electronics within the pixel. Thus, the floor plan of one pixel was very simple as shown below (Fig.2.). All the remaining area was used to route each photodiode output to the outside of the array.

II.2. For the whole system

In this solution, building the photodiode array consists in putting side by side the above pixels (they must be regularly spaced to avoid any image distortion), in drawing lines to provide the power supply to each photodiode, and in designing the lines going from each photodiode to the outside of the array. As a consequence, there are 4,096 lines going out from the array and two going in.

Each of these 4,096 lines are connected to a couple of switches in parallel as shown in the current multiplication method using the OP AMP (Chapter 6, part 1, § II, Fig.2.).

Each of these switch couples uses a control signal to command the aperture or closure of its switches. Each couple has two outputs being respectively connected to the OP AMP (associated to the column where this couple is) inverting and non inverting inputs.

These control signals come from 4,096 memory points (flip-flops) in the digital unit.

The digital unit is the one controlling the image acquirement and compression. It is the one containing the control unit, the decoder and the memory points. This digital part should have a long shape to be placed along the photodiode array.

The operational amplifiers should be placed far away from the digital part which will generate parasitic signals.

Some pads will be located at the top to supply the inputs needed by the system, and to read the outputs generated by the operational amplifiers and the digital unit.

Below (Fig.3.) is thus the obtained floor plan for the whole system.
II.3. Estimation of the size of the system

The photodiode is of dimensions 30µm x 30µm. On its left, there are 32 lines representing the routing of the outputs of the photodiode to the outside of the array (see Fig.3.). Their minimal width is of 0.5µm and the minimum enclosure is of 0.5µm too. So, the width of this bus of 32 lines is of exactly 32µm. Thus, the width of one pixel is exactly of 62µm. Therefore, the array width is precisely of 3.968mm which is too large for us since we still need to integrate pads, the digital part and the analog part. Besides, with this solution, the digital part is of about 3.5 million micro meter squares because of the use of a 12 bit decoder plus 4,096 memory points with their associated logic of 680µm² each. The width of the system thus goes over our 5mm limit. We thus need another solution for the floor plan to reduce the area. This will imply to modify the digital part.

III. Our final idea for the floor plan

In this solution, the photodiode layout is the same than in the first solution. What changes is the pixel and the whole system floor plan.

III.1. Storing the coefficient within the pixel

In the previous solution, as we used the already built digital unit, the coefficients were stored outside the array within memory points. These coefficients were generated by a LFSR and placed in the memory point selected by the decoder. We had to rescan all the pixels of the array each time we were doing a measurement. All this implied to draw many lines to the outside of the array. Thus the routing was occupying a large area in the array. Besides, the area below the photodiode and the photodiode electronics was left empty. Only the part on the left of the photodiode was used and the rest is simply wasted.
Fig. 3. Floor plan of the whole system: solution 1.

Fig. 4. How memory points (flips-flops) are connected together.

Fig. 5. Floor plan of the pixel: solution 2.
It is to avoid this waste that in the new solution the memory elements and switches are located within the pixel (see Fig.5.). The flip-flop is still controlling the two switches aperture and closure, the switches are still respectively plugged to the inverting and non inverting input of the OP AMP (associated to the column where these switches are), and on the photodiode output. The difference is that each pixel (in yellow in Fig.4.) contains a memory element (in red in Fig.4.) and that all these elements are plugged in series (thanks to wires in blue in Fig.4) thus building a shift register of length 4,096.

We could add some buffers from time to time on the clock signal path (blue line of Fig.4.) to refresh this clock. These additional buffers will be needed if we notice that the clock is degraded when it travels across the array.

In the case were the LFSR length is different from 12 bits, the LFSR generates the coefficients in permanence and these are sent into the shift register (through its input at the top left hand corner of the array). In this case, we do not reuse the coefficients that are going out from the array. In the case where the LFSR length is of 12 bits, we firstly load 4,096 coefficients into the shift register, thus assigning one to each pixel. When this is done, the system is ready to perform one measurement. After the end on this measure, we just apply a rising edge in the clock to shift by one step all coefficients towards the output. One coefficient goes out (through the output in the bottom left hand corner of the array) and is to be reloaded into the array. We thus obtain the next configuration of coefficients for the array enabling us to perform the next measure.

The digital unit thus has to be modified to be able to either load 4,096 coefficients into the shift register using the LFSR, then shift them by one step and reinsert the coefficient going out from the shift register into the shift register or to load the shift register in permanence using the LFSR.

III.2. The new floor plan of the pixel

The pixel now includes the photodiode, the photodiode biasing (and eventually amplifying circuit), a flip-flop and two switches (see Fig.5.). Here, the coefficient that will multiply the current of the photodiode is thus stored within the pixel.

In this case, we only have to provide the supply voltages (GND and VDD), clock and reset signals for the flip-flop and photodiode, and what will be returned are the signals going to the inverting and non inverting inputs of the OP AMP (associated to the column of the array were this pixel is located). There are also the data lines for the flip-flop (lines going to its input D and coming out from its output Q) but these ones come from or go to the neighbouring pixels and are not lines crossing the whole array. Thus, the routing is much simpler and the routing area is tremendously reduced thanks to this method.
III.3. Ideas for the new floor plan of the whole system

We have added to Fig. 5, three horizontal blue lines (clock for the flip-flops, GND and VDD supply lines), two brown lines going to the inverting and non inverting inputs of the OP AMP (associated to the considered column) and two other brown lines for the reset of the photodiode and for the reset of the flip-flop.

We have also added to this figure 5 the OP AMP bank containing the 64 operational amplifiers. Each of them has a size of about 1,000µm² (i.e. a block of dimensions 32µm x 32µm). We thus require between 60,000 and 70,000µm² for this bank.

We then added the digital unit which is now between 50,000 and 100,000µm² (about 1200µm² for the clock divider, 10,000µm² for the LFSR clock generation and the load technique choice, 7000µm² for the SLF generation block and 10,000µm² for the LFSR plus some area for the routing) versus 3.5 million µm² for the previous solution. We have taken care in this floor plan to place the analog part far away from the digital part.

Afterwards, we have added 68 pads. We obtained this number by estimating the total number of pads that we will need. The estimation process is explained here.

First of all we have to know that, for each group of 8 pads, we need two pads to supply the VDD and GND to these 8 pads. The VDD and GND pads to supply the VDD and GND to a group of pads must be different for the analog and the digital parts and must be separated by an insulation pad.

Then, we require six pads for the power supplies of the digital and analog part and for the array. We need two others to isolate the power supply pads of these three different parts to limit interferences.

After, we will use 16 pads that will be connected to 8 randomly chosen operational amplifiers (two lines per OP AMP since it is a fully differential OP AMP) in the OP AMP bank. These 16 pads will be used for the characterisation of the array.

Then, two pads are required for the output of the OP AMP bank. A block will have to be designed to multiplex the two groups of 64 outputs into only two outputs and will be placed inside the additional block of the floor plan.

Afterwards, the digital unit will require about 14 pads for its inputs and outputs (pads for the supply voltages non-included) plus additional pads (seven of them) used later to test this digital part. One of these additional pads will be connected to the output of the LFSR and will enable us, when the chip will be made, to monitor the sequence generated by the LFSR.
The other additional pads will permit us to monitor the various clocks (Clk, Clk_refresh…) and important signals of the control unit (like SLF, MEM_out…).

Finally, we added five or six additional pads just in case we forget some.

Therefore, we require 27 pads in total for the digital part, and 24 for the analog part. Thus we require 14 additional pads to bring the power to these pads (eight for the digital part and six for the analog part) and an additional one for insulation. Table 1 below summarizes all the required pads. As a consequence, we need 68 pads.

<table>
<thead>
<tr>
<th>Number of pads</th>
<th>Reason</th>
<th>Analog pads?</th>
<th>Digital pads?</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Pads for power supply of the digital part</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Pads for power supply of the analog part</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>Supply pads for the array</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Insulation pads</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>16</td>
<td>Pads for monitoring the outputs of eight randomly chosen operational amplifiers</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>Outputs of the OP AMP bank</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>14</td>
<td>Digital part I/O</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>Pads for chip testing</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Additional pads for digital part</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Additional pads for analog part</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>Pads for power supply of digital pads</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>Pads for power supply of analog pads</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>1</td>
<td>Separation between analog and digital supply pads</td>
<td>/</td>
<td>/</td>
</tr>
</tbody>
</table>

Table 1. Summary of the required pads.

What we did not represented on Fig.6. are two vertical lines each eight pixels that will connect the VDD and GND of each row together (see layouts in Fig.17 and 18). This will enable to equalize the value of VDD and GND in the whole array. A Matlab simulation showed that the mean output current going out from one column of the array, i.e. from input+ or input- (see Fig.14), is of 3.2mA, and we round it to 4mA. The rule being that a 1µm wide line is needed to carry a current of 1mA, we thus require 4µm wide lines for these vertical supply lines. The horizontal VDD and GND lines will have a width of 1 µm.

What we can not see in Fig.6. is that photodiodes must be regularly spaced to avoid any image distortion. So, as the flip-flop block and the electronic logic block are quite high (13.61µm) and that on the right of the photodiode the area is only used for routing, we will have some free space in this area. We can not do anything to solve it. We can not merge the NTUBs (the minimal distance between them being 3µm), we can not change the photodiode size (otherwise the pixel size increases or the photodiode layout becomes uncommon if it is rectangular) and we have no logic to add there. So we have no choice but to leave it like this.

The obtained floor plan is shown in Fig.6.
III.4. Estimation of the array area

The width of one pixel is thus of about 36µm. The array width is therefore of about 2.3mm therefore leaving a very large area (22.7mm²) for the digital unit, the OP AMP bank and the pads. This solution is as a consequence viable. The only drawbacks are a slightly more complex pixel and a change in the digital part.

![Figure 6](image)

**Fig.6. Final idea for the floor plan of the system.**

IV. Why avoiding using too many metal layers

We tried to use only three metal layers (out of four) to do the routing for the whole array. This is to avoid losses in vias that should be limited since currents going out from the array are quite low and given that contact resistances between vias and metal layers they link are generally high thus implying Joules losses.

V. The schematics and layouts

V.1. The photodiode

V.1.1. The schematic

The photodiode schematic (see fig.7. below) is a simple well diode. This schematic was taken from the library and we just had to specify the perimeter (120µm) and the area (900µm²) of the photodiode.
V.1.2. The layout

The photodiode layout (see fig. 8. above) was not present in the library and was designed by us in the way described in § I.2. Its width and height are both of exactly 30µm. We can see all the layers described in § I.2. and the two lines for contacts (in purple at the right bottom hand corner of the image).

V.2. The logic

V.2.1. The schematic

This schematic was designed by us. It has four inputs and two outputs.

Inputs:  
- Control and Controlbar coming respectively from the Q and QN outputs of the flip flop and controlling the switches. These two signals are the complement of each others (when one is at 0, the other is at 1).
- Reset: signal to reset the photodiode.
- Photodiode: input to plug the photodiode on the logic.

Outputs:  
- Input+ going to the non-inverting input of the OP AMP associated to the column where this logic is located.
- Input- going to the inverting input of the OP AMP associated to the column where this logic is located.
Basically, there are three components:

- a switch MP0: when turned on (Reset at 0), the photodiode is reset.
- a first transmission gate (I4) that is closed when Control is at 1. In this case (multiplication by the coefficient 1), the photodiode output goes directly to the non inverting input of the OP AMP.
- a second one (I5) closed when Control is at 0. In this case (multiplication by the coefficient -1), the photodiode output goes directly to the inverting input of the OP AMP.

Finally, we should add that the sizes of the transistors in the logic were set to the maximum values, for this logic to be as fast as possible, and to reasonable values so that the logic block can be placed below the photodiode and next to the flip-flop.

V.2.2. The layout

The layout is very simple. It was designed by us. It contains five transistors (two transmission gates and one switch). The p diffusion regions of all PMOS were merged to gain space. We can see the labels showing the inputs and outputs of the logic.
V.3. The flip-flop

V.3.1. The schematic

This flip-flop is from the Corelib library of Cadence. There, it is called DFC3. We can see its inputs and outputs on Fig. 11. We see that we have chosen a normal flip-flop with a clock (C), a data input (D), a reset (RN) and the Q and complemented Q outputs. The flip-flop inner schematic is shown in Fig. 12.

![Fig.11. The flip-flop component.](image1)

![Fig.12. The flip-flop schematic.](image2)

V.3.2. The layout

The layout is also from the library and was attached to the schematics in Fig. 11. and 12. It is simply the transcription of the schematic in Fig. 12. into a layout.
V.4. The pixel

V.4.1. The schematic

The pixel contains the flip-flop, the photodiode and the logic described above. We have a reset signal for the flip-flop and one for the photodiode. There are also one input, called Data_in, to load the coefficient into the flip-flop, and another output, called Data_out, to send the coefficient contained in this flip-flop, into the next memory point.

Fig.14. The pixel schematic.
V.4.2. The layout

The layout (Fig.15.) gathers all the blocks contained in the schematic of the pixel. The two output lines called Input+ and Input- have a width of 4µm (because of the Matlab simulation result that we spoke about before). Lacking of space, we were forced to superimpose two 2µm wide lines using two different metals (metal 1 and 3) that we linked together thanks to many vias (orange squares). The widths of the two reset lines were kept minimal (i.e. 0.6µm) since these are control signals (i.e. lines carrying a voltage only). The layout size was of exactly of 46.3µm x 46.3µm. The active area of the photodiode being of 19.4µm x 19.4µm (green area in Fig.15.), the fill factor that we obtain is of exactly 17.56% which is not so bad with respect to fill factors generally obtained in photosensors arrays (generally around 25%).
V.5. The array

V.5.1. The schematic

The schematic will not be shown. It is simply 4,096 times the one of the pixel. All the Reset (flip-flop reset signal) are connected together. Same thing for the Reset_photodiode (the photodiode reset signal), VDD, GND and Clock signals. There are 64 couples of output lines, each couple containing signals Input+ and Input-.

V.5.2. The layout

The photodiode array layout is shown below (Fig.19.). In Fig.20. is shown a zoom on the top left hand corner of the array. Each 8 columns, as explained above, we added two vertical lines (metal 3, yellow colour) to make the VDD and GND uniform along the array. These two lines are more visible on Fig.17 and 18 (see the right of the image where there are lines called VDD and GND).

To build this array, we used four different pixel layouts that are shown in figures 15 to 18. They are exactly the same, but they only differ in the presence or not of VDD and GND large supply lines, and in the positions of the vertical signal lines (at the left, at the right or on both sides of the pixel).
The row located at the top of the array is built first. Its floor plan is shown in Fig.25. One red block represents a cell or a group of cells. From left to right on figure 25 we have:

- The pixel presented in figure16.
- M17: 6 pixels like the one in figure 15.
- One pixel identical to the one in figure 17.
- M16, M15, M14, M13, M12 and M11: all contain seven pixels like the one in figure15.
- The blocks in between M16 and M15, M15 and M13, M13 and M14, M14 and M12, M12 and M11, M11 and M10 are the pixel presented in figure 17.
- Finally, the last pixel is the one in figure 18.

As soon as we got this first row, we copied it and performed an axial symmetry on it to get the second row (the axis being the vertical line in the middle of this first row). This second row was placed below the first one. Then, these two rows were copied and the copy was placed below the original rows. We perform this process till obtaining 64 rows in the array. We finally and automatically get the whole layout for the array with the floor plan corresponding to the one presented in Fig.6. above.

Finally, when we got the array, we linked all Reset lines together. Same thing for the Reset_photodiodes lines and Clk lines. In all cases, we just used a 0.6µm wide line to connect the lines together since they all carry control signals. All the VDD and GND lines are also connected together thanks to very wide lines.

The photodiode array is then finished and thus has the dimensions 2.9632µm x 2.9632µm (46.3µm x 64, without the above final connections).

Below are shown some zooms on the array borders. In Fig.21. we see the line through which we will supply the coefficients into the array (white line at the top left hand corner of the image). On Fig.22., we see the place where the output Data_out of the pixel at the end of the first row is plugged to the input Data_in of the pixel located below. In Fig.23., we see where the output Data_out of the first pixel of the second row is plugged to the input Data_in of the first pixel of the third row. Finally, in Fig.24. (at the bottom left hand corner of the image, white line), we see the line through which the coefficients will go out from the array.
V.6. The pad

V.6.1. The Schematic

The role of a pad is to enable us to access the inputs and outputs of the microsystem and also to protect them from the electrostatic discharges that could occur when we touch the chip with our hands. As a consequence, a pad is basically composed of two parts:
- A part containing diodes and acting as a fuse. Any incoming high electrostatic charge will flow across these diodes instead of flowing across our circuit and damaging or destroying it.

- A part that enables us to access to the input or output we want to monitor. This is mainly a resistive part.

Any pad requires usual supply voltages and therefore VDD and GND (or VSS). Additional pads are thus required to be able to supply these voltages for about eight pads in our case since the switching activity is not so intense.

The schematic of such a component is shown in Fig.26. We find our two different parts and also see that VDD and GND supply voltages need to be supplied. This schematic is from an IO library of Cadence.

![Fig.26. The schematic of the pad.](image)

**V.6.2. The layout**

We can now have a look to the layout of a pad. The layout corresponding to the schematic shown above is shown below in Fig.27. We see, from right to left, the contact pad, diodes and resistors. The layout size is of 104µm x 346µm in this example.
V.7. The floor plan of the whole system: a more precise insight

We have here gathered all the blocks shown above. This enables us to have a much more precise insight of the final layout of our system, much more accurate than our drawing in figure 6.

This floor plan glimpse is shown below in Fig.28. We can now see our array, and some pads. The white blocks represent the blocks that we have not finished yet and from which we only know an estimation of the size of their layout. We have thus replaced them by an empty block of the corresponding size. In this case, we have the OP AMP bank, the digital unit and the additional block.

We can therefore see that our design will fit in a square shape that will have a width (and height) of exactly 3877µm.

The factory will make its invoice according to the area in micrometer square of our design. For this foundry, building a one micro meter square area cost close to 600Fr (about 380€). The area of our design being of 15.03µm², our design will cost approximately 5.7k€.

What we can say is that we notice that the first limitation of this size is the number of pads. So, we will try, in the future, to remove four or six of them for example. We could also try to choose another position for few of these pads, thus using the remaining free area. After doing that, the limiting factor of the area will be the array size that will be very hard to change.
Fig. 28. Precise insight of the floor plan of the whole system.
Conclusions and future work

The algorithm (issued from compressive sampling theory) managing the image compression was studied successfully. Thanks to a great amount of simulations, the structure of the whole system was found. One pixel of the 64 x 64 array was designed and we obtained a fill factor of 17.56%. With the layout of this pixel, we deduced the layout of the whole array successfully. Afterwards, a first idea for digital unit implementing this compression was designed with success. It included the design of a variable length LFSR which was playing the role of a pseudo random number generator (generating the two different coefficients -1 and 1) and that was extremely important for the algorithm. We also began to think about the new digital part and started to design the fully differential operational amplifier but we have not totally finished yet.

At the end, the floor plan of the whole system was very promising since it was small (3.8mm x 3.8mm), had a perfect square shape and was leaving enough space for the non finished blocks.

What will be done and shown in the final report for Grenoble are the final operational amplifier and the new digital part designs. We will also present the system with all gathered blocks. A PCB will be built to gather the elaborated system and a testing digital unit if we do not lack of time.
Thanks to all readers for their attention when reading my report.
Appendixes
Appendixes: Table Of Contents

Appendix 1: Acronyms .................................................................97

Appendix 2: Variables and signals names ........................................... 98

Appendix 3: The used software .......................................................... 100

Appendix 4: LFSR .................................................................101

Appendix 5: Results of the compression algorithm with three different coefficients...102

A5.1. Obtaining probabilities close to 1/6 for -1 and 1 and to 2/3 for 0 ............102
A5.2. Injecting a column vector X with only one non null component supplies
    a peak in X_new .................................................................103
    A5.2.1. Observation of the 32nd column of the 'AA matrix .........................103
    A5.2.2. Observation of the complete 'AA matrix .......................................104
A5.3. Compression and decompression of tetep.pgm (8 bits, 64x64) .................104
A5.4. Compression and decompression of tete.pgm (8 bits, 128x128) ..................105
A5.5. Why does it work so well ..............................................................105
A5.6. Variation of the computation time and of the mean value (when k varies) of
    the difference between the maximum value of X and of X_new as a function of the
    number of measures ..............................................................106
A5.7. Conclusion ........................................................................107

Appendix 6: Improvements that can be done to reduce the noise .....................108

A6.1. A means to quantify the noise in the diagonal matrix .......................108
A6.2. Increasing the number of intervals in the decision of the values of the
    coefficients according to the binary number going out from the LFSR .........109
    A6.2.1. 12 bit LFSR ................................................................109
        A6.2.1.1. Observation of the 32nd column of the 'AA matrix .................109
        A6.2.1.2. Observation of the complete 'AA matrix ..................................109
        A6.2.1.3. Observation of the tetep.pgm (64x64 image) .......................110
A6.2.1.4. Conclusion ................................................................................. 110
A6.2.2. 14 bit LFSR .................................................................................. 111
  A6.2.2.1. Observation of the 32\textsuperscript{nd} column of the \textsuperscript{t}AA matrix .............. 111
  A6.2.2.2. Observation of the complete \textsuperscript{t}AA matrix .......................... 111
  A6.2.2.3. Conclusion ............................................................................. 112
A6.3. Proceeding with a memory containing all the coefficients for all measures .... 112
  A6.3.1. Algorithm principle ................................................................... 112
  A6.3.2. Probabilities to obtain -1, 0 and 1 ........................................... 112
  A6.3.3. Observation of the 32\textsuperscript{nd} column of the \textsuperscript{t}AA matrix .............. 113
  A6.3.4. Observation of the \textsuperscript{t}AA matrix ...................................... 113
  A6.3.5. Observation of the tetep.pgm (64x64 image) .............................. 113
  A6.3.6. Conclusion ............................................................................. 114

Appendix 7: Code for the algorithm .......................................................... 115

  A7.1. PGMA\_read: Matlab code .............................................................. 115
  A7.2. Code for the compression: multlogicLFSR: C code ....................... 117
  A7.3. Code for the decompression: multadlogicLFSR: C code ............... 121
  A7.4. Code for the Matlab master .......................................................... 125

Appendix 8: Additional and simple components ...................................... 133

  A8.1. Flip-flop .......................................................................................... 133
  A8.2. Two to one multiplexer ................................................................. 133
  A8.3. Delay .............................................................................................. 133

Appendix 9: VHDL codes for the digital unit .......................................... 134

  A9.1. AND21 .......................................................................................... 134
  A9.2. XOR21 .......................................................................................... 135
  A9.3. XOR41 .......................................................................................... 136
  A9.4. MUX21 .......................................................................................... 137
  A9.5. MUX71 .......................................................................................... 138
  A9.6. Flip-flop.......................................................................................... 139
  A9.7. Delay .............................................................................................. 140
  A9.8. Clk\_divider .................................................................................. 141
A9.9. SLF_generation.................................................................142
A9.10. Generate_Clk_LFSR.........................................................146
A9.11. Control_unit...............................................................148
A9.12. LFSR........................................................................153
A9.13. The 12 bit decoder.........................................................159
   A9.13.1. Decoder 6 bits.....................................................159
   A9.13.2. Decoder 12 bits second.........................................162
   A9.13.3. Decoder 12 bits..................................................164
A9.14. Memory.................................................................166
A9.15. Part1: the top level code.............................................168
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Analog to Digital Converter.</td>
</tr>
<tr>
<td>AMS</td>
<td>Austrian Micro System.</td>
</tr>
<tr>
<td>APS</td>
<td>Active Pixel Sensor.</td>
</tr>
<tr>
<td>CCD</td>
<td>Charged Coupled Devices.</td>
</tr>
<tr>
<td>CDS</td>
<td>Correlated Double Sampling.</td>
</tr>
<tr>
<td>CID</td>
<td>Charge Integration Device.</td>
</tr>
<tr>
<td>CM</td>
<td>Common Mode</td>
</tr>
<tr>
<td>CMFB</td>
<td>Common Mode FeedBack</td>
</tr>
<tr>
<td>CMIM</td>
<td>Metal-Insulator-Metal Capacitor.</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor.</td>
</tr>
<tr>
<td>CMRR</td>
<td>Common Mode Rejection Ratio.</td>
</tr>
<tr>
<td>CS</td>
<td>Compressive Sampling.</td>
</tr>
<tr>
<td>DMD</td>
<td>Digital Micromirror Device.</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic Random Access Memory.</td>
</tr>
<tr>
<td>FPN</td>
<td>Fixed Pattern Noise.</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine.</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>ICMR</td>
<td>Input Common Mode Range.</td>
</tr>
<tr>
<td>LFSR</td>
<td>Linear Feedback Shift Register.</td>
</tr>
<tr>
<td>LFSR-4</td>
<td>4 tap Linear Feedback Shift Register.</td>
</tr>
<tr>
<td>OP AMP</td>
<td>Operational Amplifier.</td>
</tr>
<tr>
<td>PCB</td>
<td>Printed Circuit Board.</td>
</tr>
<tr>
<td>PGA</td>
<td>Programmable Gate Array.</td>
</tr>
<tr>
<td>PPS</td>
<td>Passive Pixel Sensor.</td>
</tr>
<tr>
<td>PRNG</td>
<td>Pseudo Random Number Generator.</td>
</tr>
<tr>
<td>PSNR</td>
<td>Peak Signal-to-Noise Ratio.</td>
</tr>
<tr>
<td>SoC</td>
<td>System-on-a-Chip.</td>
</tr>
<tr>
<td>TFT</td>
<td>Thin Film Transistor.</td>
</tr>
</tbody>
</table>
APPENDIX 2: VARIABLES AND SIGNAL NAMES

Bit_Nb_Sel: Selection signal to select the LFSR feedback and output in the LFSR circuit.

Clk: Clock for the FSMs.

Clk_CU: Control unit clock.

Clk_fast: Fastest clock.

Clk_LFSR: LFSR clock.

Clk_LFSR_apply: Externally applied LFSR clock.

Clk_memory: Control unit clock.

Clk_refresh: Clock used to refresh the FSMs state.

Count: Output signal from the counter in the SLF and seed generation system.

Decoder_12bits_out: Decoder output signal.

Decoder_output: Signal from the decoder to control all memories.

Equal: Signal at logic one when Count and Threshold are identical.

Im_done: Tells that one image has been completed when going to logic 1.

k: Row of X where the non null coefficient is located.

LFSR_output: LFSR output.

Load_seed: Externally defined signal telling the seed application has to begin (when at 1) or
telling the LFSR is used in its normal operation mode (when at 0).

Meas_done: Tells that one measure has been done when going to logic 1.

MEM_out: Generated seed.

N: LFSR number of bits.

nb_dim: Number of pixels in the photodiode array.

nb_meas: Number of measurements to make one image.

Number_meas: Externally applied signal telling to the system how many measures are
needed to make one image.

Pixel_selection: Signal from the control unit to select a pixel.

Pixel_selection_12bits: Decoder input signal.

Q(i): i\textsuperscript{th} switch control signal.

Reset: Resets the system to its idle state.

Seed_applied: Externally applied LFSR seed.

Sel_LFSR_Clk: LFSR clock selection signal.

Sel_seed: Seed selection signal.

SLF: Signal telling the seed has been loaded when at logic 1.

Threshold: 5 bit signal representing the LFSR number of bits.

X: Image turned into a column vector.
\textbf{X\_new}: Column vector containing the compressed and then decompressed image.

\textbf{Y}: Matrix containing the compressed image.
- Simulator: Matlab 7.4.0.
- Image converters: Image Converter.
  Advanced Batch Converter.
- Compilator: Gcc 4.2.0.
- VHDL code editor, compiler and simulator: Modelsim.
- Synthesizer: Design Vision.
- Placement and routing (digital part): Encounter.
- Schematic editor and simulator: Cadence, Assura (DRC, LVS, RCX).
- Layout editor: Virtuoso.
APPENDIX 4: LFSR

A linear feedback shift register is a shift register whose input bit is a linear function of its previous state.

The only linear functions of single bits are XOR and inverse-XOR. Thus it is a shift register whose input bit is driven by the exclusive-OR (XOR) of some bits of the overall shift register value.

The initial value of the LFSR is called the seed, and because the operation of the register is deterministic, the sequence of values produced by the register is completely determined by its current (or previous) state. Likewise, because the register has a finite number of possible states, it must eventually enter a repeating cycle. However, an LFSR with a well-chosen feedback function can produce a sequence of bits which appears random and which has a very long cycle [1].

Reference for appendix 4

APPENDIX 5: RESULTS OF THE COMPRESSION ALGORITHM WITH THREE DIFFERENT COEFFICIENTS

A5.1. Obtaining probabilities close to 1/6 for -1 and 1 and to 2/3 for 0

These probability estimations were done using the LFSR code only (i.e. without the compression and decompression algorithm) and thus just the seed, nb_meas (taken equal to 2000), the structure and nb_dim were supplied. The thresholds chosen to choose for a -1, 0 or 1 coefficient according to the decimal value of the N bit number going out from the LFSR are $2^{N/6}$ and $5.2^{N/6}$. If this number was between 0 and $2^{N/6}$ the PRNG generated a -1 (probability of 1/6), if between $2^{N/6}$ and $5.2^{N/6}$ it supplied a 0 (probability of 2/3) and if between $5.2^{N/6}$ and $2^{N}$ it provided a 1 (probability of 1/6). These thresholds were chosen to be in agreement with the compression principle knowing that the probabilities of obtaining each integer between 1 and $2^{N}$ were all equal.

a) For 12 bits (if we consider a 64x64 array thus containing 4096=2^{12} pixels, see conclusion to see why we chose 12 bits) the obtained probabilities (Pr) were (as shown in Fig.1.) 0.1668 (about 1/6) for -1, 0.1659 (about 1/6) for 1 and 0.6673 (about 2/3) for 0.

![Fig.1. 12 bit LFSR probabilities for -1, 1 and 0.](image1)

b) For 14 bits (if we consider a 128x128 array thus containing 16384=2^{14} pixels, see conclusion to see why we chose 14 bits) the obtained probabilities were (as shown in Fig.2.) 0.1669 (about 1/6) for -1, 0.1660 (about 1/6) for 1 and 0.6671 (about 2/3) for 0.

![Fig.2. 14 bit LFSR probabilities for -1, 1 and 0.](image2)

So, the probabilities of -1, 1 and 0 are close to the expected ones and that in both cases (12 and 14 bit LFSR). It is very important for the algorithm. Another important thing, that is assumed and not shown here, is the good randomness of the number generation. This has already been shown for LFSRs. Please refer to an appropriated document.
A5.2. Injecting a column vector $X$ with only one non null component supplies a peak in $X_{\text{new}}$

We have here chosen an $X$ that is particular: its rows are all at 0 except the $k^{\text{th}}$ row where there is a 1. We have then inserted $X$ in the multlogiLFSR and the obtained $Y$ was the input of the multadlogiLFSR. We took a $\text{nb\_meas}$ equal to 2000 and a $\text{nb\_dim}$ to 128. We thus directly worked with our created $X$ and so without image. We worked with the 12 bit LFSR and then with the 14 bit LFSR.

When doing the compression and decompression we do: $Y=AX$, then $X_{\text{new}}=\text{\,}'AY$ and thus we have $X_{\text{new}}=(\text{\,}'AA)X$. But by taking such a $X$, $(\text{\,}'AA)X$ is simply the $k^{\text{th}}$ column of $\text{\,}'AA$. Therefore, by doing a loop on $k$ (from 1 to 128) we can obtain the whole $\text{\,}'AA$ matrix. The latter should be as close as possible to the identity matrix to have a good compression and decompression quality. We have first observed one $X_{\text{new}}$ with such a $X$ (and thus one column of $\text{\,}'AA$ and so $X_{\text{new}}$ must have a peak in its $k^{\text{th}}$ row as close to one as possible) and secondly we have observed the complete $\text{\,}'AA$ by plotting it with the Matlab imagesc function. The bright regions are values close to 1 and dark regions have values close to 0.

A5.2.1. Observation of the $32^{\text{nd}}$ column of the $\text{\,}'AA$ matrix

![Fig.3. 32\text{nd} column of $\text{\,}'AA$ matrix for 12 bit LFSR.](image1)

![Fig.4. 32\text{nd} column of $\text{\,}'AA$ matrix for 14 bit LFSR.](image2)

We thus gain a quite sharp peak in $k=32$. We however notice a bit of noise around the peak which is due to the fact that two following generated coefficients by the LFSR are not completely independent.
A5.2.2. Observation of the complete $^1$AA matrix

![Fig.5. $^1$AA matrix for a 12 bit LFSR-4](image1)

![Fig.6. $^1$AA matrix for a 14 bit LFSR-4](image2)

We thus obtain a very good identity matrix in both cases. We however notice a bit of noise in the diagonal region which is simply due to the same fact that was explained previously.

We can conclude by saying that the 12 and the 14 bit LFSR have very similar performances in terms of quality of the compressed and decompressed image at the output.

A5.3. Compression and decompression of tetep.pgm (8 bits, 64x64)

![Fig.7. 12 bits, PSNR: 8.01dB.](image3)

![Fig. 8. Initial image.](image4)

![Fig.9. 14 bits, PSNR: 7.36 dB.](image5)

The images were plotted for 200,000 measures (this number is here higher than the number of pixels because the decompression algorithm is not efficient).

The quality of the two images (12 bit and 14 bit LFSR) is mediocre, a good quality of image being between 30 and 40dB.

Finally, the quality obtained in the 14 bits case is less than in the 12 bits case.
A5.4. Compression and decompression of tete.pgm (8 bits, 128x128)

The images were plotted for 200,000 measures.

The obtained image qualities (with 12 bit and 14 bit LFSRs) are also mediocre.

In the 12 bits case there is a problem. It is due to the fact that the LFSR sequence length is 4095 ($2^{12}-1$) and that the pixel number is 16384 ($128^2$) and so, before the end of the image the sequence repeats close to 4 times (16384/4095) and that is why 4 patterns can be seen in the image.

A5.5. Why does it work so well

One could say for instance for the 14 bit LFSR with the 128x128 image and the 200,000 measures:

“Your image is 128x128 and you do 200,000 measures. Therefore the A matrix has to contain exactly $3.2768.10^9$ coefficients. But your LFSR sequence length is only 16383 so it shouldn’t work at all because there is some periodicity in that”. And, we would answer by: “You are right but the number of pixel is 16384 and the LFSR sequence length is 16383. So, when we proceed with the $2^{nd}$ measure we will not proceed with the same seed as in the $1^{st}$ measure (initial seed defined in Matlab master) but the one generated by the LFSR (at its output) when we compute the coefficient of the 16383$^{rd}$ pixel. So, each time we do a new measure, the new seed becomes the LFSR output that is, in the LFSR sequence, the seed before the seed that was used in the previous measurement.”
A5.6. Variation of the computation time and of the mean value (when k varies) of the difference between the maximum value of X and of X_new as a function of the number of measures

This study will enable us to choose an optimal value for \( nb\_meas \).

We have here used the 14 bit LFSR. We used a X with only one non null component equal to 1 located in the \( k^{th} \) row. \( nb\_meas \) was varied from 1 to 128.

We notice in the results of this study that:

- The computation time increases linearly (see red line in Fig.13.) with \( nb\_meas \) with a slope of 570\( \mu s/meas \). thus pushing us to choose a small \( nb\_meas \).

- The mean difference between the X maximum value and the one of X_new and this when k varies is like \( 1-e^{1/nb\_meas} \) (see blue curve in Fig.13.) thus pushing us to increase \( nb\_meas \).

In the current case, we could take \( nb\_meas \) equal to 128 that gives a difference close to zero which is excellent for a good compression and decompression quality.

![Fig. 13. Variation of the computation time and of the mean value (when k varies) of the difference between the maximum value of X and of X_new with the number of measurements.](image)

The number of measures was however very high in the experiences above (2,000 and 200,000) because the decompression algorithm is not efficient since we built a very simple one and much more efficient ones do exist but it is not our goal here to build one of them.
A.5.7. Conclusion

We therefore conclude by saying that the number of bits $N$ of the LFSR-4 must be sufficient enough so that $2^N - 1 > \text{nb\_dim}$.

We must have a large enough number of measures to have an as close image as possible to the initial one.

A more efficient decompression algorithm should be built but it is not our purpose here since we just show that the compression works and that it is possible to decompress such a data. We could think, for example, to use the matching pursuit algorithm.
APPENDIX 6: IMPROVEMENTS THAT CAN BE DONE TO REDUCE THE NOISE

A6.1. A means to quantify the noise in the diagonal matrix

A means to quantify the noise in the diagonal matrix constituted by the plot of X_new as a function of X (with k is varying) was found. It is a three steps process.

- First step:
  
  - We take the matrix $^\dagger$AA (obtained with a LFSR for example and a square image) and we remove completely its diagonal thus giving a matrix F with one less column.
  - Then, we compute the standard deviation of F thanks to the std function of Matlab returning a column matrix.
  - All the elements of this column matrix are summed and give a result called sigma.
  - 1/sigma is then computed.

- Second step (done 200 times):

  - A random matrix A of coefficients -1, 0 and 1 is computer generated with the rand function of Matlab and other locally defined functions. This matrix has the same size as the one built by the LFSR and thus we use the same number of measures here.
  - The product $U = ^\dagger$AA is computed.
  - The diagonal of U is removed giving F.
  - The standard deviation of F is computed and we obtain $\sigma_0$ which is the perfect randomness $\sigma$ and used as a reference.
  - 1/$\sigma_0$ is computed.
  - The mean value of 1/$\sigma_0$ is computed.

- Third step:

  - $(1/\sigma)/(1/\sigma_0)$ is computed.

The higher this ratio is, the lower the noise we have and the easiest to decompress the image it will be. We have computed this ratio in the following parts to compare the different techniques together.
A6.2. Increasing the interval number in the coefficients value decision according to the binary number going out from the LFSR

A6.2.1. 12 bit LFSR

For the 12 bit LFSR we have a sequence of 4095 \( (2^{12} - 1) \) binary numbers that are, in decimal scale, between 1 and 4095. This interval, previously divided into 3 parts, has been further divided into 12 and then into 24 parts. We will see the effects of this division.

A6.2.1.1. Observation of the 32\textsuperscript{nd} column of the †AA matrix

We notice that by increasing the intervals number, the X\textsubscript{new} curve matches the X curve better (plots below obtained for 2,000 measures). However, there are two noise peaks appearing near the main peak when we further increase the intervals number and this is troublesome as we will see in the § A6.2.1.3.

![Fig.1. Increasing the number of intervals to 12.](image1)

![Fig.2. Increasing the number of intervals to 24.](image2)

A6.2.1.2. Observation of the complete †AA matrix

![Fig.3. Increasing the number of intervals to 12.](image3)

![Fig.4. Increasing the number of intervals to 24.](image4)
We here have two clear identity matrixes (Fig.3. and Fig.4. obtained for 2,000 measures). We however see that the diagonal in the 24 intervals case is closer to one than the one in the 12 intervals case. A drawback in the 24 interval case is a clearly higher noise outside the peak area.

In the case where the interval is cut into 12 parts and where we perform 1,434 measures, the \( \frac{1}{\sigma}/(1/\sigma_0) \) ratio is of 228 (0.4108/0.0018). In the case where we have 24 intervals, this ratio is of 367 (0.6616/0.0018) which is much better.

**A6.2.1.3. Observation of the tetep.pgm (64x64 image)**

![Figure 5: 12 intervals: PSNR~7dB.](image1)

![Figure 6: 24 intervals: PSNR~3dB.](image2)

We have plotted the decompressed images obtained by compressing the tetep.pgm image with LFSRs having 12 and 24 intervals respectively (see Fig.5. and Fig.6.). The number of measures was of 200,000. We here notice that, despite the fact that the X_new curve matches better the X curve on the peak, we have a lower image quality with 24 intervals than with 12 intervals. This is due to the fact that in the 24 intervals case, the noise next to the peak (two peaks going downwards) is much higher than in the 12 intervals case. We should thus rather divide the interval into 12 parts.

**A6.2.1.4. Conclusion**

If we may choose the solution with the 12 bit LFSR to generate the pseudo random sequence of coefficient, then we should choose to increase the number of intervals to twelve, solution which gives acceptable results.
A6.2.2. 14 bit LFSR

A6.2.2.1. Observation of the 32\textsuperscript{nd} column of the $\text{^1AA}$ matrix

Fig.7. Increasing the number of intervals to 12. 
Fig.8. Increasing the number of intervals to 24.

We here also notice a better matching of $X$ and $X_{\text{new}}$ in the peak area at the cost of a higher noise around this peak area.

A6.2.2.2. Observation of the complete $\text{^1AA}$ matrix

In the case where the interval is cut into 12 parts and where we perform 5,734 measures, the $(1/\sigma)/(1/\sigma_0)$ ratio is of 0.325290 (0.2076/0.6382). In the case where we have 24 intervals, this ratio is of 0.59589 (0.38/0.6377) which is much better.

Fig.9. Increasing the number of intervals to 12. 
Fig.10. Increasing the number of intervals to 24.
A6.2.2.3. Conclusion

If we may choose the solution with the 12 bit LFSR to generate the pseudo random sequence of coefficient, then we should choose to increase the number of intervals to twelve.

We have seen that increasing the number of intervals but not too much is good to obtain a better image quality. However, this technique, if we want to implement it in our circuit, will cost too much area and therefore will be too expensive.

A6.3. Proceeding with a memory containing all the coefficients for all measures

A6.3.1. Algorithm principle

Within a main Matlab M-file, we created a nb_meas x nb_dim A matrix containing some random coefficients between 0 and 1 generated using the Matlab rand function. The X matrix was generated at the same time.

A C file then takes this matrix and turns it into a matrix with the same dimensions but containing our -1, 0 and 1 coefficients and this just by comparing the coefficients between 0 and 1 to two thresholds i.e. 1/6 and 5/6. If below 1/6, we opt for -1, if above 5/6 we obtain a 1 and if in between we get a 0. Afterwards this matrix is multiplied by X thus supplying Y. The Y value is sent to a second C file.

The second C file takes Y and A as inputs and generates \( A \). From this it computes \( X_{\text{new}} \) and returns it to the main Matlab file.

Finally, the main Matlab file writes a PGM image.

A6.3.2. Probabilities to obtain -1, 0 and 1

The bar plot (Fig.11. below, obtained for 2,000 measures) shows respective apparition probabilities for -1, 0 and 1 of 0.1667, 0.1668 and 0.666. They are very close to the case of the LFSR which is a very good point for our LFSR since it shows a very good randomness for the coefficients generated by this component.

![Fig.11. Probabilities to get -1, 1 and 0 (from left to right).](image)
A6.3.3. Observation of the 32\textsuperscript{nd} column of the $^1$AA matrix

The X\_new curve (in green in Fig.12.) matches perfectly the X curve (in blue in the same figure) in the peak region. There is however a bit of noise elsewhere and this noise is higher than the one we usually have with LFSRs.

Because of this the image quality will be much lower than in the LFSR case.

Fig.12. 32\textsuperscript{nd} column of $^1$AA for 2000 measures.  
Fig.13. $^1$AA matrix for a memory PRNG.

A6.3.4. Observation of the $^1$AA matrix

We here have a clear diagonal matrix (obtained for 2,000 measures, see Fig.13.). But what we easily notice is the much higher background outside the diagonal of the matrix. This will for sure induce highly blurred images and we will thus require more measures to achieve a similar results to the one obtained in the LFSR case, which is not acceptable for us since we want a number of measures much lower than the number of pixels.

A6.3.5. Observation of the tetep.pgm (64x64 image)

Here (Fig.14.) is the obtained 64x64 image for 500,000 measures. The obtained result is comparable to the one obtained in the LFSR case. We thus need much more measures than with a LFSR to gain an acceptable result.

Fig.14. tetep for 500,000 measures.
A6.3.6. Conclusion

For our purpose (compressing an image and thus taking a measure number lower than the pixel number) it is better not to use the PRNG using a memory to store the coefficients. Besides, such a memory will be very large (4096.nb_meas memory points which is huge and much larger than a simple 12 bit LFSR).
APPENDIX 7: CODES FOR THE ALGORITHM

A7.1. PGMA_read: Matlab code

% PART 1: FUNCTION READING THE PGM FILE AND WRITING IT INTO i_X
% function [i_X, ncol, nrow] = pgma_read(file_name)
% PGMA_READ opens an ASCII PGM file and reads the data.
% The PGM file is assumed to have the format:
% P2
% # feep.pgm
% ncol nrow
% maxgray, the largest legal value for the data.
% row 1, a list of ncol numbers between 0 and maxgray.
% row 2, a list of ncol numbers between 0 and maxgray.
% ...
% row nrow, a list of ncol numbers between 0 and maxgray.
% Lines beginning with '#' are comments.
% Example:
% P2
% # feep.pgm
% 24 7
% 15
% 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
% 0 3 3 3 0 0 7 7 7 0 0 11 11 11 11 0 0 0 0
% 0 3 0 0 0 0 0 0 0 11 0 0 0 0 0 0 15 0 0
% 0 3 3 3 0 0 0 0 0 0 11 11 0 0 0 0 15 0 0
% 0 3 0 0 0 0 0 0 0 0 11 11 11 11 0 0 15 0 0
% 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
% % Created in February 2008
% % Author: Alexandre Bibet
% % Parameters:
% % Input, character *FILE_NAME is the name of the PGMA file to read.
% % Output, integer GRAY[NROW,NCOL], is the gray scale data read from the file.

FALSE = 0;
TRUE = 1;

gray = [ ];

fid = fopen ( file_name, 'r' );

if ( fid < 0 )
    fprintf ( 1, 'un' );
    fprintf ( 1, 'PGMA_READ - Fatal error!un' );
    fprintf ( 1, 'Could not open the input file.un' );
    error ( 'PGMA_READ - Fatal error!' );
    return;
end

% Read the first line.

line = fgets ( fid );

% Verify that the first two characters are the "magic number".
% Matlab strncmp returns 1 for equality, and 0 for inequality.

if ( strncmp ( line, 'P2', 2 ) == 0 )
    return;
end
while ( 1 )
    line = fgets ( fid );
    if ( line(1) ~= '#' )
        break;
    end
end

% Extract NCOL and NROW.

[ array, count ] = sscanf ( line, '%d' );
ncol = array(1);
nrow = array(2);

% Move to the next non-comment line.

while ( 1 )
    line = fgets ( fid );
    if ( line(1) ~= '#' )
        break;
    end
end

% Extract MAXGRAY, and ignore it.

[ array, count ] = sscanf ( line, '%d' );
maxgray = array(1);

% Set aside space for GRAY.

gray = zeros ( nrow, ncol );
i = 1;
j = 0;
done = FALSE;
while ( done == FALSE )
% Move to the next non-comment line.

while ( 1 )
    line = fgets ( fid );
    if ( line(1) ~= '#' )
        break;
    end
end

[ array, count ] = sscanf ( line, '%d' );
% Each value that you read goes into the "next" open entry in GRAY.

for k = 1 : count
    j = j + 1;
    if ( ncol < j )
        j = 1;
        i = i + 1;
    end
    if ( i <= nrow )
        gray(i,j) = array(k);
    end
    if ( i == nrow & j == ncol ) done = TRUE;
end
t = ncol*nrow;
i_X = reshape(gray,t,1);
close ( fid );
A7.2. Code for the compression: multlogicLFSR: C code

/* multphi: perform random projection of a vector using a random matrix commuted on fly
* thanks to the logistic equation.
* Syntax: y = multlogi(x, seed, nb_measure, Structure).
* Ref: Xueyi, Z.; Lu, J.; Kejun, W. & Dianpu, L.
* Logistic-map chaotic spread spectrum sequences under linear transformation.
* Intelligent Control and Automation, 2000. Proceedings of the 3rd World Congress on,
* 2000, 4.
* The documentation is available in the corresponding M-file.
* Mansec{License}.
* This file is part of YAW Toolbox (Yet Another Wavelet Toolbox).
* You can get it at ["http://www.fyma.ucl.ac.be/projects/yawtb"]{"yawtb homepage"}.
* Modified by Alexandre Bibet in March 2008.
* Copyright (C), the YAWTB Team (see the file AUTHORS distributed with this library).
* (See the notice at the end of the file.) */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mex.h>

/* Input data */

/* Vector to multiply (size n \times 1) */
#define i_x         prhs[0]

/* Define the seed */
#define i_seed      prhs[1]

/* Number of random projection */
#define i_nb_meas   prhs[2]

/*Definition of the structure*/
#define i_Structure prhs[3]

/* Number of inputs */
#define i_narg  4

/* Output data : the projected vector */
#define o_y         plhs[0]
void  mexFunction ( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{

/* Input/Output arrays (in C) */
double *x, *seed, *Structure, nb_meas;
double *y, *cur_y;

/* Misc */
int i, m, j, k, h, f;
double count, nb_dim, bern_val, bern_coeff;

/**************************************************************************
*                               Input Checkings                                *
************************************************* *************************/
if (nrhs != i_narg)
{
    mexErrMsgTxt("There are not enough parameters");
}

/**************************************************************************
*                               Initializations                                *
************************************************* *************************/
x   = mxGetPr(i_x);
b_dim = mxGetM(i_x)*mxGetN(i_x); 
seed  = mxGetPr(i_seed);
nb_meas  = mxGetScalar(i_nb_meas);
Structure = mxGetPr(i_Structure);

*****************************/
o_y = mxCreateDoubleMatrix (nb_meas, 1, mxREAL);
y   = mxGetPr(o_y);
bern_val = sqrt(3.0/nb_meas);

/* Definition of the LFSR nb of bits */
int n1, n2, n3, n4;
n1=Structure[0];
n2=Structure[1];
n3=Structure[2];
n4=Structure[3];

/* Elements */
int R_prev[n1-1], R_cur[n1-1];

/* Put seed in R_prev */
for (j=0;j<=n1-1;j++)
{
    R_prev[j]= seed[j];
}
for(m = nb_meas - 1; m >= 0; m--)
{
    cur_y = (y + (long int) m);
    *cur_y = 0;
    for( i = nb_dim - 1; i >= 0; i-- )
    {
    
    /* Below: translation of: cur_logi = logi(prev_logi); */
    R_cur[0] = (R_prev[n1-1]+R_prev[n2-1]+R_prev[n3-1]+R_prev[n4-1])%2;
    for (k=1; k<=n1-1; k++)
    {
        R_cur[k]= R_prev[k-1];
    }
    
    /* printf("%.4f", cur_logi); */

    /* Bernoulli coefficients */
    if ( R_prev[n1-1]==0 )
    {
        *cur_y -= *(x + (long int) i);
        /* printf(" (-1) "); */
    }
else if (R_prev[n1-1]==1 )
{
    *cur_y += *(x + (long int) i);
    /* printf(" (+1) "); */ */
}

for (f=0;f<=n1-1;f++)
{
    R_prev[f]=R_cur[f]; /* prev_logi = cur_logi; */
}
}

*cur_y *= bern_val; /* printf("n"); */
}

/* EndOf mexfunction */
A7.3. Code for the decompression: multadlogicLFSR: c code

/* multadphi: perform the adjoint of a random projection of a vector using a random matrix 
* commuted on fly thanks to the logistic equation.
* Syntax: x = multadlogiLFSR(y, seed, nb_dim, Structure).
* Ref: Xueyi, Z.; Lu, J.; Kejun, W. & Dianpu, L.
* Logistic-map chaotic spread spectrum sequences under linear transformation.
* Intelligent Control and Automation, 2000. Proceedings of the 3rd World Congress on,
* 2000, 4.
* The documentation is available in the corresponding M-file.
* Mansec{License}.
* This file is part of YAW Toolbox (Yet Another Wavelet Toolbox).
* You can get it at \url{http://www.fyma.ucl.ac.be/projects/yawtb}[/"yawtb homepage"].
* Modified by Alexandre Bibet in March 2008.
* Copyright (C) 2001-2002, the YAWTB Team (see the file AUTHORS distributed with this
* library).
* (See the notice at the end of the file.) */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mex.h>

/* Input data */

/* Vector to multiply (size n \times 1) */
#define i_y prhs[0]

/* Define the seed */
#define i_seed prhs[1]

/* Number of random projection */
#define i_nb_dim prhs[2]

/* Define the seed */
#define i_Structure prhs[3]

/* Number of inputs */
#define i_narg 4

/* Output data : the projected vector */
#define o_x plhs[0]
/* Mexfunction == main */
void mexFunction ( int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
    /* Input/Output arrays (in C) */
    double *y, nb_dim,*seed, *Structure;
    double *x, *cur_x, cur_y_val;
    /* Misc */
    long int i, m, j, k, h, f;
    double count, nb_meas, bern_val, bern_coeff;

    /**************************************************************************
    *                               Input Checkings                                *
    ************************************************** ************************/
    if (nrhs != i_narg)
    {
        mexErrMsgTxt("There are not enough parameters ");
    }
    /* Check inputs sizes */
    if ( (mxGetM(i_y) == 0) ||
        (mxGetN(i_seed) == 0) ||
        (mxGetM(i_nb_dim) == 0) ||
        (mxGetN(i_Structure) == 0) )
    {
        mexErrMsgTxt("Check inputs, one is of zero size!");
    }
    /**************************************************************************
    *                               Initializations                                *
    ************************************************** ************************/
    y = mxGetPr(i_y);
    seed = mxGetPr(i_seed);
    nb_dim = mxGetScalar(i_nb_dim);
    nb_meas = mxGetM(i_y)*mxGetN(i_y);
    Structure = mxGetPr(i_Structure);
    o_x = mxCreateDoubleMatrix (nb_dim, 1, mxREAL);
    x = mxGetPr(o_x);
    for ( i = nb_dim - 1; i >= 0; *(x+i) = 0, i--);
bern_val = sqrt(3/nb_meas);

/* Definition of the LFSR nb of bits */
int n1, n2, n3, n4;
n1=Structure[0];
n2=Structure[1];
n3=Structure[2];
n4=Structure[3];
int R_prev[n1-1], R_cur[n1-1];

/* Put seed in R_prev */
for (j=0;j<=n1-1;j++)
{
   R_prev[j]= seed[j];
}
for(m = nb_meas - 1; m >= 0; m--)
{
   cur_y_val = *(y + m);
   for( i = nb_dim - 1; i >= 0; i--)
   {
      /* Pseudo random generator : logistic equation */
      R_cur[0] = (R_prev[n1-1]+R_prev[n2-1]+R_prev[n3-1]+R_prev[n4-1])%2;
      for (k=1; k<=n1-1; k++)
      {
         R_cur[k]= R_prev[k-1];
      }
      /*printf("%.4f", cur_logi);*/
      /* Bernoulli coefficients */
      if (R_prev[n1-1]==0)
      {
         *(x + i) -= cur_y_val;
       /*printf(" (-1) ");*/
      }
      else if (R_prev[n1-1]==1)
      {
         *(x + i) += cur_y_val;
       /* printf(" (+1) ");      */
      }
for (f=0;f<=n1-1;f++)
{
    R_prev[f]=R_cur[f];
}

/* printf("\n"); */

for (i = nb_dim - 1; i >= 0; *(x+i) *= bern_val, i--);
A7.4. Code for the Matlab master

%% Author: Bibet Alexandre

% % Code to do Random Projections and Transpose them

%% X definition
n = 64;                         % nb_dim
k = 32;
v=zeros(n,1);
v(k)=1;

% Meas definition
meas = 3300;

% Number of bits of the ADC definition
nb_bit=11;

% Structure of the LFSR definition
Structure=[12 6 4 1 0];
n1=Structure(1);

% Seed definition
seed=zeros(1,n1);
seed(1)=1;
seed(3)=1;
seed(5)=1;
seed(7)=1;
seed(9)=1;
seed(11)=1;

av = multlogiLFSR(v, seed, meas, Structure);
avm = round(av*(2^nb_bit))/(2^nb_bit);
atav = multadlogiLFSR(avm, seed, n, Structure)/3;

plot(v);
title('Atav and V')
xlabel('Row of the matrix'), ylabel('Atav (Green) and V (Blue)')
hold on;
plot(atav, 'g')
hold off;
nb_dim = paramnbc('tetep.pgm')*paramnbrow('tetep.pgm');
Structure=[12 6 4 1 0];
n1=Structure(1);
meas = 3300;

% Seed definition
seed=zeros(1,n1);
seed(1)=1;
seed(3)=1;
seed(5)=1;
seed(7)=1;
seed(9)=1;
seed(11)=1;

A = LFSR(nb_dim, seed, Structure, meas)
bar(A)
xlabel('1: Logic 0  2: Logic 1'), ylabel('Probability');
title('Probability to get logic 0 or 1 at the LFSR output');

meas = 3300;

% Number of rows
n = 64;

% U is the matrix where put all X_new
U=zeros(n,n);

% Structure of the LFSR definition
Structure=[12 6 4 1 0];
n1=Structure(1);
% Seed definition
seed=zeros(1,n1);
seed(1)=1;
seed(3)=1;
seed(5)=1;
seed(7)=1;
seed(9)=1;
seed(11)=1;

for k = 1:n, % Where put 1 in v
  v=zeros(n,1); % Input vector
  v(k,1)=1;
  av = multilogiLFSR(v, seed, meas, Structure);
  avm = round(av*(2^nb_bit))/(2^nb_bit);
  atav = multadlogiLFSR(avm, seed, n, Structure);
  U(:,k)=atav;
end;

imagesc(U);
colormap(gray);
title('Atav as a function of V');
xlabel('V'), ylabel('Atav');

% Definition of the number of measures
meas=3300;

% Number of bits of the ADC definition
nb_bit=11;

% Number of rows
n = 64;

% U is the matrix where put all X_new
U=zeros(n,n);

% Structure of the LFSR definition
Structure=[12 6 4 1 0];
n1=Structure(1);
\begin{verbatim}
% Seed definition
seed=zeros(1,n1);
seed(1)=1;
seed(3)=1;
seed(5)=1;
seed(7)=1;
seed(9)=1;
seed(11)=1;

for k = 1:n,       % Where put 1 in v
    v=zeros(n,1);    % Input vector
    v(k,1)=1;
    av = multilogiLFSR(v, seed, meas, Structure);
    avm = round(av*(2^nb_bit))/(2^nb_bit);
    atav = multadlogiLFSR(avm, seed, n, Structure)/3;
    U(:,k)=atav;
end;

% Compute the Dk
D=zeros(127,1);
for q=-63:63,
    diagp=zeros(64-abs(q),1);
    diagn=zeros(64-abs(q),1);
    if (q>=0)
        for i=1:(64-q),
            diagp(i,1)=abs(U(i,i+q));
            D(q+64)=mean(diagp);
        end;
    elseif(q<0)
        for j=(1-q):64
            diagn(j+q,1)=abs(U(j,j+q));
            D(q+64)=mean(diagn);
        end;
    end;
end;

alpha=10;

for l=-63:-alpha,
    moy2(l+64)=D(64+l);
end;
\end{verbatim}
for m=alpha:63,
moy2(64-2*alpha+m+1)=D(64+m); 
end;

mu=mean(moy2);
sigma=std(moy2);
T=mu+3*sigma;
center=zeros(2*alpha-1,1);
over=0;

for h=(64-alpha+1):(64+alpha-1),
    if (D(h)>T)
        over=over+1;
    end;
end;

over_final=over
O(m)=over_final;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%Code to take image tetep and compress it and decompress it%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 

%% Meas definition
meas = 3300;

%% Number of bits of the ADC definition
nb_bit=11;

%% Structure of the LFSR definition
Structure=[12 6 4 1 0];
n1=Structure(1);

%% Seed definition
seed=zeros(1,n1);  
seed(1)=1;       
seed(3)=1;       
seed(5)=1;       
seed(7)=1;       
seed(9)=1;       
seed(11)=1;
% Open image and take parameters
nb_dim = paramnbcol('tetep.pgm')*paramnbrow('tetep.pgm');
x = pgma_read('tetep.pgm');

y = multlogiLFSR(x, seed, meas, Structure);
ym = round(y*(2^nb_bit))/(2^nb_bit);
x_new = multadlogiLFSR(ym, seed, nb_dim, Structure);

m = reshape(x_new, paramnbrow('tetep.pgm'), paramnbcol('tetep.pgm'));
imagesc(m);
title('Compressed and decompressed 64x64 image')
colormap gray;

% Number of bits of the ADC definition
nb_bit=11;
for m=1:41;

% Meas definition
meas=100*m;

% Structure of the LFSR definition
Structure=[12 6 4 1 0];
n1=Structure(1);

% Seed definition
seed=zeros(1,n1);
seed(1)=1;
seed(3)=1;
seed(5)=1;
seed(7)=1;
seed(9)=1;
seed(11)=1;

% Open image and take parameters
nb_dim = paramnbcol('tetep.pgm')*paramnbrow('tetep.pgm');
x = pgma_read('tetep.pgm');
y = multlogiLFSR(x, seed, meas, Structure);

ym = round(y*(2^nb_bit))/(2^nb_bit);

x_new = multadlogiLFSR(ym, seed, nb_dim, Structure);

P(m)=PSNR(x,x_new);

end;

plot(P, 'r')
xlabel('Number of measurements divided by 100'), ylabel('PSNR');
title('PSNR as a function of the number of measurements divided by 100');

meas=3300;
P=zeros(15,1);

% Open image and take parameters
nb_dim = paramnbcol('tetep.pgm')*paramnbrow('tetep.pgm');
x = pgma_read('tetep.pgm');

% Structure of the LFSR definition
Structure=[12 6 4 1 0];
n1=Structure(1);

% Seed definition
seed=zeros(1,n1);
seed(1)=1;
seed(3)=1;
seed(5)=1;
seed(7)=1;
seed(9)=1;
seed(11)=1;

for nb_bit=1:1;
    y = multlogiLFSR(x, seed, meas, Structure);
    ym = round(y*(2^nb_bit))/(2^nb_bit);
    x_new = multadlogiLFSR(ym, seed, nb_dim, Structure);
    PSNR(x,x_new);
end;
PSNR_final=P;
plot(PSNR_final);
xlabel('Number of bits of the ADC'), ylabel('PSNR');
title('PSNR as a function of the number of bits of the ADC');
APPENDIX 8: ADDITIONAL AND SIMPLE COMPONENTS

A8.1. Flip-flop

Each time there is a rising edge on the D flip-flop (DFF) clock input, the input D is copied to the output Q of the DFF. Otherwise, the Q signal keeps the same value. We have a memory effect. A reset signal is present to force Q to logic 0.

The Modelsim simulation below clearly shows the proper DFF behaviour.

![Fig.1. Simulation of the DFF.](image1)

A8.2. Two to one multiplexer

The 2 to 1 multiplexer (MUX21) copies its input a/b to its output z when its selection input is 0/1.

The simulation gave the result below. We check that the MUX21 works properly.

![Fig.2. Simulation of the MUX21.](image2)

A8.3. Delays

The delay system delays its input signal of a given amount of time (here, one fourth of the clock period). These delays are very useful to synchronize signal together when, for instance, one of them is too much in advance with respect to the others thus causing the failure of the system.

The simulation gave the result below. We check that the delay works properly.

![Fig.3. Simulation of the delay](image3)
APPENDIX 9: VHDL CODES FOR THE DIGITAL UNIT

A9.1. AND21

    -- Author: Bibet Alexandre
    -- Date: April 2008
    -- Place: EPFL

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity AND_21 is
    port (a : in std_logic;
          b : in std_logic;
          z : out std_logic);
end AND_21;

architecture behav5 of AND_21 is
    begin
        z <= a AND b;
    end architecture behav5;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity XOR21_created is
port (a : in std_logic;
b : in std_logic;
z : out std_logic);
end XOR21_created;

architecture behav1 of XOR21_created is
begin
z <= a XOR b;
end architecture behav1;
A9.3. XOR41

-- Author: Bibet Alexandre
-- Date: April 2008
-- Place: EPFL

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity XOR41_created is
port ( a : in std_logic; b : in std_logic; c : in std_logic; d : in std_logic; z : out std_logic);
end entity XOR41_created;

architecture behav2 of XOR41_created is
begin
z <= a XOR b XOR c XOR d;
end architecture behav2;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity MUX21_created is
port ( a : in std_logic;
       b : in std_logic;
       sel : in std_logic;
       z : out std_logic);
end MUX21_created;

architecture behav4 of MUX21_created is
begin
process(a, b, sel)
begin
  case sel is
    when '0' => z <= a;
    when '1' => z <= b;
    when others => z <= '0';
  end case;
end process;
end architecture behav4;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity MUX71 is
port (a : in std_logic;
      b : in std_logic;
      c : in std_logic;
      d : in std_logic;
      e : in std_logic;
      f : in std_logic;
      g : in std_logic;
      sel : in std_logic_vector(2 downto 0);
      z : out std_logic);
end MUX71;

architecture behav5 of MUX71 is
begin
  process(a, b, c, d, e, f, g, sel)
  begin
    case sel is
      when "001" => z <= a;
      when "010" => z <= b;
      when "011" => z <= c;
      when "100" => z <= d;
      when "101" => z <= e;
      when "110" => z <= f;
      when "111" => z <= g;
      when others => z <= '0';
    end case;
  end process;
end architecture behav5;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity Flip_Flop is
    port ( 
        clk    : in  std_logic;      -- Clock signal
        Reset  : in  std_logic;     -- Reset signal
        d      : in  std_logic;      -- Input signal
        q      : out std_logic      -- Output signal 
    );
end Flip_Flop;

architecture behav3 of Flip_Flop is
    begin
        process(clk,Reset)
        begin
            if (Reset='0') then
            q <= '0';
            elsif (clk'event AND clk='1') then
            q <= d;
            end if;
        end process;
    end architecture behav3;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- CONSTRUCTION OF THE DELAY USING ONE FLIP FLOP
entity delay is
port ( 
A          : in  std_logic;
Reset     : in  std_logic;
Clk_fast  : in  std_logic;
Z          : out std_logic );
end delay;

architecture behav of delay is

-- COMPONENT DFF
component Flip_Flop port( 
clk : in  std_logic; -- Clock signal
Reset : in  std_logic; -- Reset signal
d : in  std_logic; -- Input signal
q : out std_logic); -- Output signal
end component;

begin

Flip_Flop_1 : component Flip_Flop
port map (clk => Clk_fast, Reset => Reset, d => A , q => Z);

end architecture;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity Clk_divider is
  port(
    Clkin, rst_b: in std_logic;
    Clkout : out std_logic_vector(0 to 2));
end entity Clk_divider;

architecture rtl of Clk_divider is
  constant NSTAGES:natural:=2;
  signal w_reg, w_next: std_logic_vector(0 to NSTAGES);
begin
  w_reg(0) <= clkin;

  STAGES: for s in 1 to NSTAGES generate
    -- memory element(register)
    REG: process(w_reg(s-1),rst_b)
    begin
      if rst_b='0' then
        w_reg(s)  <= '1';
        Clkout(s) <= '1';
      elsif w_reg(s-1)'event and w_reg(s-1)='1' then
        w_reg(s) <= w_next(s);
        Clkout(s) <= w_next(s);
      end if;
    end process REG;
    -- next state logic
    w_next(s) <= not w_reg(s);
  end generate STAGES;
end architecture rtl;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity SLF_generation is
port (
    Clk              :  in   std_logic;                        -- Clock of control unit
    Reset             :  in   std_logic;                          -- Reset signal active when 0
    Bit_Nb_Sel    :  in   std_logic_vector (2 down to 0);     -- To select the number of bits of the LFSR
    load_seed        :  in   std_logic;
    SLF               :  out  std_logic;                          -- Seed has been loaded when SLF=0
    MEM_out       :  out  std_logic);                          -- Output of the MEM block integrated in SLF block
end entity SLF_generation;

architecture behav of SLF_generation is
    type states is (idle, State1, State2);
    constant RST_STATE: states := idle;
    signal state_reg, state_next: states;
    signal count_reg  : unsigned(4 downto 0);
    signal count_next : unsigned(4 downto 0);
    signal threshold_reg : unsigned(4 downto 0);
    signal threshold_next : unsigned(4 downto 0);
    signal equal       : std_logic;
    signal inc_count   : std_logic;
begin
    REG0: process (Clk)
    begin
        if Clk'event and Clk = '1' then
            threshold_reg <= threshold_next;
        end if;
    end process REG0;
Compute_threshold: process(threshold_reg, Bit_Nb_Sel)
begin
  threshold_next <= "01100";
  if (Bit_Nb_Sel="001") then
    threshold_next <= "10101";
  elsif (Bit_Nb_Sel="010") then
    threshold_next <= "10010";
  elsif (Bit_Nb_Sel="011") then
    threshold_next <= "01111";
  elsif (Bit_Nb_Sel="100") then
    threshold_next <= "01001";
  elsif (Bit_Nb_Sel="101") then
    threshold_next <= "01100";
  elsif (Bit_Nb_Sel="110") then
    threshold_next <= "00110";
  elsif (Bit_Nb_Sel="111") then
    threshold_next <= "00011";
  end if;
end process Compute_threshold;

-------------------------------------------------------------------------------
-- FSM
-------------------------------------------------------------------------------
-- memory element (register)
REG: process (Clk, Reset)
begin
  if Reset = '0' then
    state_reg <= RST_STATE;
  elsif Clk'event and Clk = '1' then
    state_reg <= state_next;
  end if;
end process REG;

-- next-state logic
NSL: process (state_reg, load_seed, Reset, equal)
begin
  state_next <= state_reg;  -- to avoid latches
  case state_reg is
    when idle =>
      if Reset='0' or load_seed='0' then
        state_next <= idle;
  end case;
end process NSL;
else
    state_next <= State1;
end if;
when State1 =>
    if equal='0' and Reset='1' then
        state_next <= state2;
    else
        state_next <= idle;
    end if;
when State2 =>
    if equal='0' and Reset='1' then
        state_next <= state1;
    else
        state_next <= idle;
    end if;
end case;
end process NSL;

OL1: with state_reg select
MEM_out <= '0' when idle,
    '1' when State1,
    '0' when State2;

OL2: with state_reg select
inc_count <= '0' when idle,
    '1' when State1,
    '1' when State2;

-- Counter of the number seed parts sent

counter_seed_parts_reg: process (Clk, inc_count) begin
    if inc_count = '0' then
        count_reg <= (others => '0');
    elsif Clk'event and Clk = '1' then
        count_reg <= count_next;
    end if;
end process counter_seed_parts_reg;
--combinational process to describe the counter increment
counter_seed_parts: process(count_reg, inc_count)
begin

--define default values
  count_next <= count_reg;
  if (inc_count = '1') then
    count_next <= count_reg + "00001";
  end if;
end process counter_seed_parts;

----------------------------------------------------------------------------
-- Comparator counter threshold
----------------------------------------------------------------------------

Comp_count_threshold: process(count_reg, threshold_reg)
begin

--define default values
  equal <= '0';
  if count_reg = (threshold_reg-"00001") then
    equal <= '1';
  else
    equal <= '0';
  end if;
end process Comp_count_threshold;
SLF <= inc_count;
end architecture behav;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity generate_Clk_LFSR3 is
  port(
    Clk_refresh : in std_logic;
    Clk         : in std_logic;   -- Clock of the whole system
    SLF         : in std_logic;   -- Seed has been loaded when SLF=0
    Reset       : in std_logic;   -- Reset signal active when 0
    Clk_LFSR    : out std_logic; -- Clock of LFSR
    Clk_Memory  : out std_logic); -- Clock of Memory
end entity generate_Clk_LFSR3;

architecture behav of generate_Clk_LFSR3 is
  type states is (idle, idle1, state1);
  constant RST_STATE    : states := idle;
  signal state_reg, state_next : states;
begin

  -- memory element (register)
  REG: process (Clk_refresh, Reset)
  begin
    if Reset = '0' then
      state_reg <= RST_STATE;
    elsif Clk_refresh'event and Clk_refresh='1' then
      state_reg <= state_next;
    end if;
  end process REG;

  -- next-state logic
  NSL: process (state_reg, SLF, Reset, Clk)
  begin
    Clk_LFSR <='0';
    Clk_memory <='0';
    state_next <= state_reg;  -- to avoid latches
  end process NSL;
end architecture behav;
case state_reg is
  when idle =>
    Clk_Memory <= '0';
    Clk_LFSR <= '0';
    if (SLF='0') then
      Clk_Memory <= '0';
      Clk_LFSR <= '0';
      state_next <= idle;
    elsif (SLF='1') then
      Clk_Memory <= '0';
      Clk_LFSR <= Clk;
      state_next <= idle1;
    end if;
  when idle1 =>
    Clk_Memory <= '0';
    Clk_LFSR <= Clk;
    if (SLF='0' and Reset='0') then
      Clk_Memory <= '0';
      Clk_LFSR <= '0';
      state_next <= idle;
    elsif (SLF='1') then
      Clk_Memory <= '0';
      Clk_LFSR <= Clk;
      state_next <= idle1;
    elsif (Reset='1' AND SLF='0') then
      Clk_Memory <= NOT Clk;
      Clk_LFSR <= NOT Clk;
      state_next <= state1;
    end if;
  when state1 =>
    Clk_Memory <= Clk;
    Clk_LFSR <= Clk;
    if (Reset = '0') then
      Clk_Memory <= '0';
      Clk_LFSR <= '0';
      state_next <= idle;
    end if;
end case;
end process NSL;
end architecture behav;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity control_unit2 is
    port (  
        nb_meas : in   std_logic_vector (11 downto 0);  -- Contains number of measures we want to do  
        Clk_CU :  in   std_logic;                               -- Clock of control unit  
        SLF :  in   std_logic;                         -- Seed has been loaded when SLF=0  
        Reset :  in   std_logic;                             -- Reset signal active when 0  
        Pixel_selection :  out  std_logic_vector (1 1 downto 0);   -- To select pixels form 0 to 40  
        Meas_done  :  out  std_logic;                          -- 1 meas has been done when =1  
        Im_done  :  out  std_logic);                           -- 1 picture has been done when =1
    end entity control_unit2;

architecture behav of control_unit2 is

    type states is (idle, Select_pix);
    constant RST_STATE: states := idle;

    signal state_reg, state_next : states;
    signal Number_meas_reg   :unsigned (11 downto 0);  
    signal Number_meas_next  :unsigned (11 downto 0);  
    signal Pixel_sel_reg      :unsigned (11 downto 0);  
    signal Pixel_sel_next     :unsigned (11 downto 0);  
    signal inc_pix            :std_logic;
    signal inc_meas           :std_logic;
    signal rst_pix            :std_logic;
    signal rst_meas           :std_logic;
    signal nb_pix             :std_logic_vector(11 downto 0);  
    signal Meas_done_m        :std_logic;  
    signal Im_done_m          :std_logic;  
    signal rst_meas_tmp       :std_logic;

begin

nb_pix <= (others=> '1');
-- FSM
-- memory element (register)
REG: process (Clk_CU, Reset)
begin
  if Reset = '0' then
    state_reg <= RST_STATE;
  elsif Clk_CU'event and Clk_CU = '1' then
    state_reg <= state_next;
  end if;
end process REG;

-- next-state logic
NSL: process (state_reg, SLF, Reset, Meas_done_m, Im_done_m)
begin
  state_next <= state_reg; -- to avoid latches
  case state_reg is
    when idle => if SLF='0' and Reset='1' then
      state_next <= Select_pix;
    end if;
    when Select_pix => if Reset='0' then
      state_next <= idle;
    end if;
  end case;
end process NSL;

OL: process(state_reg, SLF, Reset, Meas_done_m, Im_done_m)
begin
  inc_pix <= '0';
  inc_meas <= '0';
  rst_pix <= '1';
  rst_meas_tmp <= '1';
  case state_reg is
    when idle => if Reset='0' or SLF='1' then
      inc_pix <= '0';
      inc_meas <= '0';
      rst_pix <= '0';
      rst_meas_tmp <= '0';
    elsif Reset='0' then
      inc_pix <= '0';
      inc_meas <= '0';
      rst_pix <= '0';
      rst_meas_tmp <= '0';
    end if;
end process OL;
when Select_pix => if Reset='1' and Im_done_m='0' and Meas_done_m='0' then
    inc_pix <= '1';
    inc_meas <= '0';
    rst_pix <= '1';
    rst_meas_tmp <= '1';
elsif Reset='1' and Im_done_m='0' and Meas_done_m='1' then
    inc_pix <= '1';
    inc_meas <= '1';
    rst_pix <= '1';
    rst_meas_tmp <= '1';
elsif Reset='1' and Im_done_m='1' then
    inc_pix <= '1';
    inc_meas <= '0';
    rst_pix <= '1';
    rst_meas_tmp <= '0';
end if;
end case;
end process OL;

---------------------------------------------------------------------------
<table>
<thead>
<tr>
<th>-- Counter of the number of pixels</th>
</tr>
</thead>
</table>
---------------------------------------------------------------------------
counter_clk_1: process (Clk_CU, rst_pix)
begin
    if rst_pix = '0' then
        Pixel_sel_reg <= (others => '0');
    elsif Clk_CU'event and Clk_CU = '1' then
        Pixel_sel_reg <= Pixel_sel_next;
    end if;
end process counter_clk_1;

--combinational process to describe the counter increment
Counter_pix: process(Pixel_sel_reg, inc_pix)
begin
    --define default values
    Pixel_sel_next <=(others => '0');
    if (inc_pix='1') then
        Pixel_sel_next <= Pixel_sel_reg + "000000000001";
    end if;
end process Counter_pix;
-- Counter of the number of measures

counter_clk_2: process (Clk_CU, rst_meas)
begin
    if rst_meas = '0' then
        Number_meas_reg <= (others => '0');
    elsif Clk_CU'event and Clk_CU = '1' then
        Number_meas_reg <= Number_meas_next;
    end if;
end process counter_clk_2;

--combinational process to describe the counter increment
Counter_meas: process(Number_meas_reg, inc_meas)
begin
    --define default values
    Number_meas_next <= Number_meas_reg;
    if (inc meas = '1') then
        Number_meas_next <= Number_meas_reg + "000000000001";
    end if;
end process Counter_meas;

-- Register to cut the path between Q/QN and RN port for Numb_meas registers
-- used to break a timing loop
REG1: process(Clk_CU, rst_meas_tmp, Reset)
begin
    if Reset = '0' then
        rst_meas <= '0';
    elsif Clk_CU'event and Clk_CU = '1' then
        rst_meas <= rst_meas_tmp;
    end if;
end process REG1;

-- Comparator number of pixels
Comp_pix: process(Pixel_sel_reg, nb_pix)
begin
--define default values
  Meas_done_m <= '0';
if Pixel_sel_reg = unsigned(nb_pix) then
  Meas_done_m <= '1';
else
  Meas_done_m <= '0';
end if;
end process Comp_pix;

Comp_meas: process(Number_meas_reg, nb_meas, Meas_done_m)
begin
  --define default values
  Im_done_m <= '0';
if Number_meas_reg = unsigned(nb_meas) then
  Im_done_m <= '1';
else
  Im_done_m <= '0';
end if;
end process Comp_meas;

pixel_selection <= std_logic_vector(Pixel_sel_reg);
Im_done <= Im_done_m;
Meas_done <= Meas_done_m;
end architecture behave;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- CONSTRUCTION OF THE VARIABLE LENGTH LFSR

entity LFSR is
  port (
    Bit_Nb_Sel : in  std_logic_vector(2 downto 0);
    Seed  : in  std_logic;
    Clk  : in  std_logic;
    Reset  : in  std_logic;
    SLF  : in  std_logic;
    LFSR_output : out std_logic);
end entity LFSR;

architecture behav0 of LFSR is

-- COMPONENT XOR21_created
component XOR21_created port(a : in  std_logic;
                              b : in  std_logic;
                              z : out std_logic);
end component;

-- COMPONENT XOR41_created
component XOR41_created port(a : in  std_logic;
                              b : in  std_logic;
                              c : in  std_logic;
                              d : in  std_logic;
                              z : out std_logic);
end component;
-- COMPONENT DFF
component Flip_Flop port( clk : in std_logic;  -- Clock signal
Reset  : in std_logic;  -- Reset signal
d      : in std_logic;  -- Input signal
q      : out std_logic );  -- Output signal
end component;

-- COMPONENT MUX21_created
component MUX21_created port( a  : in std_logic;
b  : in std_logic;
    sel: in std_logic;
z  : out std_logic);
end component;

-- COMPONENT MUX71
component MUX71 port( a  : in std_logic;
b  : in std_logic;
c  : in std_logic;
d  : in std_logic;
e  : in std_logic;
f  : in std_logic;
g  : in std_logic;
    sel: in std_logic_vector(2 downto 0);
z  : out std_logic);
end component;

-- INTERMEDIATE SIGNALS
signal input_0   : std_logic;              signal input_12  : std_logic;
signal input_1   : std_logic;              signal input_13  : std_logic;
signal input_2   : std_logic;              signal input_14  : std_logic;
signal input_3   : std_logic;              signal input_15  : std_logic;
signal input_4   : std_logic;              signal input_16  : std_logic;
signal input_5   : std_logic;              signal input_17  : std_logic;
signal input_6   : std_logic;              signal input_18  : std_logic;
signal input_7   : std_logic;              signal input_19  : std_logic;
signal input_8   : std_logic;              signal input_20  : std_logic;
signal input_9   : std_logic;              signal input_21  : std_logic;
signal input_10  : std_logic;              signal input_11  : std_logic;
signal input_3_1 : std_logic;              signal input_15_1 : std_logic;
signal input_18_1 : std_logic;              signal input_6_1  : std_logic;
signal input_9_1 : std_logic;              signal input_12_1 : std_logic;
signal bit3      : std_logic;              signal bit15     : std_logic;
signal bit6      : std_logic;              signal bit18     : std_logic;
begin

-- All flip flops

Flip_Flop_1 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_0, q => input_1);

Flip_Flop_2 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_1, q => input_2);

Flip_Flop_3 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_2, q => input_3);

p1: process (input_3, Bit_Nb_SEL)
begin
  case Bit_Nb_SEL is
    when "111" => input_3_1 <= '0';
    when others => input_3_1 <= input_3;
  end case;
end process p1;

Flip_Flop_4 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_3_1, q => input_4);

Flip_Flop_5 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_4, q => input_5);

Flip_Flop_6 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_5, q => input_6);

p2: process (input_6, Bit_Nb_SEL)
begin
  case Bit_Nb_SEL is
    when "110" => input_6_1 <= '0';
    when others => input_6_1 <= input_6;
  end case;
end process p2;

Flip_Flop_7 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_6_1, q => input_7);

Flip_Flop_8 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_7, q => input_8);

Flip_Flop_9 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_8, q => input_9);
p3: process (input_9, Bit_Nb_Sel)
begin
  case Bit_Nb_Sel is
    when "100" => input_9_1 <= '0';
    when others => input_9_1 <= input_9;
  end case;
end process p3;

Flip_Flop_10 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_9_1, q => input_10);
Flip_Flop_11 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_10, q => input_11);
Flip_Flop_12 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_11, q => input_12);

p4: process (input_12, Bit_Nb_Sel)
begin
  case Bit_Nb_Sel is
    when "101" => input_12_1 <= '0';
    when others => input_12_1 <= input_12;
  end case;
end process p4;

Flip_Flop_13 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_12_1, q => input_13);
Flip_Flop_14 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_13, q => input_14);
Flip_Flop_15 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_14, q => input_15);

p5: process (input_15, Bit_Nb_Sel)
begin
  case Bit_Nb_Sel is
    when "011" => input_15_1 <= '0';
    when others => input_15_1 <= input_15;
  end case;
end process p5;

Flip_Flop_16 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_15_1, q => input_16);
Flip_Flop_17 : component Flip_Flop
  port map (clk => Clk, Reset => Reset, d => input_16, q => input_17);
Flip_Flop_18 : component Flip_Flop
    port map (clk => Clk , Reset => Reset, d => input_17 , q=> input_18);

p6: process (input_18, Bit_Nb_Sel)
begin
    case Bit_Nb_Sel is
        when "010" => input_18_1 <= '0';
        when others => input_18_1 <= input_18
    end case;
end process p6;

Flip_Flop_19 : component Flip_Flop
    port map (clk => Clk , Reset => Reset, d => input_18_1, q=> input_19);
Flip_Flop_20 : component Flip_Flop
    port map (clk => Clk , Reset => Reset, d => input_19 , q=> input_20);
Flip_Flop_21 : component Flip_Flop
    port map (clk => Clk , Reset => Reset, d => input_20 , q=> input_21);

-- Compute feedbacks

XOR21_1 : component XOR21_created
    port map (a => input_3, b => input_2, z => bit3);
XOR21_2 : component XOR21_created
    port map (a => input_6, b => input_5, z => bit6);
XOR21_3 : component XOR21_created
    port map (a => input_9, b => input_5, z => bit9);
XOR41_1 : component XOR41_created
    port map (a => input_12, b => input_4, c => input_1, d => input_1, z => bit12);
XOR21_4 : component XOR21_created
    port map (a => input_14, b => input_15, z => bit15);
XOR21_5 : component XOR21_created
    port map (a => input_18, b => input_11, z => bit18);
XOR21_6 : component XOR21_created
    port map (a => input_21, b => input_19, z => bit21);

-- Choose the feedback

MUX71_1 : component MUX71
    port map (a => bit21, b => bit18, c => bit15, d => bit9, e => bit12, f => bit6, g => bit3, sel => Bit_Nb_Sel ,
    z=> feedback);
-- Choose the output

MUX71_2 : component MUX71
    port map (a => input_21, b => input_18, c => input_15, d => input_9, e => input_12, f => input_6, g => input_3, sel => Bit_Nb_Sel, z => LFSR_output);

-- Choose the input of the LFSR: feedback or seed

tmp_SLF_1 <= NOT SLF;
MUX21_1 : component MUX21_created
    port map (a => Seed, b => feedback, sel => tmp_SLF_1, z => input_0);

end architecture behav0;
A9.13. The 12 bit decoder

A9.13.1. Decoder 6 bits

-- Author: Bibet Alexandre
-- Time: May 2008
-- Place: EPFL

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity decoder_6bits is
  port (  
    Pixel_selection_6bits : in   std_logic_vector (5 downto 0);
    Decoder_out_6bits      :  out  std_logic_vector (63 downto 0));
end entity decoder_6bits;

architecture behav2 of decoder_6bits is
begin

p1: process (Pixel_selection_6bits)
begin
  Decoder_out_6bits <= (others => '0');  -- Initialization
  case Pixel_selection_6bits is
  when "000000" =>  Decoder_out_6bits(0) <= '1';
  when "000001" =>  Decoder_out_6bits(1) <= '1';
  when "000010" =>  Decoder_out_6bits(2) <= '1';
  when "000011" =>  Decoder_out_6bits(3) <= '1';
  when "000100" =>  Decoder_out_6bits(4) <= '1';
  when "000101" =>  Decoder_out_6bits(5) <= '1';
  when "000110" =>  Decoder_out_6bits(6) <= '1';
  when "000111" =>  Decoder_out_6bits(7) <= '1';
  when "001000" =>  Decoder_out_6bits(8) <= '1';
  when "001001" =>  Decoder_out_6bits(9) <= '1';
  when "001010" =>  Decoder_out_6bits(10) <= '1';
  when "001011" =>  Decoder_out_6bits(11) <= '1';
  when "001100" =>  Decoder_out_6bits(12) <= '1';
  when "001101" =>  Decoder_out_6bits(13) <= '1';
  when "001110" =>  Decoder_out_6bits(14) <= '1';
  when "001111" =>  Decoder_out_6bits(15) <= '1';
  when others => Decoder_out_6bits <= "000000";
  end case;
end process p1;

end architecture behav2;
when "010001" => Decoder_out_6bits(17) <= '1';
when "010010" => Decoder_out_6bits(18) <= '1';
when "010011" => Decoder_out_6bits(19) <= '1';
when "010100" => Decoder_out_6bits(20) <= '1';
when "010101" => Decoder_out_6bits(21) <= '1';
when "010110" => Decoder_out_6bits(22) <= '1';
when "010111" => Decoder_out_6bits(23) <= '1';
when "011000" => Decoder_out_6bits(24) <= '1';
when "011001" => Decoder_out_6bits(25) <= '1';
when "011010" => Decoder_out_6bits(26) <= '1';
when "011011" => Decoder_out_6bits(27) <= '1';
when "011100" => Decoder_out_6bits(28) <= '1';
when "011101" => Decoder_out_6bits(29) <= '1';
when "011110" => Decoder_out_6bits(30) <= '1';
when "011111" => Decoder_out_6bits(31) <= '1';
when "100000" => Decoder_out_6bits(32) <= '1';
when "100001" => Decoder_out_6bits(33) <= '1';
when "100010" => Decoder_out_6bits(34) <= '1';
when "100011" => Decoder_out_6bits(35) <= '1';
when "100100" => Decoder_out_6bits(36) <= '1';
when "100101" => Decoder_out_6bits(37) <= '1';
when "100110" => Decoder_out_6bits(38) <= '1';
when "100111" => Decoder_out_6bits(39) <= '1';
when "101000" => Decoder_out_6bits(40) <= '1';
when "101001" => Decoder_out_6bits(41) <= '1';
when "101010" => Decoder_out_6bits(42) <= '1';
when "101011" => Decoder_out_6bits(43) <= '1';
when "101100" => Decoder_out_6bits(44) <= '1';
when "101101" => Decoder_out_6bits(45) <= '1';
when "101110" => Decoder_out_6bits(46) <= '1';
when "101111" => Decoder_out_6bits(47) <= '1';
when "110000" => Decoder_out_6bits(48) <= '1';
when "110001" => Decoder_out_6bits(49) <= '1';
when "110010" => Decoder_out_6bits(50) <= '1';
when "110011" => Decoder_out_6bits(51) <= '1';
when "110100" => Decoder_out_6bits(52) <= '1';
when "110101" => Decoder_out_6bits(53) <= '1';
when "110110" => Decoder_out_6bits(54) <= '1';
when "110111" => Decoder_out_6bits(55) <= '1';
when "111000" => Decoder_out_6bits(56) <= '1';
when "111001" => Decoder_out_6bits(57) <= '1';
when "111010" => Decoder_out_6bits(58) <= '1';
when "111011" => Decoder_out_6bits(59) <= '1';
when "111100" => Decoder_out_6bits(60) <= '1';
when "111101" => Decoder_out_6bits(61) <= '1';
when "111110" => Decoder_out_6bits(62) <= '1';
when "111111" => Decoder_out_6bits(63) <= '1';
when others => Decoder_out_6bits <= (others => '0');
end case;
end process p1;
end architecture behav2;
A9.13.2. Decoder 12 bits second

-- Author: Bibet Alexandre
-- Time: May 2008
-- Place: EPFL

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- CONSTRUCTION OF THE SECOND STAGE OF THE 12 BITS DECODER COMPONENT BY
-- INSTANTIATING THE 6 BITS DECODER AND AND21 GATES

entity decoder_12bits_second is
port (    Pixel_selection_6bits_second : in  std_logic_vector(5 downto 0);
    Prev_decoder_6bits             : in  std_logic;
    Decoder_6bits_out_second : out std_logic_vector(6 3 downto 0));
end decoder_12bits_second;

architecture behav4 of decoder_12bits_second is

-- COMPONENT DECODER 6 BITS
component decoder_6bits port(    Pixel_selection_6bits : in  std_logic_vector (5 downto 0);
    Decoder_out_6bits   : out std_logic_vector (63 downto 0) );
end component;

component AND_21 port(    a,b: in  std_logic;
    z : out std_logic);
end component;

-- INTERMEDIATE SIGNALS
signal tmpa : std_logic_vector (63 downto 0);

begin

decoder_6bits_1: component decoder_6bits
    port map (Pixel_selection_6bits => Pixel_selection_6bits_second(5 downto 0) , Decoder_out_6bits =>
         tmpa(63 downto 0));
one: for i in 63 downto 0 generate
    AND_21_i: component AND_21
        port map (a => Prev_decoder_6bits, b => tmpa(i), z => Decoder_6bits_out_second(i));
    end generate one;
end architecture;
A9.13.3. Decoder 12 bits

-- Author: Bibet Alexandre
-- Time: May 2008
-- Place: EPFL

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- CONSTRUCTION OF THE 12 BITS DECODER COMPONENT BY INSTANTIATING THE 6 BITS
DECODER PLUS THE SECOND STAGE COMPONENT

entity decoder_12bits is
port (  
  Pixel_selection_12bits  : in  std_logic_vector(11 downto 0);
  Decoder_12bits_out      : out std_logic_vector(4095 downto 0)));
end decoder_12bits;

architecture behav3 of decoder_12bits is

-- COMPONENT DECODER 6 BITS
component decoder_6bits port(
  Pixel_selection_6bits : in std_logic_vector(5 downto 0);
  Decoder_out_6bits : out std_logic_vector(63 downto 0)));
end component;

component decoder_12bits_second port(  
  Pixel_selection_6bits_second: in std_logic_vector (5 downto 0);
  Prev_decoder_6bits : in std_logic;
  Decoder_6bits_out_second : out std_logic_vector(63 downto 0));
end component;

-- INTERMEDIATE SIGNALS
signal tmpf : std_logic_vector (63 downto 0);
begin
  decoder_6bits_1: component decoder_6bits
      port map (Pixel_selection_6bits => Pixel_selection_12bits(11 downto 6), Decoder_out_6bits => tmpf(63 downto 0));
Second_stage: for i in 63 downto 0 generate

    decoder_12bits_second_i: component decoder_12bits_second

        port map (Pixel_selection_6bits_second => Pixel_selection_12bits(5 downto 0), Prev_decoder_6bits => tmpf(i), Decoder_6bits_out_second => Decoder_12bits_out(64*(i+1)-1 downto 64*i));

    end generate Second_stage;
end architecture;
A9.14. Memory

-- Author: Bibet Alexandre
-- Time: May 2008
-- Place: EPFL

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- CONSTRUCTION OF THE MEMORY ELEMENT
entity Memory is
port ( decoder_output : in  std_logic;
LFSR_output_delayed : in  std_logic;
Clk_LFSR_delayed : in  std_logic;
Reset   : in  std_logic;
    Q   : out std_logic);
end Memory;

architecture behav of Memory is

-- COMPONENT Flip Flop
component Flip_Flop port( clk : in  std_logic;  -- Clock signal
Reset : in  std_log ic;  -- Reset signal
d   : in  std_logic ;  -- Input signal
q    : out std_logi c);  -- Output signal
end component;

signal I1 : std_logic;
signal I2 : std_logic;

begin

Process(decoder_output, LFSR_output_delayed, Clk_LFSR_delayed)
begin
if (decoder_output='1') then
I1 <= LFSR_output_delayed;
I2 <= Clk_LFSR_delayed;
end if;
end Process;
elsif (decoder_output='0') then
    I1 <= '0';
    I2 <= '0';
end if;
end process;

Flip_Flop_1: component Flip_Flop
    port map (clk => I2, Reset => Reset, d => I1, q => Q);

end architecture;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

-- CONSTRUCTION OF THE PART WITH SLF_generation AND Control_Unit
entity part1 is
port (
  Clk_fast  : in   std_logic;
  Reset     : in   std_logic;
  Reset3    : in   std_logic;
  Sel_seed  : in   std_logic;
  Seed_applied  : in   std_logic;
  Bit_Nb_Sel : in   std_logic_vector(2 downto 0);
  load_seed : in   std_logic;
  nb_meas   : in   std_logic_vector(11 downto 0);
  Clk_LFSR_apply : in   std_logic;
  Sel_LFSR_Clk  : in   std_logic;
  Clk      : out  std_logic;
  Clk_refresh : out  std_logic;
  Clk_CU   : out  std_logic;
  SLF      : out  std_logic;
  Im_done  : out  std_logic;
  Meas_done : out  std_logic;
  Pixel_selection : out  std_logic_vector(11 downto 0);
  MEM_out   : out  std_logic;
  Decoder_12bits_out : out  std_logic_vector(4095 downto 0);
  Seed     : out  std_logic;
  Clk_LFSR : out  std_logic;
  LFSR_output : out  std_logic;
  Q        : out  std_logic;
  Clk_Memory : out  std_logic;
end entity part1;
architecture behav of part1 is

-- COMPONENT CLK DIVIDER
component Clk_divider port( Clkin, rst_b : in  std_logic;
                           Clkout : out std_logic_vector(0 to 2));
end component;

-- COMPONENT LFSR
component LFSR port( Bit_Nb_Sel : in  std_logic_vector(2 downto 0);
                     Seed : in std_logic;
                     Clk : in std_logic;
                     Reset : in std_logic;
                     SLF : in std_logic;
                     LFSR_output : out std_logic);
end component;

-- COMPONENT MEMORY
component Memory port( decoder_output : in  std_logic;
                       LFSR_output_delayed : in  std_logic;
                       Clk_LFSR_delayed : in  std_logic;
                       Reset : in  std_logic;
                       Q : out std_logic);
end component;

-- COMPONENT MUX21
component MUX21_created port( a : in  std_logic;
                              b : in std_logic;
                              sel: in std_logic;
                              z : out std_logic);
end component;

-- COMPONENT DELAY
component delay port( A : in  std_logic;
                     Reset : in  std_logic;
                     Clk_fast : in  std_logic;
                     Z : out std_logic);
end component;

-- COMPONENT SLF_generation
component SLF_generation port(clk : in  std_logic;
                              Reset : in  std_logic;
                              Bit_Nb_Sel : in  std_logic_vector(2 downto 0);
                              load_seed : in  std_logic);
SLF : out std_logic;
MEM_out : out std_logic);
end component;

-- COMPONENT control_unit2
component control_unit2 port( nb_meas : in std_logic_vector (11 downto 0);
Clk_CU            : in std_logic;
SLF               : in std_logic;
Reset             : in std_logic;
Pixel_selection : out std_logic_vector (11 downto 0);
Meas_done         : out std_logic;
Im_done    : out std_logic);
end component;

-- COMPONENT decoder_12bits
component decoder_12bits port( Pixel_selection_12bits : in std_logic_vector(11 downto 0);
Decoder_12bits_out : out std_logic_vector(4095 downto 0));
end component;

-- COMPONENT generate_Clk_LFSR3
component generate_Clk_LFSR3 port( Clk_refresh : in std_logic;
clk               : in std_logic;
SLF         : in std_logic;
Reset      : in std_logic;
Clk_LFSR          : out std_logic;
Clk_Memory        : out std_logic);
end component;

-- INTERMEDIATE SIGNALS
signal tmp_SLF_1             : std_logic;
signal tmp_SLF_2             : std_logic;
signal tmp_Clk              : std_logic;
signal tmp_Clk_2             : std_logic;
signal tmp_Pixel_selection_1 : std_logic_vector(11 downto 0);
signal tmp_MEM_out_1         : std_logic;
signal tmp_Im_done_1         : std_logic;
signal tmp_Meas_done_1       : std_logic;
signal tmp_Clk_LFSR_1        : std_logic;
signal tmp_Clk_LFSR_2        : std_logic;
signal tmp_Clk_LFSR_3        : std_logic;
signal tmp_seed_1            : std_logic;
signal tmp_LFSR_output       : std_logic;
signal tmp_dec_12bits_out : std_logic_vector(4095 downto 0);
signal tmp_Clk_Memory_1 : std_logic;
signal tmp_Clk_refresh : std_logic;
signal tmp_Clkout : std_logic_vector(0 to 2);

begin

Clk_divider_1: component Clk_divider
  port map (Clkin => Clk_fast, rst_b => Reset3, Clkout => tmp_Clkout);
  tmp_Clk <= tmp_Clkout(2);
  Clk <= tmp_Clk;
  tmp_Clk_refresh <= tmp_Clkout(1);
  Clk_refresh <= tmp_Clk_refresh;

SLF_generation_1: component SLF_generation
  port map (clk => tmp_Clk, Reset => Reset, Bit_Nb_Sel => Bit_Nb_Sel, load_seed => load_seed, SLF => tmp_SLF_1, MEM_out => tmp_MEM_out_1);
  SLF <= tmp_SLF_1;
  MEM_out <= tmp_MEM_out_1;

delay_3: component delay
  port map (A => tmp_Clk, Reset => Reset3, Clk_fast => Clk_fast, Z => tmp_Clk_2);

delay_4: component delay
  port map (A => tmp_Clk_Memory_1, Reset => Reset3, Clk_fast => Clk_fast, Z => tmp_Clk_LFSR_3);

control_unit2_1: component control_unit2
  port map (nb_meas => nb_meas, Clk_CU => tmp_Clk_Memory_1, SLF => tmp_SLF_1, Reset => Reset, Pixel_selection => tmp_Pixel_selection_1, Meas_done => tmp_Meas_done_1, Im_done => tmp_Im_done_1);
  Pixel_selection <= tmp_Pixel_selection_1;
  Im_done <= tmp_Im_done_1;
  Meas_done <= tmp_Meas_done_1;

decoder_12bits_1: component decoder_12bits
  port map (Pixel_selection_12bits => tmp_Pixel_selection_1, Decoder_12bits_out => tmp_dec_12bits_out);
  Decoder_12bits_out <= tmp_dec_12bits_out;

MUX21_1: component MUX21_created
  port map (a => tmp_MEM_out_1, b => Seed_applied, sel => Sel_seed, z => tmp_seed_1);
  Seed <= tmp_seed_1;
generate_Clk_LFSR3_1: component generate_Clk_LFSR3
    port map (Clk_refresh => tmp_Clk_refresh, clk => tmp_Clk_2, SLF => tmp_SLF_1, Reset => Reset,
        Clk_LFSR => tmp_Clk_LFSR_1, Clk_Memory => tmp_Clk_Memory_1);
    Clk_Memory <= tmp_Clk_Memory_1;
    Clk_CU <= tmp_Clk_Memory_1;

MUX21_2: component MUX21_created
    port map (a => tmp_Clk_LFSR_1, b => Clk_LFSR_apply, sel => Sel_LFSR_Clk, z =>
        tmp_Clk_LFSR_2);
    Clk_LFSR <= tmp_Clk_LFSR_2;

LFSR_1: component LFSR
    port map (Bit_Nb_Sel => Bit_Nb_Sel, Seed => tmp_seed_1, clk => tmp_Clk_LFSR_2, Reset => Reset,
        SLF => tmp_SLF_1, LFSR_output => tmp_LFSR_output);
    LFSR_output <= tmp_LFSR_output;

Memory_stage: for i in 4095 downto 0 generate
    Memory_i: component Memory
        port map (decoder_output => tmp_dec_12bits_out(i), LFSR_output_delayed => tmp_LFSR_output,
            Clk_LFSR_delayed => tmp_Clk_LFSR_3, Reset => Reset, Q => Q(i));
end generate memory_stage;

end architecture behav;