**Technical University of Cluj-Napoca**
Faculty of Automation and Computer Science
Computer Science Department

# Rolling Average System of a Parallel 8-bit Data Stream

Todea Tudor Ștefan
Reckerth Daniel Peter
Group **30414**

Coordinating Professor: Octavian Creț

Digital System Design

# Table of Contents

# I. Project's specification

## 1. Defining the task

This project and the system developed within it represents a simple signal processing system that will calculate the rolling average of a parallel 8-bit data stream.

Calculating the numerical average value for a given input data stream is always a necessity within signal processing systems. A simple low pass filter is implemented, which smooths out rapid changes in value from the data stream but maintaining the overall direction. The more smoothing will occur for a greater number of samples used for calculating the average. This system is required to run in "real time" and output the average value at the same rate as the original input data.

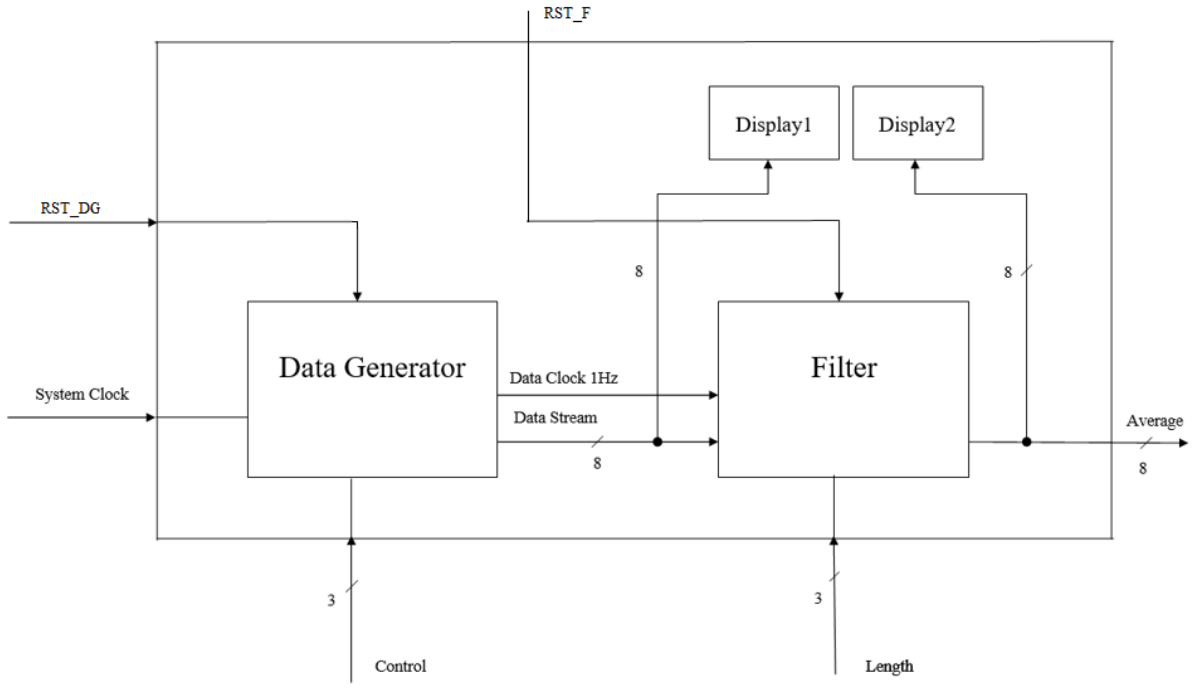
Therefore, the task was to develop a VHDL based model for this so called "Digital Filter / Rolling Average" system combined with a data stream generator. It will be implemented on a self-contained Xilinx/Digilent FPGA board to allow demonstration of a working system. Furthermore, the software tools used – VHDL and Xilinx ISE WebPack – include the use of the Modelsim simulation tools for design verification.

The system is made from two big modules, the average filter and the data stream generator, which generates number automatically. For both of these we have different input settings – representing also the system inputs - which will establish the mode and the behavior of the whole system. For the data generator we have a "Control Setting" which will generate numbers differently, depending on the inputted sequence, and for the filter we have a "Buffer Length Setting" which will sample different averages from two up to sixteen.

# II. Block Diagram and Component List

## 1. Block Diagram

The system's block diagram consists of the two main modules – data generator and average filter - and the inputs and outputs, especially the control setting for the data generator and the length one for the filter average. Below is the main system's block diagram.




## 2. Main Component List

The main modules and their components are presented below:

- Frequency Divider *(code - Annex 1.10)* : the system's clock has a frequency of 100Mhz but as the filter is required to run in real time, a frequency divider for the data clock was needed; thus, the `Data Clock` signal of the filter has a frequency of 1Hz.

- Data Generator *(code – Annex 1.2)*

  - Square wave generator *(code – Annex 1.7)* – generates an 8-bit number at a frequency of 0.25.

- The 6 repeated digits *(code – Annex 1.5 & 1.6)* - for student one and the 6 one for student two are different entities where the numbers are stored in six 8-bit circular shift registers, the numbers being already written; they could represent anything for example the telephone numbers of the two students

- Linear feedback shift register (LFSR) *(code – Annex 1.3)* - on 4 bits was used to generate pseudo random sequence of reduced range 0 to 15, using a 4-bit shift register with feedback represented by a combinational logic circuit containing XOR gates.

- Linear feedback shift register (LFSR) *(code – Annex 1.4)* - on 8 bits was used to generate pseudo random sequence of full range 0 to 15, on the same idea.

- Filter *(code – Annex 1.8)* – one of the main components used for calculating the average in real time of the numbers received from the data generator; calculates the average of 2 up to 16 numbers and for doing that it stores them in some registers and use shifting operations on bits for divisions

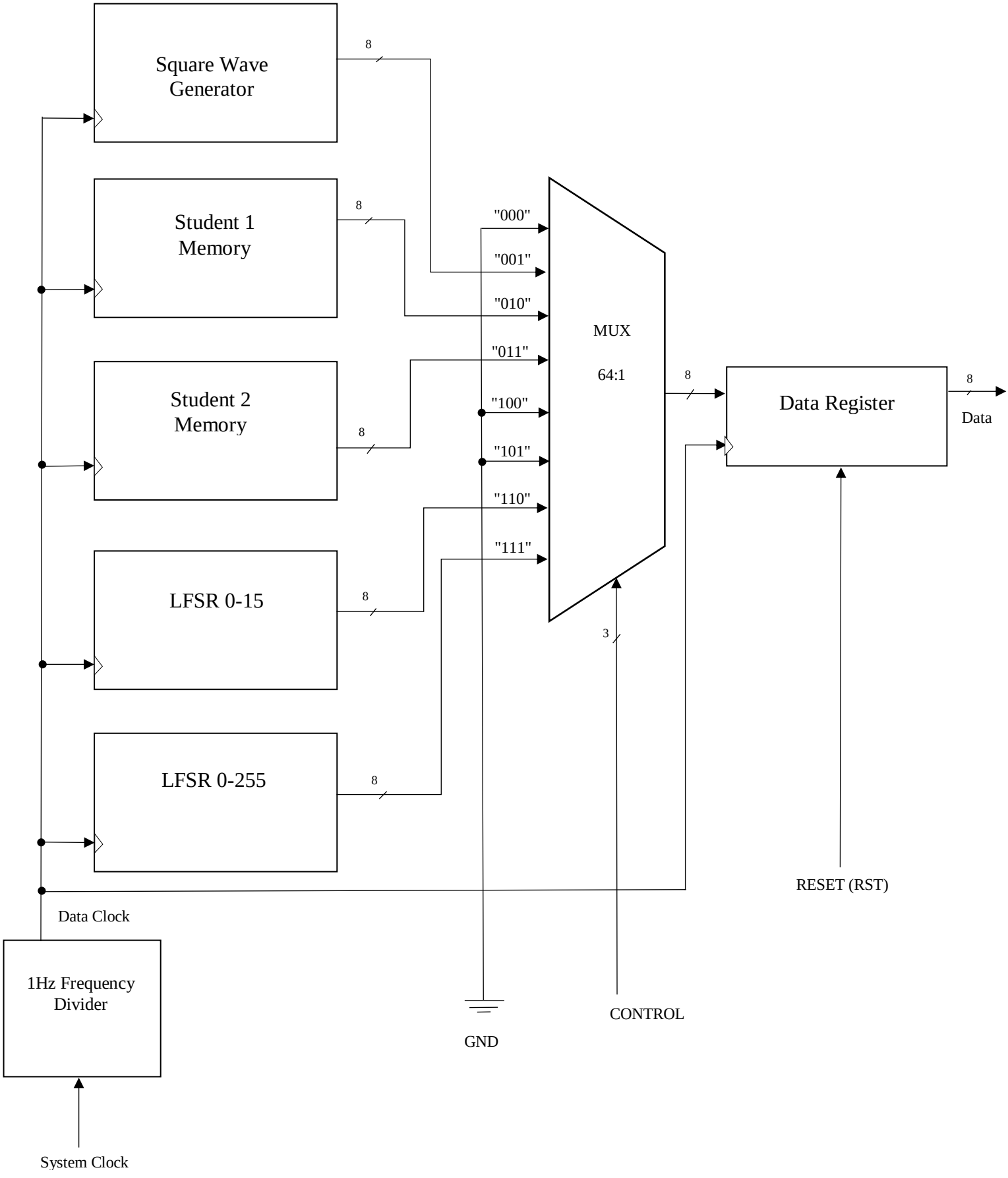
- Display *(code – Annex 1.9)* - based on 2 major components

  - Converter *(code – Annex 1.10)* - a component which tests the 4 bit number that is given and assigns a particular 7 bits sequence to its output, which will lead to the hexadecimal equivalent

  - 7 Segment Frequency Divider *(code – Annex 1.11)* - since the anodes, with their respective cathodes, light up sequentially, we need them to light up one after another at such a high frequency that we cannot tell that they are not lit up all at the same time, an optical "illusion"

# III. Designing and Implementation

## 1. Data Generator



The above detailed logic scheme represents the internal architecture of the data generator. When designing it we followed a bottom-up approach, starting in mind with the small modules that together linked form the main component. This representation is not the smallest level of the data generator, but it presents its main subcomponents.

In this sense, for designing this component we adopted a structural approach meaning that in the main module we declared as components all the submodules and we instantiated them, as we did not do everything in one big file, or process. The respective subcomponents are designed differently, some by following a combination of data flow and behavioral methods for designing.

As it can be seen, the data stream generator has as its inputs the following: `System Clock`, `Reset (RST)`, 3-bit `Control` switches.

The `System Clock` is the clock of the FPGA board, Nexys4, having a frequency of 100Mhz. This means that the clock changes 100 million times per second. However, the whole process needs to be synchronized to the `Data Clock` of 1Hz. And for this we use a frequency divider.

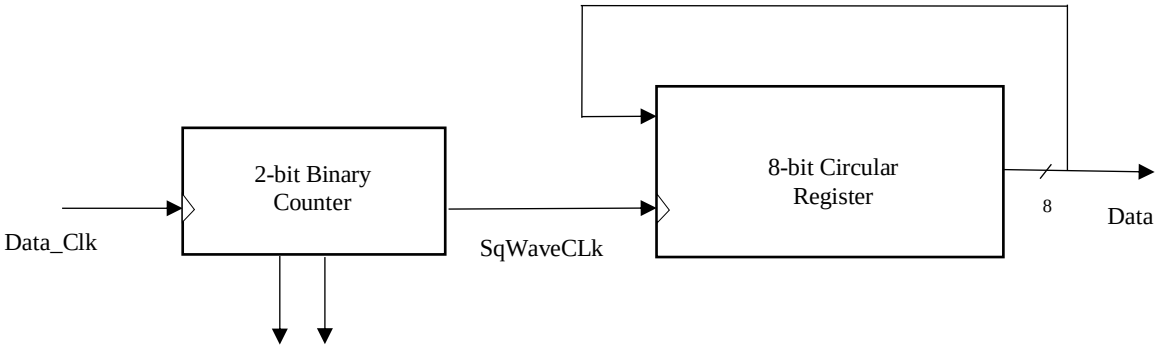
The `RST` button, when pushed will action at the level of the 8-bit register which receives the specific number from the multiplexer and will determine an 8-bit zero value.

The 3-bit `Control` will represent the selections for the cascaded multiplexers and will determine the mode for the data generator in accordance with the six modes given by the requirement. For the other two unmentioned cases, we will generate an 8-bit zero number as we would have in the case of a `Reset` state. The working modes for the data generator are:

| **Control Setting** | |
|---|---|
| Off – Off – Off ("000") | Test Mode o/p 0 (zero) |
| Off – Off – On ("001") | Square wave (0.25 x data clock) |
| Off – On – Off ("010") | Repeated 6 digit Sequence for Student 1 |
| Off – On – On ("011") | Repeated 6 digit Sequence for Student 2 |
| On – On – Off ("110") | Pseudo Random Sequence reduced range 0 to 15 |
| On – On – On ("111") | Pseudo Random Sequence full range 0 to 255 |

As we know, the data generator is made up of six subcomponents, namely the square wave generator, the two storing registers for the two students, and the reduced and full linear feedback shift registers. We will present them now in detail:

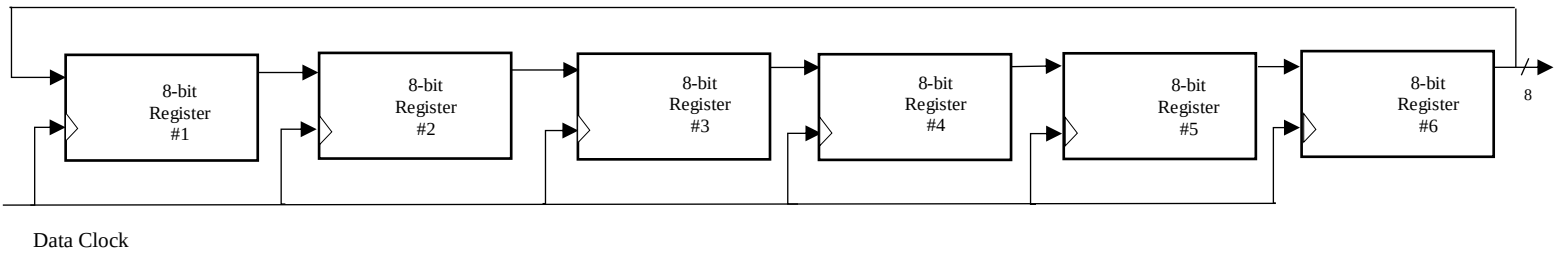
A. Square Wave Generator




The square wave component has as input the `Data_Clock` and as output the data, represented by the `OP` signal.

Therefore, we made an entity/architecture pair for the square wave generator in which we also declared an internal output signal `OP_INT` signal of 8 bits loaded with the value "00001000" which will be the changed data generated output after different circulars shifting. The operations are done in a process dependent on the `CLK`, in the sequential domain and they mainly represents the shifting movements as bits 6 down to 0 are assigned the bits from 7 down to 1, with the seventh one receiving the first one – a circular register. Also, in the declarative zone of the process we declared an integer variable count of range 0 to 3 initialized with 0, which will help us delay the data clock by four times, at a rate of 0.25Hz. For this reason, within the process, on the rising edge of the clock, the count increments by one and if hitting the value 3 then the shifting operations are performed. As the process is sensitive to the clock the count will be reinitialized with 0. For this we simply made use of the *if ... then* statement. The code is provided in the Annex at the section 1.7.

B. Memory registers for both students



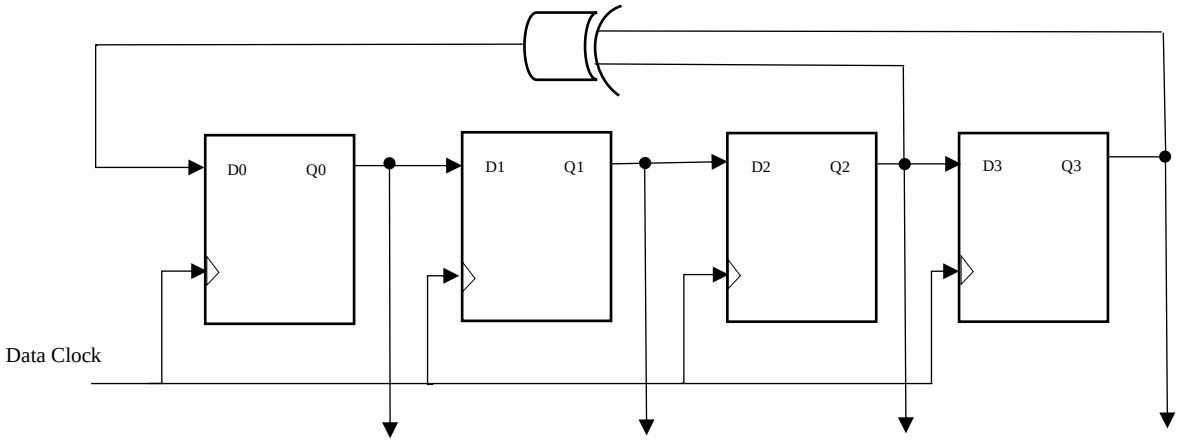

The memory registers used for storing the numbers for the both students are six per student, each of 8-bit, each one receiving data from the previous one, with the first one receiving it circularly from the last one.

For this we made two different entities, which have as input a `CLK` and as output a `DATA` signal on 8 bits. Both architectures contain a process dependent on the 1Hz Data Clock. In them, we declared a user-defined type – `StudentMEM` – an array of width 6 of 8-bit standard logic vector type (*std_logic_vector (7 downto 0)*). We also declared a variable of the type `StudentMEM` called `student1` and `student2` respectively for the other process, and initialized them with the student's respective numbers: for student one we have written the registers with numbers 7, 2, 3, 8, 9, 1, and for the second student we have the numbers 5, 8, 7, 4, 5, 3.

We also took help from a variable with auxiliary role, called `temp`, for operating and circulating the numbers in the registers. Basically, `temp` gets assigned the sixth register, then all registers from the sixth one to the second one receiving the values from the fifth one down to the first one, and the first one receives the value of `temp`, i.e. the sixth one, which is then given as data generated stream. Since we operate on variables, the assignments are immediate.

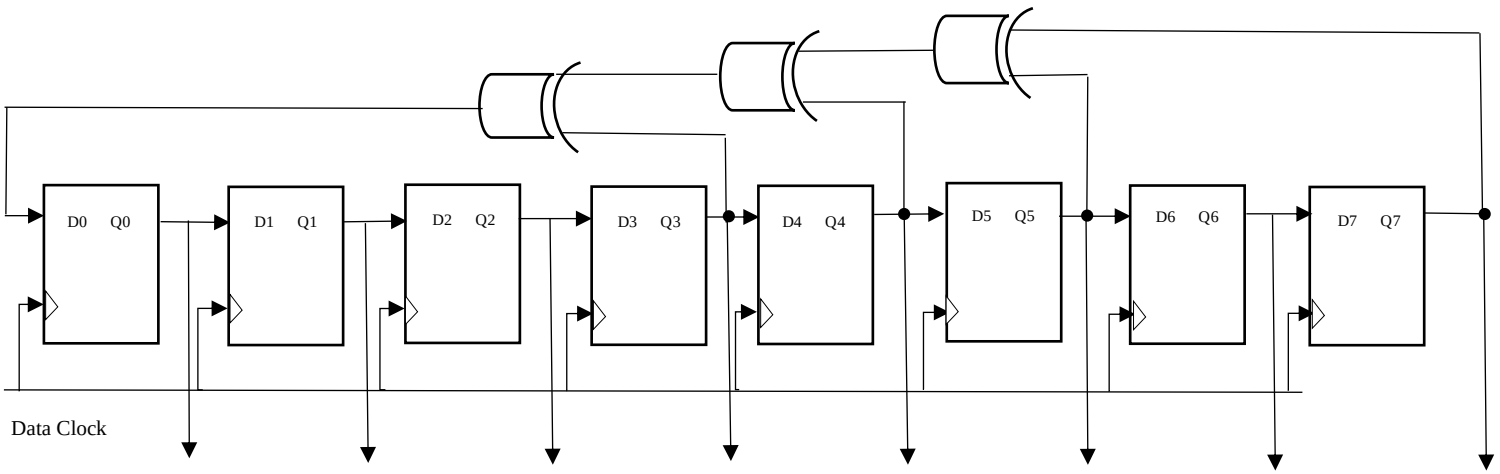
C. Linear Feedback Shift Register

Figure 1: reduced range 0 - 15




This diagram shows a 4-bit linear feedback shift register (LFSR), which is made of a shift register with a feedback through an exclusive-OR gate, acting together to produce a pseudo-random binary sequence (PRBS) at each of the flip-flop outputs. By correctly choosing the "tap points" at which we take feedback from the 4-bit shift register, we can produce a PRBS of length $2^4 - 1 = 15$, a maximal-length sequence including all possible patterns (or vectors) of 4 bits, excluding the all-zeros pattern.

Figure 2: full range 0 – 255



This is also a linear shift register, but it is on 8-bit. The difference between the previous one is that here other "tap points" were chosen. That is because we want a greater range of values, and choosing specific tap points will help generate the required maximum length pseudo-random binary sequence.

Both LFRS have been made in different entities which have as inputs the `CLK` and as outputs the `Data` on 8 bits. In designing them we used the same principles of shfiting registers, in which we have an internal signal called `OP_INT,` on 4 and respectively 8 bits both initialized with non-zero values. Then we have a process sensitive to a `CLK` and in its body we do the operations, i.e. we assign the value of the flip-flops 3 down to 1 the values of the ones from 2 down to 0, and then the first flip-flop will receive the XOR-ed feedback. For the second LFRS the operations include auxiliary signals which hold the XOR feedback as we have more "tap points". Also, as we have an 8-bit output, but a 4-bit LFSR for the first case, we assign to the 4 most significant bits "0000" and the other 4 the results from the registers. In the second LFRS this is not the case.

## 2. Filter

The Filter, as is described in the provided documentation, or the *average_computer* as we called it in our project, is required to have several inputs:

`Length` indicates the number of elements for which we will compute the average as described below:

| | |
|---|---|
| Off - Off - Off (0 - 0 - 0) | Stop - Hold Value |
| On - Off - Off (1 - 0 - 0) | 2 Sample Average |
| On - Off - On (1 - 0 - 1) | 4 Sample Average |
| On - On - Off (1 - 1 - 0) | 8 Sample Average |
| On - On - On (1 - 1 - 1) | 16 Sample Average |

`Number` is an 8-bit input provided by the data generator described previously.

`Data Clock` input is generated by the frequency divider described earlier, which converts the 100 MHz internal clock of the Nexys4 board to a 1 Hz clock that will be used for the Filter.

`Reset`, taken directly from the user and it clears the whole Filter module, loading “00000000” on all 16 registers that store our numbers.

We only have one output when it comes to the average computing module, namely the `Average`. This is the average of the numbers stored in our module depending on the length desired by the user. This output will be used by our 7 segment display module to display our number. This process will be described further down our documentation.

We receive the 8-bit `Data` input from the data generator. We store at most 16 numbers of 8 bits in our array of 16 memory registers. When a new number is received from the generator, the left most number stored in our array is discarded.

We compute the sum of the last two numbers entered and store the result in the register named *"AVG 2"* after shifting it to the right with one position in order to compute the average of those two numbers. We then compute the sum of the first 4 numbers, shift it right by two positions in order to compute the average of those four numbers, and store it in the register called *"AVG 4"*. We do the same with the first 8 numbers, shifting the result right by three positions and storing it in the register called *"AVG 8"*. Finally, we add all of the 16 numbers, shift the result by 4 positions and store it in the *"AVG 16"* register.

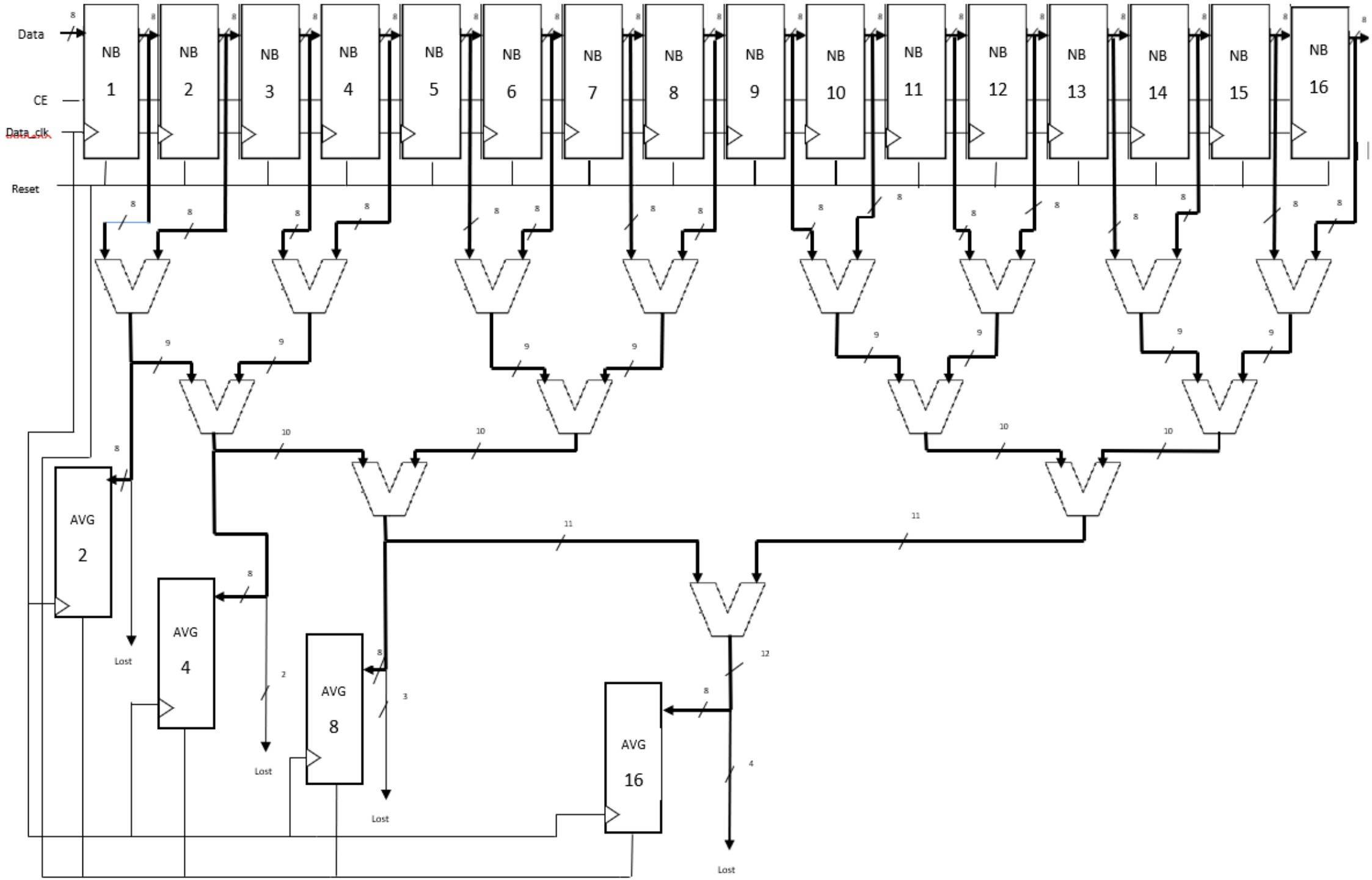
As shown in the schematics below, each memory register that stores a number has a *CE (Clock Enable)* input. The purpose of this input is to disable the `Data_clk` input if the `Length` input selected by the user is "000". By disabling this input, we will no longer store the numbers received from the generator, thus "freezing" the last displayed value of the average.

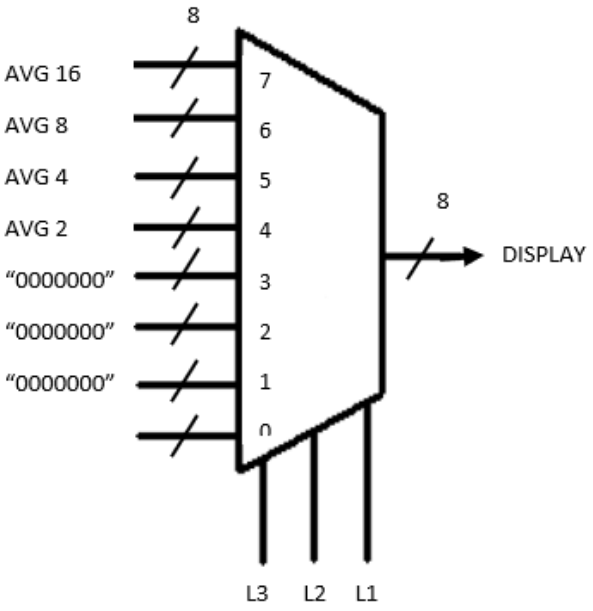
We also need a second multiplexer that has the `Length` input as the 3-bit selection. Depending on the selections we will display either the average of the last 2, 4, 8 or 16 generated numbers or the string “00000000”. The latter will be displayed if the user selects a combination of the Length switches that is not specified in the instructions.

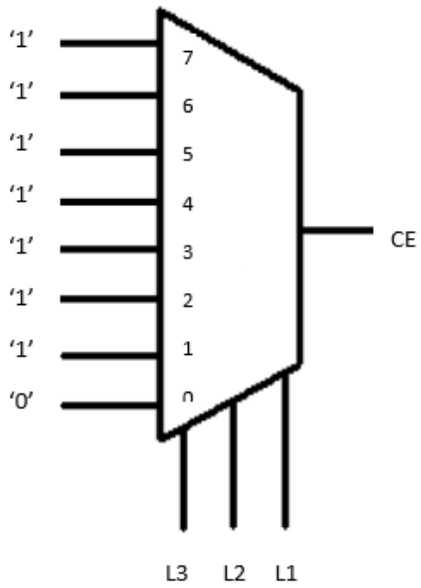
Each memory register also has a `Data_clk` input. This clock has a frequency of 1 Hz, so the 16 number array will be shifted by one position each second. Additionally, the 2, 4, 8, 16 number average will be recomputed each second.

The `Reset` button erases all of the memory registers, as a result the array “00000000” will be loaded into every memory register, thus reseting the system.

Down below is the full and detailed logic scheme of the filter, called *average_computer*.


Data
CE
Data_clk
Reset
NB 1
NB 2
NB 3
NB 4
NB 5
NB 6
NB 7
NB 8
NB 9
NB 10
NB 11
NB 12
NB 13
NB 14
NB 15
NB 16
AVG 2
AVG 4
AVG 8
AVG 16
Lost
Lost
Lost
Lost




When it comes to the VHDL code, the approach taken with this filter module was that of creating a single component described behaviorally. The description of this module required the definition of several new types aside from the ones present in the standard library. The definition of those types can be found in *Annex 1.8*. Each type present there is an array of variable length that contains *std_logic_vectors* of 12 elements. The reason we chose the size of 12 is that, in the case of adding all the 16 numbers of 8 bits, the sum of those numbers has a maximum length of 12 bits. Since VHDL does not allow the operations of addition between elements of different lengths, we considered that the best way to avoid this was to give each possible element the maximum possible length an element in our module will have. During the synthesis of our project, we will not need all of the 12 bits in a lot of cases, so we will fill all of those empty positions with '0'. The ISE will trim those registers that remain constant, thus presenting no problem when it comes to physically implementing this module.

We define our array of numbers as a signal in our architecture. After that, inside our process, we define several variables using the types we defined earlier. These variables will be the ones that we will use to compute the sum of every 2, 4, 8 and 16 numbers. The definition of these variables can be found in the *Annex 1.8*.

As the first thing that we do in our process, we check if the `Reset` button is active. If this is true, we assign "0000000" to every element of our array of numbers, and the average that will be displayed right after resetting will be "00000000" as well. Otherwise, we discard the last element of our array of numbers, and load the newly generated one on the first position.

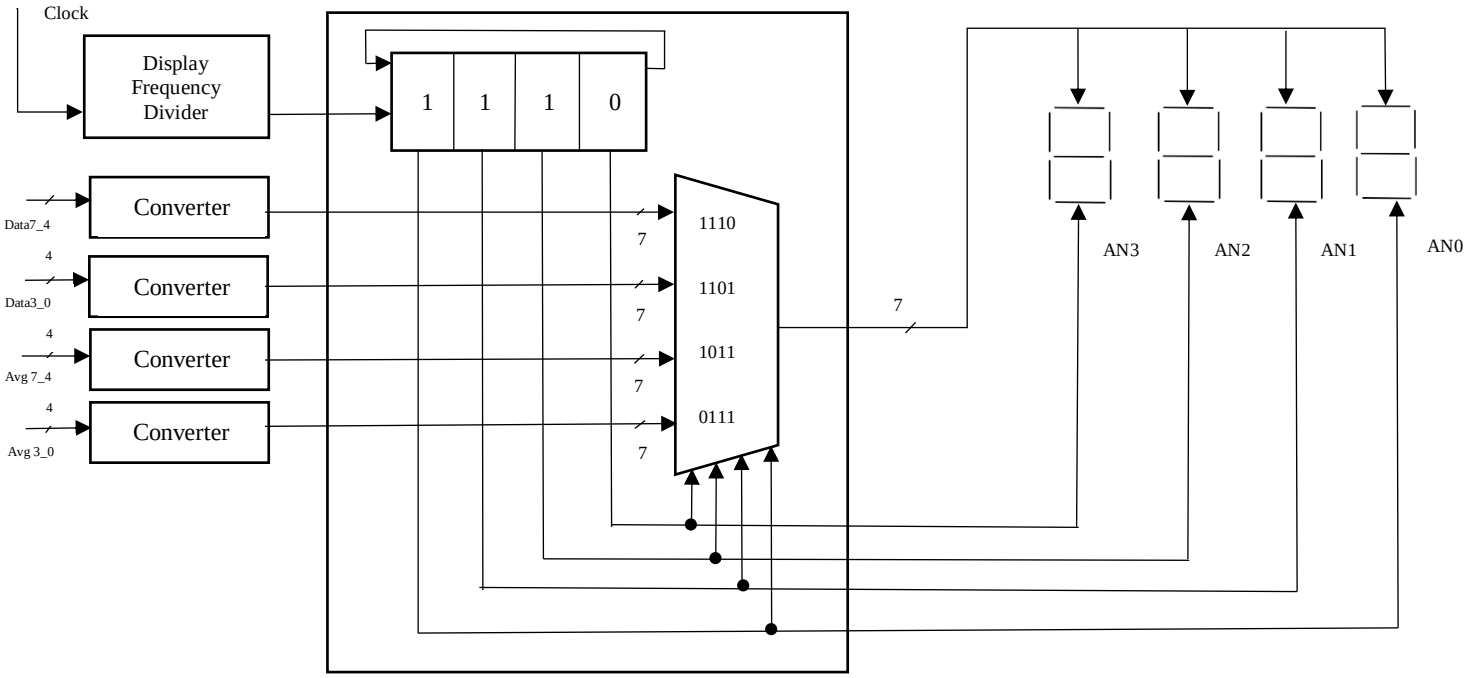
We then move on to compute the sum of every 2 numbers. The sum of the first two numbers will be used to compute the average of the first two numbers, while the rest of the sums will be used to compute the sum of every 4 numbers and so forth. We repeat the process for every 4 numbers as well as for every 8 numbers. After that, we add all of our 16 numbers.

Once the addition of the numbers stored in our array is complete, we move onto the process of division. We take the first element of our array that stores the sum of every two elements. We shift this number one position to perform the division, then we store the result. We do the same for the element that stores the sum of the first 4, 8 and 16 numbers by shifting them to the right by 2, 3 and 4 positions respectively. The code describing this process can be found in *Annex 1.8*. Since the outcome of these divisions will always be on 8 bits, the array that stores the averages will be and array of *std_logic_vectors* of 8 bits each.

We then have a case function that checks the value of the `Length` input. We then move on to display the average as requested by the user. If the `Length` input happens to have an invalid value, we display "00000000" as the average, and if the `Length` input has the value "000", the signal `clock_enable` becomes '0'. Once the `Length` switches move from the "000" position, the `clock_enable` signal becomes '1'.

The sensitivity list of our process consists of the `Reset` and `Data_clk` input, meaning that unless the `Reset` button is pushed, the process will repeat itself every second, as it is required by the project specifications.

## 3. Display




When it comes to the Display module, we have decided to split it into three components: the *Convertor*, the *7 Segment Frequency Divider* and the *Displayer*, which is basically the main component – the display. The *Convertor* and the *Frequency Divider* are a component of the Displayer.

The Displayer, or the display takes the most recently generated number, as well as the most recently generated average as inputs. The outputs of this component are the `ANOD`, which indicates which LED on the Nexys4 board will display each number, and the `CATOD`, which represents the actual number that is to be displayed. For each number that is to be displayed (that is, the generated number and the average) we have assigned two anodes, since the numbers range from 0 to 255 in decimal and require 2 anodes to be displayed in Hexadecimal.

The number and the average that are taken as inputs are on 8 bits. We then split each number into two separate 4-bit numbers that we will run through the *Convertor*. The output of the convertor is a sequence of 7 bits which indicates which LEDs of each Cathode will light up in order to display our desired number. This sequence is stored in an array of 7-bit vectors. This type definition can be found in *Annex 1.9*.

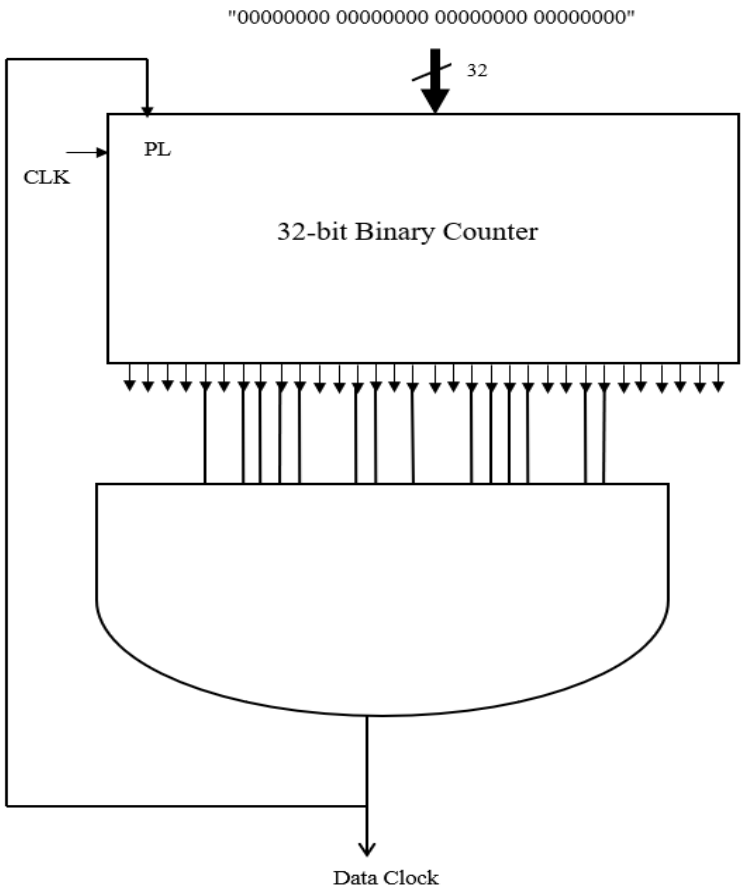
The *Convertor* component is composed of a process which contains a case function. This function tests the 4-bit number that is given and assigns a particular 7 bits sequence to its output, which will lead to the Hexadecimal equivalent of our 4 bit number to be displayed on the Cathode that we chose. This case function can be found in *Annex 1.11*.

The *Frequency Divider* is a very important part for this module. Since the Anodes, with their respective Cathodes, light up sequentially, we need them to light up one after another at such a high frequency that we can't tell that they are not lit up all at the same time. To do this we have a 4-bit register (since we only work with 4 anodes) that consists of three '1's and one '0'. Then we rotate that register at a frequency of 200 Hz, which means that it rotates 200 times a second, making it appear that all of those 4 Anodes are lit all of the time.

The anodes are active low, which means that, for instance, when our register indicates "1011" it means that the second anode is active. We then build a case function that tests the value of the register and displays our number on the respective Anode. The code for the Anode selection can be found at the end of our documentation in *Annex 1.9*.

The 200 Hz Frequency Divider is built using a process which creates a variable that counts up to 500 000. That variable is incremented each time the internal clock of the FPGA board goes through a cycle. When the counting variable has a value ranging from 0 to 250 000, the output of the Divider is '0'. When it ranges from 250 000 to 499 999, the output becomes '1'. When it reaches 500 000, the variable is reset to '0'. Since the Nexys4 board has an internal clock with a frequency of ~ 100 MHz, this counting process will create a clock signal as its output that has a frequency of ~200 Hz.

## 4. 1Hz Frequency Divider




This is also an important component of our project and it was mentioned before in the other but we now look at it more closely.

As the Nexys4 has a crystal oscillator clock of ~100Mhz – referred by us as the system's clock, and as we need our project to run in real time, to synchronize the filter and the data generator to the same clock, we need a new frequency of 1Hz, represented by the `Data Clock`. Our whole system works on this frequency of 1Hz.

For this purpose, we had made a different file, a component, called *frequency_divider*, which has as its inputs a `CLK`, namely the system's one, and as output a `CLK_OUT`, which will be the 1Hz Data Clock.

In its architecture we defined a process called *DIV* which is sensitive to the given clock, and in its declarative zone we define a variable called `counter` of type integer from 0 up to 99 million. The whole process is then defined behaviorally. On the rising edge of the given clock the counter increments by one. If the value of the counter is between 0 and 49,5 million then the new clock will be high, has the value '1'. Otherwise it will hold '0'. When the counter reached the 99 million value, we made it again '0'. And this all represents the new clock, which will be the input of many others important parts

# IV. I/O Notations and Internal Signals

As we have already extensively detailed the behavior and working methods of the components, and their inputs and outputs signal, internal signals between them or other notations, we will do an account of all the notations and signals used so far, by categorizing them in dependence of the modules:

- Main module
  - Inputs:
    `CLK` – Nexys4 100 MHz crystal oscillator clock, referred by us as system's clock
    `CONTROL` – 3-bit mode setting for data generator
    `LENGTH` – 3-bit mode setting for average/filter
    `RST` – reset button
  - Outputs:
    `ANOD` – 8-bit signal showing which led of the display will show information
    `CATOD` – 7-bit signal used for each anode to represent information

- Frequency Divider
  - Inputs: `CLK` – system's clock
  - Outputs: `DataClock` – 1Hz frequency clock

- Data Generator
  - Inputs:
    `CONTROL` – 3-bit mode setting for data generator
    `CLK` – system's clock
    `RST` – reset
  - Output
    `DATA` – 8-bit data stream which will be received by the filter

For all the subcomponents – square wave generator, student one and two memory, LFSR 0-15 and LFSR 0-255, we have as inputs the `DataClock` and as outputs a signal called `OP` on 8-bit representing the specific output for each component.

- Filter
  - Inputs:
    `DataClock` – 1Hz frequency clock
    `LENGTH` – 3-bit mode for average/filter
    `RST` – reset
    `NUMBER` – 8-bit data received from data generator
  - Outputs:
    `AVERAGE` – 8-bit resulted average

- Display
  - Inputs:
    `CLK` – system's clock
    `AVERAGE` – 8-bit resulted average
    `DATA` – 8-bit data stream

- Outputs:
  `ANOD` – 4-bit signal as we use only 4 anodes; this will link to the 8-bit `ANOD` from main module (first 4 anodes being disabled)
  `CATOD` – 7-bit signal used for each anode to represent information

- Frequency Divider for the Display
  - Inputs: `CLK` – system's clock
  - Outputs: `CLK_OUT` – 200Hz clock used for displaying

- Converter
  - Inputs: `Number` – 4-bit number for conversion
  - Outputs: `Converted` – 7-bit number indicating the state on catodes

# V. Justifying the Solution

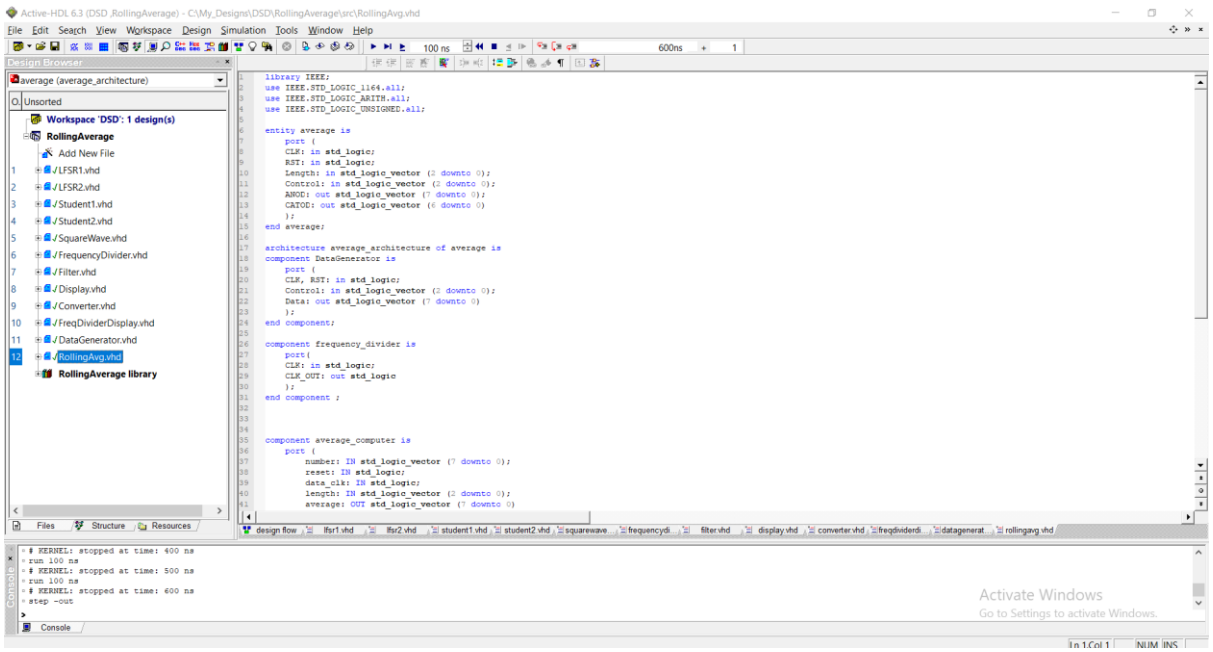
For solving this task, we took the KIS approach (Keep It Simple). We tried understanding the problem from a top-down approach, starting with the whole system's as root in mind, and then dividing it in smaller subproblems which we then tried to solve. After this we started implementing everything in a bottom-up manner, starting from the subproblems and arriving at the big one. Our project has thus in total 4 main components each with additional one, if they were fitting.

Therefore, we worked in more files, everything being well modularized and compacted. The technique we choose is to store and add the latest input data value to a running total, and to subtract the first value at the other end of the buffer. The "running total" is then shifted by a number of bits as a simple method of performing division (shifting by one bit divides by 2, shifting by 2 divides by 4, etc.). The stored values may then move along the buffer, the oldest value being discarded from the front such that the next input value may than be stored at the end. The cycle then repeats for the next input data value.

# VI. Utilization and User's Manual

## 1. Active-HDL

The user will run the Active-HDL program. Then we choose from the *File* menu the option *Open* and from the new window which opens we can browse to find the project from the directory it was saved in. Then the user opens “DSD.aws”.




On the left side of the screen, in *Design Browser* the user can see all the components of the current project. In our case we have: *LFSR1*, *LFSR2*, *Student1*, *Student2*, *SquareWave*, *FrequencyDivider*, *Filter*, *Display*, *Converter*, *FreqDividerDisplay*, *DataGenerator*, *RollingAverag*e, all of them having the extension *.vhd*. By selecting one of these components, the user will be able to see the source code of that component. It is important to mention that when solving the project, we firstly started by writing code in Active-HDL, and then moved to the ISE program.

## 2. Xilinx ISE and FPGA Board

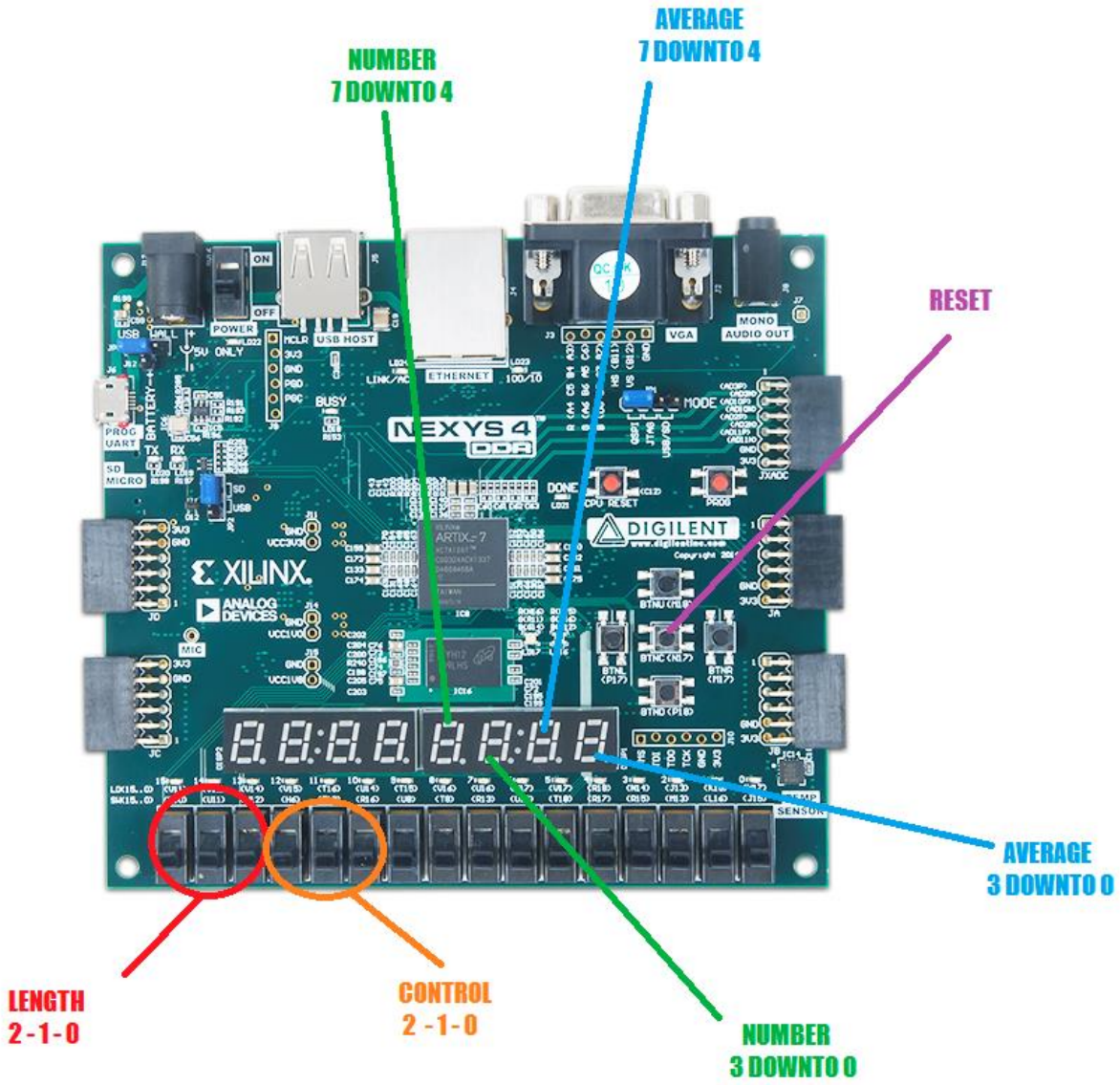
Firstly, the user will have to load the VHDL source codes with the help of the option *Add Source*. All files with the extension *.src* should be loaded at this stage. This enables the source code written in VHDL to be opened with the help of Xilinx.

On the left side of the screen, from the Processes window, we have to choose the option User

```
NET "CLK" LOC = E3 | IOSTANDARD = LVCMOS33;
NET "CATOD(6)" LOC = T10 | IOSTANDARD = LVCMOS33;
NET "CATOD(5)" LOC = R10 | IOSTANDARD = LVCMOS33;
NET "CATOD(4)" LOC = K16 | IOSTANDARD = LVCMOS33;
NET "CATOD(3)" LOC = K13 | IOSTANDARD = LVCMOS33;
NET "CATOD(2)" LOC = P15 | IOSTANDARD = LVCMOS33;
NET "CATOD(1)" LOC = T11 | IOSTANDARD = LVCMOS33;
NET "CATOD(0)" LOC = L18 | IOSTANDARD = LVCMOS33;


NET "ANOD(7)" LOC = U13 | IOSTANDARD = LVCMOS33;
NET "ANOD(6)" LOC = K2 | IOSTANDARD = LVCMOS33;
NET "ANOD(5)" LOC = T14 | IOSTANDARD = LVCMOS33;
NET "ANOD(4)" LOC = P14 | IOSTANDARD = LVCMOS33;
NET "ANOD(3)" LOC = J17 | IOSTANDARD = LVCMOS33;
NET "ANOD(2)" LOC = J18 | IOSTANDARD = LVCMOS33;
NET "ANOD(1)" LOC = T9 | IOSTANDARD = LVCMOS33;
NET "ANOD(0)" LOC = J14 | IOSTANDARD = LVCMOS33;

NET "RST_DG" LOC = N17 |  IOSTANDARD = LVCMOS33;
NET "RST_F" LOC = T16 |  IOSTANDARD = LVCMOS33;

NET "Length(2)" LOC = V10 | IOSTANDARD = LVCMOS33;
NET "Length(1)" LOC = U11 | IOSTANDARD = LVCMOS33;
NET "Length(0)" LOC = U12 | IOSTANDARD = LVCMOS33;

NET "Control(2)" LOC = H6 | IOSTANDARD = LVCMOS33;
NET "Control(1)" LOC = T13 | IOSTANDARD = LVCMOS33;
NET "Control(0)" LOC = R16 | IOSTANDARD = LVCMOS33;
```

Constraints-Assign Package Pins in order to select the input and output pins with respect to the pin's names present on the FPGA board. In our case:

We implemented and tested our project on the Nexys4 DDR FPGA board. Down below is an image of the configuration:




As it can be seen switches V10, U10, U12 represent the 3-bit Length setting, switches H6, T13, R16 the 3-bit Control setting, the button N17 represents the RST, and Anodes J17, J18 - the Average, T9, J14 – the Number. All that remains if for the project to be tested on the board.

# VII. Development Possibilities

A first improvement which could be brought to this system could be a more exact approximation of the average value of the numbers: those with decimal values larger or equal to 50 should be approximated by adding to the next number, and those smaller than 50 should be approximated by lack to the previous number. In this way the accuracy of the computed average will be higher.

Another improvement which could be made in the future is to give the user the opportunity to choose from a larger variety of options. One of these could be the possibility of having numbers larger than 8 bits. In this way we could extend the computing system to numbers larger than 255, increasing its applicability.

# VIII. Bibliography


- Received material in class, namely *LabTaskUTCN.pdf*
- https://reference.digilentinc.com/reference/programmable-logic/nexys-4-ddr/reference-manual - from where additional information and FPGA picture was taken
- https://en.wikipedia.org/wiki/Linear-feedback_shift_register

# IX. Annexes

## 1. VHDL Code

### 1.1 Rolling Average Module – main

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity average is
   port (
   CLK: in std_logic;
   RST_DG, RST_F: in std_logic;
   Length: in std_logic_vector (2 downto 0);
   Control: in std_logic_vector (2 downto 0);
   ANOD: out std_logic_vector (7 downto 0);
   CATOD: out std_logic_vector (6 downto 0)
   );
end average;

architecture average_architecture of average is
component DataGenerator is
   port (
   CLK, RST: in std_logic;
   Control: in std_logic_vector (2 downto 0);
   Data: out std_logic_vector (7 downto 0)
   );
end component;

component frequency_divider is
   port(
   CLK: in std_logic;
   CLK_OUT: out std_logic
   );
end component ;

component average_computer is
   port (
      number: IN std_logic_vector (7 downto 0);
      reset: IN std_logic;
      data_clk: IN std_logic;
      length: IN std_logic_vector (2 downto 0);
      average: OUT std_logic_vector (7 downto 0)
   );
end component ;

component DISPLAY is
   port(CLK: in std_logic;
   number: in std_logic_vector (7 downto 0);
   average: in std_logic_vector (7 downto 0);
   ANOD: out std_logic_vector(3 downto 0);
   CATOD: out std_logic_vector(6 downto 0));
end component;


signal Data_internal: std_logic_vector (7 downto 0);
signal DATA_CLK: std_logic;
signal display_internal: std_logic_vector (7 downto 0);
signal average_internal: std_logic_vector (7 downto 0);
begin
   DG: DataGenerator port map(CLK, RST_DG, Control, Data_internal);
   FDiv: frequency_divider port map(CLK, DATA_CLK);
   AC: average_computer port map(Data_internal, RST_F, DATA_CLK, length, average_internal);
   Displayer: DISPLAY port map(CLK, Data_internal, average_internal, ANOD(3 downto 0), CATOD);
   ANOD(7 downto 4) <= "1111";

end average_architecture;
```

## 1.2 Data Generator

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.STD_LOGIC_UNSIGNED.all;

entity DataGenerator is
   port( CLK: in std_logic;
   RST: in std_logic;
   CONTROL: in std_logic_vector (2 downto 0);
   DATA: out std_logic_vector (7 downto 0));
end DataGenerator;

architecture A_DataGenerator of DataGenerator is

component LFSR1 is
   port( CLK: in std_logic;
   OP: out std_logic_vector (7 downto 0));
end component;

component LFSR2 is
   port(CLK: in std_logic;
   OP: out std_logic_vector (7 downto 0));
end component;

component frequency_divider is
   port(
      CLK: in std_logic;
      CLK_OUT: out std_logic
   );
end component;
component Student1 is
   port( CLK: in std_logic;

   OP: out std_logic_vector (7 downto 0));
end component;
component Student2 is
   port( CLK: in std_logic;

   OP: out std_logic_vector (7 downto 0));
end component;

component SquareWave is
   port( CLK: in std_logic;

   OP: out std_logic_vector (7 downto 0));
end component;

signal data_clk: std_logic;
signal OP_LFSR1: std_logic_vector(7 downto 0);
signal OP_LFSR2: std_logic_vector(7 downto 0);
signal OP_student1: std_logic_vector (7 downto 0);
signal OP_student2: std_logic_vector (7 downto 0);
signal OP_SqWave: std_logic_vector (7 downto 0);


begin
   I_LFSR1: LFSR1 port map (CLK => data_clk, OP => OP_LFSR1);
   I_LFSR2: LFSR2 port map (CLK => data_clk, OP => OP_LFSR2);
   FrDiv: frequency_divider port map (CLK, data_clk);
   ST1: Student1 port map (data_clk, OP_student1);
   ST2: Student2 port map (data_clk, OP_student2);
   SqWave: SquareWave port map (data_clk, OP_SqWave);

   CLK_Process: process (data_clk, RST)

   begin
      if RST = '1' then
            DATA <= (others => '0');

      elsif (data_clk'EVENT and data_clk = '1') then
```

```
            case CONTROL is
               when "000" => DATA <= (others => '0'); -- reset/test mode
               when "001" =>  -- square wave

               DATA <= OP_SqWave;
               when "010" => -- Numbers of stud1

               DATA <= OP_student1;
               when "011" => -- Numbers of stud2

               DATA <= OP_student2;
               when "110" => -- Numbers from LFSR1(0-15)
               DATA <= OP_LFSR1;
               when "111" => -- Numbers from LFSR2(0-255)
               DATA <= OP_LFSR2;
               when others => DATA <= (others => '0'); -- other cases
            end case;

      end if;

      end process CLK_Process;

   end A_DataGenerator;
```

### 1.3 LFSR 0-15

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity LFSR1 is
   port( CLK: in std_logic;

   OP: out std_logic_vector (7 downto 0));
end LFSR1;

architecture A_LFSR1 of LFSR1 is
signal OP_INT: std_logic_vector (3 downto 0):= "1101"
begin
   process(CLK)
   begin
   if (rising_edge(CLK)) then
         OP_INT(3 downto 1) <= OP_INT(2 downto 0);
         OP_INT(0) <= (OP_INT(2) xor OP_INT(3));
      end if;
      OP <= "0000" & OP_INT;
   end process;
end A_LFSR1;
```

### 1.4 LFSR 0-255

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity LFSR2 is
   port( CLK: in std_logic;

   OP: out std_logic_vector (7 downto 0));
end LFSR2;

architecture A_LFSR2 of LFSR2 is
signal OP_INT: std_logic_vector (7 downto 0):= "00000011";
signal FB1, FB2, FB3: std_logic; -- feedback
begin
   FB1 <= OP_INT(0) xor OP_INT(2);
   FB2 <= FB1 xor OP_INT(3);
   FB3 <= FB2 xor OP_INT(4);
   process(CLK)
   begin
      if (rising_edge(CLK)) then
         OP_INT(6 downto 0) <= OP_INT(7 downto 1);
         OP_INT(7) <= FB3;
      end if;
      OP <= OP_INT;
   end process;
end A_LFSR2;
```

### 1.5 Student1

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Student1 is
   port( CLK: in std_logic;

   OP: out std_logic_vector (7 downto 0));
end Student1;

architecture A_Student1 of Student1 is

begin

   process(CLK)
   type StudentMem is array (5 downto 0) of std_logic_vector (7 downto 0);
   variable student1: StudentMem := ("00000111", "00000010", "00000011", "00001000", "00001001", "00000001"); -- numbers: 7 2 3 8 9 1
   variable temp: std_logic_vector (7 downto 0);
   begin


   if (rising_edge(CLK)) then


         temp := student1(5);
         student1(5 downto 1) := student1(4 downto 0);
         student1(0) := temp;

      end if;
      OP <= temp;
   end process;
end A_Student1;
```

### 1.6 Student2

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity Student2 is
   port( CLK: in std_logic;

   OP: out std_logic_vector (7 downto 0));
end Student2;

architecture A_Student2 of Student2 is

begin

   process(CLK)
   type StudentMem is array (5 downto 0) of std_logic_vector (7 downto 0);
   variable student2: StudentMem := ("00000101", "00001000", "00000111", "00000100", "00000101", "00000011"); -- numbers: 5 8 7 4 5 3
   variable temp: std_logic_vector (7 downto 0);
   begin

      if (rising_edge(CLK)) then


         temp := student2(5);
         student2(5 downto 1) := student2(4 downto 0);
         student2(0) := temp;

      end if;
      OP <= temp;
   end process;
end A_Student2;
```

## 1.7 Square Wave Generator

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity SquareWave is
   port( CLK: in std_logic;

   OP: out std_logic_vector (7 downto 0));
end SquareWave;

architecture A_SquareWave of SquareWave is

signal OP_INT: std_logic_vector (7 downto 0):= "00010000"; -- 16 , 32 , 64
begin


   process(CLK)
   variable count: INTEGER range 0 to 3 := 0;
   begin
   if (rising_edge(CLK)) then
            count := count + 1;
            if count = 3 then
               OP_INT(6 downto 0) <= OP_INT(7 downto 1);
               OP_INT(7) <= OP_INT(0);
            end if;
         end if;
   OP <= OP_INT;
   end process;

end A_SquareWave;
```

## 1.8 Filter

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity average_computer is
port (
      number: IN std_logic_vector (7 downto 0);
      reset: IN std_logic;
      data_clk: IN std_logic;
      length: IN std_logic_vector (2 downto 0);
      average: OUT std_logic_vector (7 downto 0)
   );
end average_computer;

architecture average_computer_architecture of average_computer is

   -- numerele vor fi stocate intr-un array de numere care va tine 16 elemente
   type number_array_t is array (15 downto 0) of std_logic_vector (11 downto 0);
   signal number_array: number_array_t := (others=>(others=>'0'));
   signal clk_enable: std_logic := '1';
begin

process_average: process(reset, data_clk)

   -- vom stoca suma a doua cate doua elemente, respectiv a 4 cate 4, 8 cate 8
   --si a tuturor celor 16 numere prezente in sirul nostru de 16 numere stocate

   type average_display_type is array (3 downto 0) of std_logic_vector (7 downto 0);
   type sum2_type is array (7 downto 0) of std_logic_vector (11 downto 0);
   type sum4_type is array (3 downto 0) of std_logic_vector (11 downto 0);
   type sum8_type is array (1 downto 0) of std_logic_vector (11 downto 0);
```

```
   variable average_display: average_display_type;
   variable sum2: sum2_type;
   variable sum4: sum4_type;
   variable sum8: sum8_type;
   variable sum16: std_logic_vector (11 downto 0);

begin


   if length /= "000" then
      clk_enable <= '1';
   end if;


   if reset = '1' then
       average <= (others => '0');
       number_array <= (others =>(others => '0'));
   elsif data_clk'event and data_clk = '1' and clk_enable = '1' then
      number_array (15 downto 1) <= number_array (14 downto 0);
      number_array(0)(7 downto 0) <= number;
   end if;
   |
sum_2:
   for index in 0 to 7 loop
      sum2(index) := number_array(index*2) + number_array(index*2 + 1);
   end loop;

sum_4:
   for index in 0 to 3 loop
      sum4(index) := sum2(index*2) + sum2(index*2+1);
   end loop;

sum_8:
   sum8(0) := sum4(0) + sum4(1);
   sum8(1) := sum4(2) + sum4(3);

sum_16:
   sum16:= sum8(0) + sum8(1);

   -- facem impartirea numerelor prin deplasare la stanga

   sum2(0)(7 downto 0) := sum2(0)(8 downto 1);
   average_display(0) := sum2(0)(7 downto 0); --media a 2 numere

   sum4(0)(7 downto 0) := sum4(0)(9 downto 2);
   average_display(1) := sum4(0)(7 downto 0); --media a 4 numere

   sum8(0)(7 downto 0) := sum8(0)(10 downto 3);
   average_display(2) := sum8(0)(7 downto 0); --media a 8 numere

   sum16(7 downto 0) := sum16(11 downto 4);
   average_display(3) := sum16(7 downto 0); --media a 16 numere

   case length is
      when "100" => average <= average_display(0);
      when "101" => average <= average_display(1);
      when "110" => average <= average_display(2);
      when "111" => average <= average_display(3);
      when "000" => clk_enable <='0';
      when others => average <= "00000000";
   end case;

end process process_average;
end   average_computer_architecture;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity DISPLAY is
   port(CLK: in std_logic;
   number: in std_logic_vector (7 downto 0);
   average: in std_logic_vector (7 downto 0);
   ANOD: out std_logic_vector(3 downto 0);
   CATOD: out std_logic_vector(6 downto 0));
end DISPLAY;

architecture A_DISPLAY of DISPLAY is
signal LED: std_logic_vector(3 downto 0) := "1110";
signal CLK200Hz: std_logic;
type LED_out_type is array (3 downto 0) of std_logic_vector (6 downto 0);
signal LED_out: LED_out_type := (others =>(others=>'0'));

component FreqDivider7Seg is
   port(CLK: in std_logic;
   CLK_OUT_2: out std_logic);
end component;

component Convertor is
port (
number: in std_logic_vector (3 downto 0);
Converted: out std_logic_vector (6 downto 0)
);
end component;

begin

   C0: Convertor port map(number (7 downto 4), LED_out(0))
   C1: Convertor port map(number (3 downto 0), LED_out(1))
   C2: Convertor port map(average (7 downto 4), LED_out(2)
   C3: Convertor port map(average (3 downto 0), LED_out(3)

   CLKDiv: FreqDivider7Seg port map(CLK, CLK200Hz);
   process(CLK200HZ, LED)
   begin
      if (rising_edge(CLK200Hz)) then
         LED(3) <= LED(0);
         LED(0) <= LED(1);
         LED(1) <= LED(2);
         LED(2) <= LED(3);
      end if;

   end process;


   afisare: process (LED)
   begin
       case LED is
         when "1110" => CATOD <= LED_out(0);
         when "1101" => CATOD <= LED_out(1);
         when "1011" => CATOD <= LED_out(2);
         when "0111" => CATOD <= LED_out(3);
         when others => CATOD <= "0000000";
      end case;
   end process afisare;
   ANOD <= LED;
end A_DISPLAY;
```

1.9 Display

1.10 Converter

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity Convertor is
  port (
number: in std_logic_vector (3 downto 0);
Converted: out std_logic_vector (6 downto 0)
);
end Convertor;

architecture Convertor_architecture of Convertor is
begin
process (number)
variable LED_out: std_logic_vector(6 downto 0);
begin
case number is
    when "0000" => LED_out := "0000001"; -- "0"
    when "0001" => LED_out := "1001111"; -- "1"
    when "0010" => LED_out := "0010010"; -- "2"
    when "0011" => LED_out := "0000110"; -- "3"
    when "0100" => LED_out := "1001100"; -- "4"
    when "0101" => LED_out := "0100100"; -- "5"
    when "0110" => LED_out := "0100000"; -- "6"
    when "0111" => LED_out := "0001111"; -- "7"
    when "1000" => LED_out := "0000000"; -- "8"
    when "1001" => LED_out := "0000100"; -- "9"
    when "1010" => LED_out := "0000010"; -- a
    when "1011" => LED_out := "1100000"; -- b
    when "1100" => LED_out := "0110001"; -- C
    when "1101" => LED_out := "1000010"; -- d
    when "1110" => LED_out := "0110000"; -- E
    when "1111" => LED_out := "0111000"; -- F
    when others => LED_out := "0000000";
end case;
Converted <= LED_out;
end process;
end Convertor_architecture;
```

1.11 Frequency Divider 1Hz

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity frequency_divider is
   port(
      CLK: in std_logic;
      CLK_OUT: out std_logic
   );
end frequency_divider;

architecture frequency_divider_architecture of frequency_divider is

begin
   DIV: process (CLK)

   variable counter: INTEGER range 0 to 99_000_000;
   begin
   if CLK'EVENT and CLK = '1' then
      counter := counter + 1;
      if counter >= 49_500_000 then
         CLK_OUT <= '1';
      else
         CLK_OUT <= '0';

      end if;
   end if;

   if counter = 99_000_000 then
      counter := 0;
   end if;

end process DIV;
end frequency_divider_architecture;
```

1.12 Frequency Divider Display

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity FreqDivider7Seg is
   port(
      CLK: in std_logic;
      CLK_OUT_2: out std_logic
   );
end FreqDivider7Seg;

architecture A_FreqDivider7Seg of FreqDivider7Seg

begin
   DIV: process (CLK)

   variable counter: INTEGER range 0 to 500_000;
   begin
   if CLK'EVENT and CLK = '1' then
      counter := counter + 1;
      if counter >= 250_000 then
         CLK_OUT_2 <= '1';
      else
         CLK_OUT_2 <= '0';

      end if;
   end if;

   if counter = 500_000 then
      counter := 0;
   end if;

end process DIV;
end A_FreqDivider7Seg;
```