POLITECNICO DI TORINO

Department of Electronics and Telecommunications

Master's Degree in Electronic Engineering

Master's Degree Thesis

Test and characterization of an integrated circuit implementing a low-energy compressed-sensing based acquisition system

Supervisor

Prof. Gianluca Setti

Co-supervisors

Prof. Fabio Pareschi

Ing. Carmine Paolino

Author

Francesco Trinca

A.Y. 2021/2022

Summary

Latest demand for advanced IC (Integrated Circuit) solutions in industrial, wireless or biomedical field, has led to a growing interest for innovation in data converters. Physical signals, such as audio, radio, or image signals, belonging to the real world, have to be converted into numerical values, with the intention of processing and storing them. ADCs (Analog-to-Digital Converters) represent the interface between the analog domain and the digital one.
In modern applications regarding the biomedical field, energy and area consumption have become critical requirements for an ADC. Converters adopted for acquire bio-signals, such as ECG, must have low power consumption and low area, to allow the highest level of integration, as well as long battery life in case of portable devices.
For this reason, signal processing has gained an increasing interest among the research community. Compressed sensing (CS) is a signal processing technique for efficiently acquire and reconstruct an analog signal from the energy consumption point of view.
The starting point of this thesis is an already developed chip which is essentially a SAR (Successive Approximation Register) ADC with CS capabilities. Every time a new IC is designed, the importance to test and characterize it is crucial. The aim of this work is to test and characterize the chip, and so the ADC, designing an ad-hoc testing platform, with the intention of providing control signals for the chip, regulated supply voltages and obviously, the analog input signals. First, an ADC must be characterized by means of the common figures of merit, through simulations, in order to verify if its performances meet the specifications of the project. Subsequently, having the physical chip, it is significant to test its functionality. To do this, a deep knowledge of both logic and analog part of the chip under test is needed, in order to design a measurement setup. The latter has been designed to host the IC, allowing the device characterization and testing.

In chapter 1, there is a brief introduction of the theory behind the idea of SAR ADC and of the Compressed Sensing. Chapter 2 is concerned with the first characterization of the ADC. Through AMS (Analog/Mixed-signal) simulations, the common figures of merit of the ADC, have been extracted. Then, the whole testing platform design is reported in chapter 3 and 4. Chapter 3 describes the PCB (Printed Circuit Board) design from the preliminary choice of the components to the final layout creation. In chapter 4, instead, there is a detailed discussion of the FPGA (Field Programmable Gate Array), which allows to generate all the control signal needed by the chip and by the board. All the digital elements have been described in VHDL (Very High Speed Integrated Circuit (VHSIC) Hardware Description Language). Finally, chapter 5 regards the final conclusions of this work.

Acknowledgements

Some people have offered both moral and technical contribution to the realization of this work, or they have made this long path pleasant and less hard, simply with their constant presence. I would like to express my gratitude to them in the following few, but significant words.

I thank my mother Rosanna and my father Renato, pillars of my life, who made possible to reach this goal, giving me the opportunity to study and grow professionally. Thank you for always believing in me and in my choices, for encouraging me in moments of difficulty and for teaching me to be a man.

I thank my sister Laura and my brother Stefano, who have been for me a source of inspiration and a model to admire for the determination and tenacity they show every day in what they do. Thank you for being the best family I could wish for.

I thank Prof. Gianluca Setti and Prof. Fabio Pareschi, for fueling my passion for this subject, and Ing. Carmine Paolino, who has always been available for explanations or to provide technical support throughout the thesis work.

I thank my friends and roommates Alfonso and Marco. Thank you for the time spent together, for being my travelling companions on this adventure in Turin, and for making everything more funny.

Finally, I thank my colleague Alessandro, for being the perfect classmate, for sharing the difficulties, concerns and joys we encountered during the university career.

Contents

List of Figures

Chapter 1

SAR-based ADC using Compressed Sensing

Signals coming from the real world are physical continuous quantities represented by analog voltages and currents [1]. These signals could be transmitted or stored, after being processed by an electronic system. These kinds of operations introduce noise which can be partially eliminated if the analog signal is converted into a digital representation. Furthermore, digital signals are far more robust than their analog form. For this reason, every electronic system has an input ADC, which allows to digitally process analog signals. The chip to be tested is essentially a switching capacitor SAR-based ADC with embedded compressed sensing capabilities. In this chapter, a brief discussion of the theory behind the realization of the main elements inside the chip, is reported. Since CS demands several requirements, the successive approximation register architecture has been resulted as the best hardware solution to meet all the constraints.
A particular structure of the capacitive array has been exploited to perform different tasks, such as the multiplication of each acquired sample by the sensing matrix, which is a fundamental element of compressed sensing.
Firstly, the successive approximation algorithm will be explained. Then, different topologies implementing a capacitive array will be presented, followed by a brief description of the compressed sensing concept. Finally, the specific architecture of the converter present in the chip will be reported.

1.1 The successive approximation algorithm

Successive-approximation ADCs are a common approach for implementing an ADC when decent conversion speed and tolerable complexity are specific requirements [2]. A SAR ADC exploits the successive approximation algorithm.

The first element of a SAR ADC is a S&H (Sample and Hold) which samples the input signal. Then, by means of a comparator, the sampled input is compared with the output of a DAC (Digital-toAnalog Converter), and a "binary search" of the value of the n bits begins, starting from the MSB (Most Significant Bit).

This binary search determines the closest digital word to represent an analog input voltage value [3].
In particular, the input signal is sequentially compared with a variable reference. Every comparison coincides with the computation of one bit, starting from the MSB. The binary search determines the state of the n bits, changing the output of the DAC until it reaches a good approximation of the sampled input signal which differs from it within less than a LSB (Least Significant Bit).

A block scheme of this kind of circuit is depicted in figure 1.1, with a diagram which describes the algorithm.

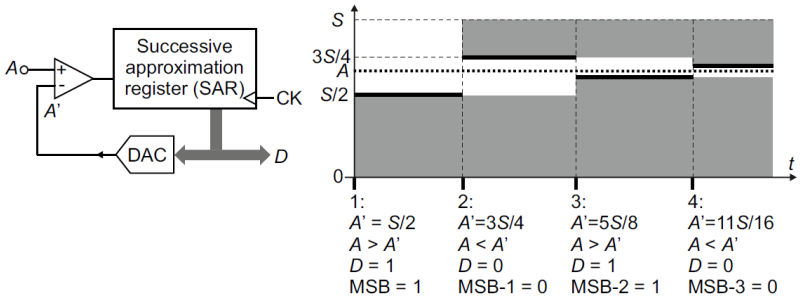


Figure 1.1: Architecture of a 4-bit SAR ADC and sequence of approximation steps [1]

If the input signal A is higher than the reference A', the latter is increased and the MSB=1 (D in the figure), or vice versa. This operation is iterated every cycle, while the approximation of the input signal is getting closer and closer to the real value. Every comparison result is stored in a register until the conversion is completed, that is when the LSB has been computed. It is clear that the number of cycles required for a complete conversion of a sample depends linearly on the number of bits: a N-bit SAR ADC needs N steps to fulfil a conversion.

In modern implementations, capacitive charge redistribution DACs are adopted.

This kind of implementation allows to combine the S&H operation in the DAC array, and it can easily be implemented in a differential configuration.

Generally, the main speed limitation for this kind of ADC comes from the DAC included in the circuit [3]: DAC settling time related to the charge of the capacitors is a critical aspect. The capacitive array can be implemented in different ways, each of which corresponds to a specific architecture.

1.2 Capacitive weighted array

Normally, the element which performs the conversion is a binary weighted array of passive elements, that can be resistors or capacitors. The latter case will be analyzed. A SAR ADC which embeds this kind of array (figure 1.2) performs the successive approximation relying on the charge redistribution. All the capacitors in the array have binary weighted values, i.e. C, 2C, 4C, ... 2^NC.

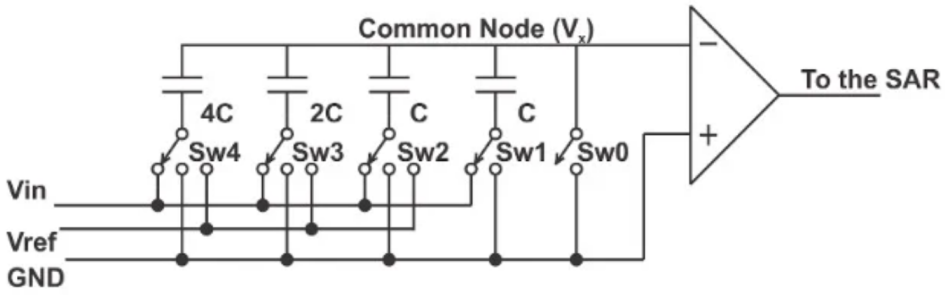


Figure 1.2: 3-bit binary-weighted capacitors array [4]

Firstly, the top plate of all capacitors is connected to ground through SW0, while bottom plate is connected to V_{in}: in this way, the array samples the input signal, storing a precise amount of charge corresponding to a voltage value at the common node V_x equal to $-V_{in}$. Then, SW0 is opened, and the array enters in a holding state. The conversion starts closing SW4 on the V_{ref} node. Since the 4C capacitor forms a 1:1 capacitance divider with the remaining capacitors connected to ground, the voltage V_x becomes $-V_{in} + \frac{V_{ref}}{2}$. Now, if $V_{in} < V_{ref}$, then $V_x > 0$, and the comparator output goes low, assigning to the MSB the logic value '0'. Vice versa, if $V_{in} > V_{ref}$, $V_x < 0$ and MSB=1.
During the second conversion step, the SW3 is closed and 2C capacitor is connected

to V_{ref} node. Depending on the previous computation of the MSB, SW4 will be connected to ground or to V_{ref} node, and V_{in} will be compared with an updated reference, producing the MSB-1. This process continues until all the bits are computed.

1.3 C-2C ladder array

As can be deduced, the ratio of capacitors strongly affects the functioning of the DAC. In particular, the output accuracy depends on the capacitors matching. Since in MOS (Metal Oxide Semiconductor) technology, the capacitance matching depends on area ratios, if one would add a bit in the DAC, a new larger capacitor should be added in the weighted array, and so the matching would become harder and harder as well as this could cause an increasing area consumption. Another solution can be adopted to solve this kind of problem and to increase the accuracy: the C-2C array (figure 1.3). Each C-2C cell acts as a capacitance divider and it can be used in addition of a weighted array, forming a mixed-type array. In this way, the capacitances employed are always the same, thus layout matching will be easier, and good resolution can be achieved.

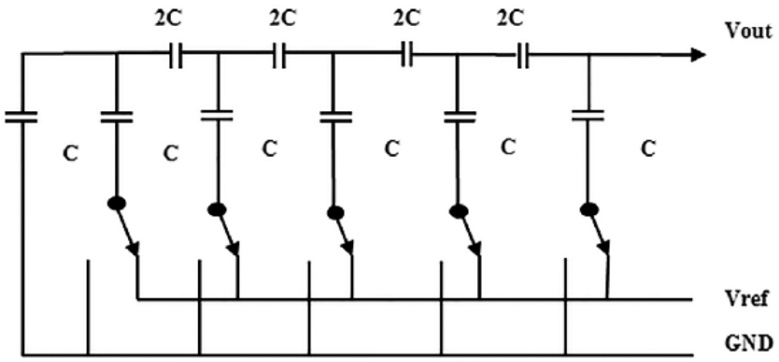


Figure 1.3: C-2C ladder array [5]

1.4 Compressed Sensing

When there is the need to process an analog signal such as voice, with a computer, it has to be converted from the continuous-time domain to a discrete-time domain [6]. This conversion is called sampling, and generally it is a lossy operation. It consists in measuring the analog signal at precise time instants and each measurement represents a sample. The Nyquist-Shannon sampling theorem establishes conditions for which a continuous-time signal of finite bandwidth can be described by its samples and recovered back from its discrete-time form. This condition concerns the sampling frequency. In particular, it must be at least twice the bandwidth of the analog signal. However, it is possible to sample under the Nyquist-Shannon rate. Compressed sensing is a signal processing technique for efficient reconstruction of signals. This is based on the principle that, through optimization, a particular property of a signal family called sparsity, can be exploited to recover them from fewer samples than required by the Nyquist–Shannon sampling theorem [7]. Therefore, CS is possible if there is the prior knowledge that the signals of interest are such that, if expressed in a suitable domain, they exhibit a large number of zero or almost-zero components, i.e., are sparse.
For specific applications, such as in biomedical fields [8], which have heavy constraints in energy and bandwidth, the CS technique can be a solution, thanks to lower acquisition rates. Measuring analog signals in the analog domain, it is possible to acquire them at their true information rate, sparing energy resources. This concept can be applied through CS, which has become an efficient method to substitute Analog-to-Digital Converters with Analog-to-Information Converters [9].

1.5 Architecture for a SAR-based ADC with compressed sensing capabilities

The analysis and the discussion reported in this paragraph refer to the work contained in [10].
As already said, a signal can be processed exploiting the CS technique if it is sparse. Considering a sparse signal in discrete-time domain described by its n consecutive discrete values x, a measurement vector y, which collects all the information contained in a signal window, can be defined as:

$$y_j = \sum_{k=1}^{n} A_{j,k} x_k \tag{1.1}$$

where A is the sensing matrix. It has been observed that, in order to achieve a simple hardware implementation and a good reconstruction quality, the elements

contained in matrix A can be limited to only the values {-1, +1}. In this way, the multiply-and-accumulate operations results to be only signed sums.. In this way, the multiply-and-accumulate operations results to be only signed sums. This is traduced in a preliminary modulation of the input signal by a stream of ±1. Since the architecture (figure 1.4) has a differential implementation, the inverted replica of the input signal is provided by an embedded capability.

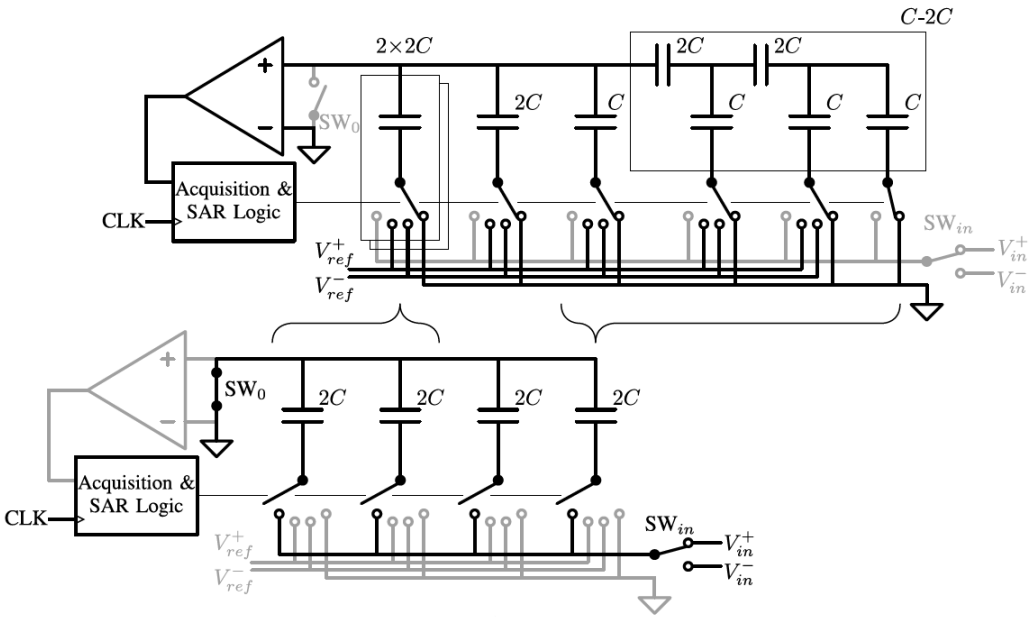


Figure 1.4: Proposed architecture during acquisition (top) and conversion (bottom)[10]

The architecture showed in the top half of figure 1.4, represent a SAR ADC composed of a capacitance array. Its role is to: sample the input signal at precise time instants, hold it during the acquisition, compute a linear combination and perform the conversion into a digital form.
In particular, the capacitive array is composed of a 3-bit weighted array, and a 2-bit C-2C array which implements the LSBs and so allows a better resolution and a low area and energy consumption. This circuit works as a traditional switched-capacitor SAR ADC, as discussed in the previous paragraphs. However, the MSB capacitors have been decomposed into smaller elements (2x2C cell), each driven by its own set of switches. Meanwhile, the smallest element of the weighted array and the C-2C array are driven together, sampling at the same time instant as a larger capacitor.

In this way, all the storing elements have the same value leading to an equal weight on the final result. The configuration during sampling is showed in the bottom half of figure 1.4: all the capacitors act as identical sampling elements, and each of them stores the modulated signal which has been sampled at different time instants.
Then, the SW0 is opened, while the bottom plates are grounded, and the voltage at the input of the comparator results to be the average of the sampled values. During the conversion instead, the sub-elements composing the largest capacitance element, are driven together. Therefore, the conversion is performed as in traditional SAR ADC.
This architecture requires only the addition of a set of switches for the capacitive sub-elements, with respect to the traditional ones.

Chapter 2

ADC characterization

Every new chip, component, or in general a new design, must be tested and characterized to determine if it meets the specifications, and to analyze its behaviour under particular conditions. After having designed a new electronic component, the design flow imposes to test and characterize it by way of simulations. The electronic circuit to be tested is a SAR ADC which contains both analog and digital blocks. For this kind of circuits, analog/mixed signal simulations are run. Providing the circuit of test and control signals, it is possible to derive the most important performance metrics to characterize an ADC. They are the quantization error, offset error, gain error, the INL (integral non-linearity) and the DNL (differential non-linearity).
In this chapter, AMS simulations are briefly introduced, regarding the software used. Then, a theoretical description of the principal performance metrics to characterize an ADC are discussed. Finally, simulation results are reported with an accurate analysis.

2.1 Analog/Mixed signal simulations

Generally, ICs are classified as analog or digital [11]. Circuits as the ADCs, are mixed-signal ICs, which means that they contain both analog and digital circuitry on the same chip. The mixed-signal capability of these ICs allows to obtain low power consumption and increasing reliability. However, the simulation process of these kind of circuits have gained an increasing complexity. In analog circuits simulations, a set of voltages and currents are provided to the circuit with the aim of verifying if it performs required functionalities. Normally, analog circuits simulations are performed at transistor level, unlike what is done for digital circuits. Indeed, digital circuits simulations are performed at gate or behavioural level. Logic and timing verifications are carried out, in order to check if the outputs follow a

specific truth table, for a given set of binary inputs, and to verify propagation delays for critical paths. To test a mixed-signal circuit, the analog part and the digital part are required to be simulated separately, respectively on a SPICE like type of simulator and on a digital logic simulator as VERILOG. Therefore, connection modules, i.e, interface elements used to connect digital and analog domains, are needed.

Modern EDA tools for electronic design, such as Cadence Virtuoso, provide simulation environment dedicated to the simulation of analog/mixed-signal circuits. In particular, the AMS environment by Cadence Virtuoso has been used. It allows to netlist, compile, elaborate, and simulate a circuit that contains analog, digital, and mixed-signal components [12].
Analog and digital blocks are distinguished through their "view". Indeed, each cell has a corresponding view. For AMS simulation, the config view is required [13]. If the design to simulate has not a config view, it will not be possible to set the simulator as AMS.
When a schematic is opened with a config view, it is called a "configured schematic". This kind of view is used by the netlister to control how the design is represented in the simulation netlist. For example, in the config view shown in figure 2.1, the “top” cell is represented by a schematic. It can also be seen that the I0, I1, and I2 inverters are represented as “verilog” and the I3 inverter is represented by a schematic. In other words, I0, I1, and I2 are digital, whereas I3 is analog.

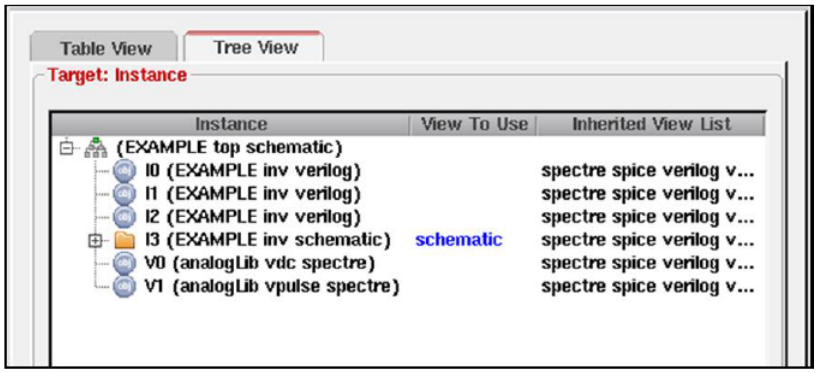


Figure 2.1: Hierarchy view example [13].

Furthermore, it is possible to change the config view of an instance, setting the instance view in the Hierarchy Editor. It will be read by the netlister, which will determine what view to use for each block when creating the netlist. Modifying the config view enables the swapping of different representations (verilog, schematic, veriloga, parasitic extracted, and so on) for the design blocks, without changing the schematic. The netlister starts at the top of the tree in the config view and

uses the views specified in the config (figure 2.2).

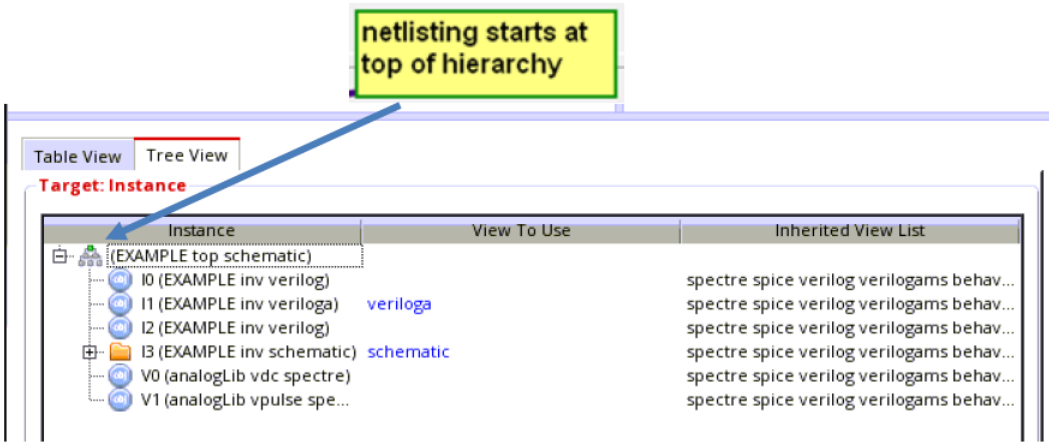


Figure 2.2: Hierarchy view example [13].

The inherited view is default set by the view list (verilog). In this case, I0 gets the first view in the list, which exists in the EXAMPLE library. For the I1 instance, the View To Use column overrides the verilog view and tells the netlister to use the veriloga representation instead. Similarly, the I3 instance has an override set to use the schematic view.

It is possible to annotate the partitioning information on the schematic through the AMS Partition Display. In addition to analog and digital, the AMS partitioning display also shows "analog/mixed", "digital/mixed", and “real/mixed” variations. Digital/Mixed means digital at the current level of the hierarchy (from where the AMS Partition Display form is opened) and mixed somewhere else in the hierarchy. Analog/Mixed means analog at the current level and mixed somewhere else in the hierarchy.
In the schematic of figure 2.3, nets A, B, and C are analog/mixed (light purple), which means they are analog on the current schematic and are mixed (analog and digital) down inside the schematic hierarchy. For an example, take net B. Instance I1 drives net B, which is an analog inverter; so, the signal is analog on the top-level schematic. However, descending into instance I2 (verilog), net B is digital, and therefore, is represented as analog/mixed in the AMS partitioning display. Analog/mixed nets appear as analog in the waveform window since the signal is analog where the probe is placed.

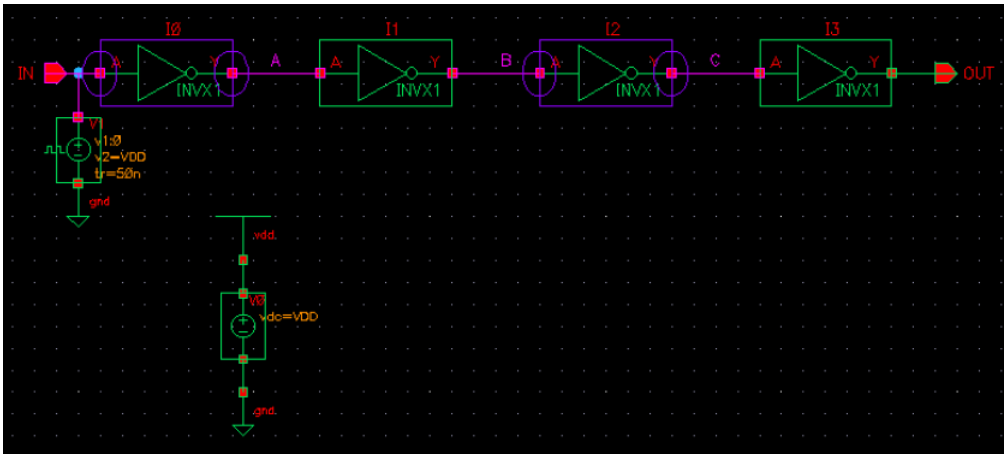


Figure 2.3: Schematic example [13].

It is also possible to select a Verilog view as top level, after having written an equivalent Verilog file which describes the circuit.

With a Verilog top, the stimulus will change, for example, from the typical analogLib vsource to a “clock” module, which is verilogams. Therefore, there is not any vsource setting the supply voltage "vdd". One need to bring the vsource by using a text file (.scs file in figure 2.4).

```
//
simulator lang=spectre
global 0  vdd!

V1 vdd! 0 vsource dc=1.8
```

Figure 2.4: .scs file example [13].

2.2 ADC performance metrics

The transfer characteristic of an ADC associates an input analog continuous quantity (V_{in}) to an unique discrete output binary number (D_{out}). As can be seen in figure 2.5, it is a staircase function, in the case of ideal ADC. Each step width of V_{in} values corresponds to a quantization interval $\Delta = \frac{V_{FSR}}{2^N}$, where V_{FSR} is the full scale voltage, while N is the number of bits. All the analog values falling in this interval are converted into a unique digital word. Step on the y-axis instead, correspond to 1 LSB. An ADC is defined linear if the quantization interval Δ have constant width. These steps lead to quantization errors because a single digital output D_{out} represents a continuous range of values of the analog input V_{in} [3].

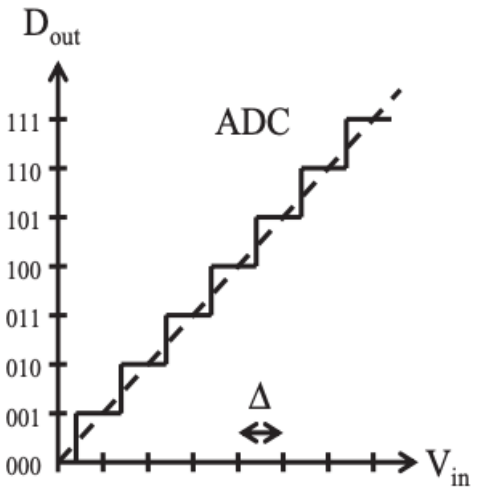


Figure 2.5: ADC transfer characteristic [3].

Depending on its resolution, an ADC can resolve 2^N distinct analog levels. Resolution is defined as the smallest step of voltage that can be recognized by the converter, causing a variation of the digital output [14].
ADCs performance can be measured by means of some metrics. They are classified in static and dynamic metrics. Static metrics refer to the errors for which the transfer characteristic of an ADC deviates from the ideal staircase.

In this paragraph, a brief summary of the static errors, which are the ones measured on the ADC under test, is reported. They are:

- Quantization error.
- Offset error.
- Gain error.
- Differential non-linearity error (DNL).
- Integral non-linearity error (INL).

2.2.1 Quantization error

An ADC translates an analog continuous signal into a digital representation over N bits. As can be seen in the transfer characteristic of figure 2.5, a single output word is associated to a defined range Δ of analog input signal, which contains infinite values. These values will be assigned to the same output word, and this introduces a quantization error, since there are only 2^N analog values which can be represented into a digital form. In other words, a digital number identifies a range, not an exact value. Even ideal ADCs are affected by quantization error. Note that only the midpoints of each interval Δ fall on the ideal transfer function (dotted trace of figure 2.5), and so they are associated to a digital word which quantifies exactly their value. The other points in the interval are associated to the same digital word, so a quantization error is committed, and it can be at most equal to $\frac{\Delta}{2}$. The characteristic for an ideal ADC with uniform distribution of the quantization error is depicted in figure 2.6.

As can be noticed, quantization error depends on the number of bits N. Indeed, an increasing number of bits would decrease Δ and so the maximum quantization error.
Quantization is an operation that introduces noise and cannot be recovered. A noise quantity V_q can be defined as:

$$V_q = V_{in} - V_D \tag{2.1}$$

where V_{in} is the analog input, and V_D is its relative digital representation re-converted into a voltage value. Furthermore, observing figure 2.6 where a voltage ramp is considered as input, it can be said that the signal V_q has an average equal to zero, but its RMS (Root Mean Square) value is equal to:

$$V_{q(RMS)} = \frac{\Delta}{\sqrt{12}} \tag{2.2}$$

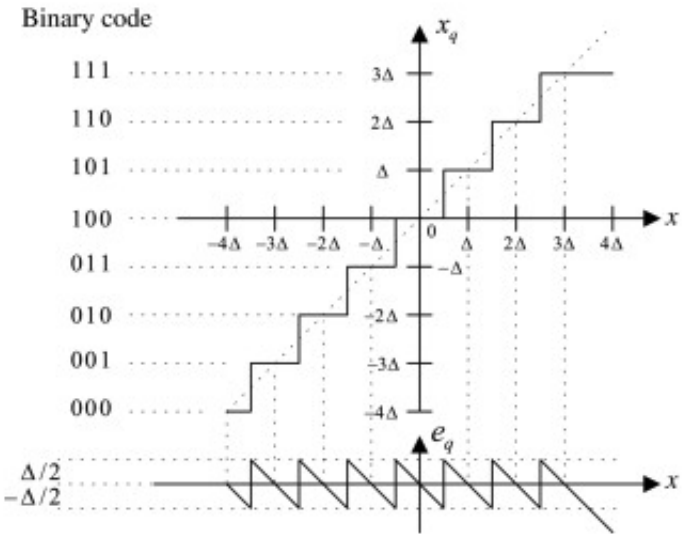


Figure 2.6: Quantization error of an ideal ADC. [15].

This is another proof that the power of the quantization noise is proportional to the step size Δ an so to the number of bits N of the converter. Clearly, a SNR (Signal-to-Noise Ratio) metric can be attributed to the quantization noise. It is defined as:

$$SNR_q = 20log(\frac{V_{in(RMS)}}{V_{q(RMS)}}) \tag{2.3}$$

It is clear that the SNR_q increases with the input signal level, or with an increasing number of bits N, as already said, as showed in figure 2.7.

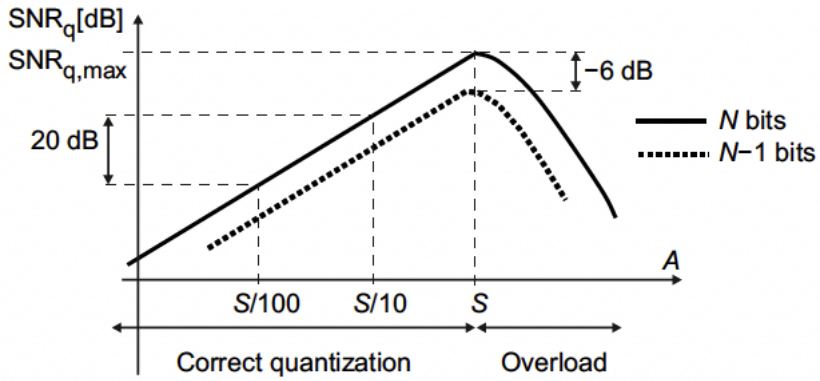


Figure 2.7: SNR_q vs signal amplitude A [1].

2.2.2 Offset and Gain errors

In an ADC, the offset error is the difference between the transition voltage associated to the LSB and the transition voltage of the same code on an ideal ADC with the same number of bits [16].
As can be seen in the figure 2.8, the offset error causes the shift of the transfer characteristic from the ideal position. However, offset error does not introduce non-linearity and it can be eliminated by adding a constant to the output.

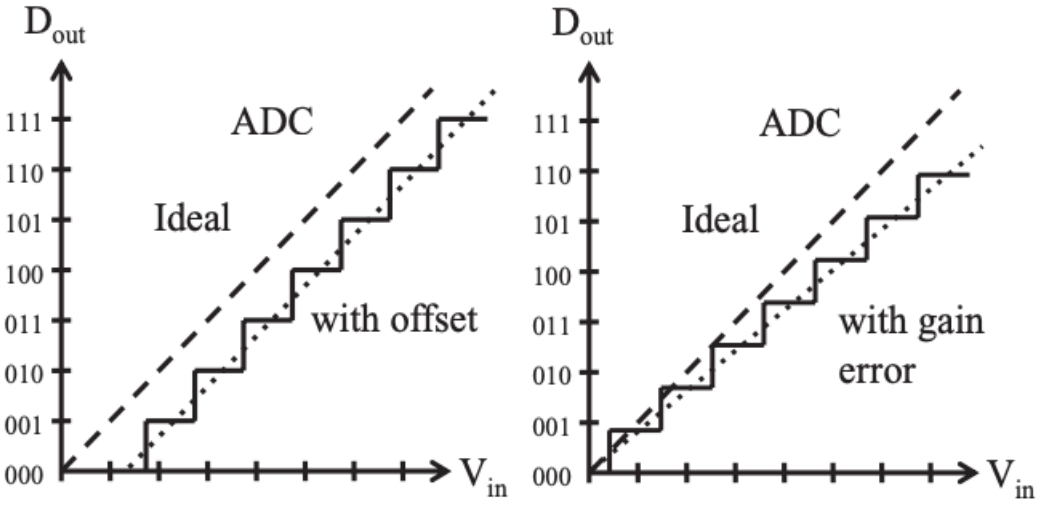


Figure 2.8: Offset and gain error in ADC [3].

The gain error instead, is the difference between the transition voltage associated to the MSB and the transition voltage of the same code on an ideal ADC with the same number of bits, after the offset error correction. Observing the transfer characteristic in figure 2.8, it can be said that the offset error causes the deviation of the slope of the actual staircase from the ideal transfer function slope.

Offset and gain error are usually reported in units of the LSB of the converter.

2.2.3 Differential non-linearity error

In an ideal ADC, the quantization steps Δ (referring to figure 2.5) are all equal to 1 LSB. However, this quantization steps could be larger or smaller than the ideal size of 1 LSB. The differential non-linearity error is defined as the difference between each actual quantization step and the ideal value. The DNL identifies localized errors in the transfer characteristic among adjacent quantization steps. Indeed, smaller, or larger analog steps increase or reduce the adjacent ones. The DNL error can be positive or negative: for example, considering the step width

which corresponds to D_{out}=001 in figure 2.9, it can be noticed that the actual step (solid line) is smaller than the ideal step (1 LSB). This deviation causes a negative DNL error. Conversely, considering the step width which corresponds to D_{out}=011, it can be observed that this time, the actual step is wider than 1 LSB. This deviation causes a positive DNL error. DNL errors are generally measured in fraction of LSB, and it can be referred to each digital word or to the maximum magnitude of the DNL values [2].

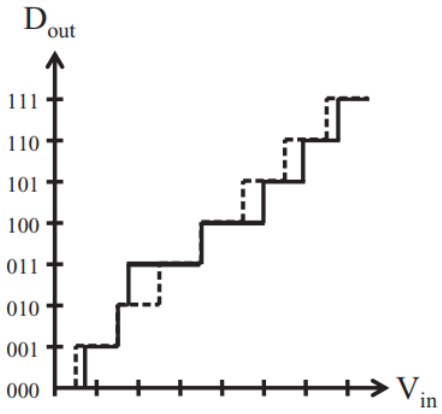


Figure 2.9: Differential non-linearity error in ADC [3].

A typical problem related to the DNL is the missing code error. This error occurs when the DNL error is greater than 1 LSB or smaller than -1 LSB. In this case, a quantization step completely suppresses the adjacent one and the corresponding code in never generated [1]. For example, in the transfer characteristic of figure 2.10, the digital word output 010 is never generated.

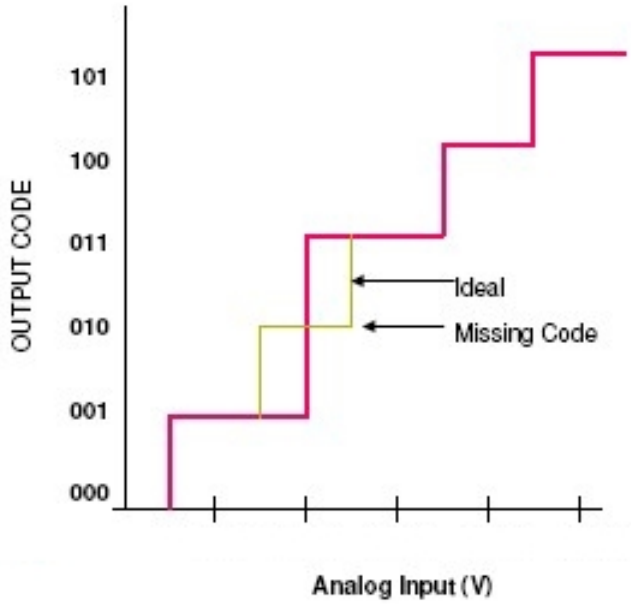


Figure 2.10: Missing code error in ADC [17].

The DNL of a digital code D is normally defined after having removed gain and offset error as:

$$DNL_D = \frac{V_D - V_{D-1} - \Delta_{ideal}}{\Delta_{ideal}} \tag{2.4}$$

where V_D is the analog voltage value corresponding to the output digital word D, and Δ_{ideal} is the ideal analog quantization step corresponding to 1 LSB, defined as:

$$\Delta_{ideal} = \frac{V_{FSR}}{2^N} \tag{2.5}$$

where V_{FSR} is the full-scale range voltage of the ADC, while N is the number of bits of the ADC.

2.2.4 Integral non-linearity error

Non-linearity error is code-dependent and is related to the step size. These kinds of errors lead to a deviation of the transfer characteristic from the ideal one. Considering an ideal straight line which connects the actual first transition (LSB) to the last transition (MSB), the actual transfer characteristic of an ADC moves away from this line. The deviation is quantified by the so called integral non-linearity error (figure 2.11). Unlike the DNL, the INL considers the combined effect

of consecutive localized step-size errors, which are propagated form one step to another, gradually moving the transfer characteristic away and back from the ideal straight line. Indeed, the INL error at a certain point, is the sum of the DNL errors from 0 to the point itself. If there is a sequence of DNL errors having the same sign, the resulting INL error is high, while sequences of DNL errors with opposite sign less contribute to the overall INL error [1].

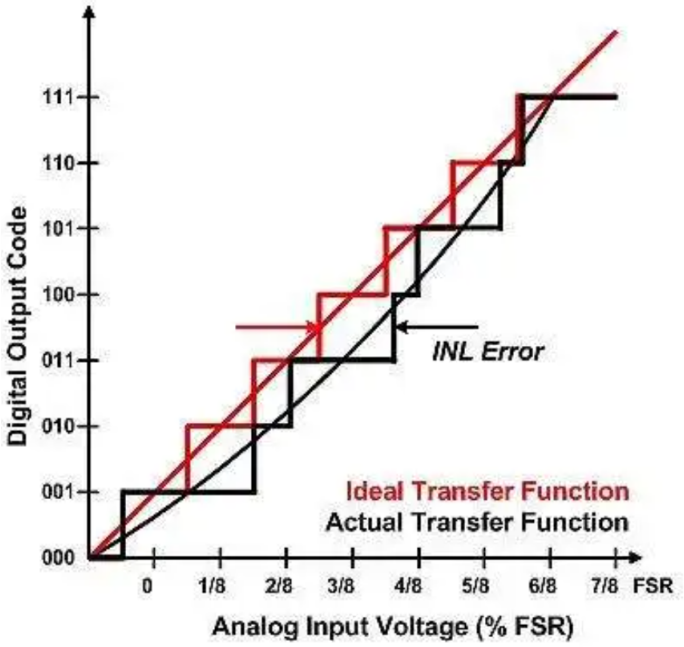


Figure 2.11: Integral non-linearity error in ADC [18].

The INL of a digital code D is normally defined after having removed gain and offset error as:

$$INL_D = \frac{V_D - V_{D_ideal}}{\Delta_{ideal}} \tag{2.6}$$

Physical limitation such as finite gain of amplifiers or mismatch in both active and passive component, strongly affect the linearity of the ADC, originating non-linearity errors. However, it is possible to minimize them by design, using proper architectural choices [19] combined with layout techniques [20].

2.3 Simulations

As already said, the ADC circuit has been simulated by means of the EDA tool Cadence Virtuoso. In particular, the environment used to program and simulate the circuit is ADE Explorer, choosing "ams" as simulator, while the waveform viewer exploited is ViVA (Virtuoso Visualization and Analysis Tool).

2.3.1 ADC internal structure

In figure 2.12, there is the symbol containing the top-level schematic which has been simulated.

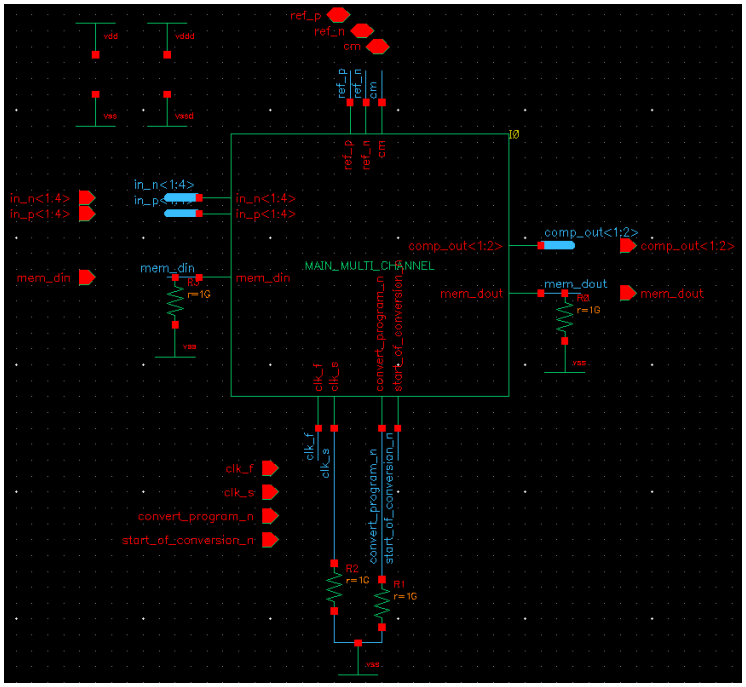


Figure 2.12: Symbol of the schematic

Below is a description of all the pins of the symbol. In order to synchronize and perform all the operations, the ADC needs to be controlled through different control signals which must be sent by the user.

The input signals that must be programmed are:

- conv_prog_n. This signal controls the internal memory programming and the beginning of the conversion. In addition, it is used also to reset the memory address counter.

- SOC. This signal (Start of Conversion) must be '0' only during the conversion, when the ADC is sending in output the 12-bit word that represents a single sample.
- clk_s. This is a clock signal used to shift data in the memory and to control the acquisition cycles.
- clk_f. This is another clock signal used to control the conversion. Every single output bit is synchronized on the rising edge of this signal.
- mem_din. This is the memory input data pin.

The other signals of the symbol are:

- in_n/in_p. These are the pins where the differential analog input signal must be provided.
- ref_p/ref_n/cm. These are the pins for the reference voltages used by the ADC.
- comp_out. Output pin of the ADC.
- mem_dout. Output pin used to check the data shifting in the memory.

The chip under test is essentially a SAR ADC. Inside the chip there is a memory which must be programmed by the user only once, before starting any operation. This memory will store information about how to acquire the input signal, controlling the number of capacitors used during every acquisition cycle. Furthermore, the memory will store information about the preliminary modulation of the input signal. The ADC has two different channels which works independently. Before the conversion, a sample can be multiplied by 1 or -1. This operation is performed by the first block of each channel, which is a modulator for the input signal. Then, there is a capacitive array, which contains 16 capacitors. The array is responsible to perform the acquisition and the conversion of the input signal. Both these blocks are controlled by the memory. Finally, there is a comparator which computes the output digital value. In figure 2.13 there is a scheme representing the layout of the memory, from cell point of view.

Each circle represents a single memory cell, which can store one bit. The first four bits, called "analog_config" bits, are don't care. Then, the instructions for the acquisition cycle 0 are stored in 20 cells. The green cells, called "input_channel_state", contains information about the modulation. The purple cells instead, called "sampling_cell_state", contain information about the acquisition. In particular, if all of these 16 bits are set to '1', the acquisition of the sample will last only one

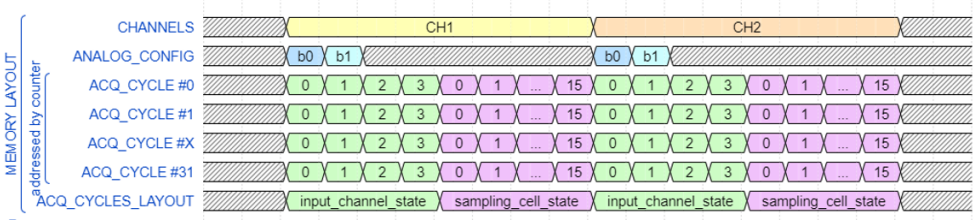


Figure 2.13: Functional layout of the memory.

clock cycle, and it will be performed using all the 16 capacitors. Otherwise, for example, if bits from 0 to 7 are set to '0' and the last 8 bits are set to '1', during the acquisition cycle 0, 8 capacitors will be used. Then, for acquisition cycle 1, the remaining cell can be used to perform another acquisition. The resulting output will be the analog mean of the two samples. In this case the acquisition process will last 2 clock cycles. This is possible programming the memory properly, and obviously, all the combinations are possible.

In figure 2.14 there is a timing diagram which describes the required timing of the input signal, and the relative operations. So, the analog-to-digital conversion is performed in the chip, following different steps. They are in order:

- MEMORY PROGRAMMING. A serial bitstream must be provided by the user at the input pin "mem_din", to program the memory. This operation must be fulfilled only once, and before the beginning of the conversion. Data are shifted-in on the rising edge of the signal "clk_s". During this operation, the signal "conv_prog_n" must be set to '0'.
- ACQUISITION. During this step, the capacitive array samples the input signal, according to the memory content. The input signal is sampled on the falling edge of the signal "clk_s" and an acquisition cycle is performed.
- CONVERSION. When the signal "SOC" goes high, the conversion starts, and the result is put in output on every rising edge of the signal "clk_s". Finally, a pulse of the signal "conv_prog_n" resets the memory address counter, and a new cycle can start.

2.3.2 Testbench and simulations

The testbench has been implemented directly assigning the stimuli for the circuit, through the "Stimuli Assignment" tool. In order to characterize the ADC, and to measure the performance metrics discussed in the previous paragraph, a voltage ramp has been selected as input signal. For this kind of characterization, all the

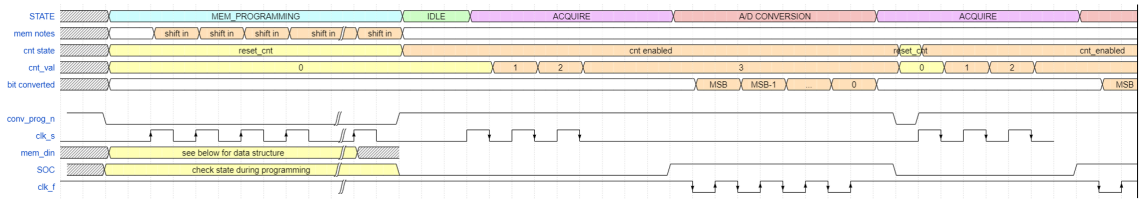


Figure 2.14: Timing diagram of chip operation/control.

input pins receive the same input signal, while its differential version has been applied to the negative pins. Then, with the aim of stimulating the entire dynamic of the ADC, the input voltage ramp has been made to vary between 0V to 1.8V. The type of generator used is the "vpwl", which is a piecewise linear function generator, selecting as "Time2" the end time of the simulation "Tsim", to stimulate and see all the transitions of the digital output levels.
Moreover, for this test, the wanted kind of conversion by the ADC is the more general. In other words, the preliminary modulation of the input signal has been disabled, having a simple multiplication by a constant equal to 1. Then, the acquisition of the input signal occurs exploiting all the 16 capacitors of the array at the same clocking edge. For this reason, the acquisition cycle will last only 1 clock cycle. Finally, it has been used only one of the 2 channel present in the ADC. It is possible to run this kind of conversion, sending to the memory the following bit sequence:

mem_din=0000000000011111111

This kind of signal has been generated using the "vbit" source: it is enough to provide the bit string, the logic high and low voltage values, the clock period, and an eventual initial delay. This voltage source has been used also to generate the input signals "clk_s" and "clk_f".
For what concerns the input signals "v_conv" and "SOC", the vpulse source has been used: for this kind of generator, it is sufficient to specify the logic high and low voltage values, the Tpulse and the Tperiod, and the eventual initial delay. In figure 2.15 there is a preview of the input signal waveforms, visualized on ViVA.

The output signal is probed on the "cmp_out1" pin, which is the output of the channel 1. As can be seen in figure 2.16, each bit of the 12-bit word is produced on the rising edge of clk_f. Obviously, bits are represented by a pulse train, where 0V is the low logic voltage, and 1.8V is the logic high voltage.

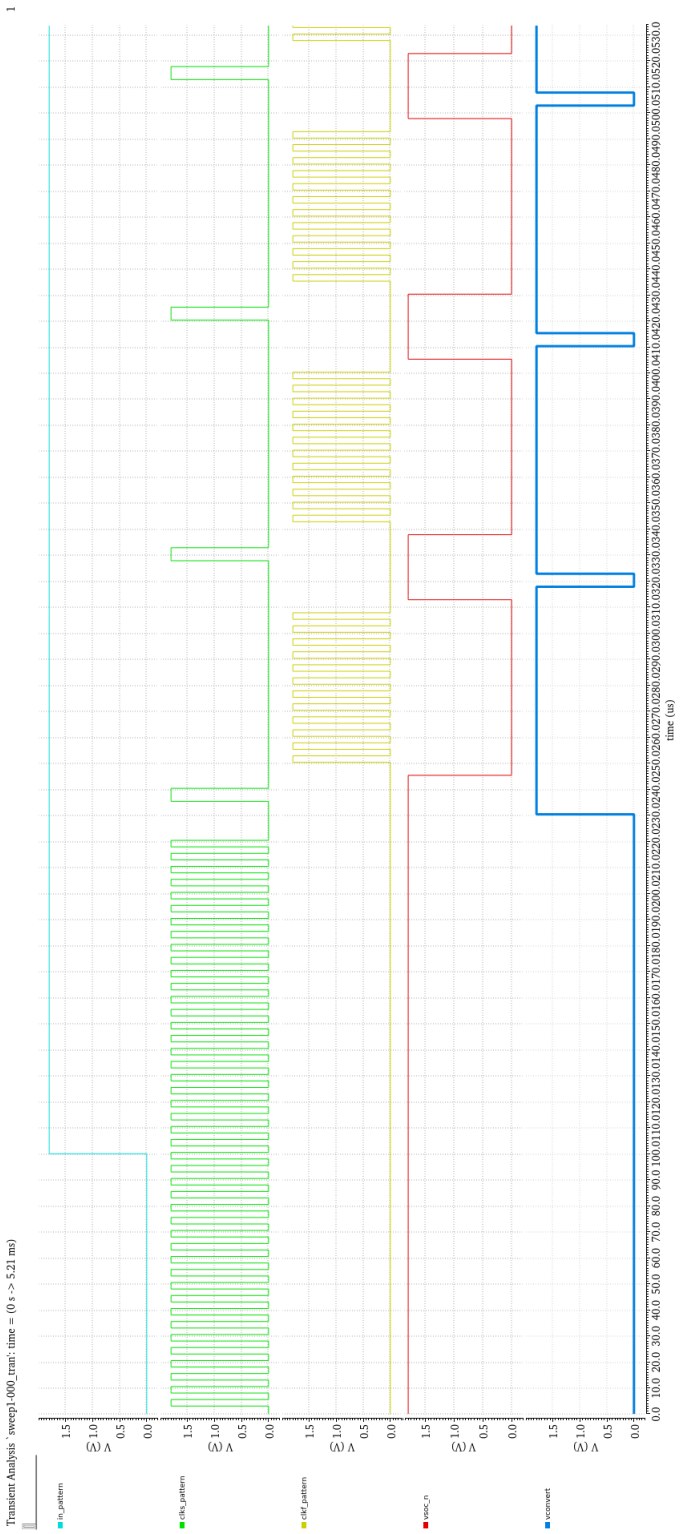


Figure 2.15: Testbench input signal patterns

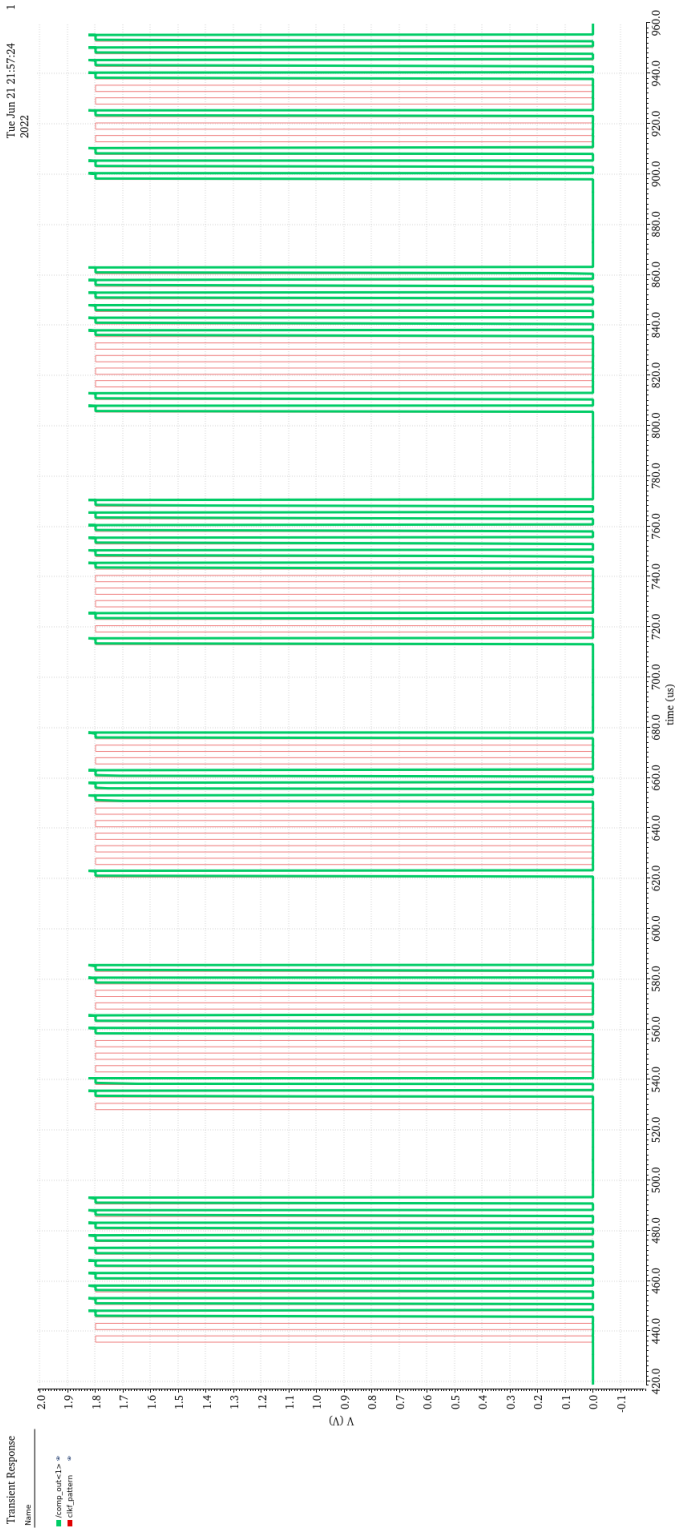


Figure 2.16: Output bitstream (in green) over clk_f (in red)

2.3.3 Numerical elaboration and results

In order to compute performance metrics and to extract a transfer characteristic, the output bitstream has to be processed and elaborated. In particular, the waveform has been exported to a .matlab file, which is a Matlab readable file, with the aim of converting the information contained in the bitstream, into numerical values. This has been possible with the use of the Matlab script reported in Appendix B.

Transfer Characteristic Since the whole input dynamic of the ADC [0; 1.8]V has been stimulated, the resulting sweep of the output goes from the output word corresponding to '0', to the output word corresponding to the full scale range voltage '4095', which is 2^N, with N=12. In order to have an estimation of the computational burden of this kind of simulations, to obtain 10108 output words, it took a 2 days lasting simulation. In figure 2.17 there is the obtained transfer characteristic (dotted line), overlapped with the ideal one.

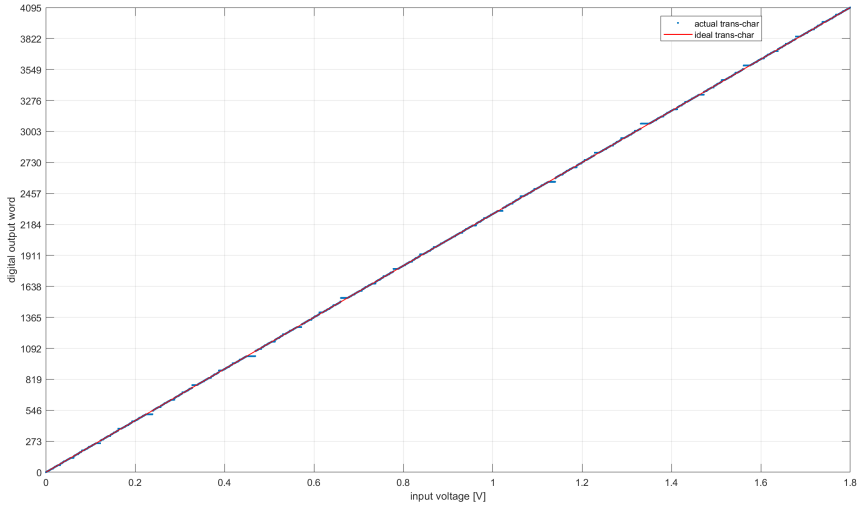


Figure 2.17: Transfer characteristic of the ADC

For a better visualization of the errors of conversion, figure 2.18 shows the conversion of the first 200 words.

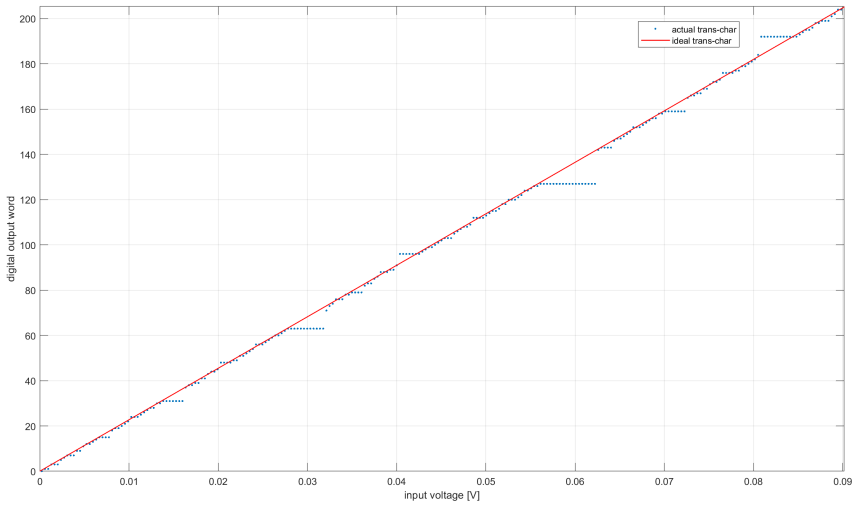


Figure 2.18: Transfer characteristic of the first 200 words

Gain and offset error Gain and offset error have been computed exploiting the Matlab app "Curve Fitting Tool". Indeed, a method to obtain these errors is to extrapolate the best fitting curve of the actual trans-characteristic; then, the resulting slope and Y-intercept of this curve correspond respectively to the gain and offset error.
In figure 2.19, the best fitting curve is depicted, while in figure 2.20, numerical result of the fitting are shown.

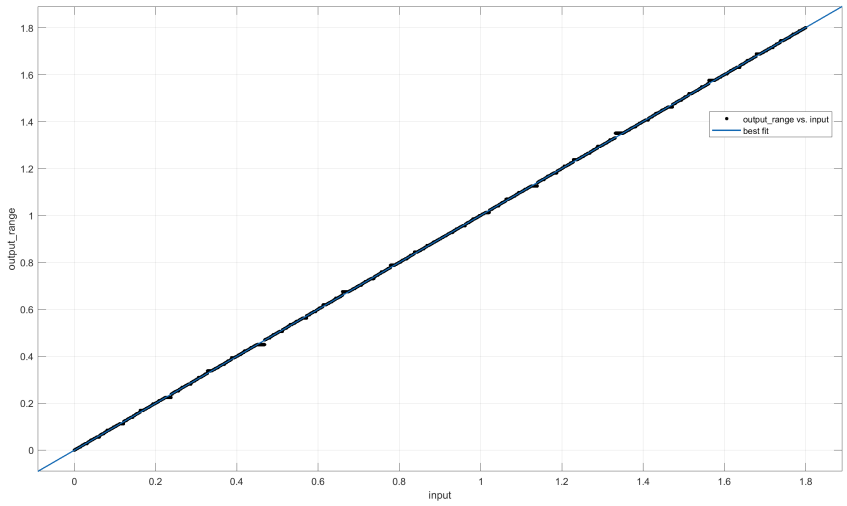


Figure 2.19: Best fitting curve of the trans-characteristic

Linear model Poly1:
f(x) = p1*x + p2
Coefficients (with 95% confidence bounds):
p1 = 1.001 (1.001, 1.001)
p2 = -0.0007789 (-0.0008896, -0.0006682)

Figure 2.20: Results of the curve fitting

As can be observed, p1 is the slope (gain), while p2 is the Y-intercept (offset). The gain can be considered to be about zero, while the offset error V_{off} is equal to -0.0007789 V. This value can be expressed as a fraction of LSB or as percentage of the FSR. Knowing that,

$$V_{LSB} = \frac{FSR}{2^N} = 440\mu V \tag{2.7}$$

offset error can be expressed as

$$V_{off[LSB]} = \frac{V_{off}}{V_{LSB}} = -1.7724\,LSB \tag{2.8}$$

or

$$V_{off\%} = 100\frac{V_{off}}{FSR} = -0.0433\% \tag{2.9}$$

Quantization error Quantization error has been computed using 2.1. Figure 2.21 shows the obtained quantization error, normalized on V_{LSB}. Note that the two red dashed horizontal lines indicate the $\frac{1}{2}LSB$ and $-\frac{1}{2}LSB$ levels. Figure 2.22 is a zoomed version on the central values of the quantization error. This waveform highlights that most of the errors are focused within the range $\frac{1}{2}LSB$ and $-\frac{1}{2}LSB$. The histogram of figure 2.23 describes the errors distribution.

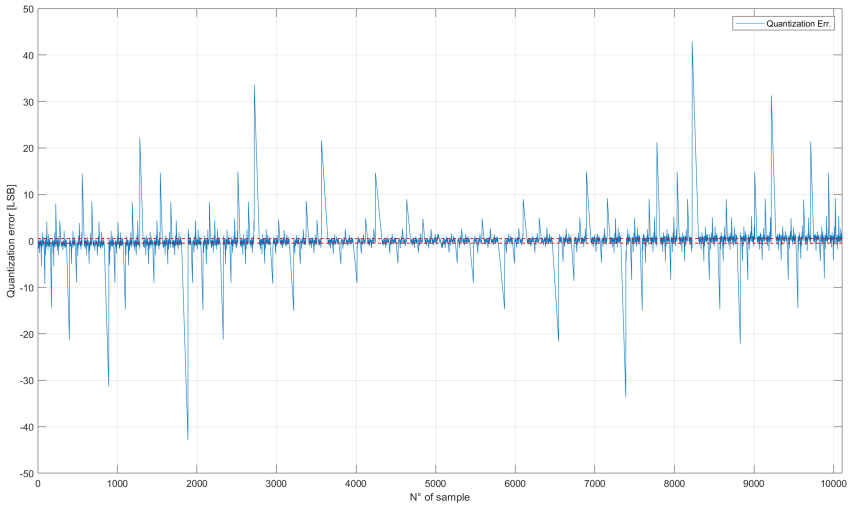


Figure 2.21: Quantization error

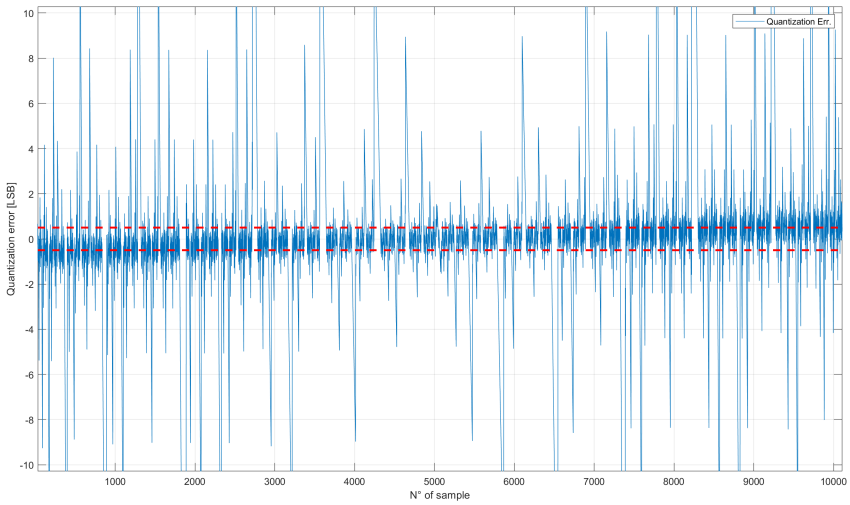


Figure 2.22: Quantization error zoomed on the central points

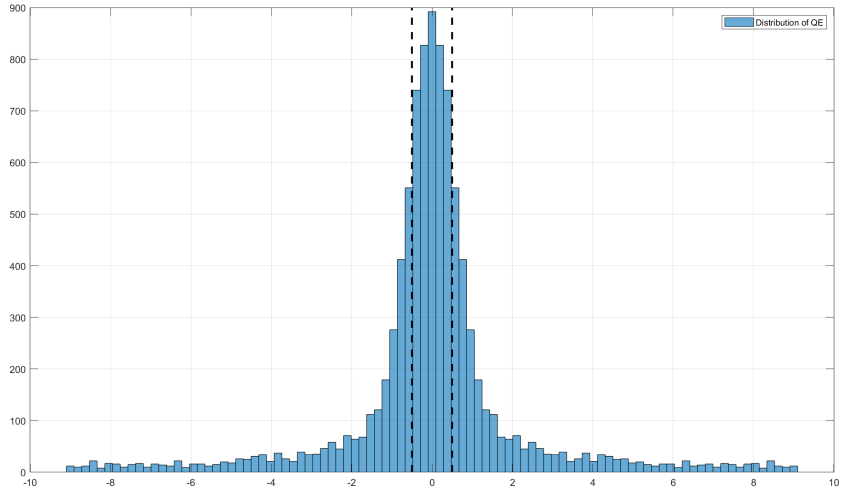


Figure 2.23: Distribution of quantization errors

Through the RMS value of quantization error, it is possible to compute the ENOB (Effective Number Of Bit) of the converter, using 2.2. The result obtained is

$$ENOB = 9.4887\,bit \simeq 9\,bit$$

Differential Non-Linearity DNL and INL computation would require a very high number of samples, resulting in a very heavy simulation. For this reason, DNL and INL have been evaluated on a narrow range of the input dynamic. The input signal used is still a voltage ramp, but varying in the range [0.9, 0.93]V. In this way, a very slow ramp is provided to the ADC, and the typical staircase transfer function is produced (figure 2.24).

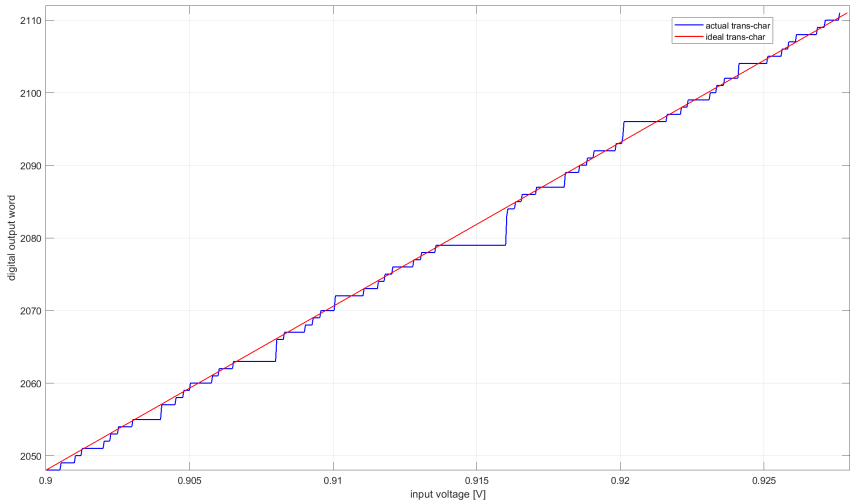


Figure 2.24: Trans-characteristic of ADC in the input range [0.9; 0.92] V

DNL has been computed using 2.4, and the graph of figure 2.25 has been obtained.

Absolute DNL is associated to the maximum value of this function, resulting to be $\sim$ 3 LSB.

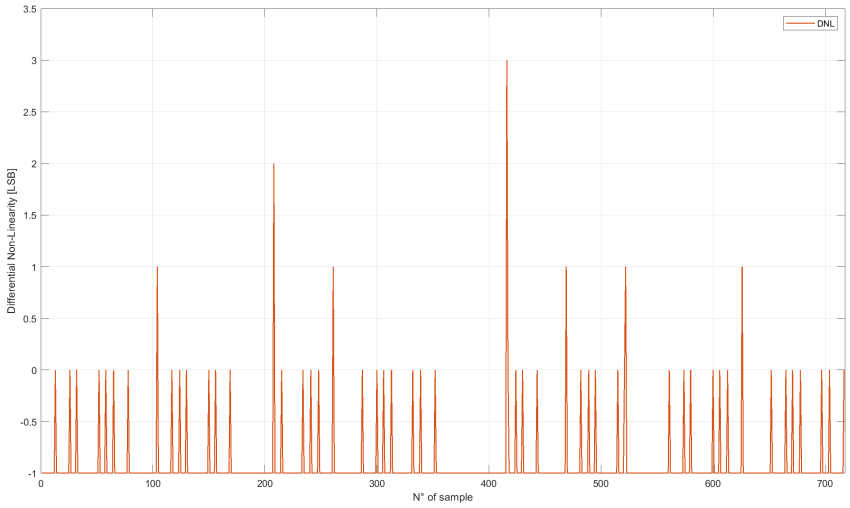


Figure 2.25: Differential non-linearity

Integral Non-Linearity INL has been computed using 2.6, and the graph of figure 2.26 has been obtained.

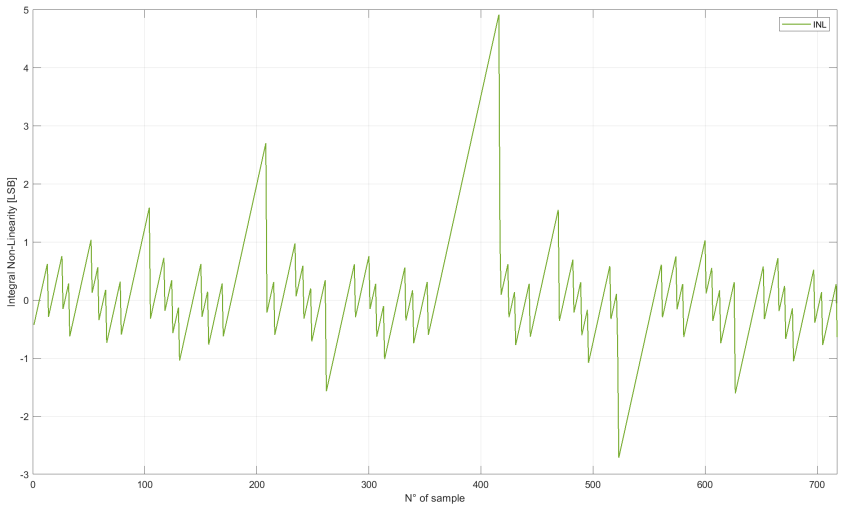


Figure 2.26: Integral non-linearity

Absolute INL is associated to the maximum value of this function, resulting to be 4.91 LSB.

Results recap In the following table there is a recap of all the results acquired from the ADC simulation.

Table 2.1: Results

Metric	Value
N	12 bit
FSR	1.8 V
V_{LSB}	440μ V
ENOB	$\sim$ 9 bit
Offset error	-1.7724 LSB
Gain error	$\sim$ 0
DNL	$\sim$ 3 LSB
INL	$\sim$ 4.91 LSB

Chapter 3

PCB design

The first element composing the test platform designed for the chip is a PCB. In this chapter, all the steps followed to design the PCB are reported. The PCB contains different components which allow to:

- Convert a digital stream representing a known test signal. This signal is converted to the analog domain and sent to chip as input signal, in order to be re-converted by the SAR ADC.

- Generate the reference voltages used by the ADC in the chip and by the components on the PCB, starting from a digital stream representing a constant voltage value in the analog domain.

- Regulate the supply voltage in order to bias the different parts of the chip with the right level of voltage.

The importance of designing this PCB lies in the possibility to have an ad-hoc testing platform for the chip. As already said, it can generate all the supply voltage levels required by the chip, and the voltage references. Furthermore, thanks to the presence of high-resolution DACs, it is possible to test the chip with a well-known input signal, which can be controlled by the user. In this way, knowing how this signal is converted from analog to digital through simulations, a direct comparison can be made with the future results of the various tests that will be performed. In particular, the PCB can be easily connected to a FPGA through a 20-pin connector, and it can directly host the chip, allowing its complete testing and characterization. The whole design has been entirely developed with the open-source software KiCad.

3.1 Choice of components

The first step of the design regards the choice of the components. In the table 3.1, a summary list of the components present on the PCB is reported.

Table 3.1: List of components

Component	N.
16-bit DAC	2
8-bit DAC	3
Linear Regulator	2
Adjustable Regulator	1
Trimming Potentiometer	1
Tantalum Capacitor	4
SMD Capacitor	50
SMD Resistor	51
LED	1
Connector 02x20	1
Connector 02x03	1
Connector 01x02	2
Barrel Jack	1
Battery Holder	1
Switch 01x03	1

A more detailed description of each component and of its usage is outlined in the following paragraph.

3.1.1 DACs for data

The first element which is needed on the board is a DAC. It is used to generate the analog test signal which will be the input signal for the chip. A digital bit stream programmed by the user, is controlled, synchronized and transmitted through a FPGA. This digital stream is received by the DAC and then converted in the analog form. In order to choose the model of DAC for this purpose, many constraints have been taken into account:

- The SAR ADC in the chip has a resolution of 12 bit, so the DAC for the data must have at least a resolution of 12 bit. Moreover, it has been intended to feed the chip with a lower supply voltage than the nominal one, aiming to observe and analyze its performances in different supply conditions. If the supply voltage of the chip is decreased, also its input dynamic range is

decreased. Consequently, the dynamic of the analog test signal to be provided must be reduced. Since the test signal is produced by these DACs, their output dynamic has to be limited. The limitation of the output dynamic corresponds to a degradation of the resolution, which means reduction of number of bits. In order to guarantee at least a resolution of 12 bits, a 16-bit DAC has been selected.

- The SAR ADC has 4 differential input for the first half of the chip, so 2 DACs with 4 output channel each, have been selected.
- Since the DAC must have 4 channel, its clock frequency could become a constraint. In particular, the effective maximum clock frequency of a DAC depends on the number of channel and on the number of bits to be sent to each channel. The selected DAC has 4 channels which require 24 bits each. These 24-bit words contain the digital data to be converted and information about the control of the DAC itself, and it has to be sent to each channel before starting the conversion. Therefore, if the nominal maximum clock frequency is 50 MHz, it must be divided by the number of channel (4) and by the number of bit of each word (24), resulting on an effective frequency of 500 kHz. This value widely meets the specification for the chip interface. A DAC with a serial interface capable of operating with input data clock frequencies up to 50 MHz, has been selected.

Considering all these constraints the DAC selected for the conversion of data was the DAC8555 by Texas Instruments, which is a 16-bit, Quad channel, voltage output DAC (figure 3.1).

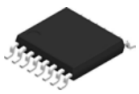


Figure 3.1: DAC 8555

3.1.2 DACs for references

The internal structure of the ADC under test requires different voltage references to work properly. In particular, the capacitive array which perform the conversion, has 4 switches that select the voltage terminal to connect to the bottom plate of the capacitors. These reference voltages are GND, V_{ref+}, V_{ref-} and V_{cm}.
Moreover, also the 2 DACs for data, placed on the PCB, require positive and negative voltage references to perform the conversion.
It has been decided to use also in this case 3 DACs, one for each voltage reference.

Each DAC receives a digital bitstream representing a constant signal. This signal is then converted by the DAC which will generate a precise constant voltage value. Since there are no stringent requirements for the resolution of these DACs, it has been stated that a resolution of 8 bit should be enough. Moreover, in this case each DAC generates a specific voltage reference, so a single channel voltage output DAC can be used.
Furthermore, it has been intended to use the same controller for all the DACs present on the board, but introducing three kinds of buses, two for the data DACs and one for all the reference DACs. The usage of only one data bus for the reference DACs is acceptable because there are no speed requirements. To share the bus, the three reference DACs must belong to the same family: they use a serial interface which is compatible with the SPI (serial peripheral interface) protocol. Furthermore, the reduced number of buses allows to spare traces on the board, and so it results to be a space-saving solution.
Considering these constraints, the DAC selected for the conversion of the voltage references is the DAC081S101 by Texas Instruments, which is a general purpose 8-bit, 1-channel, voltage output DAC (figure 3.2).

Figure 3.2: DAC 081S101

3.1.3 Regulators

All the integrated circuits soldered on the board, and of course the chip, need a certain supply voltage. Before the design of the PCB, it was intended to use three different ways to bias the circuit: using the power supply through a DC power connector jack, using 3 AAA batteries, and using the 5V output of the FPGA. Unlike the DC power supply, the usage of batteries allows to avoid the injection of additional noise in the circuit. However, the voltage provided must be regulated, to reach the precise level required by the integrated circuits and by the chip, and it must be maintained steady.
In particular, the core of the chip works properly with a power supply voltage of 1.8V, while the I/O and the pad ring work properly with a power supply voltage of 3.3V. Also, all the DACs on the PCB requires supply voltage level within a specific range. After having verified that the voltage level of 3.3V falls in the supply range of the DACs, this voltage level has been used also as supply voltage for the DACs, to reduce the number of components on the board.
Since these kinds of integrated circuits need a supply voltage as clean as possible,

an LDO (low-dropout) linear regulator has been chosen. Linear regulators use a transistor controlled by a negative-feedback circuit to produce a specified output voltage that remains stable despite variations in load current and in input voltage [21]. Also, an LDO is a particular linear regulator that can operate at very low potential difference between the input and the output voltage.
The regulator chosen for the 3.3V power supply is the ADP3300 by Analog Devices (figure 3.3), which has a supply range from 3V to 12V, and 5 fixed output voltage values, including 3.3V.

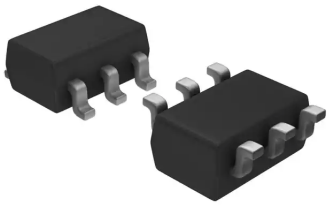


Figure 3.3: ADP 3300

On the board are present 2 ADP3300. One of them produces the "analog" 3.3V voltage, while the other produces the "digital" 3.3V voltage.

For what concern the supply voltage of 1.8V, it has been chosen an "adjustable" regulator. The output of these kinds of regulators can be programmed simply through an external resistance. The regulator chosen is the LT3085 by Analog Devices (figure 3.4).

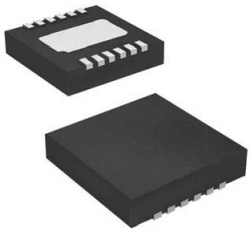


Figure 3.4: LT 3085

3.1.4 Other Components

There are other components on the PCB. Of course, resistor and capacitor have been placed in the circuit where needed.

Resistors The resistors adopted are 51 SMD type, inserted as series termination on the digital signal lines. They are there to manage transmission effects line such as ringing and oscillations from signals that have fast rise/fall edges.

Capacitors There are 50 SMD capacitors and 4 tantalum capacitors. Some of them have been directly assigned to the various integrated circuits reading their datasheets. Other capacitors have been inserted on the supply lines to obtain a filtering effect [22].

Trimmer Potentiometer As previously said, it is possible to adjust the output voltage of the regulator LT3085 through an external resistance, connecting it from its SET pin to ground [23].To have full control on this voltage, the 3362 Trimpot Trimming potentiometer by Bourns (figure 3.5) has been placed on the PCB. It is a trimmer resistor of 200kΩ, which allows, when its value is set to 182kΩ, to obtain an output voltage of 1.8V from the adjustable regulator LT3085, while trimming it, the output voltage will decrease consequently.

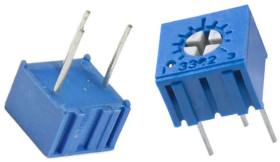


Figure 3.5: 3362 Trimpot Trimming Potentiometer

Connectors On the PCB are also present some connectors:

- A connector 02x20 (figure 3.6) has been placed to allow connection between the PCB and the FPGA.
- 2 connector 01x02, one inserted between the 3.3V power supply and the supply input of the chip, while the other inserted between the 1.8V power supply and the digital supply input of the chip. Practically, it has been planned to measure externally the current absorbed by the chip from the power supply, during its working conditions.

- A connector 02x03 allows to choose what kind of supply must be connected to the global VCC. It works properly when combined with a switch.

Figure 3.6: 02x20 Connector

Other components complete the list. They are:

- A barrel jack (figure 3.7), which allows to supply the entire board with the external 5V taken from the power supply.
- A LED, that is useful to determine rapidly if the PCB is on or not.
- A switch selects one of the 3 ways, already discussed, of biasing the board.
- A battery holder (figure 3.8), used to host 3 AAA batteries.

Figure 3.7: Barrel Jack

Figure 3.8: Battery Holder

3.2 Schematic

After having chosen all the components needed, the first step designing a PCB is to create a schematic view of the circuit. As already mentioned, the software KiCad has been exploited for the entire design of the PCB. KiCad offers the program EESchema, through which is possible to edit a schematic.
The schematic has been designed creating 3 different hierarchy levels. Two hierarchy levels contain respectively the supply part, including the regulators and the trimmer, and the data part, which includes all the DACs. Finally, the top hierarchy level contains the two instances of the lower levels, and the rest of the components, such as a schematic view of the chip, connectors, switch, etc.

3.2.1 Data part

As already said, this hierarchy level contains all the DACs, and a group of capacitances and resistances. Several 50 Ω resistors has been placed on the digital signal lines, in order to manage transmission line effects. Several capacitors instead, has been inserted on the supply lines to obtain a filtering effect.
The two 16-bit DACs shares the clock signal (CLK16), as can be seen in figure 3.9, which is connected to the SCLK pin. They are responsible to produce 4 input signal each, that are needed by the chip. Then, each DAC has its own serial data-in bus (DIN_1 and DIN_2), and its own SYNC signal (SYNC1 and SYNC2), thanks to which is possible to synchronize the transmission of data.
Furthermore, both 16-bit DACs receives the supply voltage from the regulators, and the reference voltages from the other 3 DACs that are intended to produce the references. This can be easily implemented in the schematic with global labels.
Finally, there are other signals needed by these integrated circuits, which functions are described in the next chapter.

Unlike the DAC for data, the DACs for references share not only the clock signal (CLK8), but also the serial data-in bus (it is possible because they all use the same communication protocol), as can be observed in figure 3.10. Data will be synchronized through the 3 SYNC signals during the FPGA programming. In this way, 2 bus lines are spared, and this is an acceptable solution because this kind of DACs receive as input always the same constant bit-stream.

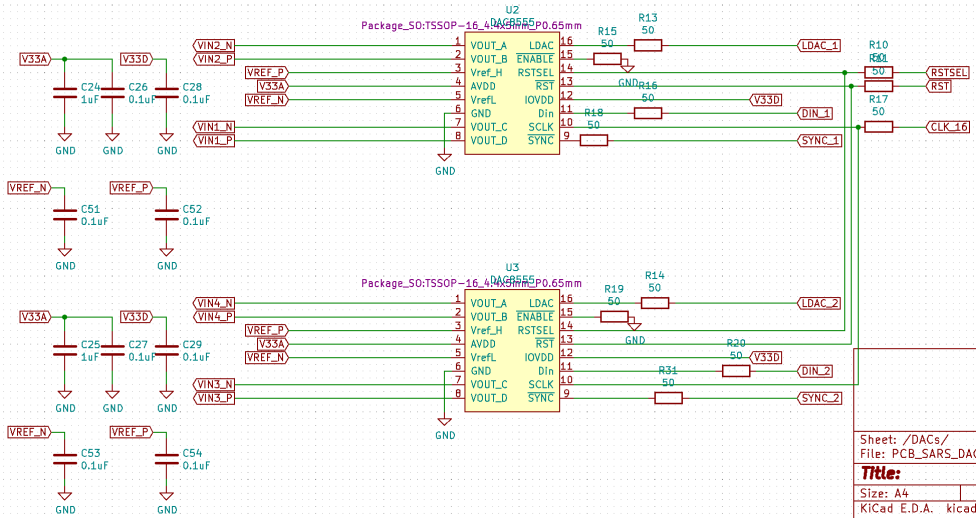


Figure 3.9: Schematic of the data part (Data DACs)

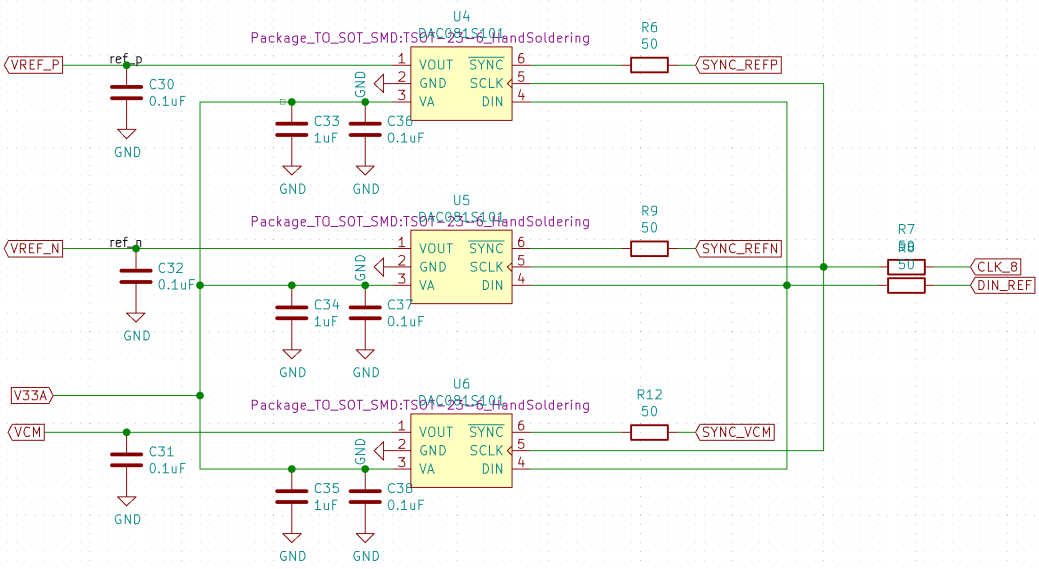


Figure 3.10: Schematic of the data part (Reference DACs)

In the following figure (3.11) there is the whole schematic of the data part.

Observing the schematic, it can be noticed that 50 Ω resitors have been placed on the digital line, where digital signals with fast rise and fall edges flow. Traces starts from the FPGA pins, and they end on a high impedance node, which is the input of a gate. These nodes provide a capacitive load on the trace, that, together with the characteristic inductance of the trace, they constitute an LC. When digital

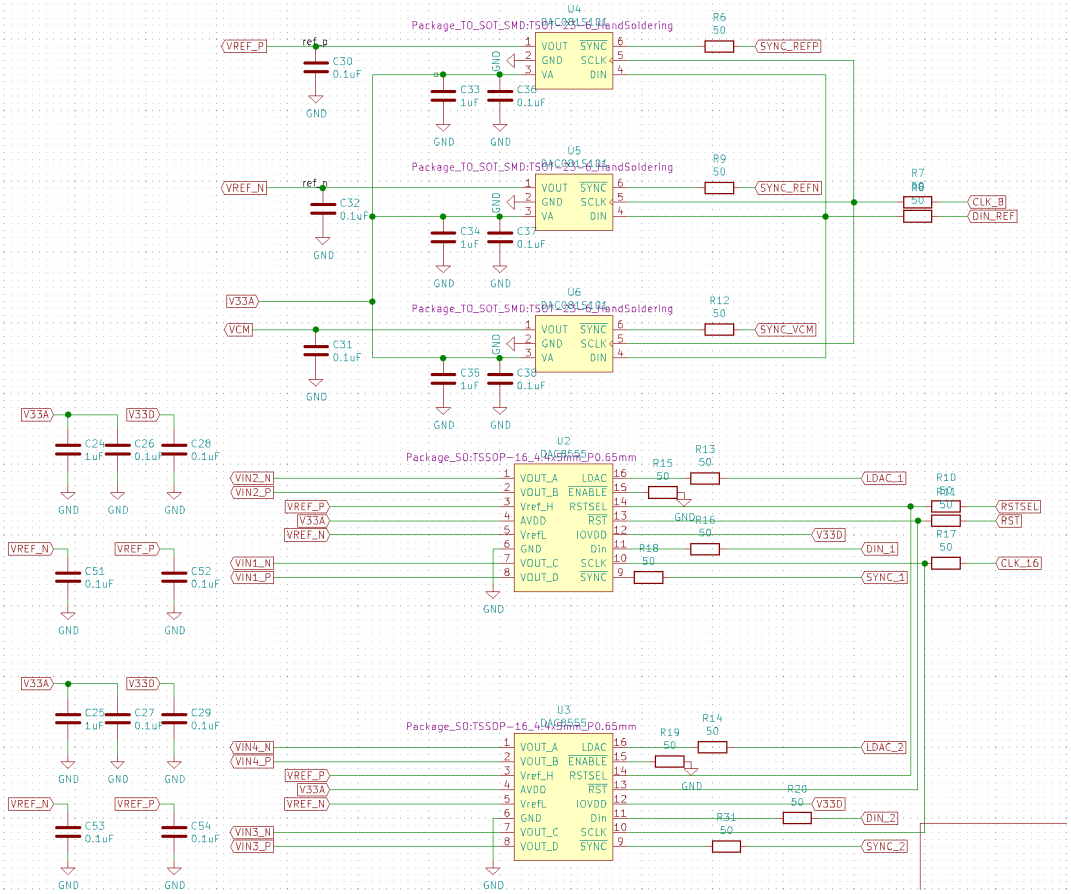


Figure 3.11: Schematic of the whole data part

signals run across these kinds of long traces, they stimulate oscillation. Inserting a resistance allows to increase the attenuation of possible oscillations.
Finally, some capacitors have been placed on the supply lines, with the aim to obtain filtering effect of the noise, while others are requested by the components, according to what is indicated in their datasheets.

3.2.2 Supply part

This hierarchy level of the schematic contains all the components responsible of generating the supply voltage levels which bias all the integrated circuits present on the PCB and the chip itself. Also in this case, in the regulators' datasheets is possible to find application information which suggests to insert several bypass capacitors, in order to guarantee stability. In figure 3.12, it can be observed the whole schematic of the supply part. The two 3.3V regulators (ADP3300) receive as input the voltage V_{cc}, which is the external 5V supply. They generate the "analog" 3.3V voltage and the "digital" 3.3V voltage. The "analog" 3.3V voltage is used to bias all the DACs, and as voltage input for the adjustable regulator, while the "digital" 3.3V voltage is used to bias the chip and to provide the digital I/O power supply voltage for the 16-bit DACs.
Then, the adjustable regulator (LT085) receives the "analog" 3.3V voltage as input, and it generates the 1.8V voltage. This voltage level can be adjusted simply trimming the external resistor connected to the SET pin. Indeed, in the schematic in figure 3.12 there is a trimmer potentiometer connected to the regulator, that allows to change the output voltage. The reason why it is necessary to control this voltage is to try to characterize the chip under test providing a supply voltage under the 1.8V level, observing how it behaves.

3.2.3 Top hierarchy level

Finally, the schematic has been completed creating the top-level hierarchy. In this level there are the instances of the two sub-parts already discussed. Then, the symbol of the chip and the symbol of the 02x20 connector has been inserted. In this way, it has been possible to connect all the pins of the chip and to configure the interface between the PCB and the FPGA.
Furthermore, a section dedicated to the supply is showed in figure 3.13: there are the symbol of the battery holder, of the barrel jack, and of the switch. As can be seen, the switch selects in which way one wants to get the 5V supply: from the 5V provided by the FPGA, from the DC jack, or from the battery. The latter is the best method to avoid insertion of a power supply, which could induce injection of additional noise in the circuit. Therefore, it has been chosen as the most suitable solution for this kind of application.

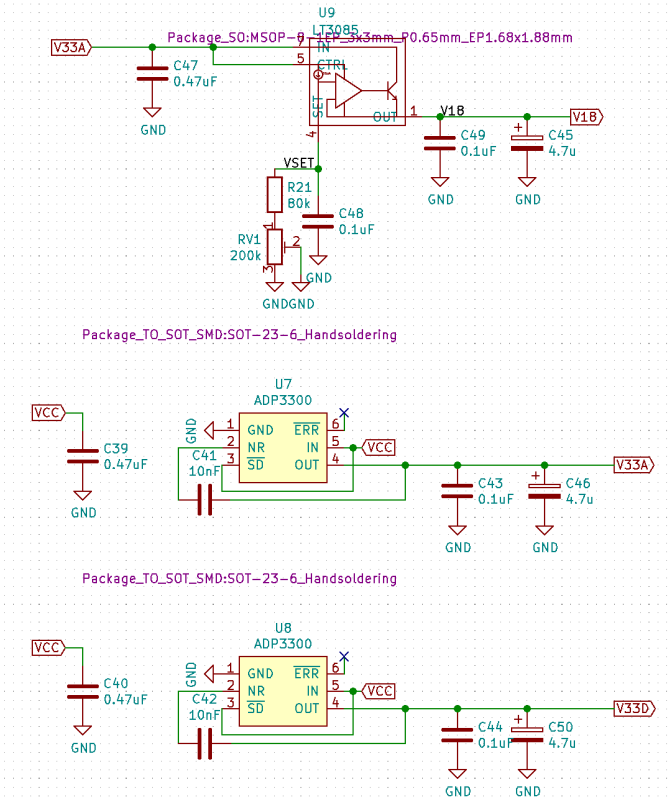


Figure 3.12: Schematic of the supply part

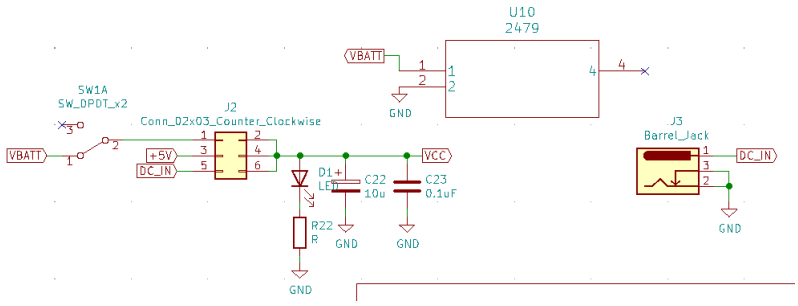


Figure 3.13: Detail of the top-level schematic

In figure 3.14 there is the whole PCB schematic. It can be noticed that there are groups of three capacitors connected between each differential input of the chip. Capacitors between the inputs and ground, allows to eliminate common mode noise, while the capacitor across the two differential inputs acts on the differential mode, filtering the signals. Then, the whole group of capacitors works as an anti-aliasing filter, which eliminates the high frequency components in the signals.

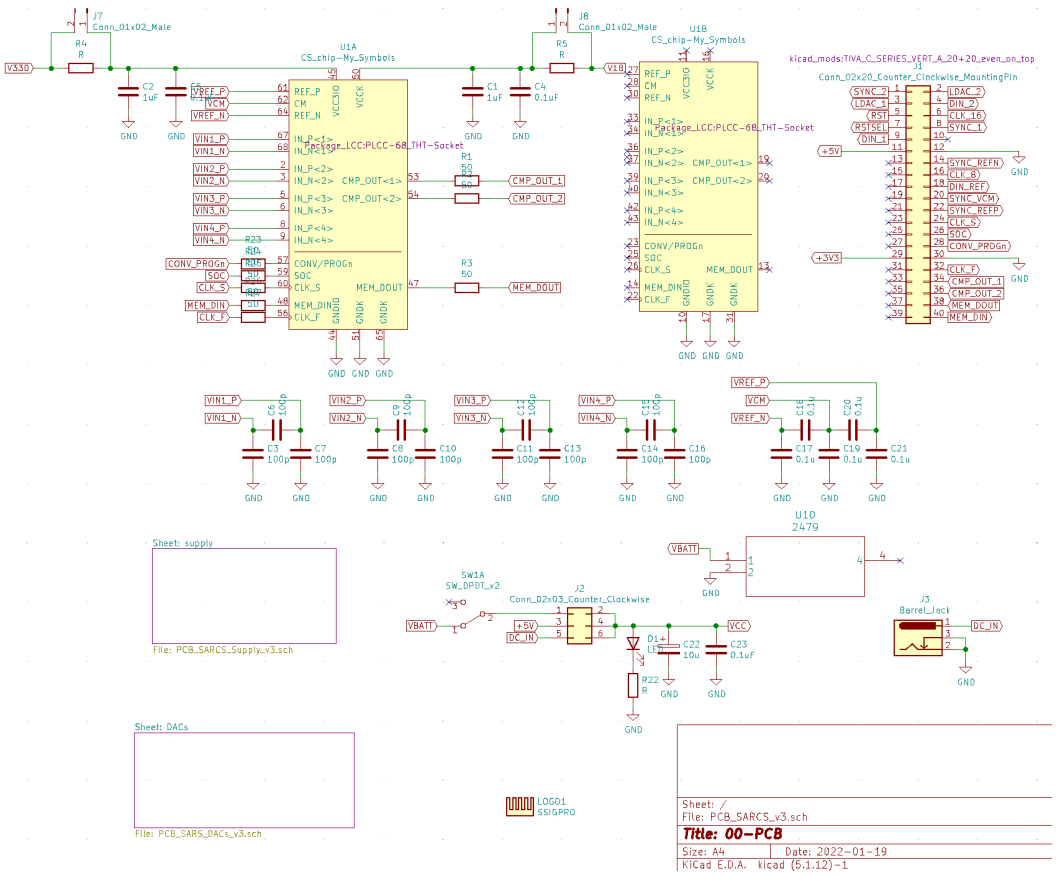


Figure 3.14: Complete PCB schematic

After having terminated the schematic, the software extracts the relative netlist which will be used to match schematic and layout. At this point, it is necessary to assign footprints to any component in the schematic. After this step, one can continue the design, starting to work on the layout.

3.3 Layout

As mentioned, a preliminary operation must be done before starting to design the layout of the PCB. It is the assignment of footprints to each component which will be soldered on the board. A footprint is the arrangement of pads (in surface-mount technology) or through-holes (in through-hole technology) used to physically attach and electrically connect a component to a printed circuit board.

After this step, is possible to design the layout. The software KiCad offers a program called PcbNew which allows to create a layout. This kind of PCB presents two different layers, which can be exploited to ease the routing and to use them as ground planes. The two layers are the front copper layer (represented on layout in red) and back copper layer (represented on layout in green). The figure 3.15 shows the layout without tracks, where only components' footprints are present, and information about the final dimensions of the board.

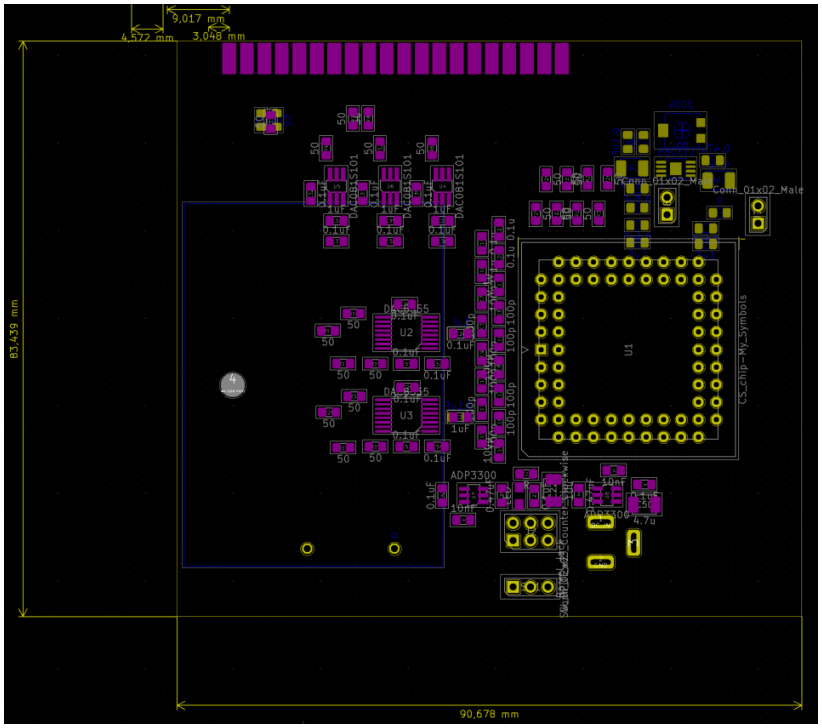


Figure 3.15: PCB layout. Copper layers are hidden

The figures 3.16 and 3.17, shows the front copper layer with all the tracks respectively hiding and showing the filled areas in zones.

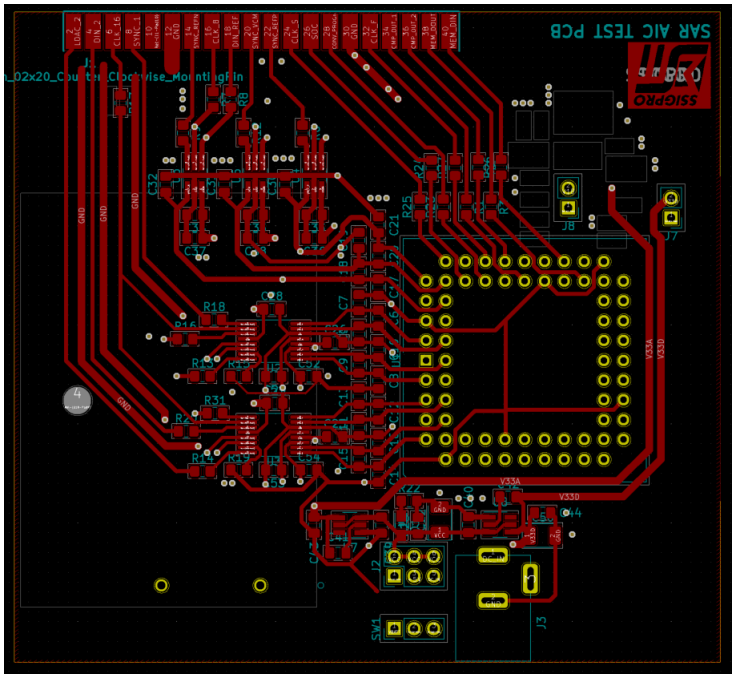


Figure 3.16: PCB layout. Only front copper layer is shown without filled areas.

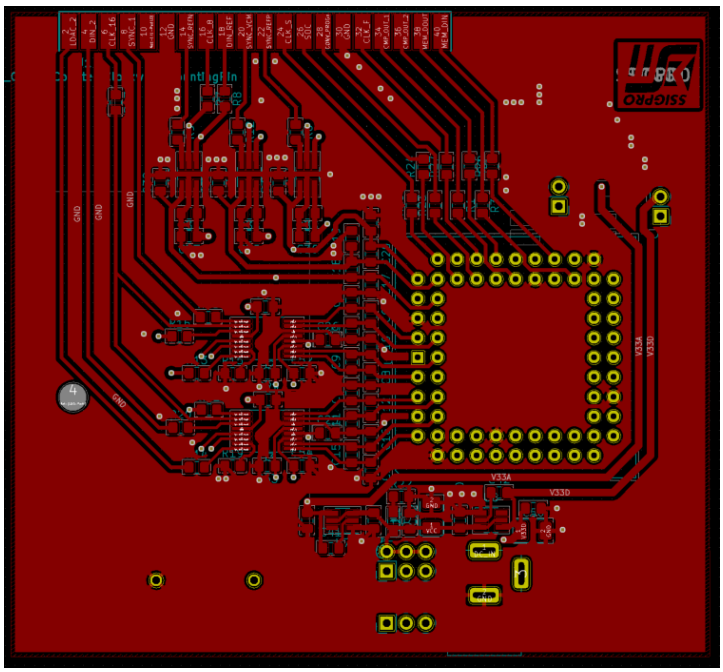


Figure 3.17: PCB layout. Only front copper layer is shown

The figures 3.18 and 3.19, shows the back copper layer with all the tracks respectively hiding and showing the filled areas in zones.

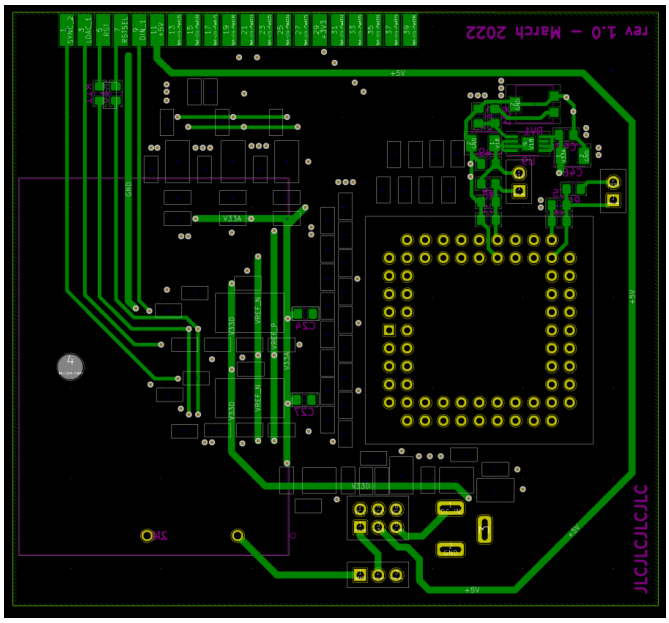


Figure 3.18: PCB layout. Only back copper layer is shown without filled areas.

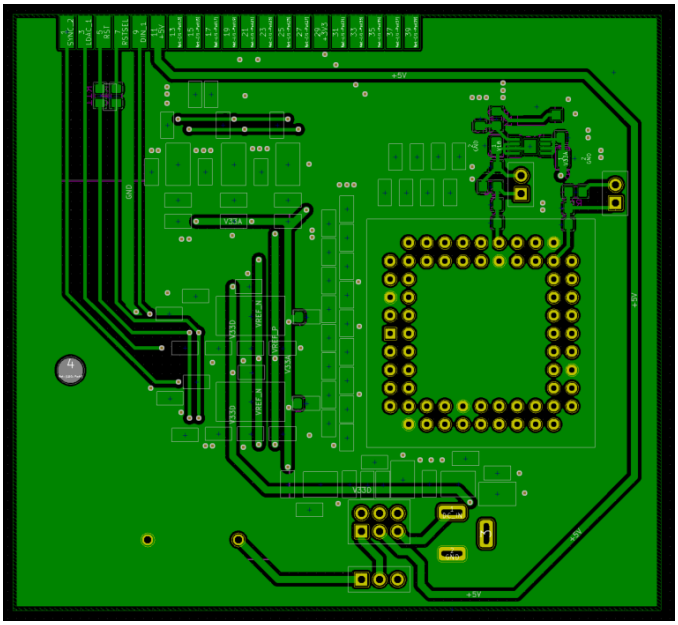


Figure 3.19: PCB layout. Only back copper layer is shown

The figure 3.20 shows the whole layout, including both copper layer with all the tracks, but hiding the filled areas in zones for a better visualization.

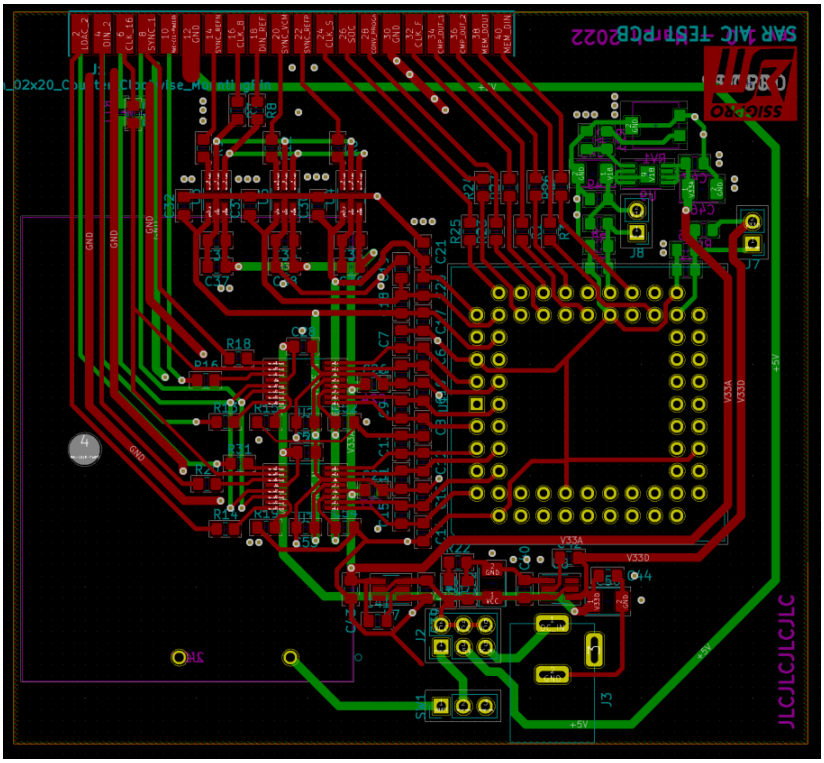


Figure 3.20: PCB layout without filled areas.

The following figures (3.21 and 3.22) are two photos of the produced PCB.

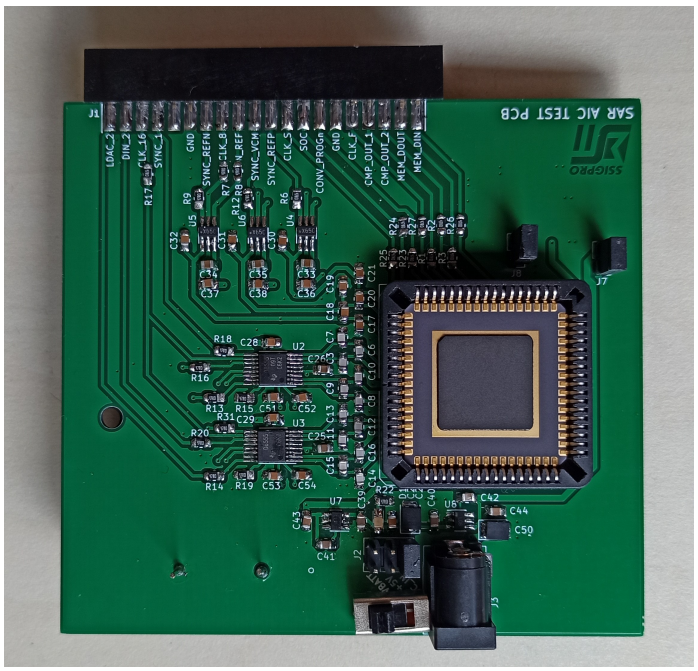


Figure 3.21: Photo of the front of the PCB

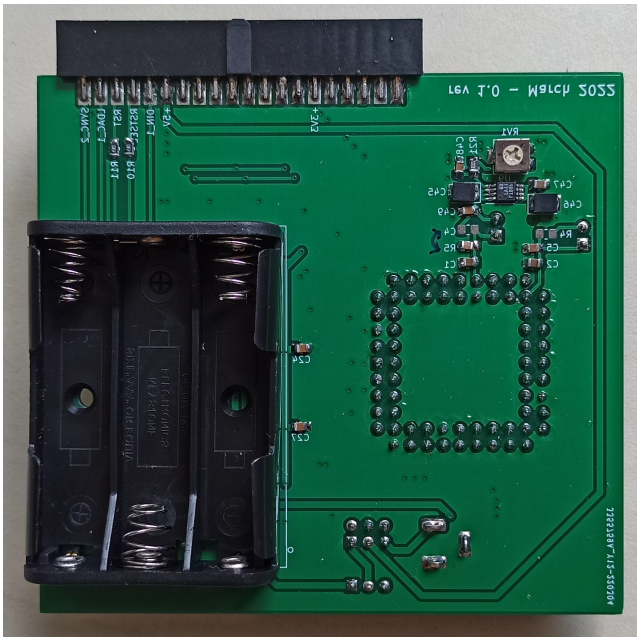


Figure 3.22: Photo of the back of the PCB.

Chapter 4

FPGA Programming

This chapter describes the design of the "digital" part included in the testing platform for the chip. It consists essentially in programming a FPGA. Components such as DACs, present on the PCB, need to be controlled. Each DAC has precise timing requirements and protocols which manage the serial write operation. In particular, data and clock signals must be sent with specific timing, and the commands exchanged between the transmitter (FPGA) and the receiver (DAC) must be synchronized.
Furthermore, the chip under test requires several control signals, to properly perform all the operations included in the analog-to-digital conversion. In particular, control signals are required to synchronize the different operations and make possible state changes. The chip has an internal memory that must be programmed at the beginning of the conversion. Obviously, both the chip and the DACs require their own clock signals, that must be controlled through the clock gating technique, in order to spare energy.
The digital design has been done in VHDL, while code simulations has been performed using the software Modelsim. Finally, having tested all the VHDL code, it has been synthetized through the software Quartus, and run on the FPGA.
There are three main blocks which have been described in VHDL: a block responsible to control the high-resolution DACs for data, a block responsible to control the reference DACs, and a block responsible to control the chip.

4.1 Data DACs control

In order to design the block which controls both high-resolution DACs for data, it is necessary to analyze which kind of signals they need as input. The starting point is to consult the datasheet. In figure 4.1 there is the pin configuration of the DAC8555 taken from its datasheet.

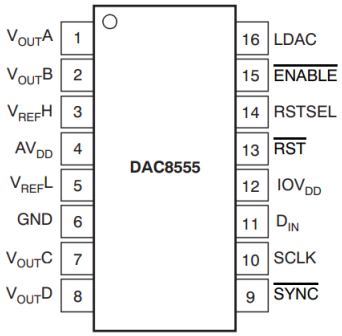


Figure 4.1: Pin configuration of DAC8555 [24]

Excluding the output pins and the ones related to the supply, the signals needed by the DAC are: SYNC, SCLK, Din, RST, RSTSEL, ENABLE and LDAC, associated respectively to pin 9, 10, 11, 13, 14, 15 and 16. However, the most important signals for the timing are SYNC, SCLK and Din, which allows to synchronize and properly perform write operations. Each DAC has its communication protocol, indeed, in every datasheet is possible to find information about how to communicate with the component, as well as useful timing diagrams. In figure 4.2 there is a timing diagram taken from the DAC8555's datasheet, which describes a serial write operation.

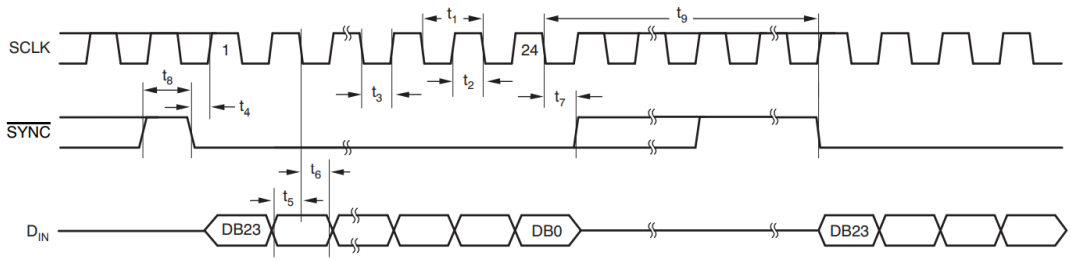


Figure 4.2: Timing diagram of serial write operation of DAC8555 [24]

As can be seen, the operation starts forcing the signal SYNC to the logic level '0'. SYNC is a level-triggered control input which is active LOW [24]. This is the frame synchronization signal for the input data. When SYNC goes LOW, it

enables the input shift register and data is transferred in on the falling edges of the following clocks.
The SCLK and Din input are respectively a serial clock input and a serial data input. Data is clocked into the 24-bit input shift register on each falling edge of the serial clock.
The write sequence begins by bringing the SYNC LOW. Data from the DIN line are clocked into the 24-bit shift register on each falling edge of SCLK. On the 24th falling edge of the serial clock, the last data bit is clocked into the shift register and the shift register gets locked. Further clocking does not change the shift register data. Once 24 bits are locked into the shift register, the eight MSBs are used as control bits and the 16 LSBs are used as data. After receiving the 24th falling clock edge, the DAC8555 decodes the eight control bits and 16 data bits to perform the required function, without waiting for a SYNC rising edge. A new SPI sequence starts at the next falling edge of SYNC. A rising edge of SYNC before the 24-bit sequence is complete resets the SPI interface: no data transfer occurs. After the 24th falling edge of SCLK is received, the SYNC line may be kept LOW or brought HIGH. In either case, the minimum delay time from the 24th falling SCLK edge to the next falling SYNC edge must be met in order to properly begin the next cycle.
The input 24-bit input shift register of the DAC8555 is 24 bits wide, as shown in figure 4.3, and is made up of eight control bits (DB23-DB16) and 16 data bits (DB15-DB0). DB 23 and DB22 should always be '0'. LD1 (DB21) and LD0 (DB20) control the updating of each analog output with the specified 16-bit data value or power-down command. Bit DB19 is a don't care bit that does not affect the operations. The DAC channel select bits (DB18 and DB17) control the destination of the data form DAC A through DAC D (see figure 4.4). The final control bit, PD1 (DB16), selects the power-down mode of the DAC8555 channels.
In figure 4.4 there is the functional block diagram of the DAC8555; it can be useful to better understand its internal structure and its control.

DB23											DB12
0	0	LD1	LD0	X	DAC Select 1	DAC Select 0	PD0	D15	D14	D13	D12
DB11											DB0
D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

Figure 4.3: DAC8555 Data input shift register format [24]

Having gathered all this information, it is possible to start designing VHDL block which will allow the DAC8555 control.

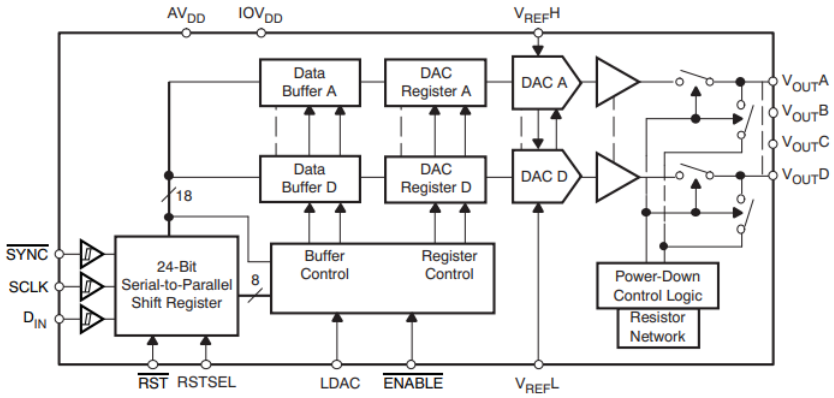


Figure 4.4: DAC8555 Functional block diagram [24]

4.1.1 Datapath

In order to describe all the hardware needed in VHDL, a "paper and pencil" approach has been adopted. First of all, a possible datapath, containing different the digital elements, has been designed. In figure 4.5 there is a simple draw of the datapath which will be then described in VHDL.

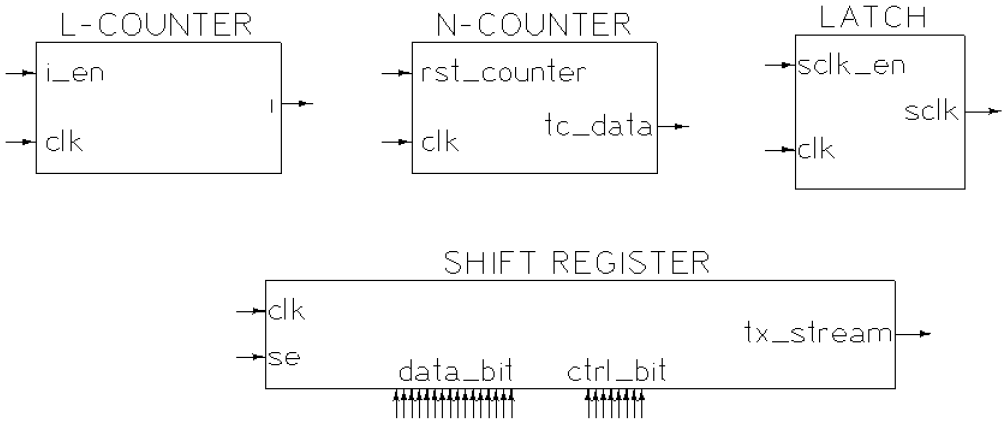


Figure 4.5: Datapath for high-resolution DACs' control

Below is a detailed description of each element and its functions.

Shift Register Since the DAC for data has a data input shift register 24-bit wide, surely, a 24-bit wide shift register is needed. It allows to serialize data represented on 24-bit words, and to send them to the DAC in the form of a bit serial stream. The shift register has 3 kind of inputs and one output:

- Clock signal. The "clk" input allows to synchronize the shift operation with a global clock.
- Data signals. The two bus signals "ctrl_bit" and "data_bit" are respectively 8-bit and 16-bit wide. Bus "ctrl_bit" represents the first 8 bits of the data to be sent, which contain information about the channel selection, the destination of the data in the DAC, and the power-down modes. Bus "data_bit" represents the last 16 bits which contain the true data to be converted by the DAC. These two strings separately come from a top level entity (described later). Thus, the padding operation is performed inside the shift register.
- Control signal. The "se" input is a shift-enable. When the signal "se" is '1', the shifting operation starts and a new bit is put in output at every clock cycle.
- Output. The signal "tx_stream" is the serial output of the shift register. Bits are sent one by one directly to the Din pin of the DAC8555.

N-counter This is a 24-counter, which is enabled every time there is a new word to be transmitted. The counter's task is to count the number of shifts and to flag to the control unit that a 24-bit word has been completely transmitted through its flag output "tc_data" (terminal count).

L-counter This is an L-counter, where L can be any number. L represents the number of 24-bit words to be transmitted, so it works as a sort of index through the output "i". Through this index, the top-level entity can control and select the word to sent.

Latch The latch implements the clock gating. Essentially, through the enable input "sclk_en", the output "sclk" can be forced to '0', when there is no transmission, or it follows the global clock signal "clk". Signal "sclk" is the serial clock used by the DAC to acquire the input data.

4.1.2 FSM

The control unit completes the design of the block which controls the DAC8555. This block is essentially a FSM, which stands for Finite State Machine. In figure 4.6 there is a simple draw that shows a block with all the inputs and outputs.

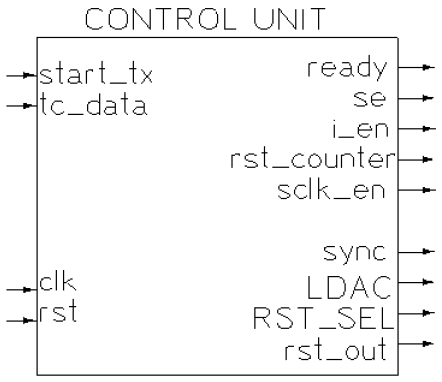


Figure 4.6: Control unit for high-resolution DACs' control

Below is a detailed description of the input/output signals of this FSM, followed by an analysis of its states and their functionality.

Inputs The FSM changes from one state to another in response to some inputs that trigger each transition.

- start_tx. This signal triggers the beginning of a transmission. It is an acknowledge signal which is interchanged with the output signal "ready". Through these two signals, it is possible to have an handshake between the TX (transmitter) and the user, who is sending data to the shift register. When "start_tx" goes high, it means that the data is valid, and a transmission can start.

- tc_data. This signal flags the end of the transmission. It is sent to the FSM by datapath. Every time a 24-bit word is fully transmitted, "tc_data" goes high and the TX returns free.

Outputs The FSM produces output signals which are effectively the control signals. Through these signals is possible to drive elements in the datapath and to communicate with the user.

- sync. This signal is the frame synchronization signal for the DAC8555's input data already discussed. It must be low during the transmission, and it must return high when it finishes.
- ready. This is the second acknowledge signal which is interchanged with the user. When a transmission ends, the counter and the shift register are reset. After this time instant, the TX is considered free, and the signal "ready" flags this condition.
- sclk_en. This is the enable for the clock gating.
- se, le. They are respectively the shift enable and load enable signals for the shift register.
- rst_counter, rst_shift. They are respectively reset signals for the 24-counter and for the shift register
- LDAC,rst,rstsel. These are signals required by the high-resolution DACs, which allow to reset it, or to manage the operations.

States The FSM follows a precise evolution, triggered by the input signals, through its states. Therefore, these signals cause a transition of the FSM, and, during each state, it produces an output pattern which controls the datapath. In figure 4.7 there is the state diagram of this FSM, followed by a detailed description.

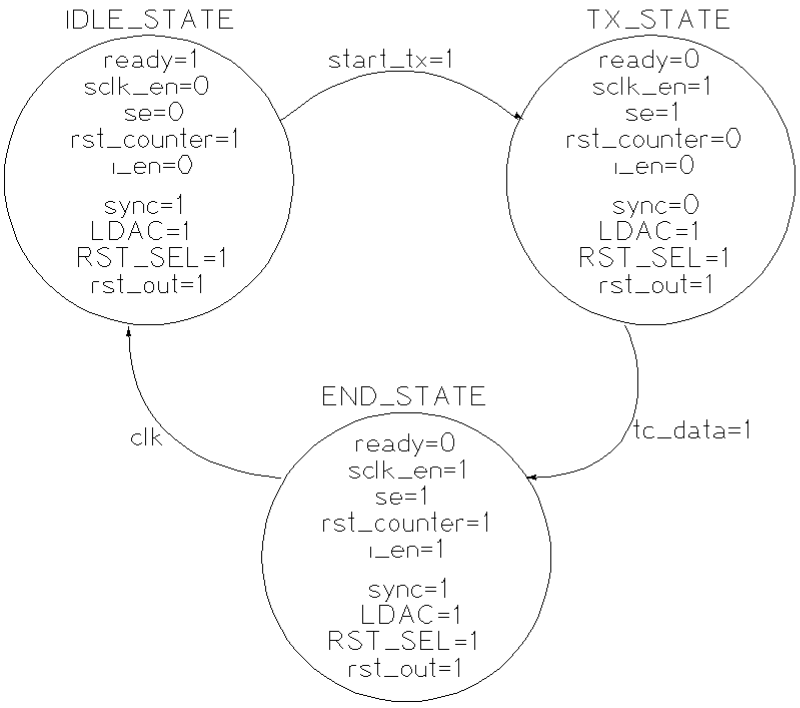


Figure 4.7: State diagram for high-resolution DACs' control

- IDLE_STATE. In this state, TX is in a waiting state and it is free, so the acknowledge signal "ready" is asserted. Since TX is not transmitting, the "sync" signal for the DAC must go '0', while the serial clock "sclk" is disabled ("sclk_en='0'). When the input "start_tx" is forced to '1' by the user, there is a transition to the next state, which is TX_state.

- TX_STATE. During this state the TX is busy because it is serializing and transmitting the word provided by the user, so the signal "ready" goes low and the shift enable signal "se" goes high. Also, "sync" and "slck_en" go respectively to '0' and '1', as required by the DAC's configuration for the serial write operation. When the 24-bit counter finishes its count, the transmission is completed and, through the input signal "tc_data" provided by the datapath, the FSM changes state.

- END_STATE. In this state, the 24-bit counter is reset, and the L-counter for the number of words sent, is enabled and so it counts up. The next transition through the IDLE_STATE is simply triggered by the next clock rising edge.

Finally, a higher level entity (figure 4.8) containing both datapath and FSM has been designed. Since there are two 16-bit DAC on the board to be controlled, two instances of the same entity have been declared.

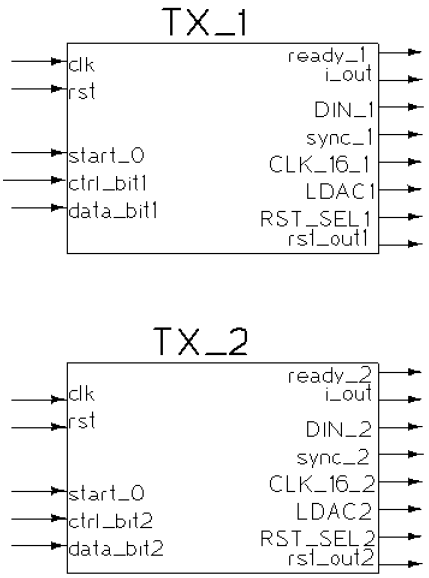
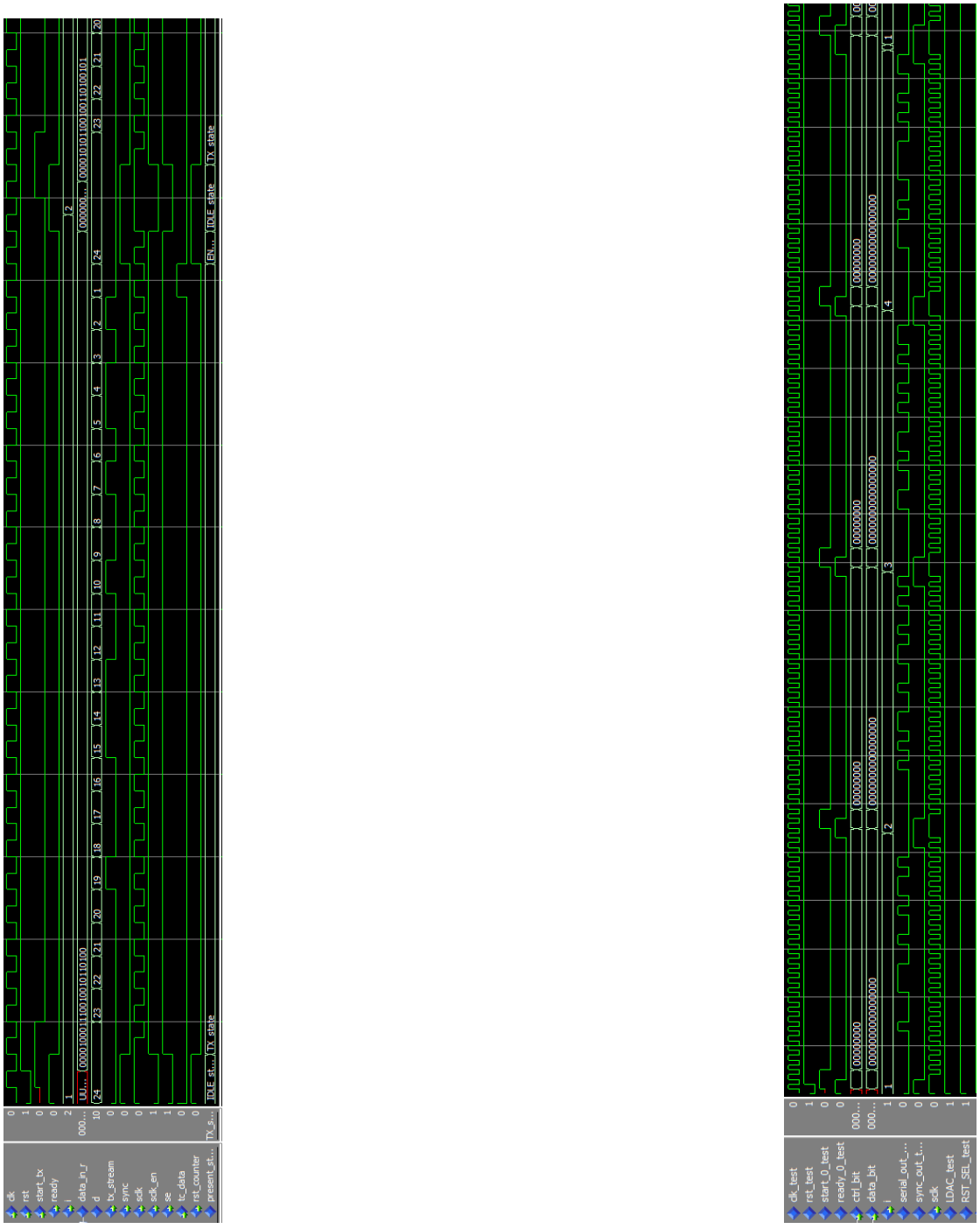


Figure 4.8: TX for high-resolution DACs' control

4.1.3 Modelsim Simulations

The hardware described in VHDL has been tested in Modelsim. Figure 4.9a shows the simulation results for a single 24-bit word ("data_in_r") transmission. The signal "tx_stream" is its serial representation.
A full transmission of 4 24-bit words is depicted in figure 4.9b. This simulation refers to the entity of figure 4.8.



(a) **(b)**

Figure 4.9: Modelsim simulation. Transmission of a single word (a) and of 4 words (b) for the high-resolution DAC.

4.2 Reference DACs control

As for the high-resolution DACs, in order to design the block which controls the DACs for references, it is necessary to analyze which kind of signals they need as input. The starting point again, is to consult the datasheet. In figure 4.10 there is the pin configuration of the DAC081S101 taken from its datasheet.

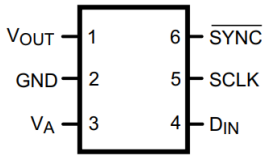


Figure 4.10: Pin configuration of DAC081S101 [25]

Excluding the output pin and the one related to the supply, the signals needed by the DAC are: SYNC, SCLK and Din, associated respectively to pin 6, 5, and 4. Also in this case, in the datasheet it is possible to find information about how to communicate with the component, as well as useful timing diagrams. In figure 4.11 there is a timing diagram taken from the DAC081S101's datasheet, which describes a serial write operation.

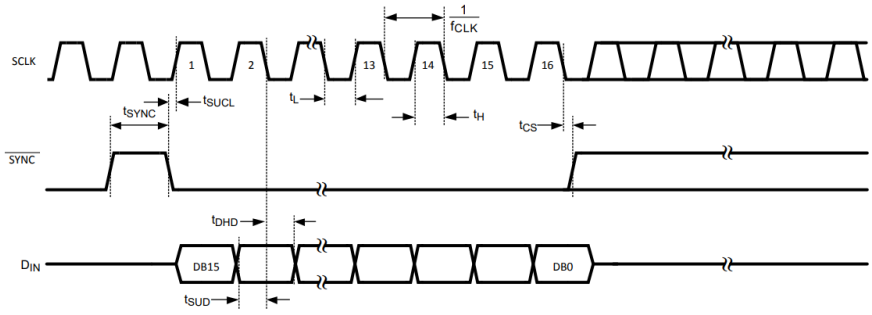


Figure 4.11: Timing diagram of serial write operation of DAC081S101 [25].

As for DAC8555, the operation starts forcing the signal SYNC to the logic level '0'. SYNC is the frame synchronization input for the data input [25].
The SCLK and Din input are respectively a serial clock input and a serial data input. Data is clocked into the 24-bit input shift register on each falling edge of the serial clock.
A write sequence begins by bringing the SYNC line low. Once SYNC is low, the data on the DIN line is clocked into the 16-bit serial input register on the falling edges of SCLK. On the 16th falling clock edge, the last data bit is clocked in and the programmed function (a change in the mode of operation and/or a change

in the DAC register contents) is executed. At this point the SYNC line may be kept low or brought high. In either case, it must be brought high for the minimum specified time before the next write sequence as a falling edge of SYNC is used to initiate the next write cycle.
The input shift register (figure 4.12) has 16 bits. The first two bits are "don't care" and are followed by two bits that determine the mode of operation (normal mode or one of three power-down modes), while the remaining 8 bits contains the data.

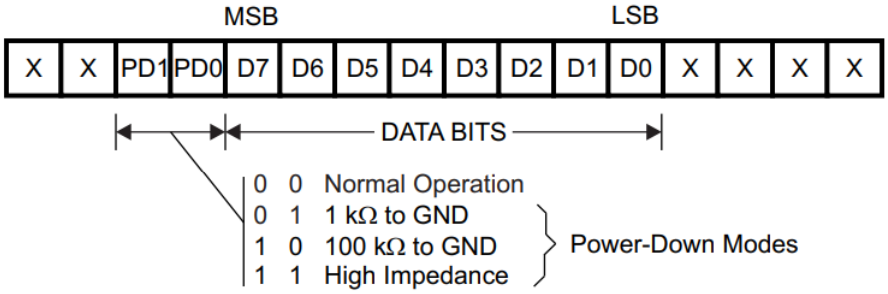


Figure 4.12: DAC081S101 Data input shift register format [25].

Having gathered all this information, it is possible to start designing VHDL block which will allow the DAC081S101 control.

4.2.1 Datapath

Also in this case, a "paper and pencil" preliminary approach has been adopted. In figure 4.13 the datapath containing all the digital blocks, is depicted.

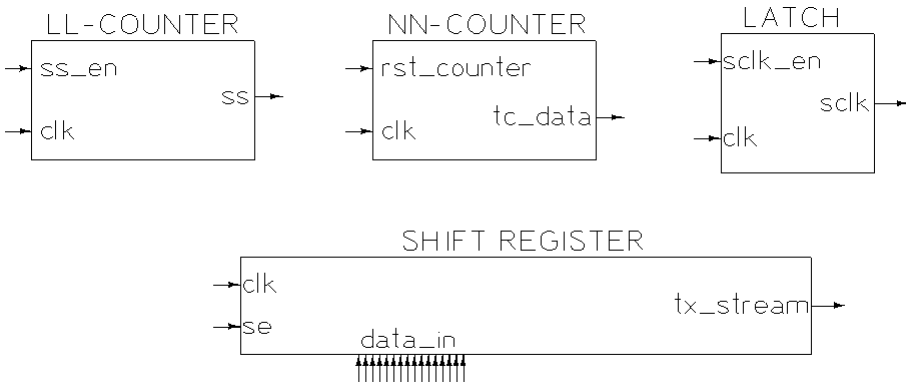


Figure 4.13: Datapath for reference DACs' control

Below is a detailed description of each element and of its functions.

Shift Register Also in this case, the DAC081S101 has a 16-bit input shift register, so a 16-bit wide shift register is needed in the datapath, in order to serialize data represented on 16 bits. It has:

- Clock signal. The "clk" input allows to synchronize the shift operation with a global clock.
- Control signal. the "se" signal is a shift enable. When it goes high, the shift register starts to send the bitstream, shifting the 16-bit word.
- Data signal. It is the input bus for data sent by the user.
- Output. The signal "tx_stream" is the serial output of the shift register. Bits are sent one by one directly to the DIN pin of the DAC081S101.

NN-counter This is a 16-counter, which is enabled every time there is a new word to be transmitted. The counter's task is to count the number of shifts and to flag to the control unit that a 24-bit data has been completely transmitted through its flag output "tc_data" (terminal count).

LL-counter This is a 3-counter which synchronizes the transmission over the three reference DACs. Since they share the same data bus and the same serial clock, it has been possible to instantiate only one block to control all the reference DACs. However, this counter controls the FSM transitions. There is a transmission state for each reference DAC which computes the right configuration of the "sync" signals needed. Furthermore, the counter is exploited to point a register which contains the three 16-bit words to be sent to the reference DACs. These data are always the same, since they represent a constant voltage.

Latch The latch implements the clock gating. Essentially, through the enable input "sclk_en", the output "sclk" can be forced to '0', when there is no transmission, or it follows the global clock signal "clk". Signal "sclk" is the serial clock used by the DAC to sample the input data. All the three reference DACs receive the same serial clock.

4.2.2 FSM

The control unit has been designed, also in this case, with a FSM. In figure 4.14 there is a simple draw that shows a block with all the inputs and outputs

Below is a description of the input/output signals of this FSM, followed by an analysis of its states and of their functionality (figure 4.15).

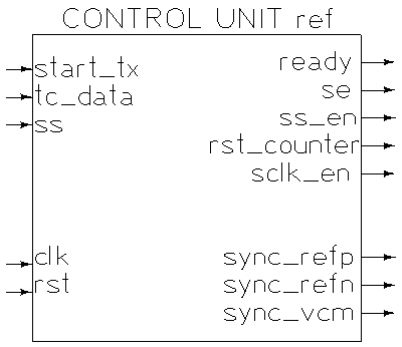


Figure 4.14: Control unit for reference DACs' control

Inputs

- start_tx. This signal has the same function of the one belonging to the data DAC's FSM. It triggers the beginning of a transmission. When "start_tx" goes high, it means that the data is valid, and a transmission can start.
- tc_data. Also this signal has the same function of the one belonging to the data DAC's FSM. It flags the end of the transmission.
- ss. This is an index computed by the 3-counter. As already said, it synchronizes the transmission over the three reference DACs. In other words, it causes a transition in the FSM, toward a transmission state related to one of the three DACs in which, the right configuration of the "sync" signals needed is computed.

Outputs

- ready. This is the second acknowledge signal which is interchanged with the user. When a transmission ends, counter and shift register are reset. After this time instant, the TX is considered free, and the signal "ready" flags this condition.
- ss_en. This control signal enables the ss-counter, which is used to synchronize and select the communication among the three reference DACs.
- sync_refp-refn-vcm. These signals are the frame synchronization signals for the DAC081S101's input data already discussed. They must be low during the transmission, and they must return high when it finishes. Each reference DAC has a proper sync signal.
- sclk_en. This is the enable for clock gating.

- se. It is the shift enable signal for the shift register.
- rst_counter. It is reset signal for the 16-counter.

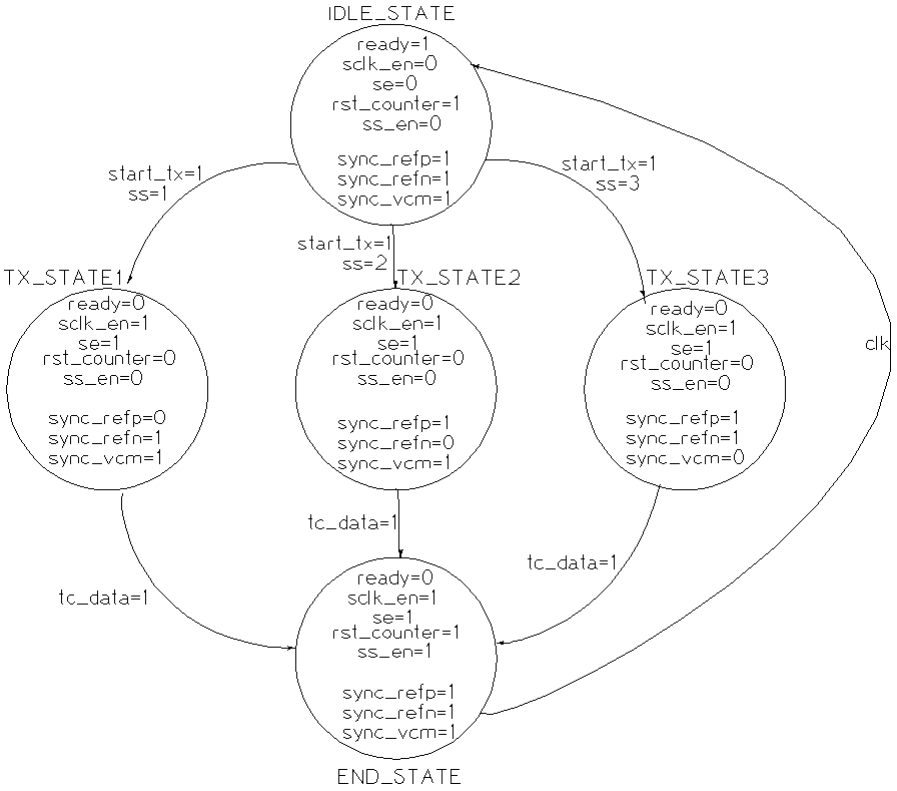


Figure 4.15: State diagram for reference DACs' control

States

- IDLE_STATE. In this state, TX is in a waiting state and it is free, so the acknowledge signal "ready" is asserted. Since TX is not transmitting, the "sync" signal for all the DACs must go '0', while the serial clock "sclk", which is shared by all the DACs, is disabled ("sclk_en='0'). When the input "start_tx" is forced to '1' by the user, there is a transition to the next state, which can be TX_state1, TX_state2 or TX_state3. Thus, the next state is selected by another input signal, which is the index "ss", and it can be 1, 2 or 3.
- TX_state_1-2-3.During these states the TX is busy because it is serializing and transmitting the 16-bit word provided by the user, so the signal "ready" goes low and the shift enable signal "se" goes high. Also, one of the three sync

signals (e.g. "sync_refp" for state TX_state1) and "slck_en" go respectively to '0' and '1', as required by the DAC's configuration for the serial write operation. When the 16-bit counter finishes its count, the transmission is completed and, through the input signal "tc_data" provided by the datapath, the FSM changes state.

- END_state. During this state the ss-counter is enabled, and the TX is not ready yet to start a new transmission. Transition through the IDLE_state happens with the next rising edge of the clock.

Finally, a higher level entity (figure 4.16) containing both datapath and FSM has been designed. Although there are three reference DAC on the PCB, it is enough to declare only one instance since the reference DACs share the clock and the data bus.

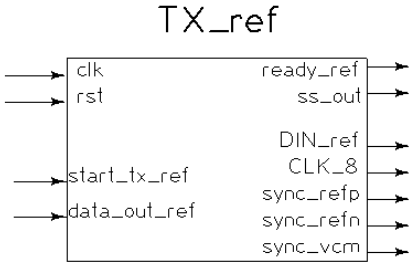
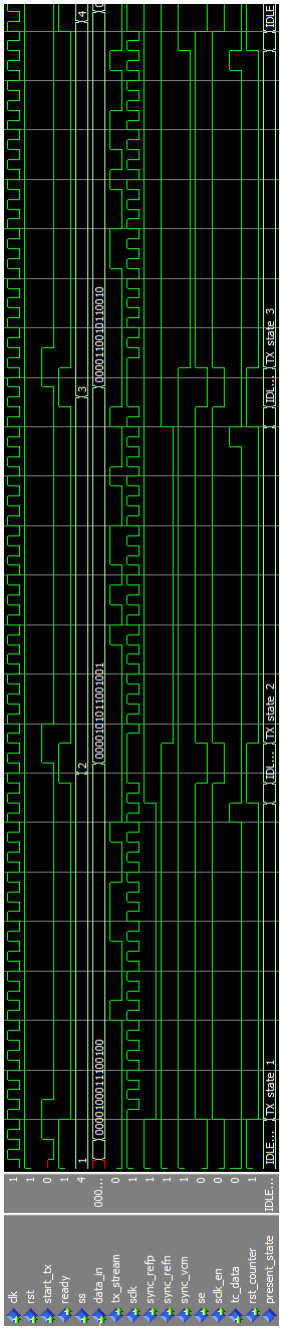


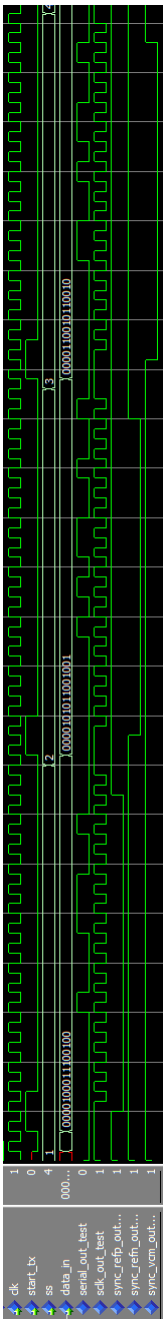
Figure 4.16: TX for reference DACs' control

4.2.3 Modelsim Simulations

Again, the hardware described in VHDL has been tested in Modelsim. Figure 4.17a shows the simulation results for 3 16-bit word ("data_in_r") transmission. The signal "tx_stream" is their serial representation.
In figure 4.17b instead, there is the same simulation but referring to the higher-level entity of figure 4.16.



(a) Lower level entity



(b) Higher level entity

Figure 4.17: Modelsim simulation. Transmission of 3 word for the reference DACs.

4.3 Chip control

The chip under test is essentially a SAR ADC. The starting point to describe in VHDL the controller, is to gather the information already provided in chapter 2 (referring to figures 2.13 and 2.14). As done for the AMS simulations, the internal structure of the ADC has to be clear to the designer. Below is a brief recap.

The memory inside the chip must be programmed by the user only once, before starting any operation. This memory will store information about how to acquire the input signal, controlling the number of capacitors used during every acquisition cycle. The capacitive array is used also to convert the signal. According to the memory layout, a 20-bit word must be sent by the user to program a single acquisition cycle for each channel.
In order to synchronize and perform all the operation, the chip needs to be controlled through different control signals which must be sent by the user. Therefore, several digital blocks must be implemented in VHDL and synthetized on the FPGA. To have an overview on the chip pin configuration, in figure 4.18 is shown the chip's symbol used in the KiCad project.

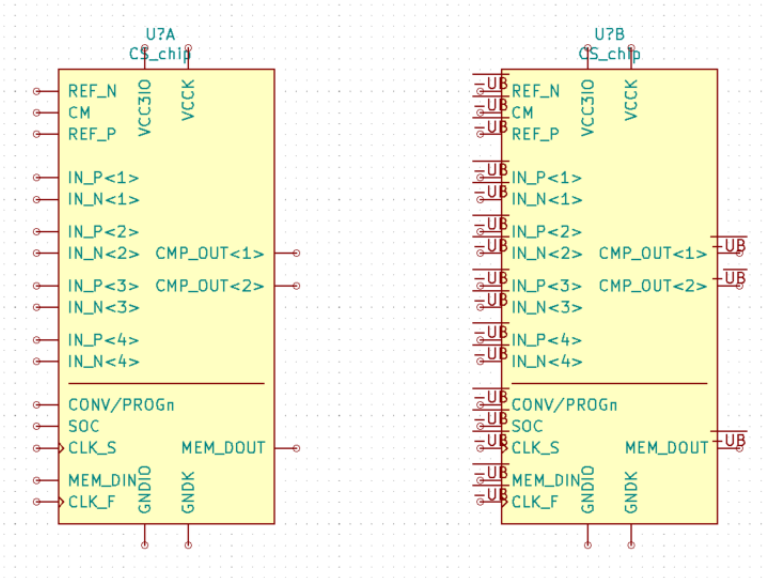


Figure 4.18: Chip's symbol (KiCad).

The analysis done in chapter 2 of the input/output pins of the chip and the timing diagram showed in figure 2.14 are still valid for this purpose.

4.3.1 Datapath

As already done for the designed datapaths related to DACs' control, a "paper and pencil" preliminary approach has been adopted. In figure 4.19, the datapath required to control the chip, containing all the digital blocks, is depicted.

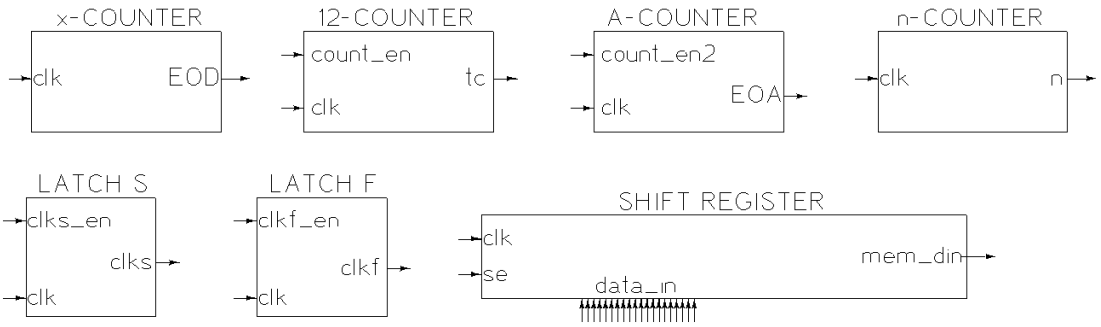


Figure 4.19: Datapath for chip's control

Below is a detailed description of each element and of its functions.

Shift Register It is needed to serialize data that must be sent to the memory. When the shift enable signal "se" goes high, the shift register begins the shifting, putting on the output "mem_din" the bits contained in the input bus "data_in".

n-counter Data sent by the user are grouped in words of 20 bits each. Thus, this is a 20-counter, which flags the end of transmission of an entire word. It enables the x-counter.

x-counter Every time a 20-bit word is transmitted to the memory, the x-counter is incremented. The "x" index can be any number, depending on the "quantity" of programming chosen by the user. When this counter reaches the end of its count, the flag signal "EOD" (End Of Data) is set to '1'. This signal flags the end of memory programming.

A-counter This counter is used to take into account of the number of acquisition cycles run. Thus, the index "A" can be any number depending on the number of cycles expected. When the acquisition is finished, the flag signal "EOA" (End Of Acquisition) is set to '1'.

12-counter Every conversion requires 12 clock cycles to be completely fulfilled. At the end of the counting, the flag signal "tc" (terminal count) is set to '1'.

Latch S and F The latches implement the clock gating. Essentially, through the enable input "clkf_en" (or "clks_en"), the output "clk_f" can be forced to '0', or it follows the global clock signal "clk".

4.3.2 FSM

The control unit has been designed, also in this case, with a FSM. In figure 4.20 there is a simple draw that shows a block with all the inputs and outputs. Below is a description of the input/output signals of this FSM, followed by an analysis of its states and of their functionality (figure 4.21).

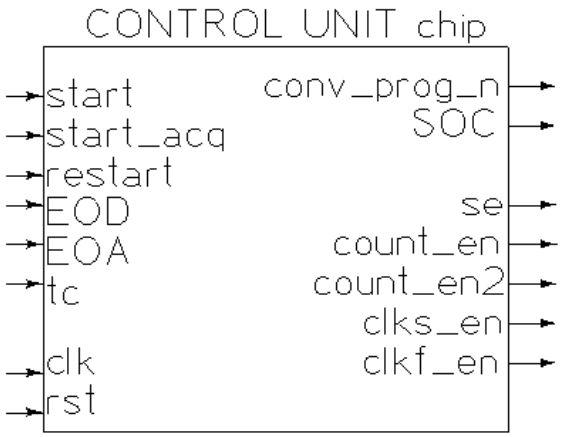


Figure 4.20: Control unit for chip's control

Inputs

- start. This signal is provided by the user and triggers the beginning of the memory programming.
- EOD. The End of Data signal is computed in the datapath and it flags the end of memory programming.
- start_acq. This signal is provided by the user and triggers the beginning of the acquisition.
- EOA. The End of Acquisition signal is produced by the datapath. When all the acquisition cycles are terminated, "EOA" goes high.
- tc. This signal flags the end of the conversion.

- restart. This signal is provided by the user, and it is used switch the FSM into a state that resets the address memory counter.

Outputs

- se. Shift enable of the shift register.
- clks_en (or f). Enable signals for clock gating.
- count_en (or 2). Counter enable for the 12-counter (or A-counter).
- SOC. This is the Start of Conversion signal needed by the chip. It must be set to '1' only during the conversion.
- conv_prog_n. This signal is used to control the chip when the operation run switches from memory programming to acquisition/conversion. In addition, the signal is used to reset the address memory counter.

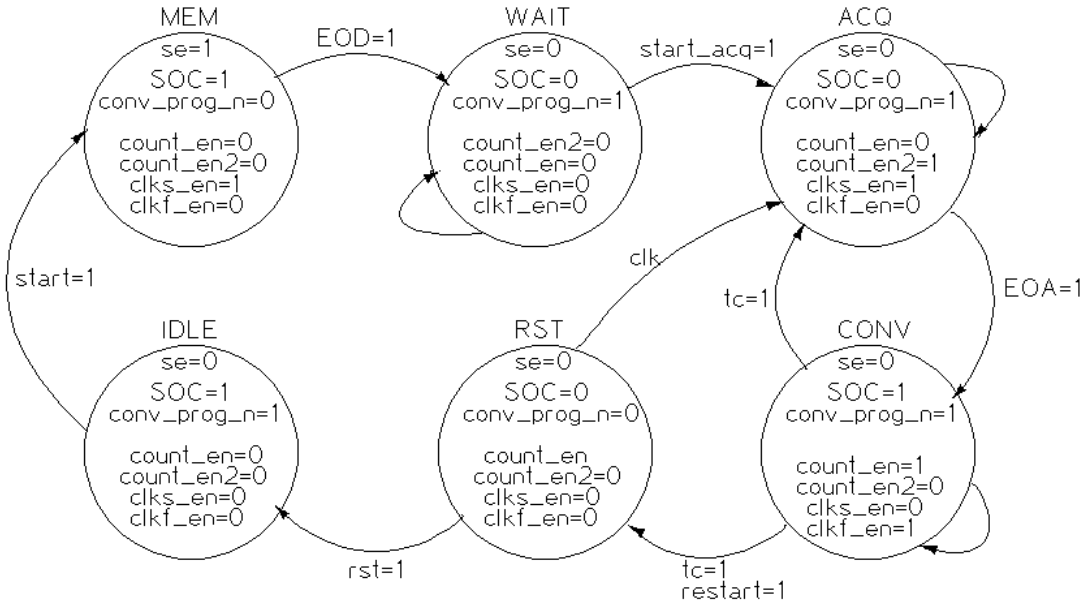


Figure 4.21: State diagram for chip's control

States

- IDLE. This is the initial state of the FSM during which all the system is waiting. When signal "start" goes high, there is a transition towards the MEM state, and the memory programming begins.

- MEM. During this state, the chip's memory is programmed. Indeed, the "clk_s" is enabled ("clks_en"='1'), signal "conv_prog_n" goes low, and the shift register is enabled ("se"='1'). When the memory programming is finished, the input signal "EOD" goes high and there is a transition towards the WAIT state.
- WAIT. Here, the system is waiting before starting the acquisition. Thus, the signal "conv_prog_n" is set to '1' and "clk_s" is disabled. When the user sets "start_acq" to '1', the acquisition begins.
- ACQ. For the acquisition operation "clk_s" is again enabled, and the A-counter is enabled. When it will reach the last counting step, "EOA" will go high and there will be a transition in the FSM.
- CONV. This state lasts 12 clock cycle (counted by the 12-counter now enabled). Signal "SOC" is set to '1' and "clk_f" is enabled. When the 12-counter reaches step 12, the signal "tc" goes high causing a transition. In this case, if the "restart" input is set to '1', next state will be RST, otherwise a new acquisition will start.
- RST. This state is used to produce a '0' pulse in the signal "conv_prog_n". This pulse will cause the reset of the address memory counter.

Finally, an higher level entity (figure 4.22) containing both datapath and FSM has been designed.

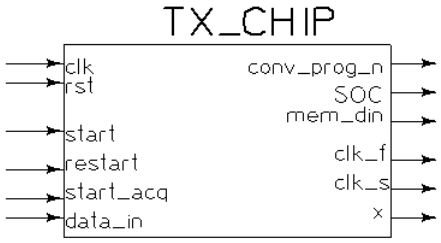


Figure 4.22: TX for chip's control

4.3.3 Modelsim Simulations

The hardware described in VHDL has been tested in Modelsim. Figure 4.23 shows the simulation results for the initial memory programming, followed by the acquisition/conversion alternation. In this simulation the acquisition lasts 30 clock cycles, and the memory address counter is reset anytime a new acquisition starts.

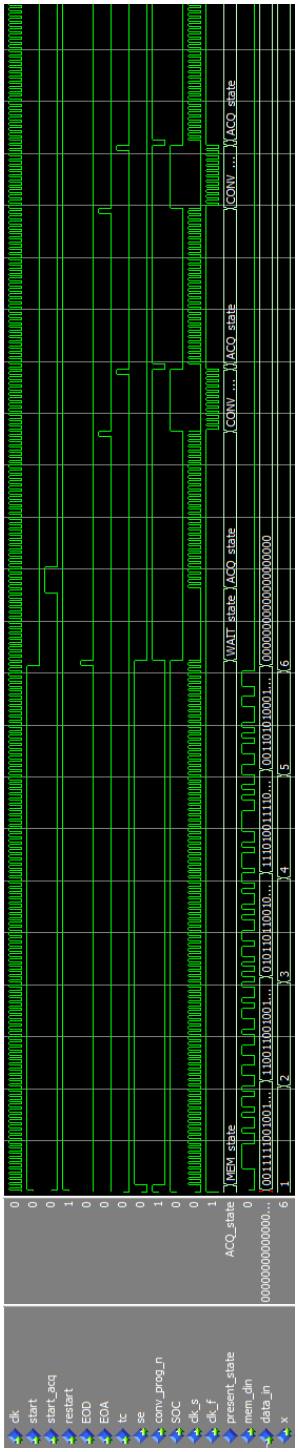


Figure 4.23: Modelsim simulation of the chip controller.

4.4 System Simulation

Having defined all the sub-blocks of the digital design, there is the need to design a top-level entity. This entity has inside all the instances of the sub-blocks already defined and it is responsible to provide them all the data required. Indeed, inside this entity, groups of registers that store data has been implemented. Furthermore, the top-level entity is the one the user directly interfaces with, and so it has to be simple to be used when it is accessed from the FPGA. In figure 4.24 there is the block of the top-level entity which has been designed.

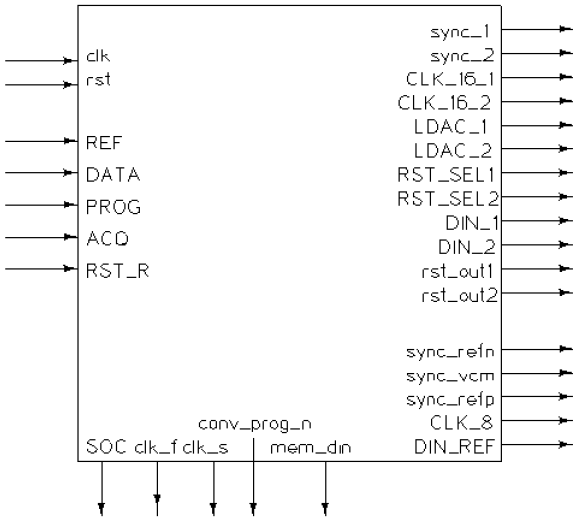


Figure 4.24: Top-level entity.

While the outputs of this block are the ones belonging to the sub-blocks already discussed, the inputs require a further explanation. They allow the external control from a user, of all the digital design, and of the chip itself. They are:

- REF. If "REF" is set to '1', the serial write operation for the reference DACs begins. Data are sequentially and continuously sent from the register file.

- DATA. If "DATA" is set to '1', the serial write operation for the high-resolution DACs begins. Data are sequentially and continuously sent from the register file.

- PROG. If "PROG" is set to '1' the memory programming begins. This operation automatically stops when all the data has been sent.

- ACQ. When "ACQ" is set to '1', the acquisition begins. Because of how the chip and the controller FSM were designed, the conversion automatically begins after every acquisition, and the cycle is continuously repeated.
- RST_R. To reset the address memory counter, "RST_R" must be set to '1' at the end of the conversion.

In figure 4.25 there is the Modelsim simulation of this top-level entity. These waveforms describe the entire sequence of data and control signals to be sent to the chip and to the PCB, by means of the FPGA. The first two operations are the memory programming and the reference DACs programming. Since the data to be sent to the reference DACs are the digital representation of constant voltages, it is enough to transmit 3 16-bit words, one for each DAC: the latter will hold its analog output voltage during time.
The third step is the high-resolution DACs programming: sets of 24-bit words are continuously transmitted, to constantly provide the analog input signal for the ADC inside the chip.
Finally, the acquisition/conversion alternation can start, and the ADC will begin the conversion.
The timing of these operations can be managed through the input signal of the top-level entity, which are accessible by the external user.

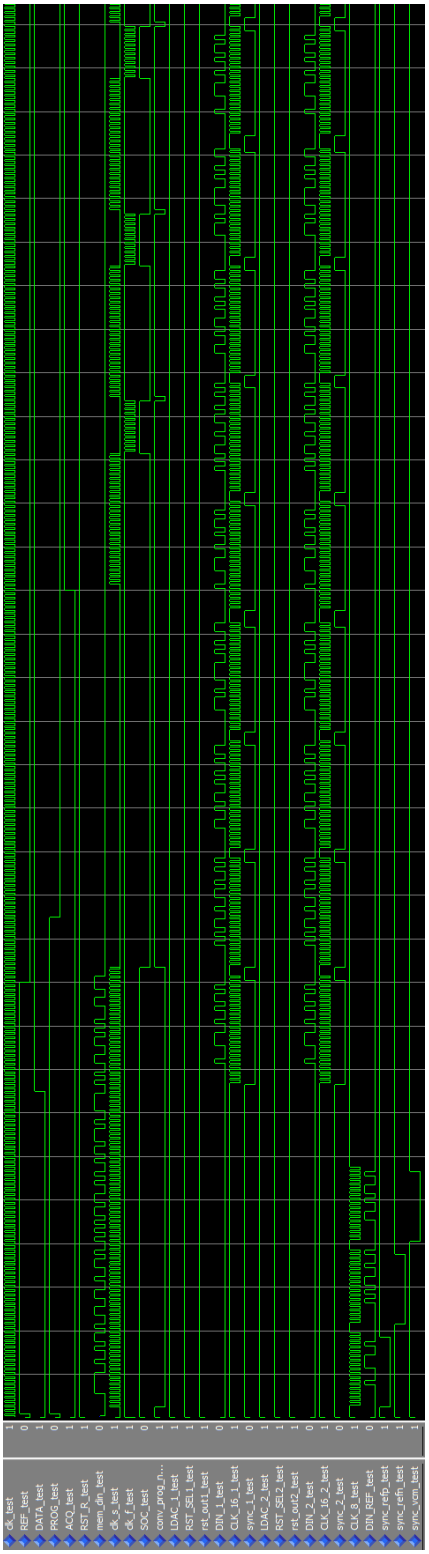


Figure 4.25: Modelsim simulation of the system.

4.5 Test board measurements

All the VHDL code has been synthetized exploiting the software Quartus on the FPGA DE0-CV by Terasic, based on the Cyclone V.
Figure 4.26 shows a photo of the PCB connected to the FPGA. With this configuration, preliminary tests of the synthetized VHDL and of the DACs present on the board have been carried out, without placing the chip.

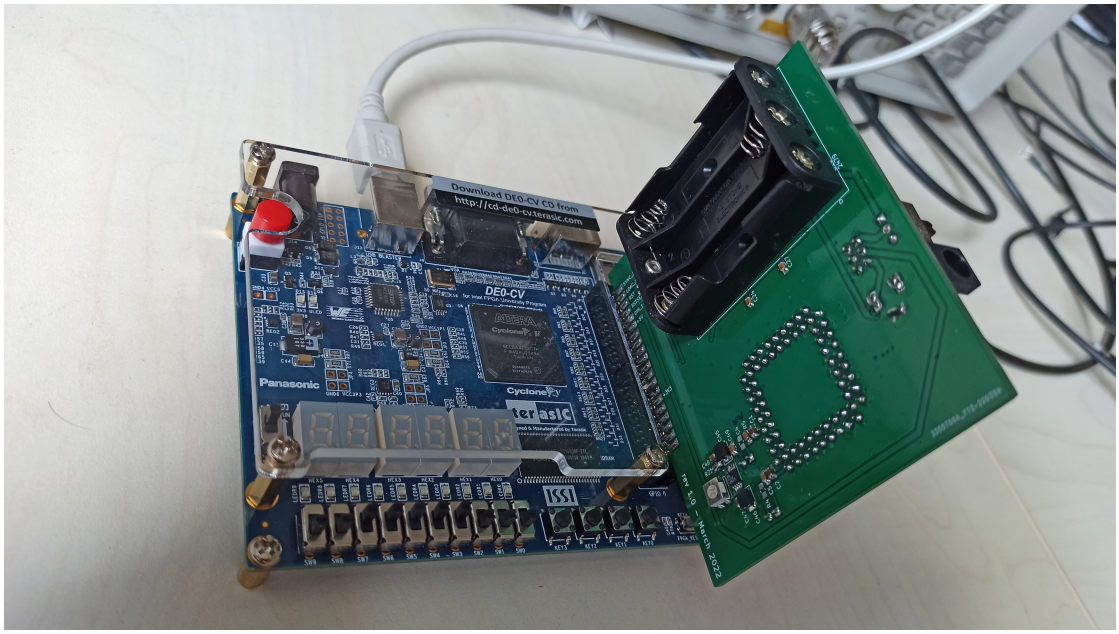


Figure 4.26: Measurement configuration (FPGA + PCB).

The FPGA has been connected to the PC through an USB cable, while the PCB has been supplied through the DC jack. The FPGA has been programmed to generate a digital sinusoidal waveform with 8 bits of resolution, which is less then their the maximum resolution: this waveform has been exploited to verify the D/A conversion performed by the high-resolution DACs on the PCB. However, this resolution is enough to verify the correct functioning of the system. The signals observed have been probed on 3 pins of the FPGA and on 2 outputs of one high-resolution DAC. They are:

FPGA pins

- CLK_16. Serial clock for the DAC.
- SYNC_1. Synchronization signal for communication.
- DIN_1. Digital data sent.

DAC outputs

- VIN1_P.
- VIN2_P.

These signals have been measured and visualized with an oscilloscope. Figure 4.27 shows the SYNC_1 (purple) and DIN_1 (yellow) signals. As can be seen, data are transmitted when the synch signal is low.

Figure 4.27: Oscilloscope capture. SYNC_1 (purple) and DIN_1 (yellow) signals.

Figure 4.28 shows the CLK_16 (yellow) and VIN1_P signals (purple). It can be seen that every 24 clock cycles, a new analog level is produced by the DAC. In figure 4.29 there is a zoomed version, where clock edges can be seen clearly.

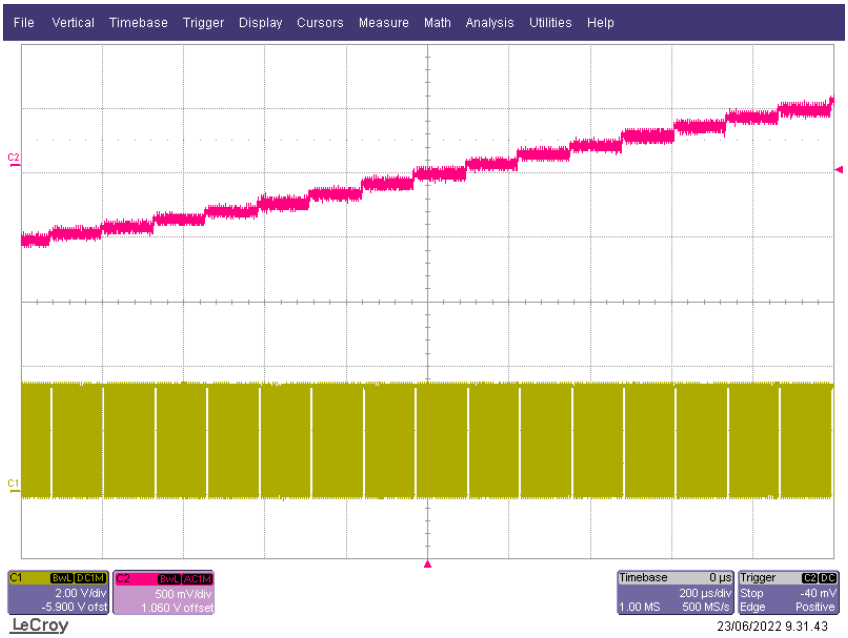


Figure 4.28: Oscilloscope capture. CLK_16 (yellow) and VIN1_P (purple) signals.

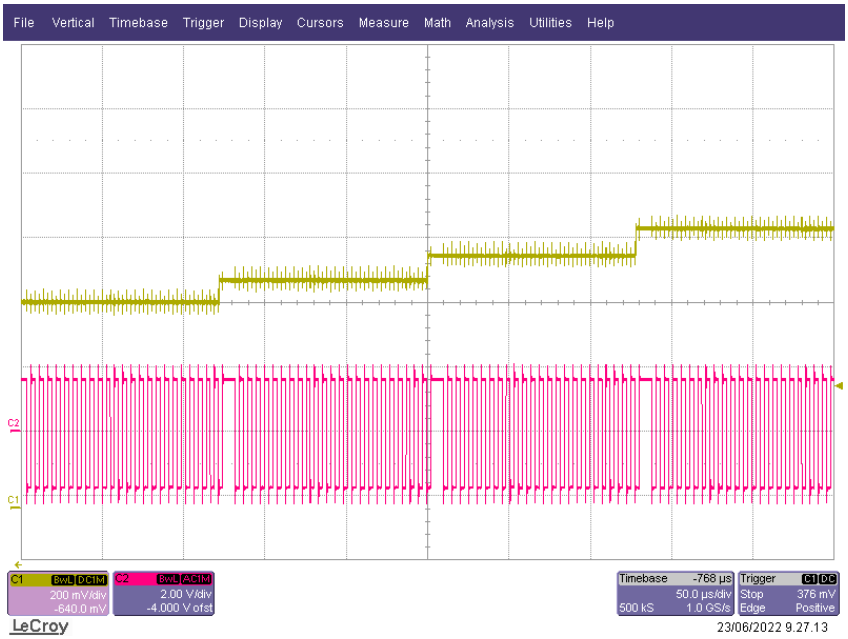


Figure 4.29: Oscilloscope capture. Zoom of CLK_16 (purple) and VIN1_P (yellow) signals.

Finally, in figure 4.30 there are the two output signals of the DAC, which are clearly sinusoidal waveforms.

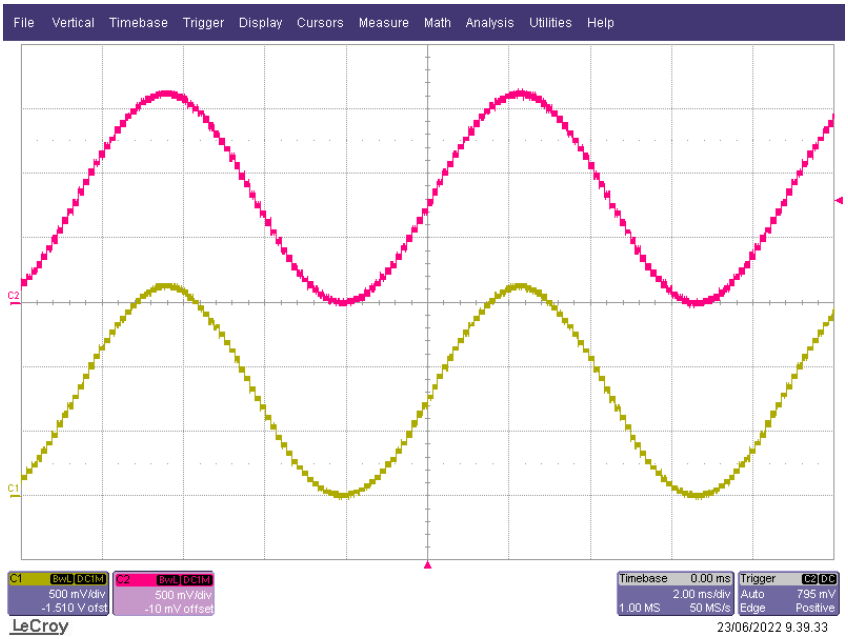


Figure 4.30: Oscilloscope capture. VIN1_P and VIN2_P output signals.

Unfortunately, due to delays in the chip production, it has not been possible to perform and conclude the complete characterization of the chip.

Chapter 5

Conclusions

This thesis has been focused on the test and characterization of an integrated circuit which implements a particular structure of acquisition system. First, an initial overview on the traditional SAR ADC topologies, has been provided, followed by a brief discussion of the theory behind the idea of the compressed sensing technique. Then, the SAR ADC included in the chip to be tested has been presented. After a preliminary detailed description of the input and output signals of the device, and of its internal logic structure, a testbench has been designed to carry out a characterization through circuital AMS simulations in the Cadence Virtuoso environment. Results have been elaborated in Matlab and different performance metrics have been obtained.
Then, an ad-hoc PCB has been designed to test the physical chip. After a rigorous selection of the components needed, based on given constraints, the testing board has been designed in KiCad, and then produced.
Through the produced VHDL code, it was possible to program a FPGA, which controls the components on the PCB and the chip itself, as well as provide the testing input signal. The correct functioning of the system composed by FPGA and PCB has been verified by means of measurements with oscilloscope.
Simulations results show that the converter has quantization error which leads to an actual resolution of 9 bit, against the aimed 12 bit. Then, non-idealities of the circuit result to be crucial in the determination of INL and DNL errors. They are respectively 4.91 LSB and 3 LSB. The impact of these errors should be estimated evaluating their effect on the final application of the chip, which is the compressed sensing operation.
Finally, the silicon device should have been tested and characterized through the designed testing platform, but unfortunately it was not possible due to delays in the chip production.
Future works would involve further simulations of the circuit, in order to test all the possible types of operations offered by this kind of ADC, and measurements

of the actual chip: its characterization could be easily completed exploiting the designed testing platform.

Appendix A

VHDL code

A.1 Data DACs control code

datapath.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity datapath is
7      generic(
8          L: integer := 4; -- numero di words da inviare ai DAC dati
9          N : integer := 24); -- numero di bit per word
10     port (
11         clk          : in  std_logic;
12         rst          : in  std_logic;   -- reset asincrono
13         ctrl_bit     : in std_logic_vector(7 downto 0); --8 bit
       contenenti informazione sul controllo dei DAC
14         data_bit     : in std_logic_vector(15 downto 0);--16 bit
       contenenti il dato vero e proprio.
15         tx_stream    : out std_logic; --uscita seriale
16         se           : in  std_logic; --shift enable
17         tc_data      : out std_logic; --terminal count
18         sclk         : out std_logic; --serial clock DAC
19         sclk_en      : in  std_logic; --sclk enable
20         rst_counter  : in  std_logic; --reset contatore
21         i            : out integer range 1 to L+1; -- indice contatore
        dei dati inviati ai DAC dati.
22         i_en         : in std_logic
23     );
24 end entity;
25
26
```

```
27 architecture rtl of datapath is
28
29 signal d : integer range 0 to N+1;
30 signal data_in_r    : std_logic_vector(N-1 downto 0);
31 signal i_int        : integer range 1 to L+1;
32
33
34 begin
35
36 tc_data <= '0' when(d>1) else '1'; --se il conteggio è terminato, tc
   =1
37
38 shifting : process(clk,se,data_in_r)
39 begin
40 data_in_r <= ctrl_bit & data_bit;
41         if (se ='0') then
42             tx_stream <= '0' ;
43         else -- se='1'
44
45             if ( rising_edge(clk) ) then
46                 tx_stream <= data_in_r(d-1);
47             end if;
48
49         data_in_r <= data_in_r;
50         end if;
51 end process;
52
53
54 conteggio : process(clk,rst_counter) --conteggio dei bit inviati
55 begin
56     if (rst_counter = '1' ) then
57         d <= N;
58     elsif (clk' EVENT AND clk= '1' ) then
59         if(d>1) then
60             d <= d - 1;
61         end if;
62     end if ;
63 end process;
64
65
66 data_count: process(clk,rst) --conteggio delle words inviate ai dac
67 begin
68     if (rst='0') then
69     i_int               <= 1;
70      elsif(i_en='1') and (clk' EVENT  AND clk= '1' ) then
71          i_int          <= i_int+1;
72     end if;
73     if (i_int=L+1) then
74     i_int <=1;
```

```
75      end if;
76      i<=i_int;
77  end process;
78
79
80  clk_gating : process(clk,rst) --latch per clock gating
81  begin
82      if(rst='0') then
83          sclk <= '1';
84      elsif(sclk_en='1') then
85          sclk <= clk;
86      else sclk<='1';
87      end if;
88  end process;
89
90  end rtl;
```

control_unit.vhd

```
1   library ieee;
2   use ieee.std_logic_1164.all;
3   use ieee.numeric_std.all;
4
5
6   entity control_unit is
7       generic(
8           N : integer := 24); -- numero di bit per word.
9       port (
10          clk          : in  std_logic;
11          rst          : in  std_logic; -- reset asincrono
12          start_tx     : in  std_logic; -- segnale di inizio
        trasmissione
13          tc_data      : in  std_logic; -- terminal count word
14          ready        : out std_logic; -- segnala trasmissione
        completata, TX disponibile
15          sync         : out std_logic; -- segnale di abilitazione per
        il DAC
16          rst_out      : out std_logic; -- reset del DAC
17          LDAC         : out std_logic; -- segnale per il DAC
18          RST_SEL      : out std_logic; -- segnale per il DAC
19          rst_counter  : out std_logic; -- reset contatore bit
20          sclk_en      : out std_logic; -- enable sclk
21          i_en         : out std_logic; -- enable contatore words
        inviate
22          se           : out std_logic -- shift enable (shift register)
23      );
24  end entity;
25
26  architecture rtl of control_unit is
```

```
27      type state is (
28              TX_state ,
29              IDLE_state ,
30              END_state );
31      signal present_state : state:=IDLE_state;
32      signal next_state    : state;
33      signal start_tx_in   : std_logic;
34
35
36  begin
37
38
39  state_register : process(clk,rst)
40      begin
41          if(rst='0') then
42          present_state <=IDLE_state;
43          elsif(rising_edge(clk)) then
44              present_state <= next_state;
45          end if;
46  end process state_register;
47
48
49  state_updating : process(present_state ,tc_data ,start_tx)
50
51  begin
52      case present_state is
53
54          when TX_state =>
55              if (tc_data='1') then
56                  next_state      <= END_state;
57              else next_state <= TX_state;
58              end if;
59
60          when END_state =>
61              next_state <= IDLE_state;
62
63          when  others =>
64              if(start_tx='1') then
65                  next_state      <= TX_state ;
66              else next_state <= IDLE_state ;
67              end if;
68
69      end case;
70  end process state_updating;
71
72
73  state_register_out : process(present_state)
74  begin
75      case present_state is
```

```
76          when TX_state =>
77                  sync        <= '0';
78                  ready       <= '0';
79                  sclk_en     <= '1';
80                  rst_out     <= '1';
81                  rst_counter <= '0';
82                  se          <= '1';
83                  RST_SEL     <= '1';
84                  LDAC        <= '1';
85                  i_en        <= '0';
86
87          when END_state =>
88                  ready       <= '0';
89                  sclk_en     <= '1';
90                  sync        <= '1';
91                  rst_out     <= '1';
92                  se          <= '1';
93                  rst_counter <= '1';
94                  RST_SEL     <= '1';
95                  LDAC        <= '1';
96                  i_en        <= '1';
97
98          when others => --IDLE_state
99                  rst_out     <= '1';
100                 ready       <= '1';
101                 sync        <= '1';
102                 sclk_en     <= '0';
103                 se          <= '0';
104                 rst_counter <= '1';
105                 RST_SEL     <= '1';
106                 LDAC        <= '1';
107                 i_en        <= '0';
108
109     end case;
110
111 end process state_register_out;
112
113 end rtl;
```

TX.vhd

```
1 library IEEE;
2 USE ieee.std_logic_1164.all;
3
4 entity TX is
5
6 generic(
7   N                 : integer := 24;
8   L                 : integer := 4);
```

```
port(clk_0          : in std_logic;
rst_0               : in std_logic;
start_0             : in std_logic;
ctrl_bit            : in std_logic_vector(7 downto 0);
data_bit            : in std_logic_vector(15 downto 0);
i_out               : out integer range 1 to L+1;
serial_out          : out std_logic;
sync_out            : out std_logic;
sclk_out            : out std_logic;
LDAC                : out std_logic;
RST_SEL             : out std_logic;
rst_out             : out std_logic;
ready_0             : out std_logic);
end entity;

architecture rtl of TX is


signal rst_counter              :  std_logic;
signal se                       :  std_logic;
signal tc_data                  :  std_logic;
signal tc                       :  std_logic;
signal sclk_en                  :  std_logic;
signal i_en                     :  std_logic;


component datapath is

 port (
  clk                       : in  std_logic;
  rst                       : in  std_logic;
  sclk_en                   : in  std_logic;
  rst_counter               : in  std_logic;
  ctrl_bit                  : in std_logic_vector(7 downto 0);
  data_bit                  : in std_logic_vector(15 downto 0);
  sclk                      : out std_logic;
  se                        : in std_logic;
  tc_data                   : out std_logic;
  i                         : out integer range 1 to L+1;
  i_en                      : in std_logic;
  tx_stream                 : out std_logic);
end component;


component control_unit is
port(
  clk                       : in  std_logic;
  rst                       : in  std_logic;
  tc_data                   : in std_logic;
```

```
58    ready                       : out std_logic;
59    start_tx                    : in std_logic;
60    sclk_en                     : out  std_logic;
61    rst_out                     : out std_logic;
62    LDAC                        : out std_logic;
63    RST_SEL                     : out std_logic;
64    sync                        : out std_logic;
65    rst_counter                 : out  std_logic;
66    i_en                        : out std_logic;
67    se                          : out std_logic);
68  end component;
69
70  begin
71
72  data: datapath port map (
73          clk          =>clk_0      ,
74          rst          =>rst_0      ,
75          ctrl_bit     =>ctrl_bit   ,
76          data_bit     =>data_bit   ,
77          sclk         =>sclk_out   ,
78          tx_stream    =>serial_out ,
79          se           =>se         ,
80          tc_data      =>tc_data    ,
81          sclk_en      =>sclk_en    ,
82          i_en         =>i_en       ,
83          i            =>i_out      ,
84          rst_counter =>rst_counter  );
85
86
87  ctrl: control_unit port map (
88          clk          =>clk_0      ,
89          rst          =>rst_0      ,
90          start_tx     => start_0   ,
91          tc_data      => tc_data   ,
92          ready        =>ready_0    ,
93          sync         =>sync_out   ,
94          rst_out      =>rst_out    ,
95          rst_counter =>rst_counter    ,
96          sclk_en      =>sclk_en        ,
97          i_en         =>i_en           ,
98          se           =>se             ,
99          LDAC         =>LDAC           ,
100         RST_SEL      =>RST_SEL
101         );
102
103 end rtl;
```

A.2 Reference DACs control code

datapath_ref.vhd

```
1
2  library ieee;
3  use ieee.std_logic_1164.all;
4  use ieee.numeric_std.all;
5
6
7  entity datapath_ref is
8  generic(
9    LL       : integer :=3;     --numero di words da inviare. .
10   NN       : integer := 16); --numero di bit per word.
11  port (
12    clk                    : in  std_logic;
13    rst                    : in  std_logic; -- reset asincrono
14    data_in                : in  std_logic_vector(NN-1 downto 0);  --
        dato da inviare
15    tx_stream              : out std_logic; --uscita seriale
16    se                     : in std_logic;
17    tc_data                : out std_logic;
18    sclk                   : out std_logic;
19    ss                     : out integer range 1 to LL+1;
20    sclk_en                : in std_logic; --slave select
21    rst_counter            : in std_logic;
22    ss_en                  : in std_logic --slave counter enable
23
24 );
25 end entity;
26
27
28 architecture rtl of datapath_ref is
29
30 signal d                      : integer range 0 to NN+1;
31 signal data_in_r              : std_logic_vector(NN-1 downto 0);
32 signal ss_int                 : integer range 1 to LL+1;
33
34 begin
35
36 tc_data  <= '0' when(d>1) else '1'; --se il conteggio è terminato, tc
       =1
37
38
39 shifting: process(clk,se ,data_in)
40
41 begin
42     if (se ='0') then
43         tx_stream    <='0' ;
```

```
44      else
45          if ( rising_edge(clk) ) then
46              tx_stream       <= data_in(d-1);
47          end if;
48      end if;
49  end process;
50
51
52  conteggio: process(clk,rst_counter)--conteggio dei bit inviati
53  begin
54
55      if (rst_counter = '1' ) then
56          d   <= NN;
57      elsif (clk' EVENT  AND clk= '1' ) then
58              if(d>1) then
59              d<= d - 1;
60              end if;
61      end if ;
62  end process;
63
64
65  ss_count: process(clk,rst,ss_en) --conteggio per sincronizzare la
        comunicazione sui 3 DAC
66  begin
67      if (rst='0') then
68      ss_int              <= 1;
69       elsif(ss_en='1') and (clk' EVENT  AND clk= '1' ) then
70           ss_int         <= ss_int+1;
71      end if;
72      -- if (ss_int=LL+1) then
73      -- ss_int <=1;
74      -- end if; per l'auto-reset
75      ss<=ss_int;
76  end process;
77
78
79  clk_gating : process(clk,rst) --latch per clock gating
80  begin
81    if(rst='0') then
82      sclk<='1';
83    elsif(sclk_en='1') then
84        sclk <= clk;
85        else sclk<='1';
86      end if;
87  end process;
88
89  end rtl;
```

control_unit_ref.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity control_unit_ref is
7  generic(
8    LL                    : integer := 3); -- numero di simboli per
       ogni dato
9   port (
10   clk                   : in  std_logic;
11   rst                   : in  std_logic; -- reset asincrono
12   start_tx              : in  std_logic; -- segnale di inizio
       trasmissione
13   ready                 : out std_logic; -- trasmissione completata,
       TX disponibile
14   sync_refp             : out std_logic; -- segnale per il DAC refp
15   sync_refn             : out std_logic; -- segnale per il DAC refn
16   sync_vcm              : out std_logic; -- segnale per il DAC vcm
17   tc_data               : in  std_logic; -- terminal count bit
18   ss                    : in  integer range 1 to LL+1; -- slave
       select per la sincronizzazione sui 3 DAC
19   rst_counter           : out std_logic; -- reset contatore
20   sclk_en               : out std_logic; -- enable sclk
21   se                    : out std_logic; -- shift enable (shift
       register)
22   ss_en                 : out std_logic  -- enable contatore slave
23  );
24
25 end entity;
26
27 architecture rtl of control_unit_ref is
28 type state is (
29                           TX_state_1 ,
30                           TX_state_2 ,
31                           TX_state_3 ,
32                           IDLE_state ,
33                           END_state  );
34
35 signal present_state        : state:=IDLE_state;
36 signal next_state           : state;
37 signal start_tx_in          : std_logic;
38
39 begin
40
41
42 state_register : process(clk,rst)
43 begin
```

```
44    if(rst='0') then
45      present_state            <= IDLE_state;
46    elsif(rising_edge(clk)) then
47      present_state            <= next_state;
48    end if;
49  end process state_register;
50
51
52  state_updating : process(
53                   present_state ,
54                   tc_data       ,
55                   start_tx      ,
56                   ss            )
57
58  begin
59   case present_state is
60
61      when  TX_state_1      =>
62        if  (tc_data='1')  then
63              next_state  <= END_state;
64        else  next_state  <= TX_state_1;
65        end if;
66
67
68      when  TX_state_2      =>
69        if  (tc_data='1')  then
70              next_state  <= END_state;
71        else  next_state  <= TX_state_2;
72        end if;
73
74
75      when  TX_state_3      =>
76        if  (tc_data='1')  then
77              next_state  <= END_state;
78        else  next_state  <= TX_state_3;
79        end if;
80
81      when  END_state       =>
82          next_state  <= IDLE_state    ;
83
84
85      when others           =>  --IDLE_state
86
87        if   (start_tx='1') and (ss=1) then
88              next_state  <= TX_state_1 ;
89
90        elsif(start_tx='1') and (ss=2) then
91              next_state  <= TX_state_2 ;
92
```

```
93        elsif(start_tx='1') and (ss=3) then
94              next_state  <= TX_state_3 ;
95
96        else    next_state  <= IDLE_state ;
97        end if;
98
99
100
101
102
103  end case;
104 end process state_updating;
105
106
107 state_register_out : process(present_state)
108 begin
109
110     case present_state is
111       when TX_state_1  => --riceve il DAC refp
112         sync_refp       <= '0';
113         sync_refn       <= '1';
114         sync_vcm        <= '1';
115         ready           <= '0';
116         ss_en           <= '0';
117         rst_counter     <= '0';
118         sclk_en         <= '1';
119         se              <= '1';
120
121
122       when TX_state_2  => --riceve il DAC refn
123         sync_refp       <= '1';
124         sync_refn       <= '0';
125         sync_vcm        <= '1';
126         ready           <= '0';
127         ss_en           <= '0';
128         rst_counter     <= '0';
129         sclk_en         <= '1';
130         se              <= '1';
131
132
133       when TX_state_3  => --riceve il DAC vcm
134         sync_refp       <= '1';
135         sync_refn       <= '1';
136         sync_vcm        <= '0';
137         ready           <= '0';
138         ss_en           <= '0';
139         rst_counter     <= '0';
140         sclk_en         <= '1';
141         se              <= '1';
```

```
142
143
144         when END_state      =>
145           ready             <= '0';
146           ss_en             <= '1';
147           sync_refp         <= '1';
148           sync_refn         <= '1';
149           sync_vcm          <= '1';
150           rst_counter       <= '1';
151           sclk_en           <= '1';
152           se                <= '1';
153
154         when others => --IDLE_state
155           ready             <= '1';
156           ss_en             <= '0';
157           sync_refp         <= '1';
158           sync_refn         <= '1';
159           sync_vcm          <= '1';
160           sclk_en           <= '0';
161           se                <= '0';
162           rst_counter       <= '1';
163
164       end case;
165
166 end process state_register_out;
167
168 end rtl;
```

TX_ref.vhd

```
1  library IEEE;
2  USE ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  entity TX_ref is
5
6  generic(
7    NN                  : integer := 16;
8    LL                  : integer := 3);
9
10 port(    clk_0                              : in std_logic;
11          rst_0                              : in std_logic;
12          start_0                            : in std_logic;
13          data_out                           : in std_logic_vector(NN-1
       downto 0);
14          ss_out                             : out integer range 1 to LL
       +1;
15          sync_refp_out                      : out std_logic;
16          sync_refn_out                      : out std_logic;
17          sync_vcm_out                       : out std_logic;
```

```
18          ready_0                         : out std_logic;
19          sclk_out                        : out std_logic;
20          serial_out                      : out std_logic
21          );
22  end entity;
23
24
25  architecture rtl of TX_ref is
26
27  signal rst_counter              :  std_logic;
28  signal se                       :  std_logic;
29  signal tc_data                  :  std_logic;
30  signal ss                       :  integer range 1 to LL+1;
31  signal tc                       :  std_logic;
32  signal sclk_en                  :  std_logic;
33  signal ss_en                    :  std_logic;
34
35
36  component datapath_ref is
37
38   port (
39    clk                         : in  std_logic;
40    rst                         : in  std_logic;
41    sclk_en                     : in  std_logic;
42    rst_counter                 : in  std_logic;
43    data_in                     : in  std_logic_vector(NN-1 downto 0);
44    sclk                        : out std_logic;
45    se                          : in std_logic;
46    ss                          : out integer range 1 to LL+1;
47    ss_en                       : in std_logic;
48    tc_data                     : out std_logic;
49    tx_stream                   : out std_logic);
50  end component;
51
52
53  component control_unit_ref is
54  port(
55    clk                         : in  std_logic;
56    rst                         : in  std_logic;
57    tc_data                     : in std_logic;
58    ready                       : out std_logic;
59    start_tx                    : in std_logic;
60    sclk_en                     : out  std_logic;
61    rst_counter                 : out  std_logic;
62    sync_vcm                    : out  std_logic;
63    sync_refp                   : out std_logic;
64    sync_refn                   : out std_logic;
65    ss                          : in  integer range 1 to LL+1;
66    ss_en                       : out std_logic;
```

```
67    se                          : out std_logic);
68  end component;
69
70
71  begin
72
73  ss_out<=ss;
74
75  data: datapath_ref port map (
76          clk          =>clk_0      ,
77          rst          =>rst_0      ,
78          data_in      =>data_out   ,
79          sclk         =>sclk_out   ,
80          tx_stream    =>serial_out ,
81          se           =>se         ,
82          tc_data      =>tc_data    ,
83          sclk_en      =>sclk_en    ,
84          ss           =>ss         ,
85          ss_en        =>ss_en      ,
86          rst_counter =>rst_counter    );
87
88
89  ctrl: control_unit_ref port map (
90          clk          =>clk_0      ,
91          rst          =>rst_0      ,
92          start_tx     => start_0   ,
93          tc_data      => tc_data   ,
94          ready        =>ready_0    ,
95          sync_refp    =>sync_refp_out     ,
96          sync_refn    =>sync_refn_out     ,
97          sync_vcm     =>sync_vcm_out   ,
98          ss           =>ss             ,
99          ss_en        =>ss_en          ,
100         rst_counter =>rst_counter    ,
101         sclk_en      =>sclk_en        ,
102         se           =>se             );
103
104 end rtl;
```

A.3 Chip control code

datapath_chip.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
```

```
5
6  entity datapath_chip is
7      generic(
8          A: integer:=30; --Numero di cicli durata acquisizione.
9          Z : integer:=5; -- Numero di words
10         M : integer := 20); --numero di bit per word da inviare alla
       memoria
11     port (
12         clk       : in  std_logic;
13         rst       : in  std_logic; -- reset asincrono
14         clk_s_en  : in  std_logic; -- enable di clk_s
15         clk_f_en  : in  std_logic; -- enable di clk_f
16         count_en  : in  std_logic; -- count enable per i 12 cicli
       necessari alla conversione
17         count_en2: in  std_logic; -- count enable per la durata dell'
       acquisizione
18         se        : in  std_logic; -- shift enable dati per la
       programmazione della memoria
19         data_in   : in  std_logic_vector(0 to M-1);
20         clk_s     : out std_logic;
21         EOD       : out std_logic := '0'; --End Of Data: fine
       trasmissione dati per la memoria
22         EOA       : out std_logic := '0'; --End Of Acquisition
23         clk_f     : out std_logic;
24         mem_din   : out std_logic := 'Z';
25         tc        : out std_logic;
26         x         : out integer:=1
27     );
28 end entity;
29
30
31 architecture rtl of datapath_chip is
32
33     signal bit_count         : integer range 1 to 12     := 1;
34     signal bit_count2        : integer range 1 to A      := 1;
35     signal n                 : integer range 0 to M      := 0;
36     signal x_int             : integer :=1;
37
38 begin
39
40 tc <= '0' when(bit_count<12) else '1'; --se il conteggio è terminato,
       tc=1
41 EOA <= '0' when(bit_count2<A) else '1'; --se il conteggio è terminato
       , EOA=1
42
43
44 shifting : process(clk) -- shift dati per programmazione memoria.
45
46 begin
```

```
47
48 if ( rising_edge(clk) ) then
49
50         if (se ='0') then
51             EOD      <= '0';
52             mem_din <= '0';
53         else --se='1';
54             mem_din<= data_in(n);
55             if (n<M-1) then --fino a 18
56                 n <= n+1;
57                 x_int<=x_int;
58                 x<=x_int;
59                 EOD<='0';
60
61                 if (n=M-3) then --n=17
62                 x_int<=x_int+1;
63                 x<=x_int;
64                 end if;
65
66             else --n=19
67                 if (x_int=Z+1)  then
68                     EOD<='1';
69                 else EOD<='0';
70                 end if;
71
72             n<= 0;
73
74             end if;
75         end if;
76 end if;
77 end process;
78
79 conteggio : process(clk) --conteggio dei 12 cicli di conversione
80 begin
81
82     if (clk' EVENT AND clk= '1' ) then
83         if (bit_count = 12 ) then
84             bit_count <= 1;
85         elsif count_en='1' then
86             bit_count <= bit_count + 1;
87         end if;
88     end if ;
89 end process;
90
91
92 conteggio2 : process(clk) --conteggio degli A-cicli di acquisizione
93 begin
94
95     if (clk' EVENT AND clk= '1' ) then
```

```
96          if (bit_count2 = A ) then
97              bit_count2 <= 1;
98          elsif count_en2='1' then
99              bit_count2 <= bit_count2 + 1;
100         end if;
101     end if ;
102 end process;
103 
104 
105 clk_s_gating : process(clk,clk_s_en) --latch per clock gating di
    clk_s
106 begin
107 
108     if(clk_s_en='0') then
109         clk_s <= '0';
110     else
111         clk_s <= clk;
112     end if;
113 end process;
114 
115 
116 clk_f_gating : process(clk,clk_f_en) --latch per clock gating di
    clk_f
117 begin
118     if(clk_f_en='0') then
119         clk_f <= '1';
120     else
121         clk_f <= clk;
122     end if;
123 end process;
124 
125 end rtl;
```

chip_ctrl.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  
5  
6  entity chip_ctrl is
7      port (
8          clk      : in std_logic;
9          rst      : in std_logic; -- reset asincrono
10         start    : in std_logic; -- segnale di inizio programmazione
11         start_acq : in std_logic; -- trigger inizio acquisizione
12         restart : in std_logic; -- segnale per resettare il puntatore
       della memoria
```

```
13          EOD      : in std_logic; -- End Of Data: fine trasmissione
        dati per la memoria
14          EOA      : in std_logic; -- End Of Acquisition
15          tc       : in std_logic; -- terminal count
16          conv_prog_n : out std_logic;
17          SOC         : out std_logic;
18          se          : out std_logic;
19          clk_f_en    : out std_logic;
20          clk_s_en    : out std_logic;
21          count_en2    : out std_logic;
22          count_en    : out std_logic
23      );
24  end entity;
25
26  architecture rtl of chip_ctrl is
27      type state is (
28              RST_state ,
29              MEM_state ,
30              WAIT_state,
31              IDLE_state ,
32              CONV_state,
33              ACQ_state );
34      signal present_state : state:=IDLE_state;
35      signal next_state    : state;
36
37
38  begin
39
40  state_register : process(clk,rst)
41  begin
42      if(rst='1') then
43          present_state <= IDLE_state;
44      elsif(rising_edge(clk)) then
45          present_state <= next_state;
46      end if;
47  end process state_register;
48
49
50
51  state_updating : process(
52              present_state ,
53              start,
54              start_acq,
55              restart,
56              EOD,
57              EOA,
58              tc,
59              rst)
60
```

```
61 begin
62     case present_state is
63 
64             when IDLE_state =>
65                 if (start='1') then
66                     next_state      <= MEM_state;
67                 else next_state <= IDLE_state;
68                 end if;
69 
70             when MEM_state =>
71                 if (EOD='1') then
72                     next_state      <= WAIT_state;
73                 else next_state <= MEM_state;
74                 end if;
75 
76             when WAIT_state =>
77                 if (start_acq='1') then
78                     next_state      <= ACQ_state;
79                 else next_state <= WAIT_state;
80                 end if;
81 
82             when ACQ_state =>
83                 if(EOA='1') then
84                     next_state      <= CONV_state ;
85                 else next_state <= ACQ_state ;
86                 end if;
87 
88             when CONV_state =>
89                 if(tc='1') then
90                     if (restart='1') then
91                         next_state      <= RST_state ;
92                     else next_state <= ACQ_state;
93                     end if;
94                 else next_state <= CONV_state;
95                 end if;
96 
97             when others => -- RST_state
98                 if (rst='1') then
99                     next_state      <= IDLE_state;
100                else next_state <= ACQ_state;
101                end if;
102 
103    end case;
104 end process state_updating;
105 
106 
107 state_register_out : process(present_state)
108 begin
109 
```

```
110     case present_state is
111
112             when IDLE_state =>
113                 conv_prog_n <= '1';
114                 SOC         <= '1';
115                 clk_f_en    <= '0';
116                 clk_s_en    <= '0';
117                 count_en    <= '0';
118                 count_en2   <= '0';
119                 se          <= '0';
120
121             when MEM_state => --Memory programming
122                 conv_prog_n <= '0';
123                 SOC         <= '1';
124                 clk_f_en    <= '0';
125                 clk_s_en    <= '1';
126                 count_en    <= '0';
127                 count_en2   <= '0';
128                 se          <= '1';
129
130             when WAIT_state => --Waiting for start Acquisition
131                 conv_prog_n <= '1';
132                 SOC         <= '0';
133                 clk_f_en    <= '0';
134                 clk_s_en    <= '0';
135                 count_en    <= '0';
136                 count_en2   <= '0';
137                 se          <= '0';
138
139             when ACQ_state => --Acquisition
140                 conv_prog_n <= '1';
141                 SOC         <= '0';
142                 clk_f_en    <= '0';
143                 clk_s_en    <= '1';
144                 count_en    <= '0';
145                 count_en2   <= '1';
146                 se          <= '0';
147
148             when CONV_state => --Conversion
149                 conv_prog_n <= '1';
150                 SOC         <= '1';
151                 clk_f_en    <= '1';
152                 clk_s_en    <= '0';
153                 count_en    <= '1';
154                 count_en2   <= '0';
155                 se          <= '0';
156
157             when others => -- Reset del puntatore memoria
158                 conv_prog_n <= '0';
```

```
                    SOC         <= '0';
                    clk_f_en    <= '0';
                    clk_s_en    <= '0';
                    count_en    <= '0';
                    count_en2   <= '0';
                    se          <= '0';

    end case;
end process state_register_out;

end rtl;
```

chip.vhd

```
library IEEE;
USE ieee.std_logic_1164.all;

entity chip is
    generic(
        Z : integer:=5;
        M : integer := 20);

port(   clk             :in std_logic;
        rst             :in std_logic;
        start           :in std_logic;
        start_acq       :in std_logic;
        restart         :in std_logic;
        data_in         :in std_logic_vector(M-1 downto 0) ;
        x               :out integer :=1;
        mem_din         :out std_logic;
        clk_s           :out std_logic;
        clk_f           :out std_logic;
        SOC             :out std_logic;
        conv_prog_n     :out std_logic
        );
end entity;

architecture rtl of chip is

signal tc                       :  std_logic;
signal a                        :  std_logic;
signal b                        :  std_logic;
signal count_en                 :  std_logic;
signal count_en2                :  std_logic;
signal se                       :  std_logic;
signal EOD                      :  std_logic;
signal EOA                      :  std_logic;


```

```
36 component chip_ctrl is
37
38 port (
39   clk                              : in  std_logic;
40   rst                              : in  std_logic;
41   start                            : in  std_logic;
42   start_acq                        : in  std_logic;
43   restart                          : in std_logic;
44   EOD                              : in  std_logic;
45   EOA                              : in  std_logic;
46   tc                               : in  std_logic;
47   conv_prog_n                      : out  std_logic;
48   SOC                              : out  std_logic;
49   se                               : out std_logic;
50   clk_f_en                         : out  std_logic;
51   clk_s_en                         : out  std_logic;
52   count_en2                        : out  std_logic;
53   count_en                         : out  std_logic) ;
54
55 end component;
56
57
58 component datapath_chip is
59
60  port (
61   clk                          : in  std_logic;
62   rst                          : in  std_logic;
63   clk_f_en                     : in  std_logic;
64   clk_s_en                     : in  std_logic;
65   data_in                      : in std_logic_vector(M-1 downto 0);
66   x                            : out integer:=1 ;--range 1 to Z+1 :=1;
67   clk_f                        : out  std_logic;
68   clk_s                        : out  std_logic;
69   EOD                          : out  std_logic;
70   EOA                          : out  std_logic;
71   count_en2                    : in std_logic;
72   count_en                     : in std_logic;
73   se                           : in std_logic;
74   mem_din                      : out  std_logic;
75   tc                           : out  std_logic);
76 end component;
77
78
79 begin
80
81 c1: chip_ctrl port map (
82   clk                  =>clk,
83   rst                  =>rst,
84   start                =>start,
```

```
85    start_acq              =>start_acq,
86    restart                =>restart,
87    EOD                    =>EOD,
88    EOA                    =>EOA,
89    tc                     =>tc,
90    se                     =>se,
91    conv_prog_n            =>conv_prog_n,
92    SOC                    =>SOC,
93    clk_f_en               =>a,
94    clk_s_en               =>b,
95    count_en2              =>count_en2,
96    count_en               =>count_en) ;
97
98  c2: datapath_chip port map (
99    clk                     =>clk    ,
100   rst                     =>rst    ,
101   clk_f_en                =>a  ,
102   clk_s_en                =>b  ,
103   se                      =>se,
104   data_in                 =>data_in,
105   x                       =>x,
106   EOA                     =>EOA,
107   EOD                     =>EOD,
108   clk_f                   =>clk_f,
109   clk_s                   =>clk_s  ,
110   count_en                =>count_en,
111   count_en2               =>count_en2,
112   mem_din                 =>mem_din,
113   tc                      =>tc  );
114
115 end rtl;
```

A.4 System code

sistema.vhd

```
1  library IEEE;
2  USE ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity sistema is
6  generic(
7    Z                  : integer := 5; -- numero di words per la
       programmazione della memoria
8    M                  : integer := 20; -- numero di bit per ogni word
       per la programmazione della memoria
```

```
9   L                   : integer := 4; -- numero di words da inviare al
      DAC dati.
10  N                   : integer := 24;-- numero di bit per ogni word da
      inviare al DAC dati.
11  NN                  : integer := 16; -- numero di bit per ogni word
      da inviare al DAC riferimenti.
12  LL                  : integer := 3); -- numero di words da inviare al
      DAC riferimenti.
13
14 port(clk, rst , REF, DATA,PROG,ACQ,RST_R: in std_logic;
15         mem_din, clk_s, clk_f, SOC, conv_prog_n,
16         LDAC_1, LDAC_2, RST_SEL1,RST_SEL2,
17         DIN_1, DIN_2, CLK_16_1, CLK_8,CLK_16_2,
18         DIN_REF, sync_vcm, sync_1, sync_2,
19         sync_refp, sync_refn, rst_out1, rst_out2 : out std_logic);
20 end entity;
21
22
23 architecture dut of sistema is
24
25 component chip is
26 port(clk, rst,  start,start_acq, restart            : in std_logic;
27 data_in                                    : in std_logic_vector(M-1
       downto 0) ;
28 x                                          : out integer :=1;--range
       1 to Z+1 :=1;
29 mem_din, clk_s, clk_f, SOC, conv_prog_n    : out std_logic);
30 end component;
31
32
33 component TX_ref is
34 port(clk_0, rst_0 , start_0                      : in std_logic;
35 ss_out                                          : out integer range 1
       to LL+1;
36 sync_refp_out,sync_refn_out,sync_vcm_out, ready_0: out std_logic;
37 data_out                                        : in std_logic_vector
     (NN-1 downto 0);
38 sclk_out,serial_out                            : out std_logic);
39 end component;
40
41
42 COMPONENT TX is
43 port(clk_0, rst_0, start_0                      : in std_logic;
44 i_out                                           : out integer range 1
       to L+1;
45 ctrl_bit                                        : in std_logic_vector
     (7 downto 0);
46 data_bit                                        : in std_logic_vector
     (15 downto 0);
```

```
47 serial_out, sync_out, sclk_out,LDAC,RST_SEL, rst_out, ready_0    : out
       std_logic);
48 end component;
49
50
51 ----------------------------------------------------------------------
52
53 type reg_file_type is array(1 to LL) of std_logic_vector(NN-1 downto
       0);
54 constant reg_file: reg_file_type:=
55 (                        -- Esempio di sequenza di dati
56 "0000100011100100",      -- dato inviato al DAC refp
57 "0000101011001001",      -- dato inviato al DAC refn
58 "0000110010110010"       -- dato inviato al DAC vcm
59 );
60 ----------------------------------------------------------------------
61
62
63 type reg_file_data_type is array(1 to L) of std_logic_vector(15
       downto 0);
64 constant reg_file_data1: reg_file_data_type:=
65 (
66 "1110010010110100", -- Esempio di sequenza di dati da inviare al
       primo DAC
67 "1100100110100101",
68 "1011001010010110",
69 "1001111001010011"
70 );
71
72
73 constant reg_file_data2: reg_file_data_type:=
74 (
75 "1110010010110100", -- Esempio di sequenza di dati da inviare al
       secondo DAC
76 "1100100110100101",
77 "1011001010010110",
78 "1001111001010011"
79 );
80
81
82 type reg_file_ctrl_type is array(1 to L) of std_logic_vector(7 downto
       0);
83 constant reg_file_ctrl1: reg_file_ctrl_type:=
84 (
85 "00001000", -- Esempio di sequenza di bit di controllo da inviare
86 "00001010", -- al primo DAC, per scrivere nei relativi 4 registri e
87 "00001100", -- inviare simultaneamente i dati sulle 4 uscite.
88 "00001110"
89 );
```

```
90
91  constant reg_file_ctrl2: reg_file_ctrl_type:=
92  (
93  "00001000", -- Esempio di sequenza di bit di controllo da inviare al
94  "00001010", -- secondo DAC, per scrivere nei relativi 4 registri e
95  "00001100", -- inviare simultaneamente i dati sulle 4 uscite.
96  "00001110"
97  );
98  ------------------------------------------------------------------------
99  type reg_file_prog_type is array(1 to Z) of std_logic_vector(M-1
        downto 0);
100 constant reg_file_prog: reg_file_prog_type:=
101 (
102 "00111110010010110100", -- Esempio di sequenza di dati da inviare
103 "11001100100110100101", -- al chip per programmare la memoria
104 "01011011001010010110",
105 "11101001111001010011",
106 "00110101000110011100"
107 );
108
109
110 ------------------------------------------------------------------------
111 --Segnali per il TX riferimenti
112 signal ready_ref : std_logic;
113 signal start_ref : std_logic;
114 signal ss : integer range 1 to LL+1; --Permette di contare i dati
        inviati
115 signal data_out_ref : std_logic_vector(NN-1 downto 0);
116
117 ------------------------------------------------------------------------
118 --Segnali per i TX dati
119 signal i : integer range 1 to L+1; --Permette di contare i dati
        inviati
120 signal ctrl_bit1 : std_logic_vector(7 downto 0);
121 signal data_bit1 : std_logic_vector(15 downto 0);
122 signal ctrl_bit2 : std_logic_vector(7 downto 0);
123 signal data_bit2 : std_logic_vector(15 downto 0);
124 signal ready_1 : std_logic;
125 signal ready_2 : std_logic;
126 signal start_0 : std_logic;
127
128 ------------------------------------------------------------------------
129 --Segnali per il controllo del chip
130 signal x : integer ;--Permette di contare i dati inviati
131 signal start: std_logic; --Fa iniziare la programmazione della
        memoria
132 signal start_acq: std_logic; -- Fa iniziare l'acquisizione
133 signal restart:std_logic; --Fa resettare il contatore che punta alla
        memoria
```

```
134 signal data_in: std_logic_vector(M-1 downto 0);
135
136 ------------------------------------------------------------------------------
137
138 begin
139
140
141 chip1 : chip port map(
142 clk             =>clk           ,
143 rst             =>rst           ,
144 x               =>x             ,
145 data_in         =>data_in       ,
146 start           =>start         ,
147 start_acq       =>start_acq     ,
148 mem_din         =>mem_din       ,
149 clk_s           =>clk_s         ,
150 clk_f           =>clk_f         ,
151 SOC             =>SOC           ,
152 conv_prog_n     =>conv_prog_n   ,
153 restart         =>restart       );
154
155
156 DAC8 : TX_ref port map(
157               clk_0           =>clk  ,
158               rst_0           =>rst  ,
159               ready_0         =>ready_ref,
160               start_0         =>start_ref,
161               ss_out          =>ss,
162               data_out        =>data_out_ref,
163               serial_out      =>DIN_REF  ,
164               sync_refp_out   =>sync_refp,
165               sync_refn_out   =>sync_refn,
166               sync_vcm_out    =>sync_vcm,
167               sclk_out        =>CLK_8 );
168
169
170 DAC16_1: TX port map (
171 clk_0       =>clk,
172 rst_0       =>rst,
173 start_0     =>start_0,
174 serial_out  =>DIN_1,
175 sync_out    =>sync_1,
176 RST_SEL     =>RST_SEL1,
177 LDAC        =>LDAC_1,
178 i_out       =>i,
179 rst_out     =>rst_out1,
180 ready_0     =>ready_1,
181 ctrl_bit    =>ctrl_bit1,
182 data_bit    =>data_bit1,
```

```
183 sclk_out     =>CLK_16_1);
184
185
186 DAC16_2: TX port map (
187 clk_0        =>clk,
188 rst_0        =>rst,
189 start_0      =>start_0,
190 serial_out   =>DIN_2,
191 sync_out     =>sync_2,
192 RST_SEL      =>RST_SEL2,
193 LDAC         =>LDAC_2,
194 rst_out      =>rst_out2,
195 ready_0      =>ready_2,
196 ctrl_bit     =>ctrl_bit2,
197 data_bit     =>data_bit2,
198 sclk_out     =>CLK_16_2);
199 ------------------------------------------------------------------------
200
201 --Process per programmare i riferimenti.
202 reference: process (clk, REF)
203
204 --Per scrivere nei DAC ref, bisogna asserire REF.
205 --L'operazione si interrompe automaticamente quando
206 --saranno stati inviati ai DAC tutti le words presenti nel
      RegisterFile
207 --REF va tenuto ='1' per tutta la trasmissione, poi don't care.
208
209 begin
210 if (clk'event and clk='1') then
211     if  REF='1' then
212
213         if (ss<LL+1) then
214
215             if ready_ref='1' then --A ogni ready invio una nuova word
216             start_ref<='1';
217             data_out_ref<=reg_file(ss);
218             else start_ref<='0';
219             end if;
220
221         else start_ref<='0';
222         data_out_ref<=(others=>'0');
223         end if;
224
225     else start_ref<='0';
226     data_out_ref<=(others=>'0');
227     end if;
228
229 end if;
230
```

```
231 end process;
232 ------------------------------------------------------------------------
233
234 --Process per programmare i DAC dati.
235 dac_data: process (clk, DATA)
236
237 --Per scrivere nei DAC dati, bisogna asserire DATA.
238 --I dati arrivano sincronizzati su entrambi i DAC.
239 --L'operazione si interrompe automaticamente quando
240 --saranno stati inviati ai DAC tutte le words presenti nel
    RegisterFile
241 --DATA va tenuto ='1';
242
243 begin
244 if (clk'event and clk='1') then
245
246     if  DATA='1' then
247
248         if (i<L+1) then
249
250             if ready_1='1' then
251             start_0<='1';
252             ctrl_bit1<=reg_file_ctrl1(i);
253             data_bit1<=reg_file_data1(i);
254             else start_0<='0';
255             ctrl_bit1<=(others=>'0');
256             data_bit1<=(others=>'0');
257             end if;
258
259             if ready_2='1' then
260             start_0<='1';
261             ctrl_bit2<=reg_file_ctrl2(i);
262             data_bit2<=reg_file_data2(i);
263             else start_0<='0';
264             ctrl_bit2<=(others=>'0');
265             data_bit2<=(others=>'0');
266             end if;
267
268         else start_0<='0';
269             ctrl_bit1<=(others=>'0');
270             data_bit1<=(others=>'0');
271             ctrl_bit2<=(others=>'0');
272             data_bit2<=(others=>'0');
273         end if;
274
275     else start_0<='0';
276             ctrl_bit1<=(others=>'0');
277             data_bit1<=(others=>'0');
278             ctrl_bit2<=(others=>'0');
```

```
279                 data_bit2<=(others=>'0');
280
281         end  if;
282 end if;
283
284 end process;
285
286 ------------------------------------------------------------------------
287
288 --Process per programmare la memoria.
289 progamming: process (clk, PROG)
290
291 --Per iniziare la programmazione della memoria, bisogna asserire PROG
        .
292 --L'operazione si interrompe automaticamente quando
293 --saranno stati inviati al chip tutti i dati presenti nel
        RegisterFile.
294 --dopodichè il chip va in stato di WAIT, finchè non viene inviato il
        segnale
295 --di inizio acquisizione (ACQ).
296 --PROG va tenuto ='1'  per tutta la programmazione, poi don't care.
297
298 begin
299 if (clk'event and clk='1') then
300
301     if  PROG='1' then
302         if (x<Z+1) then
303             start<='1';
304             data_in<=reg_file_prog(x);
305         else
306             data_in<=(others=>'0');
307             start<='0';
308         end if;
309     else start<='0';
310     data_in<=(others=>'0');
311     end if;
312 end if;
313
314 end process;
315
316 --Process per iniziare l'acquisizione/conversione
317 acquisition: process (clk, ACQ)
318
319 -- Per iniziare l'acquisizione bisogna asserire ACQ.
320 -- La conversione inizia automaticamente dopo l'acquisizione.
321 -- E' possibile programmare la durata dell'acquisizione mediante il
        segnale "A"
322 -- nel file datapath_chip.vhd
```

```
323 -- ACQ dopo essere passato la prima volta da '0' a '1' diventa don't
        care.
324
325 begin
326 if (clk'event and clk='1') then
327
328     if  ACQ='1' then
329         start_acq<='1';
330     else start_acq<='0';
331     end if;
332 end if;
333 end process;
334
335 --Process per resettare il contatore della memoria.
336 restart_mem_counter: process (clk, RST_R)
337 -- Per resettare il contatore della memoria bisogna asserire RST_R.
338 -- NB: Il segnale RST_R deve essere '1' quando è finita la
        conversione.
339 begin
340 if (clk'event and clk='1') then
341
342     if  RST_R='1' then
343         restart<='1';
344     else restart<='0';
345     end if;
346
347 end if;
348 end process;
349
350 end architecture;
```

sistema_tb.vhd

```
1  library IEEE;
2  USE ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity sistema_tb  is
7  end entity;
8
9  architecture dut of sistema_tb is
10
11
12 signal clk_test:  std_logic:='0';
13 signal rst_test:  std_logic;
14 signal  REF_test:  std_logic;
15 signal DATA_test :  std_logic;
16 signal PROG_test:  std_logic;
```

```
17 signal ACQ_test:  std_logic;
18 signal RST_R_test:  std_logic;
19 signal mem_din_test:  std_logic;
20 signal clk_s_test:  std_logic;
21 signal clk_f_test:  std_logic;
22 signal SOC_test: std_logic;
23 signal conv_prog_n_test: std_logic;
24 signal LDAC_1_test:  std_logic;
25 signal LDAC_2_test: std_logic;
26 signal RST_SEL1_test: std_logic;
27 signal RST_SEL2_test: std_logic;
28 signal DIN_1_test: std_logic;
29 signal DIN_2_test: std_logic;
30 signal CLK_16_1_test: std_logic;
31 signal CLK_8_test: std_logic;
32 signal CLK_16_2_test: std_logic;
33 signal DIN_REF_test: std_logic;
34 signal sync_vcm_test: std_logic;
35 signal sync_1_test: std_logic;
36 signal sync_2_test: std_logic;
37 signal sync_refp_test: std_logic;
38 signal sync_refn_test: std_logic;
39 signal rst_out1_test: std_logic;
40 signal rst_out2_test: std_logic;
41
42 component sistema is
43     port(clk, rst, REF,ACQ,PROG,RST_R,DATA: in std_logic;
44         mem_din, clk_s, clk_f, SOC, conv_prog_n,
45         LDAC_1, LDAC_2, RST_SEL1,RST_SEL2,
46         DIN_1, DIN_2, CLK_16_1, CLK_8,CLK_16_2,
47         DIN_REF, sync_vcm, sync_1, sync_2,
48         sync_refp, sync_refn, rst_out1, rst_out2 : out std_logic);
49 end component;
50
51
52 begin
53
54 s1: sistema port map (
55
56         clk=> clk_test,
57         rst=>rst_test,
58         REF=>REF_test,
59         DATA=>DATA_test,
60         RST_R=>RST_R_test,
61         ACQ=>ACQ_test,
62         PROG =>PROG_test,
63         mem_din=> mem_din_test,
64         clk_s=> clk_s_test,
65         clk_f=>clk_f_test,
```

```
66          SOC=>SOC_test,
67          conv_prog_n=>conv_prog_n_test,
68          LDAC_1=>LDAC_1_test,
69          LDAC_2=>LDAC_2_test,
70          RST_SEL1=>RST_SEL1_test,
71          RST_SEL2=>RST_SEL2_test,
72          DIN_1=>DIN_1_test,
73          DIN_2=>DIN_2_test,
74          CLK_16_1=> CLK_16_1_test,
75          CLK_8=>CLK_8_test,
76          CLK_16_2=>CLK_16_2_test,
77          DIN_REF=>DIN_REF_test,
78          sync_vcm=>sync_vcm_test,
79          sync_1=>sync_1_test,
80          sync_2=>sync_2_test,
81          sync_refp=> sync_refp_test,
82          sync_refn=>sync_refn_test,
83          rst_out1=> rst_out1_test,
84          rst_out2=>rst_out2_test);
85
86
87  clk_test <= not(clk_test) after 1 ns;
88  stimoli: process
89
90  begin
91  ACQ_test<='0';
92  RST_R_test<='1';
93  DATA_test<='0';
94  REF_test<='0', '1' after 2 ns;
95  PROG_test<='0', '1' after 2 ns;
96  wait for 150 ns;
97  DATA_test<='1';
98  wait for 50 ns;
99  REF_test<='0';
100 wait for 30 ns;
101 PROG_test<='0';
102 wait for 150 ns;
103 ACQ_test<='1';
104
105 wait;
106 end process;
107 end dut;
```

Appendix B

MatLab code

numerical_elaboration

```
%%
%read .matlab file

T=readtable('cmp_out.matlab','Filetype','text');
time=T.(3);
voltage=T.(4);

time_input=T.(1);
voltage_input=T.(2);

%%
%output acquisition

Nbit=12;
FSR=1.8; %Full scale range voltage
offset=-0.0007789; %computed with curve fitting tool

y=1;
k=1;
z=0;
j=0;

%to be programmed
Tfirst=2.5254e-4; %time instant of the first output bit
Tck=5e-6; %clock period
Tlat=37.5e-6; %time interval between the end of an output word and
    the beginning of the next one
Nsamples=10108; %Number of output word contained in the output signal
N=Nbit*Nsamples;  %Number of points to be saved from the time vector
time_saved=zeros(N,1); %Vector containing the saved index of the time
     vector.
```

```
30
31
32 vlsb=FSR/(2^N);
33
34 %Index saving
35 for i=1:N
36
37     if k==Nbit+1
38         j=j+1;
39         k=1;
40         z=z-1;
41     end
42
43     while time_saved(i)==0
44             if time(y)<Tfirst+z*Tck+j*Tlat
45              y=y+1;
46             else
47
48              time_saved(i)=y;
49             end
50     end
51     k=k+1;
52     z=z+1;
53 end
54
55
56 %Comparator which generate vector of logic values
57  for i=1:length(time_saved)
58
59         if voltage(time_saved(i))<0.9
60           bit(i)=0;  %vector containing the serial bits of the ouput
61         else
62           bit(i)=1;
63         end
64
65  end
66
67
68 M=reshape(bit,Nbit,Nsamples); %Matrix for a better visualization of
       the output words
69
70
71 %binary-decimal conversion
72 for i=1:Nsamples
73     words(i)=0;
74     for j=1:Nbit
75         num(i)=M(Nbit+1-j,i)*2^(j-1);
76         words(i)=words(i)+num(i);    %Vector containing the 12-bit
       output words
```

```
    end
end


%mapping digital word over the voltage range [0, 1.8]V
range = words*FSR/(2^Nbit-1);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%input acquisition

y2=1;

%da programmare
Tstart=0.00024051; %time instant of the first falling edge of clks
Tck2=9.25e-05;  %clks period
time_saved2=zeros(Nsamples,1); %Vector containing the saved index of
    the time vector.

%Index saving
for i=1:Nsamples
    while time_saved2(i)==0
            if time_input(y2)<Tstart+(i-1)*Tck2
                y2=y2+1;
            else
            time_saved2(i)=y2;
            end
    end
end


for i=1:length(time_saved2)
    input(i)=voltage_input(time_saved2(i)); %vector containing the
    values of the input signal which has been converted by the ADC
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Metrics computing

%DNL
for i=1:Nsamples-1
DNL(i)=(((range(i+1)-range(i))-vlsb)/vlsb);
end
DNL_max=max(DNL);

%INL
for i=1:Nsamples
    INL(i)=(input(i)-range(i))/vlsb;
end
INL_max=max(INL);
```

```
124
125 %Quantization error
126 for i=1:Nsamples
127     Eq(i)=range(i)-input(i);
128 end
129
130 Eq_norm=Eq/vlsb;
131
132 %ENOB
133 Vqrms=rms(Eq);
134 ENOB=log2(FSR/Vqrms);
135
136
137 ideale=linspace(0,4095,Nsamples);
138 ideale2=linspace(0,FSR,Nsamples);
139 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
140 %%
141 %plotting
142
143 figure(1)
144 plot(input,words,'.');
145 set(gca,'ytick', 0:273:4095)
146 ylim([0 4095])
147 xlabel('input voltage [V]');
148 ylabel('digital output word');
149 hold on;
150 plot(ideale2,ideale,'r', 'LineWidth',1);
151 legend('actual trans-char','ideal trans-char');
152 grid on;
153
154
155 figure(2)
156 plot(input,words,'b', 'LineWidth',1);
157 xlabel('input voltage [V]');
158 ylabel('digital output word');
159 hold on;
160 plot(ideale2,ideale,'r', 'LineWidth',1);
161 legend('actual trans-char','ideal trans-char');
162 grid on;
163
164
165 figure(3)
166 plot(Eq_norm);
167 xlabel('N   of sample');
168 ylabel('Quantization error [LSB]');
169 xlim([0 10108])
170 line(xlim, [0.5,0.5], 'Color', 'r', 'LineWidth', 1, 'LineStyle', '--'
    );
```

```
171 line(xlim, [-0.5,-0.5], 'Color', 'r', 'LineWidth', 1, 'LineStyle', '
    --');
172 legend('Quantization Err.');
173 grid on;
174
175
176 figure(4)
177 histogram(Eq_norm,95, 'BinLimits',[-9.1,9.1]);
178 line([0.5,0.5], ylim, 'Color', 'k', 'LineWidth', 2, 'LineStyle', '--')
    ;
179 line([-0.5,-0.5], ylim, 'Color', 'k', 'LineWidth', 2, 'LineStyle', '
    --');
180 legend('Distribution of QE');
181 grid on;
182
183
184  figure(5)
185 plot(DNL,'Color','#D95319','LineWidth',1);
186 xlabel('N   of sample');
187 ylabel('Differential Non-Linearity [LSB]');
188 xlim([0  718])
189 legend('DNL');
190 grid on;
191
192
193 figure(6)
194 plot(INL,'Color','#77AC30','LineWidth',1);
195 xlabel('N   of sample');
196 ylabel('Integral Non-Linearity [LSB]');
197 xlim([0  718])
198 legend('INL');
199 grid on;
```

Bibliography

[1] D. Del Corso, V. Camarchia, R. Quaglia, and P. Bardella. *Telecommunication Electronics*. Ed. by Artech House. February 2020 (cit. on pp. 1, 2, 14, 16, 18).

[2] D. Johns and K. Martin. *Analog Integrated Circuit Design*. John Wiley and Sons, Inc., 1997 (cit. on pp. 2, 16).

[3] G. Manganaro. *Advanced Data Converters*. Cambridge, 2012 (cit. on pp. 2, 3, 12, 15, 16).

[4] *SAR ADCs: Architecture, Applications, and Support Circuitry*. `https://embedded-tutorial.blogspot.com/2020/06/`. Accessed: 2022-05-31 (cit. on p. 3).

[5] I. B. Sharuddin and L. Lee. «An ultra-low power and area efficient 10 bit digital to analog converter architecture». In: *2014 IEEE International Conference on Semiconductor Electronics (ICSE2014)* (2014) (cit. on p. 4).

[6] J. Proakis and D. Manolakis. *Digital Signal Processing : Principles, Algorithms, and Applications*. Macmillan Publishing Company, 1992 (cit. on p. 5).

[7] D. L. Donoho. «Compressed sensing». In: *IEEE Transactions on Information Theory* 52.4 (2006), pp. 1289–1306 (cit. on p. 5).

[8] M. Mangia, F. Pareschi, V. Cambareri, R. Rovatti, and G. Setti. «Rakeness-based design of low-complexity compressed sensing». In: *IEEE Transactions on Circuits and Systems I: Reg. Papers* 64.5 (2017), pp. 1201–1213 (cit. on p. 5).

[9] F. Pareschi, P. Albertini, G. Frattini, M. Mangia, R. Rovatti, and G. Setti. «Hardware-algorithms co-design and implementation of an analog-toinformation converter for biosignals based on compressed sensing». In: *IEEE Transactions on Biomedical Circuits and Systems I: Reg. Papers* 10.1 (2016), pp. 149–162 (cit. on p. 5).

[10] C. Paolino, F. Pareschi, M. Mangia, R. Rovatti, and G. Setti. «A Practical Architecture for SAR-based ADCs with Embedded Compressed Sensing Capabilities». In: *Proc. 15th Conference Ph.D. Reserch in Microelectronics and Electronics* (2019), pp. 133–136 (cit. on pp. 5, 6).

[11] P. P. Fasang. «Simulation considerations for analog-digital ASICs». In: *Third Annual IEEE Proceedings on ASIC Seminar and Exhibit* (1990) (cit. on p. 8).

[12] *Virtuoso® AMS Environment User Guide*. Cadence, 2004 (cit. on p. 9).

[13] *Introduction to AMS Designer Simulation - Rapid Adoption Kit (RAK)*. Cadence, 2020 (cit. on pp. 9–11).

[14] NXP. *How to Increase the Analog-to-Digital Converter Accuracy in an Application*. 2016 (cit. on p. 12).

[15] F.M. Remley and J.F.X. Browne an S.N. Baron. *Reference data for engineers (Ninth edition)*. Ed. by Newnes. 2002 (cit. on p. 14).

[16] *Measuring Offset and Gain Errors in ADC*. `https://www.mathworks.com/help/msblks/ug/offset-error-and-gain-error.html`. Accessed: 2022-06-11 (cit. on p. 15).

[17] *Terms A/D converter characterization*. `http://www.atx7006.com/articles/terms/adc`. Accessed: 2022-06-12 (cit. on p. 17).

[18] *Signal Chain Basics 87: ADC DNL in Precision Signal Chain Error Analysis*. `https://www.planetanalog.com/signal-chain-basics-87-adc-dnl-in-precision-signal-chain-error-analysis/`. Accessed: 2022-06-12 (cit. on p. 18).

[19] L. A. Singer and T. L. Brooks. «A 14-bit 10-MHz calibration-free CMOS pipelined A/D converter». In: *VLSI Circuits Conference* (1996), pp. 94–95 (cit. on p. 18).

[20] A. Hastings. *The Art of Analog Layout*. Ed. by Prentice Hall. 2005 (cit. on p. 18).

[21] Analog Devices - H.J. Zhang. *Basic Concepts of Linear Regulator and Switching Mode Power Supplies*. 1993 (cit. on p. 37).

[22] *SMT / SMD Capacitor*. `https://www.electronics-notes.com/articles/electronic_components/capacitors/smd-smt-surface-mount-capacitor.php`. Accessed: 2022-05-31 (cit. on p. 38).

[23] *3362 Trimpot Trimming Potentiometer datasheet*. Bourns (cit. on p. 38).

[24] *DAC8555 datasheet*. Texas Instruments (cit. on pp. 52–54).

[25] *DAC081S101 datasheet*. Texas Instruments (cit. on pp. 61, 62).