Analog synthesis of a sine wave using a pure digital FPGA chip

Giulia Gioachin
dept. of Physics of Turin
Turin, Italy
giulia.gioachin@edu.unito.it

Flavio Galaverna
dept. of Physics of Turin
Turin, Italy
flavio.galaverna@edu.unito.it

***Abstract*—The FPGA (Field Programmable Gate Array) is a 100% digital chip that can be programmed "on field", i.e. at any moment, and reprogrammed at any time, to obtain very large different functions. This article describes two different ways for the synthesis of a sinusoidal signal and two different ways to generate an analog signal using the FPGA chip will be presented: the PWM technique and the analog conversion using the dedicated module DAC.**

I. The PWM technique

A. What is a PWM

The PWM (Pulse Width Modulation) technique allows to program the duty cycle of a periodic signal at will. This first section illustrates the procedure performed to obtain an analog signal using this technique.

B. The Project

To begin the project, 64 values belonging to a sine wave with amplitude between 0 and 1000 have been selected. These decimal numbers are converted to hexadecimal using a python script and then, the values are stored in a ROM (Read Only Memory) to be then "converted" in a analog output. The memory has been synthesized so that its width was of 32-bit and then it was allocated in a B-RAM in FPGA.[1]
Using a 32-bit counter, a pulse generator has been created that generates a Tick (the output) every 1000 clock posedge. It is assigned to the ROM so that each time there is a tick, the memory read one data.
Using the software VIVADO (a mixed-language simulator that supports Verilog, SystemVerilog and VHDL language), it is possible to visualize the ROM output from an "analog point of view" and to confirm the sinusoidal trend of the data (as shown in Fig.1).

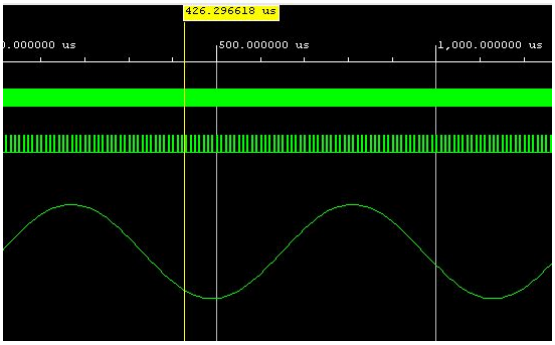

Fig. 1. ROM data. Starting from the top: CLK $200MHz$, $Tick$ and the ROM data visualizing as "analog values".

At this point the data is introduced into a binary comparator (the heart of the PWM block) which consists a 32-bit counter that is activated when the output tick is equal to 1. If the ROM data value is greater than the counter, the output is 1; otherwise it is 0.
This procedure is repeated for all 64 sine values. In the following figure it is possible to see the effect of PWM on the ROM values. In fact, the time in which the signal is 1 increases with the increase of the sine value.

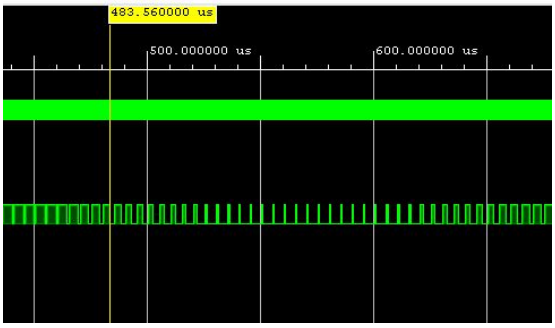

Fig. 2. Sine wave using the PWM technique. In the upper signal there is the Clock at $200MHz$, but because of its high frequency it's impossible to see every single clock posedge. In the lower one, it is represented the effect of PWM on the ROM valuesand it is possible to see the changes in the duty cycle of signal.

C. IP Blocks

In order to improve the project, 2 IPs (Intellectual Property) have been created:

1) PLL: Thanks to the IP Catalog provided by VIVADO, it was chosen a Phase Locked Loop (PLL) capable of providing a 200 MHz clock as input for the full project.

[1]Because of the sine data range, the first two bytes of each ROM address will be equal to zero.

2) IP-ROM: In addition, it was chosen a ROM with the same size to the previous one to replace it. Also in this case, the memory was loaded with the same sine values.

D. Displaying the Sine Wave using LTspice

To visualize the previous output as an analog output (without the need to own an FPGA and a board) it is possible to use LTspice, a simulation software for analog electronic circuits.
Writing a script with Python, it was created a file with two columns returns the respective time (on ns) and sine value (exiting from the PWM block) at every clock posedge. Then, always using Python, the time is converted into seconds and the digital sine values are multiplied by 3.3 (the power voltage of the FPGA core). At this point, it was build in LTspice a circuit with:

- A voltage generator able to read the file with the time and voltage values
- A resistance of 50 Ohm
- A capacity of 400 nF
- An operational amplifier

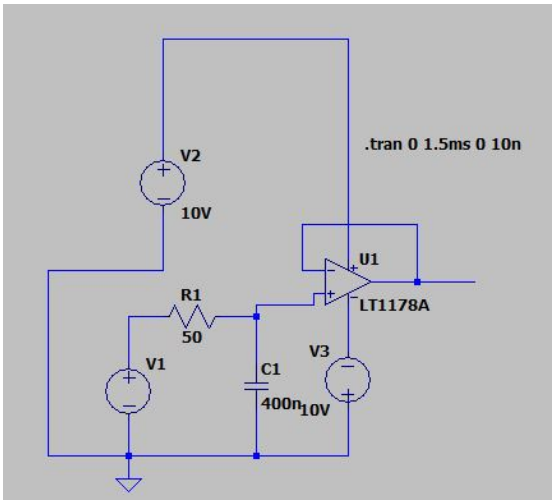

Fig. 3. LTspice circuit.

Simulating the circuit, it is possible to get the following sine wave outputs (Fig. 4).

E. Project implementation on FPGA using VIVADO

To conclude the first part of the project, it was decided to implement it on FPGA using VIVADO.
Therefore, some constraints have been compiled to enrich the project:

- The input ports (EN and CLK $100MHz$ which enters in the PLL block to be "converted" into a $200MHz$ clock) and output port ($SINE$) have been mapped out in different Pins. In the final block of the project there is also a switch (EN) to turn on or off the output Sine signal

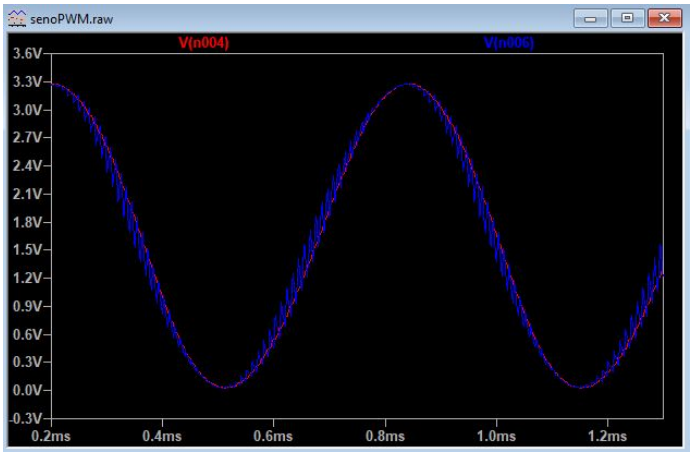

Fig. 4. Sine wave from the simulation. The blue line is the output from the RC circuit, while the red one is the output from the OP-AMP.

- In order to avoid timing violation, it was defined an input clock of 100 MHz, with a duty cycle of 50%
- It was set a capacity of 20 pF at the output. This value corresponds to a typical oscilloscope probe capability
- A 10 ns delay has been put on all input ports

After the implementation it was possible to verify that the Total Negative Slack (TNS) was equal to 0. The slack represents the difference between the constraint time and the analyzed value. If the time taken to travel the entire circuit is greater than the theoretical time set, there is no more margin for error and it is possible to enter in a "time violation" condition.
The implementation scheme is presented in figure 5.

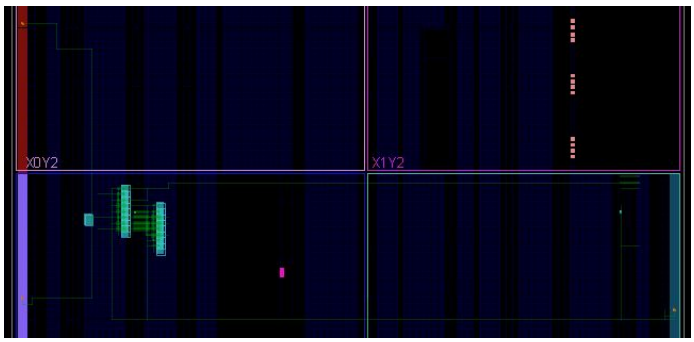

Fig. 5. The implementation scheme of the project. At the edges of the figure it is possible to see 3 colored squares corresponding to the inputs and output. While in the centre of the graph (in the large structure highlighted in green) there is the B-RAM with the sine values.

II. DAC DEDICATED MODULE

A. What is a DAC

A DAC (digital to analog converter) is a mixed signal electronic block which is able to achieve the analog conversion of a digital signal. It's impossible to implement a DAC on a FPGA chip because of its analog parts. However, the simulation software VIVADO is able to read and simulate the System Verilog sources, so, even if you are unable to implement a pure analog signal on FPGA, you can still simulate it.

B. The core

The core of the digital to analog conversion performed by the DAC is, actually, the following mathematical formula:

$$A_{out} = V_{ref} \sum_{i=0}^{N} \frac{B_i}{2^{N-i}} \tag{1}$$

where N is the number of bit of the DAC, B_i is the i-th bit and V_{ref} is the reference voltage, which often coincides with the supply voltage of the DAC block. Using System Verilog code, it is possible to implement the DAC core with the following lines:

```
always @(posedge clk) begin
   if ( en == 1'b1 ) begin
      A_out = 0.0 ;
      for ( i=0 ; i <= 11 ; i = i + 1 ) begin
         real a = 2.0 ;
         for ( j = 11 ; j > i ; j = j - 1 ) begin
            a = a*2.0 ;
         end // for
         assign digit = I_data[i] ;
         A_out = A_out + Vref*(digit/a) ;
      end // for
   end // if
end // always
```

Listing 1. Core of a 12 bit DAC code.

The listing above, which is the central part of the source code of a 12 bit DAC, takes a 12 bit bus I_data as input and obtains the analog result of conversion A_out as output.

C. The electronic block

As one can see in figure 6, the real representation of a DAC electronic block is actually more complicated that the one just presented.
In addition to the 12 bit DAC, the other components (except the amplifier that must be considered as included in the core of the analog conversion described above) are of no other use than to allow the serial transmission of the input data.

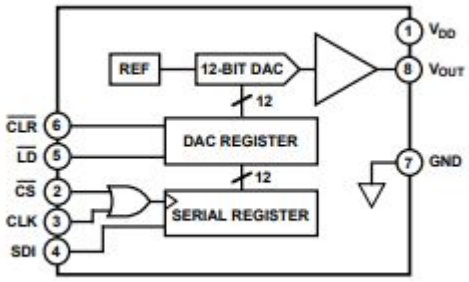

Fig. 6. Example of a real 12 bit DAC electronics block.

D. The project

The DAC that was implemented in the project presented in this article is a serial data input 12 bit digital to analog converter, exactly like the one in figure 6. The HDL (hardware description language) used was System Verilog, and the software used to simulate the project was, as also in section I, VIVADO. The DAC top level module describes a block with four [2] input ports, and one output port. The functionality of the ports are briefly listed below.

- $\overline{CS}$: Chip Select is the input port that acts as an active low enable of the input shift register of the DAC. Actually, because of the OR gate, if it is set at the low logical level, then the input data are able of being carried forward in the serial register.
- CLK: It is the input port that allows the sharing of the clock with the desired digital device, in order to achieve a synchronous data transmission.
- SDI: It is the input port that connects the DAC serial shift register to the serial output of the digital device (FPGA in this case).
- $\overline{LD}$: Active low input which writes the serial register data into the DAC register.
- V_{out}: The analog output port with the result of the conversion.

As reference voltage was used the value $V_{ref} = 5V$. In figure 7 is presented the DAC simulation made using VIVADO and immediately after the stimulus used in the test bench.

[2] the **CLR** port was not used in the project.

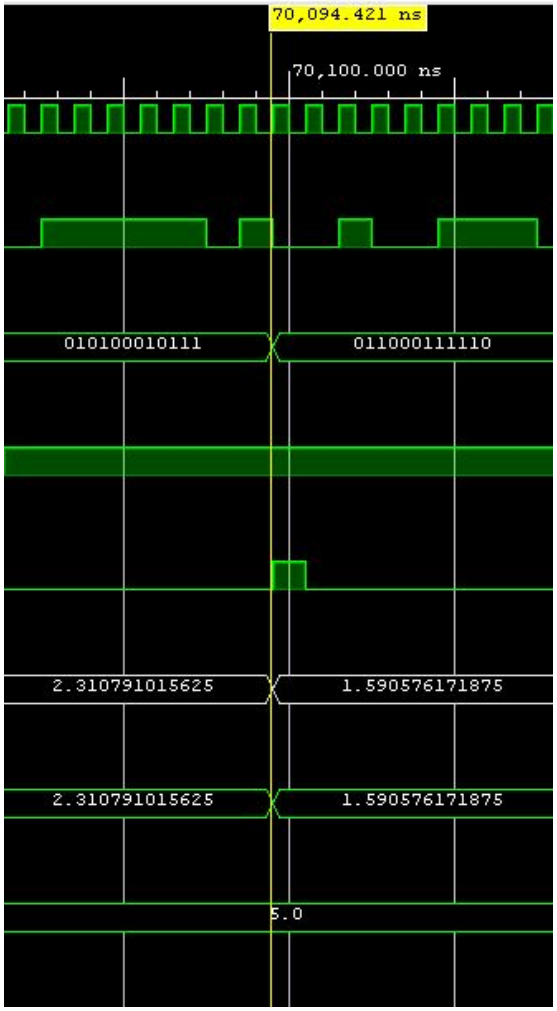

Fig. 7. Simulation of the serial input 12 bit DAC using VIVADO simulator. Starting from the top: CLK, SDI, I_data, CS, $\overline{LD}$, A_out, $A\_out\_expected$, V_ref.

```
always #(10000) begin
    CS = $random ;
end // always
always @ ( posedge clk100 ) begin
    SDI = $random ;
end // always
always #(10005) begin
    NOT_LD = 1'b1 ;
    #10 NOT_LD = 1'b0 ;
end // always
always @( posedge NOT_LD ) begin
    Aout_expected = 0.0 ;
    for ( i = 0 ; i <= 11 ; i = i + 1 ) begin
        real a = 2.0 ;
        for ( j = 11 ; j > i ; j = j - 1 ) begin
            a = a*2.0 ;
        end // for
        assign digit = I_data[i] ;
        Aout_expected = Aout_expected + Vref*(digit
/a) ;
    end // for
    $display ( "\n\n D_out should be : %f V at
time %d ns \n\n" , Aout_expected , $time ) ;
end // always
initial begin
    #100000 $finish ;
end // initial
```

Listing 2. Test bench of the serial input DAC.

As one can notice looking at figure 7, I_data are not the data in the 12 bit shift register (not represented in the picture); they are instead the data loaded into the dac register every time $\overline{LD}$ is set to the high logical level.

E. The FPGA set up

As said before, the serial input 12 bit DAC module is not synthesizable on FPGA, however, everything related to the logic of data transmission is. First of all, it must be said that the implementation of a synchronous transmission protocol, like the I^2C or the SPI, was not the goal of this project; that was, instead, synthesize a sinusoidal waveform in different ways and visualize it. Despite this, it was necessary to implement on the FPGA a certain amount of logic blocks, to allow the chip to communicate correctly with the DAC. All the transmission procedure is described in the next chapter.

F. The serial transmission

To make data transmission more similar to that of the protocols used in practice, it was implemented on the FPGA an 8 bit PISO (parallel in serial out) shift register. Because of its 8 bit size, the register allows to simulate an 8-bit data transmission, which is more similar to those usually used. The synthesizable code written to manage the serial transmission mode can be thought as divided into two parts.

1) 8-bit shift register management: the purpose of this code is to convert the 32 bits of the i-th address of the ROM in 2 bus of 1 Byte each (as also already said in section I, the first 2 more significant Bytes of the i-th address of the ROM are nothing but zero). The implemented module, therefore, takes as input a 32 bit bus $data\_bus$, a clock and a reset signal, while it has as output SO, which corresponds to the output of the

8-bit shift register, and a port here called LD_pdata, which is the signal that allow the parallel data to be loaded into the same shift register. The latter signal was introduced as module output because it is useful for the transmission logic described later. Thanks to the RST port, all the logic could restart at the right time if necessary. The simulation of this module and the test bench code used for the stimulus are presented in figure 8 and in listing 3 respectively.

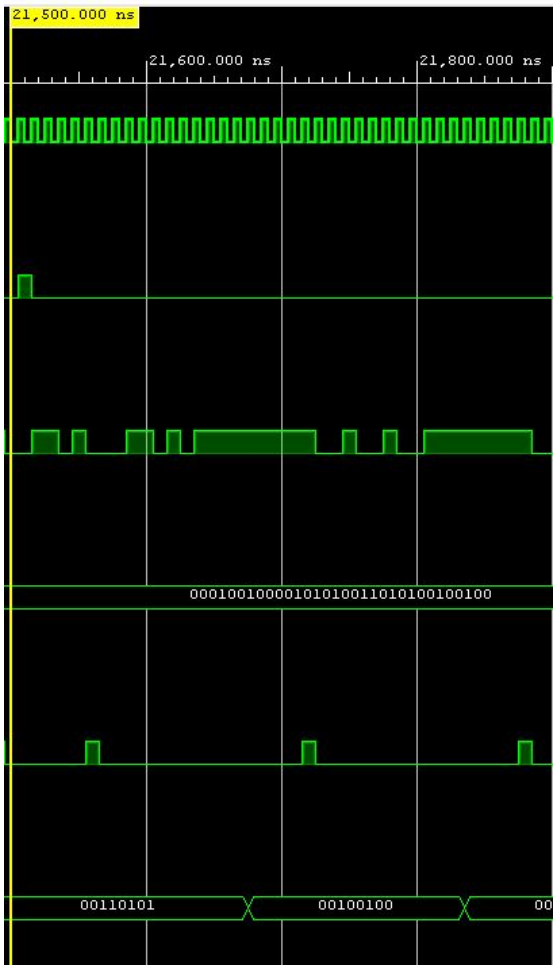

Fig. 8. Simulation of the module used to simulate a 1 - Bytes serial transmission. Starting from the top: CLK, RST, SO, $data\_bus$, LD_pdata, $pdata$.

```
always #(20500) begin

   data_bus = $random ;
   #1005 RST = 1'b1 ;
   #10 RST = 1'b0 ;

end // always

initial begin

   #100000 $finish ;

end // initial
```

Listing 3. Stimulus for the test of the module used to transmit the 32 bit of the i-th address of the ROM in serial mode.

The signal $pdata$ corresponds to the 1 Byte which will be loaded to the 8-bit shift register at the posedge of LD_pdata. As one can notice looking at figure 8, once the whole Byte has been extracted from the shift register, the module was built to assign the high logical value to SO up to the next LD_pdata posedge.

2) Transmission management: this module takes care of all the interface management between the FPGA and the DAC. Takes as input a clock, a reset and the LD_pdata signal from the output of the module just described, and has as output CS and $\overline{LD}$. It was thought as the last step of logic before the interface with the DAC, that's why the source code was written using a FSM (Finite State Machine), in order to take into account all the possible combinations of the ports involved in the transmission. The simulation, the stimulus, and the FSM code are presented in figure 9, listing 4 and listing 5 respectively.

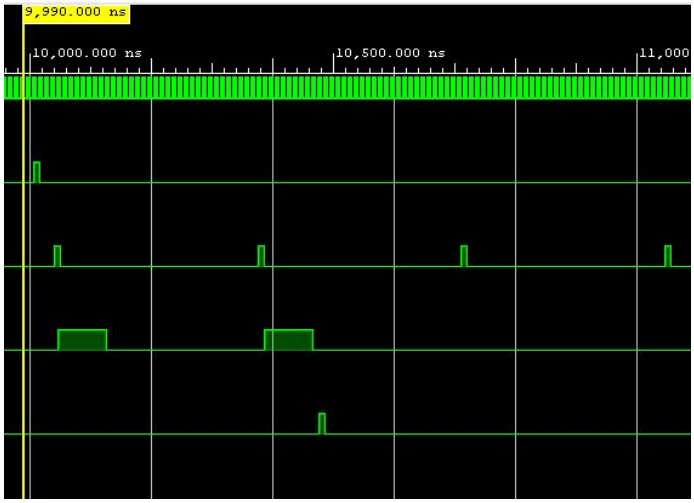

Fig. 9. Simulation of the module for the interface with the DAC. Starting from the top: CLK, RST, LD_pdata, CS $\overline{LD}$.

```
always #( 32*PERIOD ) begin

   #5  LD_pdata = 1'b1 ;
   #10 LD_pdata = 1'b0 ;

end // always

always #(10000) begin

   #5  RST = 1'b1 ;
   #10 RST = 1'b0 ;

end // always

initial begin

   #100000 $finish ;

end // initial
```

Listing 4. Stimulus for the test of the module used to manage the interface logic.

```
always @(*) begin

   case(STATE)

      default : NEXT_STATE <= END ;

      STOP1 : begin

         CS      = 1'b0 ;
         NOT_LD  = 1'b0 ;

         if ( LD_pdata ) begin

            en_counter  <= 3'd7 ;
            NEXT_STATE  <= WRITE1 ;

         end // if

         else

            NEXT_STATE <= STOP1 ;

      end
      //_____________________________________________
      //
      WRITE1 : begin

         CS      = 1'b1 ;
         NOT_LD  = 1'b0 ;

         if ( en_counter == 3'd7 ) begin

            NEXT_STATE <= STOP2 ;

         end // if

         else

      NEXT_STATE <= WRITE1 ;

      end
      //_____________________________________________
      //
      STOP2 : begin


         CS      = 1'b0 ;
         NOT_LD  = 1'b0 ;

            if ( LD_pdata ) begin

               en_counter  <= 3'd7 ;
         NEXT_STATE <= WRITE2 ;

            end // if

            else

               NEXT_STATE <= STOP2 ;

      end
      //_____________________________________________
      //
      WRITE2 : begin

         CS      = 1'b1 ;
         NOT_LD  = 1'b0 ;

         if ( en_counter == 3'd7 ) begin

            NEXT_STATE <= STOP3 ;

         end // if

         else

            NEXT_STATE <= WRITE2 ;

      end
      //_____________________________________________
      //
      STOP3 : begin

         CS      = 1'b0 ;
         NOT_LD  = 1'b0 ;

         NEXT_STATE <= SOC ;

      end
      //_____________________________________________
      //
     SOC : begin

         CS      = 1'b0 ;
         NOT_LD  = 1'b1 ;

            NEXT_STATE <= END ;

      end
      //_____________________________________________
      //
      END : begin

         CS      = 1'b0 ;
         NOT_LD  = 1'b0 ;

         if ( RST )

            NEXT_STATE <= STOP1 ;

         else

            NEXT_STATE <= END ;

      end

   endcase

end   // always
```

Listing 5. Combinatory part of the FSM used to implement the logic interface.

As one can see from figure 9, once the LD_pdata signal is high, then the CS remains high for exactly eight clock hits, thus enabling the first Byte transmission. After the LD_pdata has activated the second time, indicating that the second Byte has been loaded on the 8 bit shift register of the FPGA, then the logic repeats again. After this two time, the $\overline{LD}$ becomes high, recording the two Bytes on the DAC register. All the logic then can be restarted at the posedge of RST. Here it must be notice that not only the first two Bytes of the i-th ROM address are zero, but also the first four most significant bits of the third most significant Bytes. This is not a problem, because those are simply kicked out from the 12 bit shift register of the DAC during the data transmission.

G. The top level module

For the simulation of both the DAC and the FPGA, where all the data and all the transmission logic are located, it was implemented a top level module, which includes all the blocks described above. This module is actually very simple, it takes

as input a $100MHz$ clock and an enable, and has as output the DAC conversion analog signal of the data located on the FPGA ROM described in section I. The simulation and the stimulus code are presented in figure 10 and in listing 6, while the graph of the DAC analog output in function of the conversion step and in function of time are represented in figure 11 and 12 respectively.

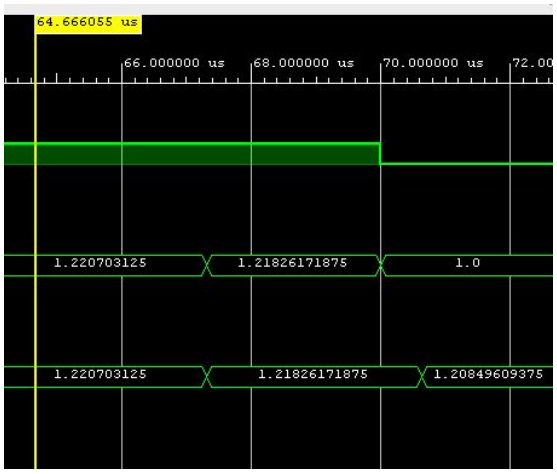

Fig. 10. Top level module simulation. Starting from the top: *dac_enable*, *Aout*, *Aout_expected*. The CLK signal is not reported for clarity.

```
   initial begin

      #70000 dac_enable = 1'b0 ;
      #30000 dac_enable = 1'b1 ;
      #30000 dac_enable = 1'b0 ;
      #30000 dac_enable = 1'b1 ;
      #100000  $finish  ;

   end

   always @(posedge NOT_LD) begin

      Aout_expected = 0.0 ;

      for ( i = 0 ; i <= 11 ; i = i + 1 ) begin

         real a = 2.0 ;

         for ( j = 11 ; j > i ; j = j - 1 ) begin

            a = a*2.0 ;

         end // for

         assign digit = I_data[i] ;

         Aout_expected = Aout_expected + Vref*(digit
    /a) ;

      end // for

      $display ( "\n\n D_out should be : %f V at
    time %d ns \n\n" ,  Aout_expected ,  $time  ) ;

   end // always
```

Listing 6. Stimulus for the top level module test.

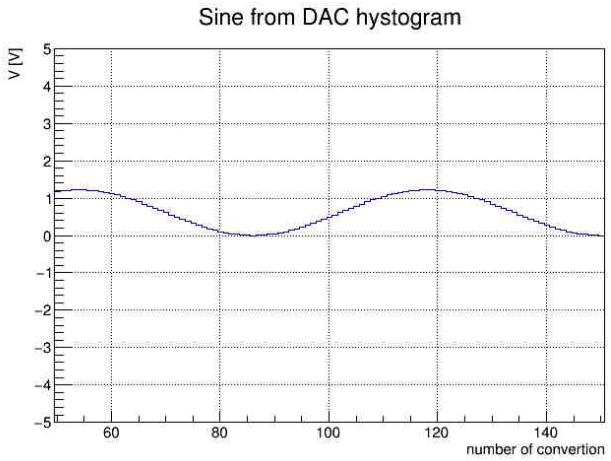

Fig. 11. The sine histogram in function of the conversion number (every $\overline{LD}$ posedge).

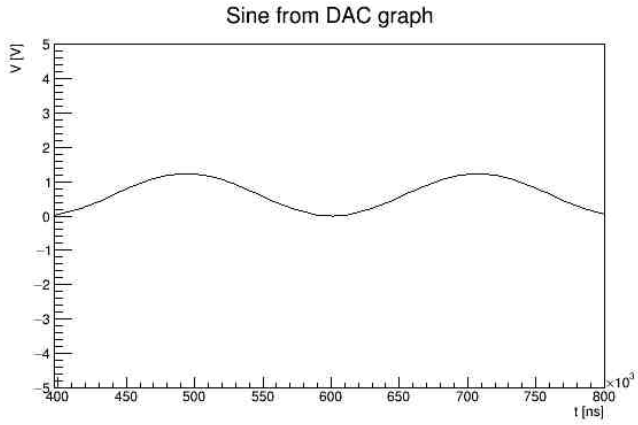

Fig. 12. The sine graph in function of time.