TECNOLÓGICO DE MONTERREY

Laboratory work: 2

Author:
Dr. Alejandro GALAVIZ

Research Group Name
School of Engineering and Sciences

May 2, 2019

Contents

List of Figures

List of Tables

List of Abbreviations

CAD	**C**omputer-**A**ided **D**esign
DAC	**D**igital to **A**nalog **C**onverter
DDS	**D**irect **D**igital **S**ynthesis
FPGA	**F**ield-**P**rogrammable **G**ate **A**rray
NCO	**N**umerically **C**ontrolled **O**scillator
PLL	**P**hase **L**ocked **L**oop
RTL	**R**egister **T**ransfer **L**evel
VHDL	**V**HSIC **H**ardware **D**escription **L**anguaje

Physical Constants

Speed of Light	$c_0 = 2.997\,924\,58 \times 10^8\,\mathrm{m\,s^{-1}}$ (exact)

List of Symbols

a	distance	m
P	power	W (J s^{-1})
ω	angular frequency	rad

Chapter 2

Synchronous counters, embedded memory, and digital generators

2.1 Practical application of the digital signal generator

The synchronous pulse counters are widely used for dividing the frequency at arbitrary values, in particular for timers implementations. Any timer is based on a counter. In the microcontrollers, timers are used to form time intervals. For example, in the interruption of a timer with a frequency of 1 KHz, the switching of the processes of the operating system can be performed. Or, after interrupting the timer with a certain periodicity, the processor can exit the sleep mode (with reduced power consumption), read the data from the sensors and transmit them to the network, and then go back to sleep mode until the next timer interruption. In general, frequency division is one of the highest priority task, since digital operations often require different operations with different frequency. This can be achieved by dividing the reference input frequency by different values and performing operations with binding to the received frequencies. Pulse counters are used for pulse-width modulation, which will be considered in the following laboratory work. Also, the synchronous counter is the main component of the well known **Numerically Controlled Oscillator (NCO)**, by which you can generate analog signals, or implement frequency and amplitude modulation, in particular for radio data transmission. Generating analog signals using NCO is called **Direct Digital Synthesis (DDS)** and is used in all modern digital signal generators.

2.2 Theoretical part

2.2.1 Conditional symbol and logic of the synchronous counter

The conditional graphical symbol of a synchronous counter is shown in Fig. 2.1. As can be seen from Fig. 2.1. a typical synchronous counter has one-bit synchronization and asynchronous reset inputs (in this case it's i_clk and i_rst_n inputs) and a multi-bit output that displays the counter content (in this case it is an output o_counter).

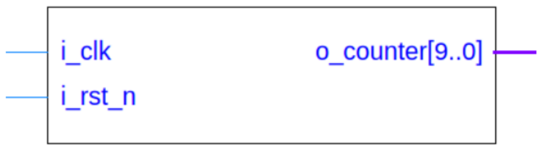


FIGURE 2.1: Conditional graphic symbol of a synchronous counter with an input asynchronous reset.

The main function of the pulse counter is to count the pulses applied with the synchronization input. For counters that count up, after each active front on the synchronization input, the content of the counter increases by 1. For counters that counting, after each rising edge at the input of the synchronization, the content of the counter decreases by 1. The counters whose contents are binary is called binary counters. Binary counters are most commonly used in digital circuitry. In addition to binary counters there are also one-hot counters and counters in the Gray code counters. Further in this laboratory work will considering only binary counters. A counter with a custom code will be considered in the next laboratory work.

When the active logic level is applied to the asynchronous asserted input, the value of the counter is shifted to 0. The active level of asynchronous throttling in many counters is the logical zero. However, by constructing a schematic of a counter in a schematic editor, or by describing such a design in the Verilog language, you can, at your own discretion, determine the active logical level of asynchronous assertion input signal. Some counters may also have an input of synchronous throttling. If the active logic level is given to the synchronous drop input, after the input of the active active front to the synchronization input, the content of such a counter will go to 0. In the counter bit number of N bit can count from 0 to $2^N - 1$. If the N-bit N counting up contains a number $2^N - 1$, the next momentum on the input synchronization will lead to overflow of the counter (counter overflow) and the occurrence in it of the value 0. In other words, in such a counter there will be 2^N stable states (from 0 to $2^N - 1$). The number of states that a counter can accept is called its account module M. For some counters, you can specify their account module with a separate input. This feature is often used in controlled frequency dividers, which will be considered in the next laboratory work. With the counter with the module of the account M you can create a frequency divider on M. Some counters have a content download input that allows you to load a certain value in the counter from which the account will continue.

2.2.2 Synchronous T-trigger

Before studying the circuit of the synchronous binary counter, one needs to understand the T-trigger synchronous input, the conditional graphic designation of which is given in Fig. 2.2. The simplest T-trigger has a synchronization input T, which is fed by a clock signal.

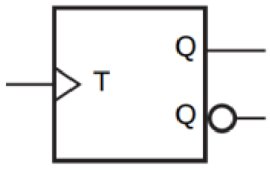


FIGURE 2.2: Conditional graphic symbol T-trigger synchronous.

At output Q there is a trigger value (0 or 1). For each active front of the synchronization input, the contents of the T-trigger are inverted - Fig. 2.3.

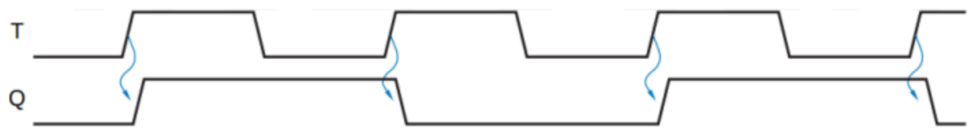


FIGURE 2.3: Timeline of signal changes in the T-trigger asynchronous along the rising edge.

As can be seen from Fig. 2.2, the simplest T-trigger works as a frequency divider at 2, since the pulse signal output at Q is twice as long as the signal signal period at the input T (correspondingly, the output frequency is 2 times lower). The simplest T-trigger can be built on the basis of the D-trigger synchronous on the front - Fig. 2.4. As shown in Fig. 2.4, the values of the inverse trigger output are fed to the data input of the trigger D. If the trigger stores a value of Q = 1, QN will have 0 on its inverse output, and this 0 will be fed to the data input. So, on the next active front, this 0 will be written to the trigger and 0 will be output at Q. There is also a T-trigger with the input of the signal EN (enabling). Conditional graphic designation of such a trigger is shown in Fig. 2.5, and timimng signal chart of work - in Fig. 2.6

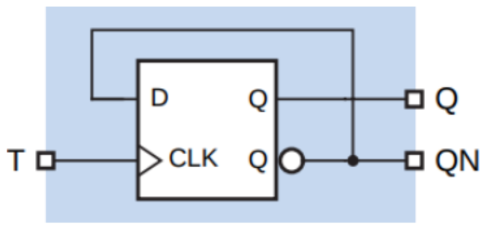


FIGURE 2.4: Symbol scheme of a synchronous T-trigger.

In such a T-trigger, the output Q value is inverted on the active front of the sync signal at the T input only provided that the active logical level is present at the EN input. If the input of the EN is not active logical level - after the arrival of the active front of the state of the trigger does not change.

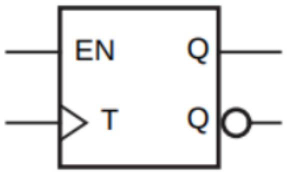


FIGURE 2.5: Symbol conditional graphic designation of the T-trigger synchronous on the rising edge with the input of permission with EN (Enable).

This is clearly seen from Fig. 2.6. Typically, the EN input has an active high level.

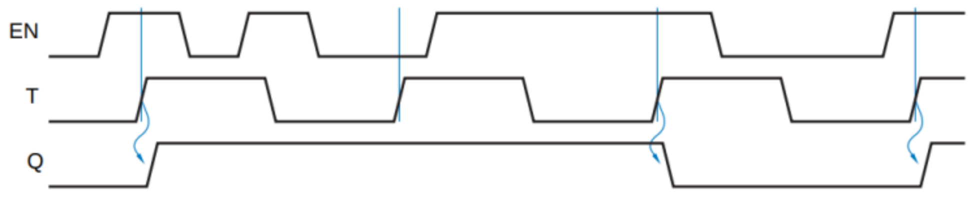


FIGURE 2.6: Conditional graphic designation of the T-trigger synchronous on the rising edge with the input of permission EN (Enable).

The scheme of the T-trigger synchronous on the rising edge with the input of permission statement EN is shown in Fig. 2.7

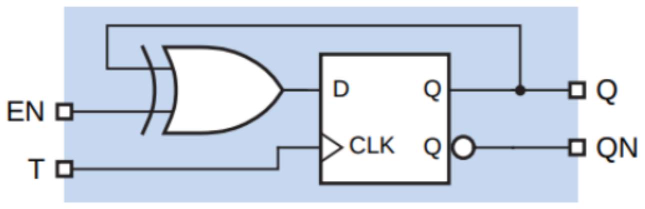


FIGURE 2.7: Scheme of a T-trigger synchronous with rising edge using the input of permission EN.

2.2.3 Scheme and principle of the synchronous counter

Consider the timing chart of the counter, counting up - Fig. 2.8.

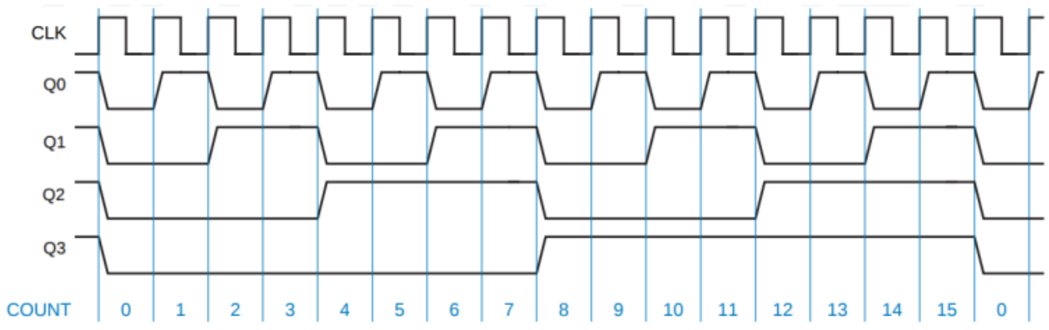


FIGURE 2.8: Timing Chart Signals Changes with a Synchronous Count, Counting Up.

From the diagram in Fig. 2.8, it is evident that each subsequent digit is inverted at the moment of receipt of the active (forward) front of the CLK sync signal, provided that all previous digits take values of units. Taking into account the above observation, it is not difficult to construct a circuit of a synchronous counter, counting upwards, on the basis of the non-fronted T-flip-flops with the input of the permission Fig. 2.9.

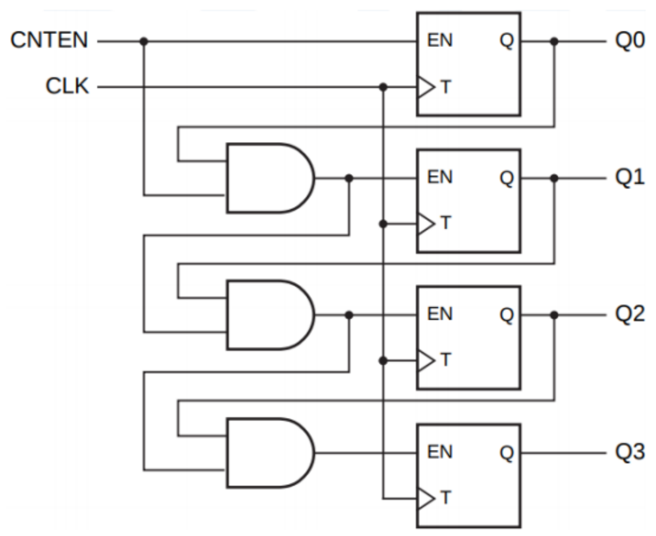


FIGURE 2.9: Scheme of synchronous 4-digit counter upwards with sequential signal permission EN.

As can be seen from Fig. 2.9, at the EN input of each T-trigger a XOR will be used. 1 only provided that the outputs Q of all previous digits will be unity. Such a scheme is easy to scale to a larger bit. In this case, the delay of forming the signal EN for the older one arrangement of the counter (in this case, Q3) will be equal to the amount of delay of 3 logic elements AND. You can reduce the delay of the formation of signals EN using the circuit as shown in Figure 2.10. Note that the CNTEN input, which allows the counter count. In case if CNTEN = 0, at the inputs of EN T-triggers there is 0 and the state of the triggers after the incoming active front does not change. If CNTEN = 1, the counter begins to count the pulses sync input signal.

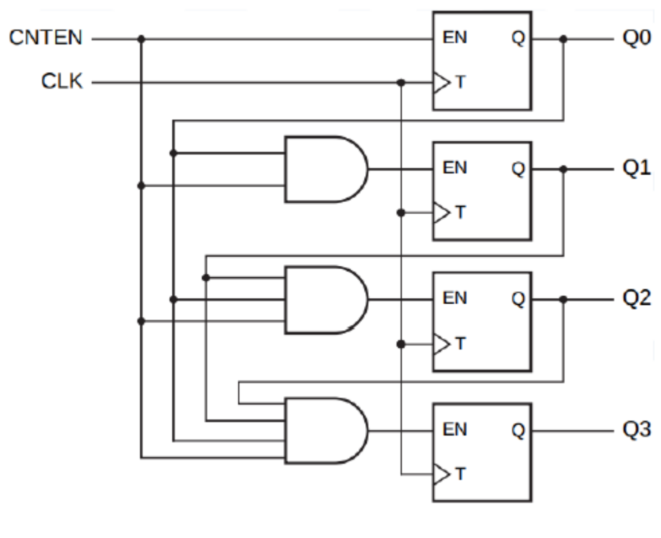


FIGURE 2.10: Scheme of synchronous 4-digit counter up to parallel signal formation EN.

Note that the frequency of the pulse signal from Fig. 2.2 in each subsequent one the discharge is twice smaller than in the previous digit. That is why a regular counter can be used to divide the frequency by 2^N. To construct a counter counting down the log inputs. elements AND must be to provide data not from direct outputs

Q, but from inverse NQ outputs (or inverted ones value Q using inverters). If necessary, you can create a counter with managed direction of the account (up/down), if you choose between direct and inverted value Q using a multiplexer.

2.2.4 Description of Synchronous Clock in Verilog

A description template in Verilog for a counting counter listed in Listing 2.1. As can be seen from listing 2.1. on the front front i_clk, if the signal i_rst_n is not active, to the value o_counter is added 1 and the result of the amount is written to multi-bit variable o_counter t idp reg. Since this logic corresponds to 7 the counter, the CAD synthesizes the specified code template in the counter. The size of the count is determined by the bit number of the o_counter variable. Similarly, you can describe a counting counter, but instead of the operator "+" you must use the operator "-". The operator "+" in this case is synthesized into the adder.

Listing 2.1 - A synchronous counter topology pattern template in Verilog language

```
module counter(i_clk, i_rst_n, o_counter);

input                   i_clk;
input                   i_rst_n;
output reg [9:0]        o_counter;

always @(posedge i_clk, negedge i_rst_n) begin
    if(~i_rst_n) begin
        o_counter <= 0;
    end else begin
        o_counter <= o_counter + 1'b1;
    end
end

endmodule
```

The result of the synthesis of this source code in Quartus Prime is shown in Figure 2.11. The specified synthesis result can be viewed independently in the RTL Viewer. Information on using the RTL Viewer is provided in lab work Lab0.

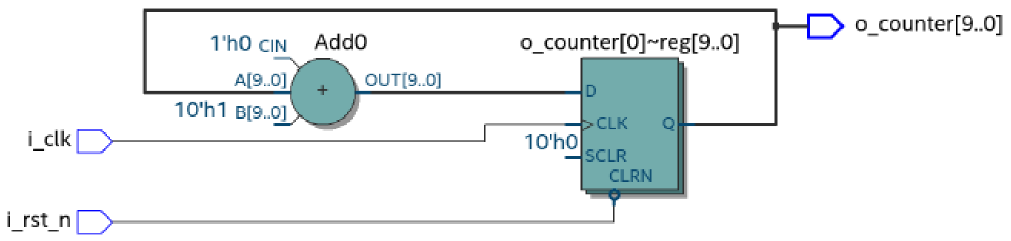


FIGURE 2.11: The result of synthesis of a synchronous counter from listing 2.1.

Note that the expression on the right side of the non-blocking assignment operator is synthesized into a combinational schema, the output of which is connected to the input of the o_counter data, which is always described in the block. The scheme in Fig. 2.11 is another embodiment of the implementation of a synchronous double counting, using a parallel register and adder, which adds to the content of the register unit and the result of the sum on the active front of the signal synchronization is written in the register.

2.2.5 Description of a synchronous counter using a source-code in Verilog language

The Verilog description template for the synchronous counter is shown in Listing 2.2.

Listing 2.2 - A description of the counter with a source-code in Verilog language

```
module counter(i_clk, i_rst_n, i_load, i_data, o_counter);

input                   i_clk;
input                   i_rst_n;
input                   i_load;
input       [9:0]       i_data;
output reg  [9:0]       o_counter;

always @(posedge i_clk, negedge i_rst_n) begin
    if(~i_rst_n) begin
        o_counter <= 0;
    end else begin
        if (i_load)
            o_counter <= i_data;
        else
            o_counter <= o_counter + 1'b1;
    end
end

endmodule
```

The result of the synthesis of Listing 2.2 can be seen in Figure 2.12. As you can see, the scheme is very similar to the counter from Fig. 2.11, but in Fig. 2.12 there is an additional multiplexer, by which it is possible to connect the input of the case data either to the i_data input, or to the adder output.

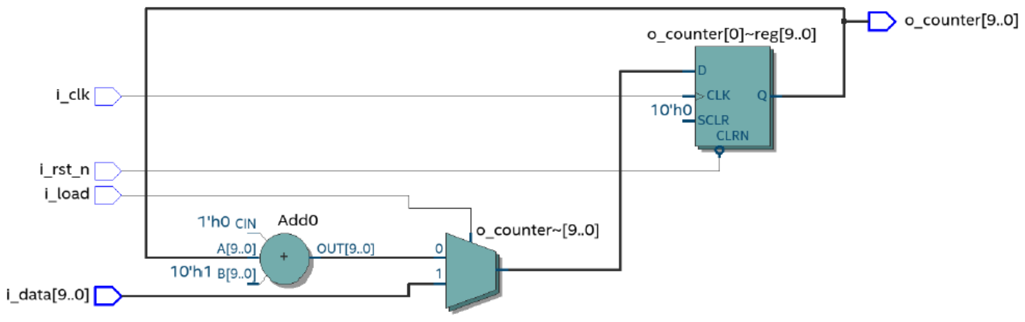


FIGURE 2.12: The result of the synthesis for the counter from Listing 2.2 in the RTL Viewer

In Figure 2.12, the multiplexer has one-bit address input and two multi-bit inputs with numbers 0 and 1. Enter the address of the multiplexer connected to the i_load input. If the one-digit address of the input takes the value 0, the output of the multiplexer is transmitted data from the input with the number 0. If the one-bit address input takes value 1, the output of the multiplexer transmit data from the input number 1. More in detail, the multiplexer will be considered in one of the following lab works. Thus, when the i_load input takes a log value. 1, in the register that stores the counter, after the active front front at the i_clk sync input, the i_data input data is recorded. If at the input i_load there is a log. 0, for each active front edge at the input of the synchronization, the content of the counter increases by 1.

2.2.6 Arithmetic Operators in Verilog (+, -, *, /,%)

In Lists 2.1 and 2.2, an arithmetic operator + languages Verilog was used to implement the addition in the counter. In addition to adding the operator, Verilog contains arithmetic operators:

TABLE 2.1: Arithmetic sintax operators in Verilog

Operation	**Symbol**
Subtraction	-
Multiplication	*
Division	/
Modular division	%

If the above arithmetic operators are used to describe the digital circuits, operators are synthesized into combinational schemes that perform the required function (for example, the operator + is synthesized into the adder). The operators *, / and % are synthesized in complex combinational schemes that have a significant delay and require significant hardware costs (a large number of logical elements) for implementation. Therefore, these operators need to be used wisely. Later in laboratory work, it will be consider the schemes and source code for operations multiplication / division for several cycles with low hardware costs.

2.2.7 Digital-to-analog converter (DAC) on resistors

This material is needed to understand the following sections of laboratory work. Digital-to-Analog Converter (DAC) is a device for converting a digital code into a voltage. The digital input of the DAC is given by the number of digits N, and at the output of the DAC there is a voltage corresponding to the digital code at the input. Also, the V_{ref} reference voltage is supplied to the DAC. The voltage on the DAC output is directly proportional to the digital code at its input. The zero at the digital input of the DAC corresponds to zero output voltage. The maximum digital code at the input of the N-bit DAC ($2^N - 1$) corresponds to the maximum possible output voltage value, which is called Reference Voltage (Reference Voltage, V_{ref}) and is often equal to the supply voltage. The voltage dependence formula V_o on the output of the DAC from the digital N-bit code D on the input and the reference voltage V_{ref}:

$$V_o = \frac{V_{ref}}{2^{N-1}} \cdot D \tag{2.1}$$

Conditional graphic designation of the DAC is shown in Fig. 2.13. The dependence of the voltage on the output of a 4-digit DAC from the digital code at its input is shown in Fig. 2.14. For neighboring digital codes at the input of the DAC voltage at its output will differ in magnitude $h = \frac{V_{ref}}{2^{N-1}}$. The larger the digit of the DAC, the less will be h.

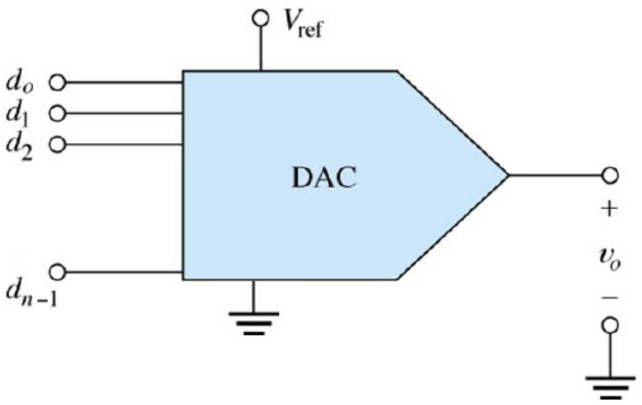


FIGURE 2.13: Conditional graphic assigments of DAC

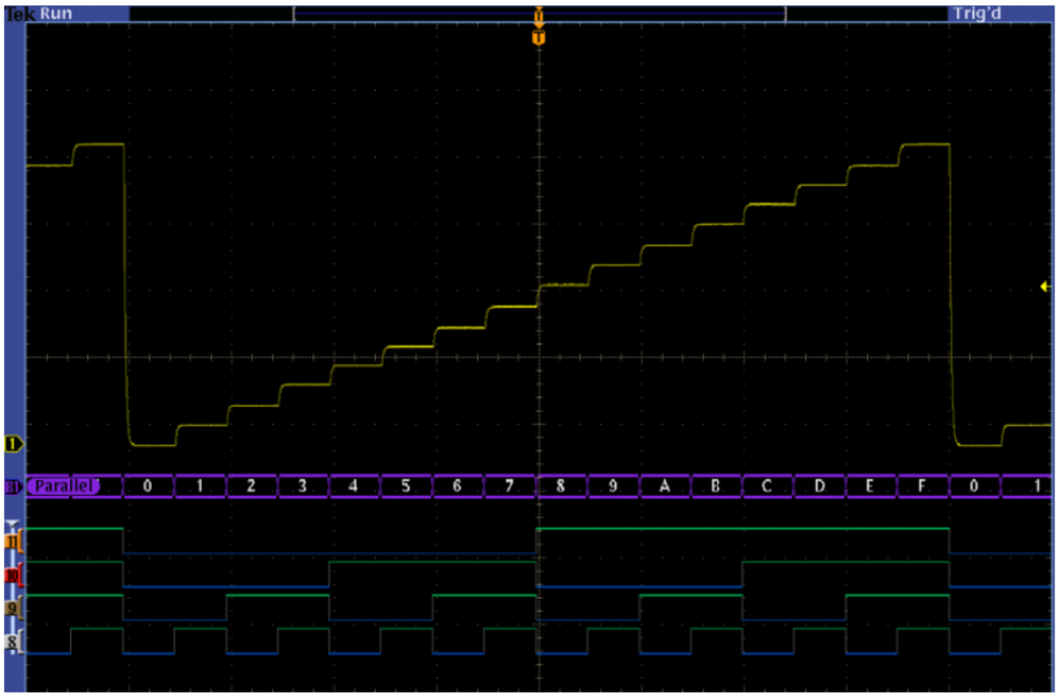


FIGURE 2.14: Dependence of voltage at output of 4-bit DAC from the digital code at its entrance

Most widely used DACs have a bit in the range of 8-16 bit. However, sometimes the DAC uses both very lower bits (4 bits) and higher bit length (24 bits). Professional DACs are issued in the form of discrete chips, but for many tasks suitable DAC is composed of resistors. There are two DAC circuits on resistors: R-2R DACs and DACs with it ighted resistors. The R-2R DACs are very common and can be found more technical compliance in the following link [2.1]. Hoit ver, in laboratory work, it will be used DACs with it ighted resistors, because such a design has a 4-digit DAC VGA interface on the board of the Intel-Altera FPGA, which it will use in generator signals. Therefore, consider more DACs with it ighted resistors. The schematic of a 4-bit DAC with it ight resistors is shown in Figure 2.15.

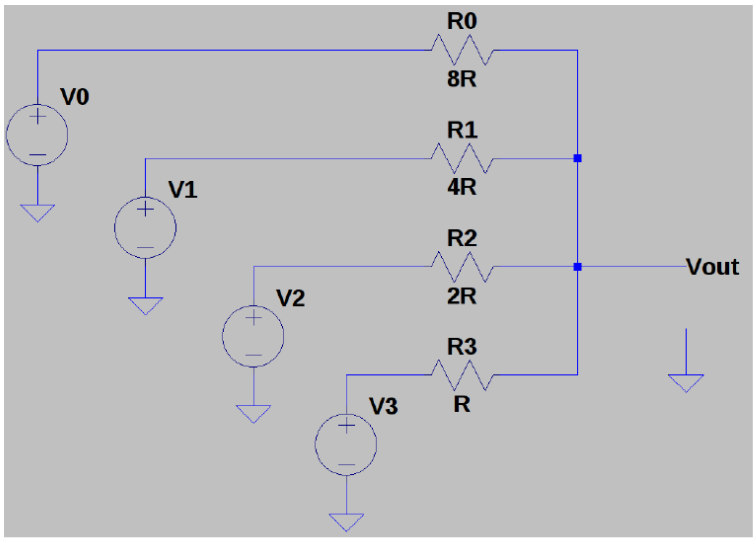


FIGURE 2.15: Scheme of 4-digit DAC with it ight-value resistors

In the DAC in Fig. 2.15, the input digital code is given by voltage sources that simulate logic levels. Any such source at the input may issue or logic 0 (0 volts), or logic 1. (V_{ref} = V_{dd} Volt, where V_{dd} is the supply voltage). The voltage of each source with the number i can be represented by the formula $V_i = V_{ref} \cdot D_i = V_{dd} \cdot D_i$, where D_i represents the *i*-th digit by digit of the digital input on the DAC input and can accept values of 0 or 1. Let us write the first *Kirchoff law* for currents through resistors:

$$\sum_{k=1}^{n} I_k = I_0 + I_1 + I_2 + I_3 + \ldots + I_n = 0 \tag{2.2}$$

Also valid for complex circtuits

$$\sum_{k=1}^{n} \tilde{I}_k = 0 \tag{2.3}$$

From *Kirchhoff's* second law, the voltage on the *i*-th resistor V_{Ri} can be represented by the formula (where V_i is the voltage of the *i*-th input source, and V out is the voltage at the output of the DAC):

$$V_{Ri} = V_i - V_{out} \tag{2.4}$$

Substituting formula (2.4) in (2.2) for each and from 0 to 3 and applying the Ohm law, is obtained:

$$\frac{V_0 - V_{out}}{R_0} + \frac{V_1 - V_{out}}{R_1} + \frac{V_2 - V_{out}}{R_2} + \frac{V_3 - V_{out}}{R_3} = 0 \tag{2.5}$$

Regrouping the terms is obtained:

$$V_{out} = \frac{\frac{V_0}{R_0} + \frac{V_1}{R_1} + \frac{V_2}{R_2} + \frac{V_3}{R_3}}{\frac{1}{R_0} + \frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}} \tag{2.6}$$

Given that $R_3 = R, R_2 = 2 * R, R_1 = 4 * R, R_0 = 8 * R$, it substitute these values in (2.6) and get (D - digital code at the DAC input at bit N bit, D_i - discharge this digital code):

$$V_{out} = \frac{\frac{V_0}{8R} + \frac{V_1}{4R} + \frac{V_2}{2R} + \frac{V_3}{R}}{\frac{1}{8R} + \frac{1}{4R} + \frac{1}{2R} + \frac{1}{R}} = \frac{V_0 + 2 \cdot V_1 + 4 \cdot V_2 + 8 \cdot V_3}{15} =$$

$$\frac{V_{dd}}{15} \cdot (D_0 + 2 \cdot D_1 + 4 \cdot D_2 + 8 \cdot D_3) = \frac{V_{dd}}{15} \cdot D \tag{2.7}$$

2.2.8 Generator of saw-type voltage based on clock and DAC

The simplest saw-tension generator can be created by connecting the output of the binary counter to the digital input of the DAC. Changing the signals on the output of the clock and the voltage at the output of the DAC in such a scheme is shown in Figure 2.14. If the DAC bit is large enough (10-16 bit), the voltage change corresponding to the neighboring digital codes will be very small and almost imperceptible to the eye.

2.2.9 Description of the constant ROM in Verilog language Arrays

To create a sinusoidal generator, it will neccesary a read only memory (ROM) component. Let us consider how to describe such memory in Verilog language. Conditional graphic designation of ROM's permanent memory is shown in Fig. 2.16.

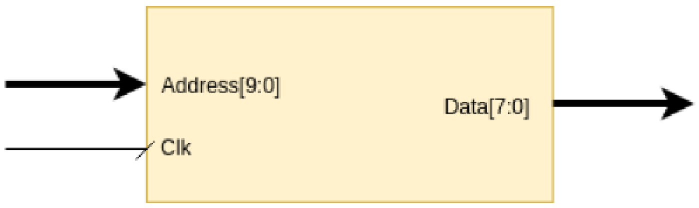


FIGURE 2.16: Conditional graphic designation of a permanent memory (ROM) component

Permanent memory is designed to save data. Memory consists of cells. Each cell stores the data word of a certain bit. Permanent memory has N-bit address input (in Figure 2.16, the address of the address is called Address, N = 10) and M-bit data output (in Fig. 2.16 output data called Data, M = 8). Some instances of permanent memory may have input synchronization, often referred to as Clk. Permanent memory with a N-bit address of an address contains 2N cells, with numbers from 0 to $2^N - 1$. At the output of the permanent memory, the contents of the cell with the number appearing on the address of the address appears. If the permanent memory does not have an input synchronization, the appearance of data on the memory output occurs with a certain delay after filing a new address admission. This delay is called read-write delay from memory - Fig. 2.17. If the permanent memory has an input synchronization, the new data appears on the output with a slight delay after the input of the address of the new memory cell number and the receipt of the active front on the synchronization input - Fig. 2.18.

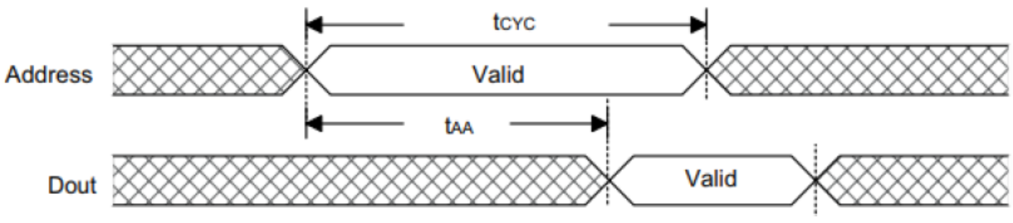


FIGURE 2.17: Timestamp reading from the post. ROM memory (t_{AA} - read delay)

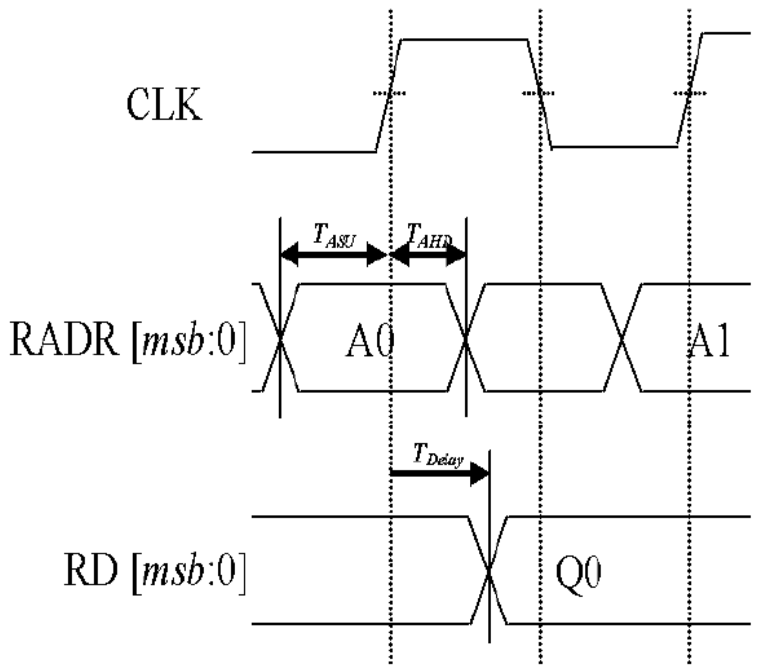


FIGURE 2.18: Clock timing reading from a permanent ROM memory with CLK sync input. New data (cell content with RADR address) appear on RD output from T delay after active CLK front

Some instances of permanent memory have CS-access permission (Chip Select). In the case when the CS has an active level, the ROM memory works as above. If on CS there is an inactive log. level, the memory data output is disconnected from the internal driver, which creates the output logical levels and goes to the so-called third state. The Verilog permanent memory description template is listed in Listing 2.3. Using the specified code, the Quartus Prime CAD replaces it with the RAM/ROM component. This source code leads to the creation of a permanent ROM memory with the sync. This is the most common ROM description template for FPGA. To describe the memory in the Verilog language, use arrays. You can define an array in Verilog using the following code:
reg [7 : 0] rom[1023 : 0];
First, specify the type and size of cell arrays (in this example, cell arrays have the type **reg** and bit 8 bit). The following is the name of the array (in this example, the array has the rom name). After the name of the array in square brackets determine the number of array cells. The right index specifies the number of the initial cell (in this for example, this is 0). The left index specifies the number of the final cell of the array (in this example it is 1023). So in this example, the array has 1024 cells with numbers from 0 to 1023. Verilog language arrays can have the type **reg**, **wire**, **integer**.
Listing 2.3 - The description of the permanent memory (ROM) in Verilog language.

```
module rom_mem(i_clk, i_addr, o_data);

input                i_clk;
input        [9:0]   i_addr;
output reg   [7:0]   o_data;

reg [7:0]   rom[1023:0];

initial $readmemh("rom_init.hex", rom);

always @(posedge i_clk)
    o_data <= rom[i_addr];

endmodule
```

In the Listing 2.3 for each rising edge of the i_clk sync signal, the contents of the component with the i_addr number of the rom array are written to the o_data variable. The result of the synthesis of the source code from Listing 2.3 can be shown in RTL Viever (Fig. 2.19).

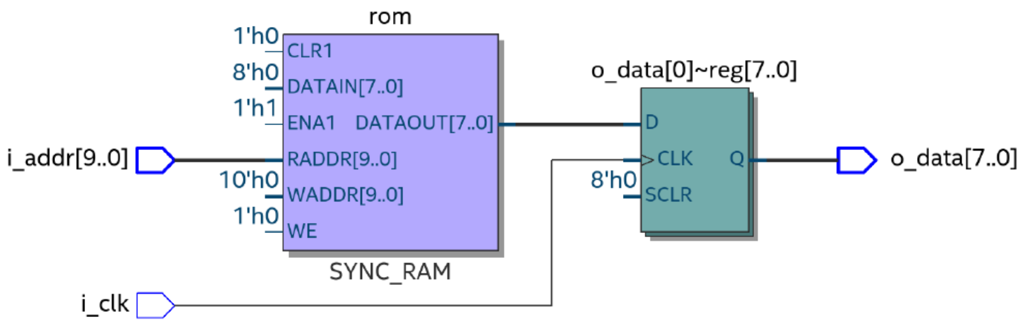


FIGURE 2.19: RTL Vieit r of synthetized block design for source code from Listing 2.3

As shown in the Figure 2.19, the source code of Listing 2.3 corresponds to static memory without a synchronization input, to the output of which the input of parallel register data is connected, which is tact from the sync input i_clk. To initialize arrays in the Verilog language, use the system function $readmemh. The system function $readmemh initializes an array of data from a text file. System functions Verilog supports all modern synthesizers and simulators. If you call the system function $readmemh at the beginning of the **initial** block, the array will be initialized at zero moment of time, immediately at the beginning of the simulation. In the case of synthesis for FPGA, calling the $readmemh function at the beginning of the **initial** block leads to adding memory initialization data to the FPGA configuration file. In this case, after the configuration in the FPGA, an object is created permanent ROM memory, whose cells are initialized by values from a file that was passed as an argument to $readmemh. The structure of the text file from which the content of the array is initialized is very simple. Each line of the text file contains the content of the Verilog cell array (or the memory that is being synthesized from that array). The starting line corresponds to the original cell of the array (memory). The final line corresponds to the final cell of the array (memory). If the array is initialized by the function $readmemh, the data in a text file must be represented in the sixteenth format (00, 11, 35, 7C, 8A, BC, AB, FF, etc.). If an array is initialized by the function

$readmemb, the data in a text file must be presented in binary format (00000000, 00010001, 00110101, 01111100, etc.). If only certain cell arrays are to be initialized, they use a slightly more complicated structure of the initialization file using the cell addresses for which the data is assigned in each line of the file. The cell address is written before the cell's value in the @address format. For example, **@7 FA** means that the cell with the address **7** need to initialize the **FA** value. The cells for which the values in the initialization file are not defined are not initialized. Example of the contents of the rom_init.hex file to initialize an array using the system function $readmemh:

```
00
7C
8A
//...The contents of the remaining cells
AB
FF
```

The memory inside the FPGA can be implemented both with the help of classical components (truth cards, synchronous triggers), and with the help of the so-called block memory, which is part of the modern FPGA. Realization of memory with the use of truth and synchronous triggers does not result in the rational use of chip resources. On the other hand, even the younger and less expensive family of FPGAs (Cyclone V, Max10) contain megabytes of blocked memory. More serious FPGAs can contain dozens of megabytes of memory. The amount of post-synthesis of blocked memory can be vieit d in the Compilation Report window, under the section "Total block memory bits". Placement of memory blocks on a FPGA crystal can be vieit d in the Chip Planner - Fig.2.20. How to call Chip Planner is told in one of the past laboratory work.

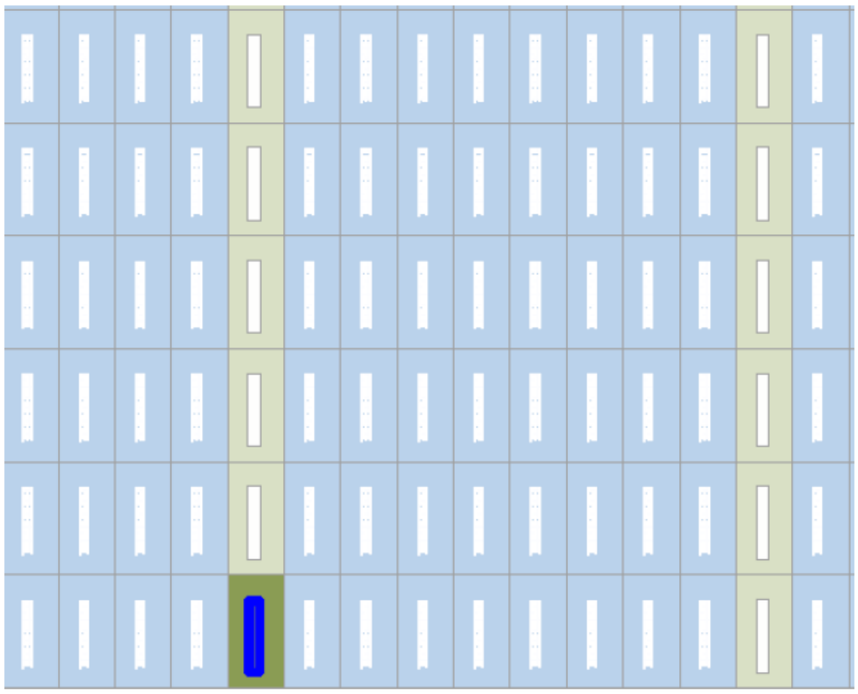


FIGURE 2.20: FPGA logic memory blocks (light green bars)

If you click on into the property of a separate logic memory block, you can see its structure - Fig 2.21. Note that the memory blocks contain D-triggers that are synchronous to the front for data on memory outputs, so general-purpose triggers are not used for this purpose.

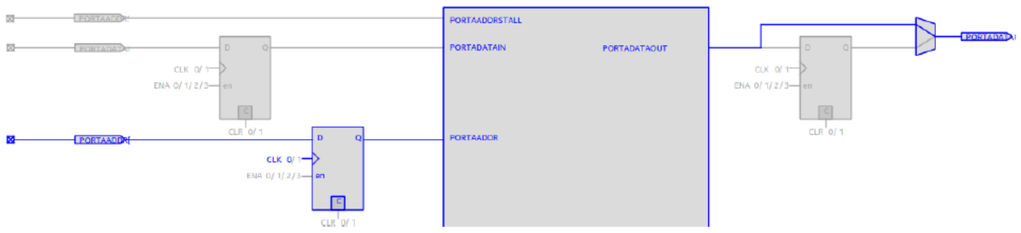
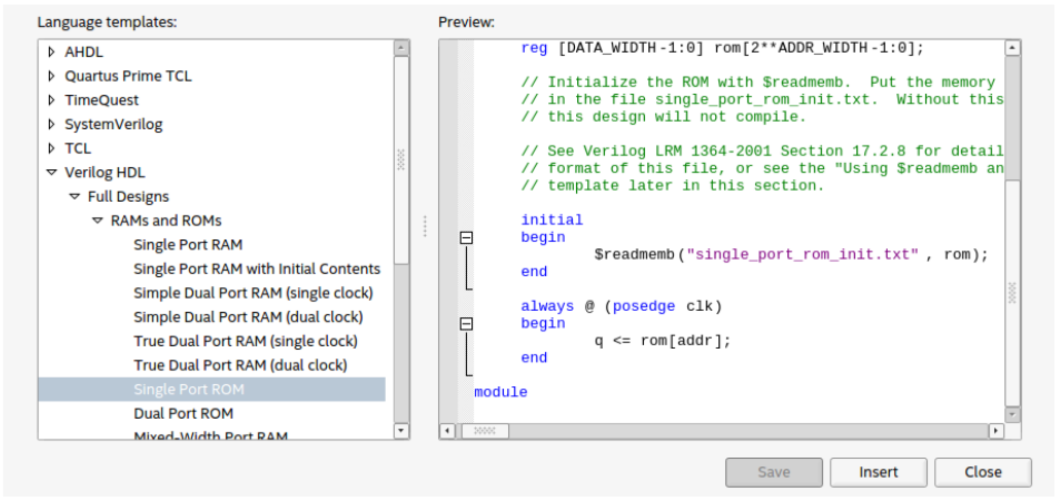


FIGURE 2.21: A separate memory module is displayed in the Intel FPGA Chip Planner

2.2.10 Verilog templates for description of digital circuits in Quartus software

Several times it has been mentioned several times that there are certain Verilog code templates for describing the various components of the digital chip and for correct synthesis it is necessary to use the correct Verilog code templates. Examples of such Verilog code templates can be found in the software **Intel® Quartus® Prime**.

To start a new project consider open a template using a Verilog file in Quartus Prime, and select Edit → Insert Template from the menu item. A window opens with code templates in different languages (Verilog, System Verilog, VHDL, AHDL). For the Verilog language, select Verilog HDL, and on the Full Designs tab can be seen the Verilog templates for describing the various components of the digital chip - Fig. 2.22.



```
	reg [DATA_WIDTH-1:0] rom[2**ADDR_WIDTH-1:0];

	// Initialize the ROM with $readmemb.  Put the memory
	// in the file single_port_rom_init.txt.  Without this
	// this design will not compile.

	// See Verilog LRM 1364-2001 Section 17.2.8 for detail
	// format of this file, or see the "Using $readmemb an
	// template later in this section.

	initial
	begin
		$readmemb("single_port_rom_init.txt", rom);
	end

	always @ (posedge clk)
	begin
		q <= rom[addr];
	end

endmodule
```

FIGURE 2.22: Verilog templates for custom code to describe different digital circuits components

2.2.11 Digital signal generator: sinusoidal

To generate a sinusoidal signal, it is necessary to give on the DAC digital codes of the values of the sinusoid which the DAC converts to voltage at the same time intervals

T_d (sampling period). The frequency of issuing digital codes to the input of the DAC is called the sampling rate $F_d = \frac{1}{T_d}$. The described idea is illustrated in Fig. 2.23.

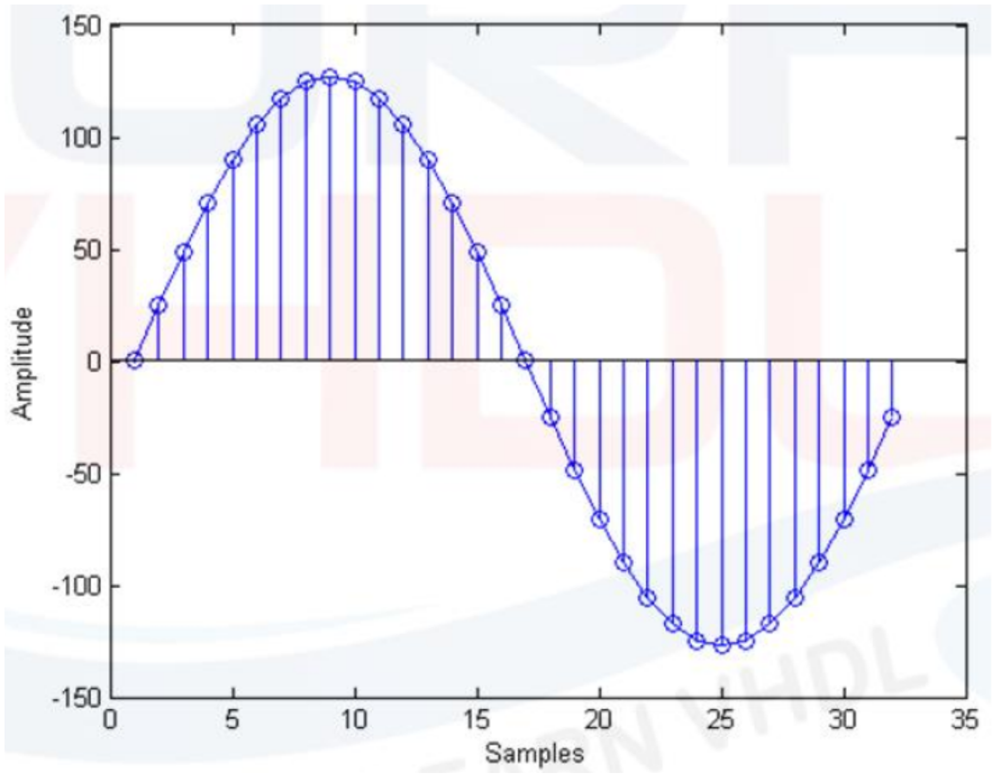


FIGURE 2.23: The sinewave (solid line) and its value at discrete time points (round markers) with the sampling period T_d (time interval betit en two round markers)

In fact, in Fig. 2.23 a simplified approach is presented. In a real scheme, it is necessary to take into account several features. **First**, a N-bit digital code at the DAC input can accept a limited number of values from 0 to $2^N - 1$. In total $2N$ values. This means that the value of the output voltage of the DAC will be rounded to one of these values. Recall that a specific value of the voltage at the output N-bit DAC can be calculated by formula (2.1).

Since there is a permanent digital code on the DAC input at the T_d input, this means that at the output of the DAC, during the time interval T_d, there will be a constant voltage depending on the digital code according to formula (2.1), which means that the voltage values at the output of the DAC will have the form of steps - Fig. 2.24. **Secondly**, the DAC based on resistors can only produce positive voltage. Therefore, you need to add a constant component equal to the amplitude of the sinusoidal signal to obatain the signal as shown in the graph of the Fig. 2.23. The result of this bias is shown in Fig. 2.24. Now the sinusoid accepts only positive values.

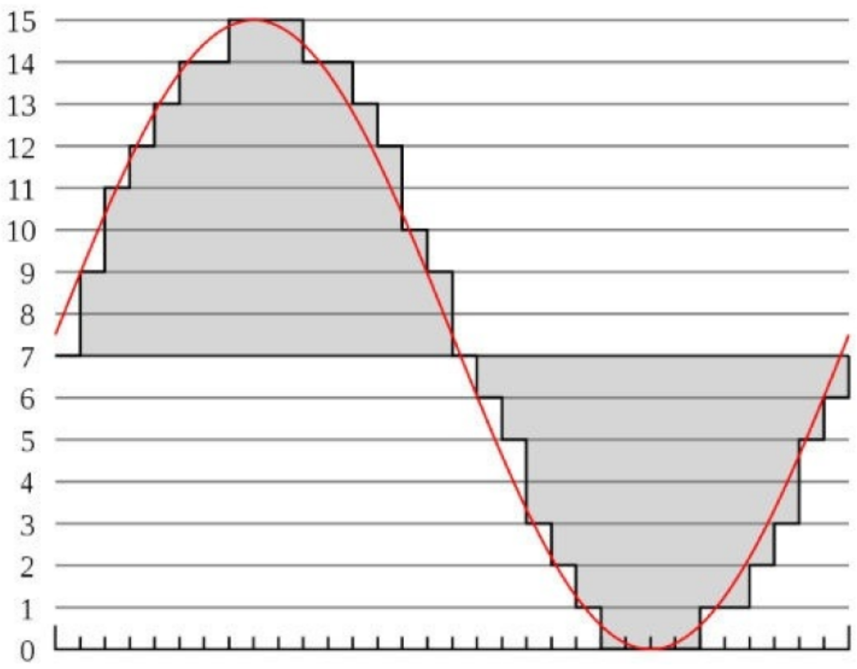


FIGURE 2.24: The value of the sinewave voltage at the output of the 4-digit DAC (the output voltage of the DAC has 16 values corresponding to the digital codes from 0 to 15) with the sampling rate F_d (F_d $=\frac{1}{T_d}$, where T_d - the time interval betit en issuing to the DAC of the neighboring digital codes)

The logic of the digital generator of the sinusoidal signal described above is quite simple to implement on the basis of the counter of pulses and microcircuits of permanent memory. The output of the pulse counter is connected to the input of the permanent memory address. In logic cells of permanent memory in the form of digital codes the values of the sinusoid are stored. The output of the permanent memory data is connected to the digital input of the DAC. The described structure is depicted in Fig. 2.25.

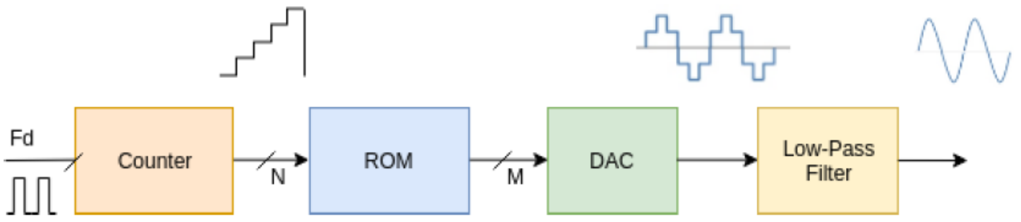


FIGURE 2.25: The block diagram of a digital generator for a sinewave voltage

For each active front of the synchronization signal (having a F_d frequency), the content of the counter that determines the address of the fixed memory cell is increased by one (a new address is formed) and a digital code stored in the cell of the permanent memory appears on the output of the permanent memory. such address. The digital output from the permanent memory is fed to the DAC, and the DAC forms a new voltage value of the sinusoid at its output. The source code for the digital generator of the sinusoidal signal in the Verilog language is listed in Listing 2.4.
Listing 2.4 - The source code of the sinusoidal generator in Verilog language.

```
module sin_gen(i_clk, i_rst_n, o_dac);

input           i_clk;
input           i_rst_n;
output  [3:0]   o_dac;

reg     [3:0]   sin_table_rom[1023:0];
reg     [9:0]   phase;
reg     [3:0]   dac_data;

assign  o_dac = dac_data;

initial $readmemh("sin_table_4bit.hex", sin_table_rom);

always @(posedge i_clk)
    dac_data <= sin_table_rom[phase];

always @(posedge i_clk, negedge i_rst_n) begin
    if(~i_rst_n) begin
        phase <= 0;
    end else begin
        phase <= phase + 1'b1;
    end
end

endmodule
```

The result of the synthesis of the source code from Listing 2.4 is shown in Figure 2.26.

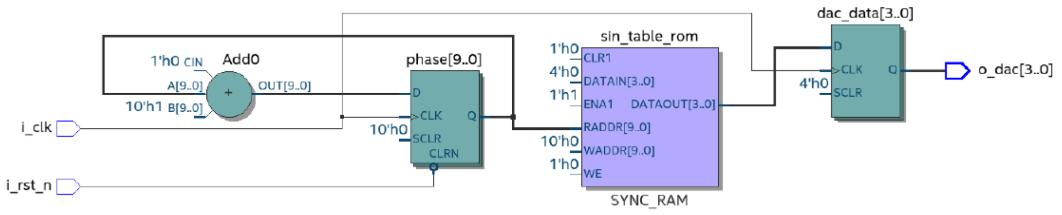
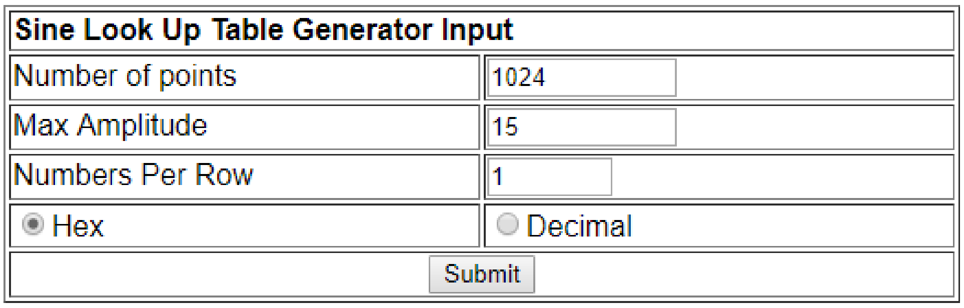


FIGURE 2.26: RTL synthesis result for the source code from listing 2.4

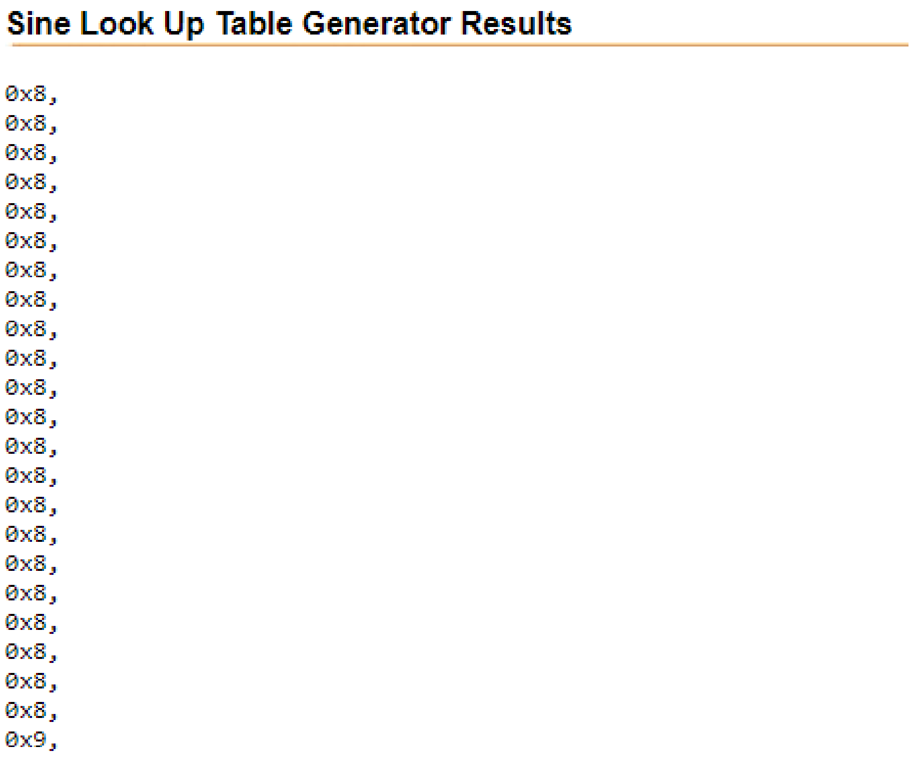
As can be seen from Fig. 2.26, the counter forming the address of a permanent memory is implemented on the basis of the phase register and the adder. As can be seen from Fig. 2.26, the counter forming the address of a permanent memory is implemented on the basis of the phase register and the adder. Output o_dac must be connected to the digital input of the DAC. The memory cell size in Listing 2.4 is 4 bits since the DE0-CV and DE10-Lite debugging boards contain a 4-digit VGA interface duplex (see section 2.3.3). Optionally, it is not difficult to increase the bit digital codes representing a sinusoid. The number of cells in the sin_table_rom array in listing 2.4 is 1024. This means that from this source code 10-bit 1024-cell memory is being synthesized. This memory will store 1024 points of the full sinusoidal period. Accordingly, the period of the generated sinusoidal signal will be equal to the number of points in the period multiplied by the time interval betit en two adjacent points (T_d). Consequently, the period of the generated sinusoidal signal in this case will be equal to $T = 1024 \cdot T_d$. If necessary, the output of the DAC can be connected to the low frequency filter to remove (filter) the "steps" that can be observed in Fig.

2.24. The permanent component of the sinusoidal signal at the output of the DAC can be removed by connecting the capacitor in series with the output of the DAC (the simplest high-pass filter that will not pass the constant component of the voltage). The content of the initialization file of permanent memory sin_table_4bit.hex is given by the link [2.2]. You can calculate the values of the sinusoid points contained in this file in two ways. **First**, you can use the converter it bpage at the link [2.3]. The converter app is depicted in Fig. 2.27. It is necessary to specify the number of points for the sinusoid period (**Numbers of points**, in our case, 1024 points), the number of digits per file line (**Numbers Per Row**, in our case 1) and the numeric code corresponding to the max. value of the sinusoid equal to $2^N - 1$, where N is the DAC bit (**Max Amplitude**, for 4-digit DAC maximum code is 15). After entering the required values, you must click "Submit". The result of the service is shown in Fig. 2.28.



Sine Look Up Table Generator Input	
Number of points	1024
Max Amplitude	15
Numbers Per Row	1
[✓ Hex]	Decimal

FIGURE 2.27: Sinewave data points App Window for digital generator initialization file



Sine Look Up Table Generator Results

```
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x8,
0x9,
```

FIGURE 2.28: Result of the calculating app for the sinewave points

The result of the calculator app [2.3]. depicted in Fig. 2.28, must be copied to a text file and using the automatic search tools with the replacement remove prefixes

0x and commas after each value (replace them with a space, or absence of a character). **Secondly**, the value of the points of the sinusoid to initialize the permanent memory of the digital generator can be calculated independently. It is known that the dependence of the sine function in the time can be represented using the formula (do not forget that the sines takes values from -1 to 1):

$$x(t) = sin(2 \cdot \pi \cdot F \cdot t), \tag{2.8}$$

where F is the frequency of the sinusoidal signal in Hertz;
$T = 1/F$ - period of sinusoidal signal.
In the case of a sinusoid represented by only discrete points, the distance betit en which is equal to the period of the sampling frequency T_s (Fig. 2.23), any time t from the start of generation of the sinusoid can be rounded to the value $k \cdot T_s$, where k is the number of passed T_s periods from the zero point of time. The smaller the value T_s, the more precisely you can represent any value of time t using this approach. So substituting $k \cdot T_s$ for formula (2.8) instead of t, is obtained the formula for representing a sinusoid defined by a discrete set of points:

$$x(k) = \sin(2 \cdot \pi \cdot F \cdot k \cdot T_s) = \sin\left(2 \cdot \pi \cdot k \cdot \frac{F}{F_s}\right), \tag{2.9}$$

where k - is the number of the sinusoid point taking values from 0 to $K - 1$ (K is the number of points for the sinusoid period);
F - frequency of sinusoidal signal in Hertz;
T_s - is the sampling period of the sinusoid in seconds. In other words, this is the time interval betit en adjacent points representing a sinusoid (see Fig. 2.23);
F_s - is the sampling frequency of the sinusoid in Hertz. $F_s = \frac{1}{T_s}$
At first glance, from the formula (2.9) one can already calculate the values of the points of the sinusoid. In this formula, there is only one variable - the point k. Other values are known constants. It is also know the frequency of the sinusoid F that want to be generated. The period of sinusoidal signal T contains K points, the distance betit en which is identical and equal to T_s. This means that the period T consists of K periods T_s, i.e. $T = K \cdot T_s$. So, set the period of sinusoidal signal T and the number of points K for a period, you can calculate the value of T_s with a formula:

$$T_s = \frac{T}{K} \tag{2.10}$$

Substituting (2.10) in (2.9), is obtained:

$$x(k) = \sin\left(2 \cdot \pi \cdot \frac{k}{K}\right) \tag{2.11}$$

Next, it must be taken into account that the resistor based DAC can not generate negative voltage. So you need to add a constant offset to the sinusoid so that the function takes only positive values (this has already been mentioned above).
The Formula (2.11) takes values from -1 to 1. So that is neccesary to create a function that takes only positive values by adding to (2.11) a constant bias equal to 1:

$$y(k) = \sin\left(2 \cdot \pi \cdot \frac{k}{K}\right) + 1 \tag{2.12}$$

The obtained function now varies according to the law of the sine, takes the value from 0 to 2 and has a constant component equal to 1. At the last stage it is necessary to convert the values of the sinusoids calculated by the formula (2.12) into numerical codes for the N-bit DAC. This can be done using observation:

1. The maximum value of $y(k)$ equal to 2 must correspond to the maximum digital code of the N-bit DAC, that is, $2^N - 1$;
2. The value $y(k)$ must correspond to the numeric code;

Having made a proportion is obtained:

$$\text{code} = y(k) \cdot \frac{2^{N-1}}{2} \tag{2.13}$$

So by choosing the number of points for the period K and frequency F of the sinusoidal signal, one can calculate T_s by the formula (2.10) and calculate the values (2.12) of (2.13) the digital codes representing each of the K points of the sinusoid (with numbers k from 0 to $K - 1$). The source code of the program that performs such a calculation is given by the link [2.4]. Figures 2.29-2.30 show the simulation results of a digital sinusoidal generator in the Modelsim simulator. On the charts, there are two variants for displaying the o_dac conductor bus on which the 4-digit numeric codes for the DAC are displayed. In one version of o_dac, you can see the digital codes issued on the bus. Another version of the o_dac display is the representation of the mentioned numeric codes in the form of an analog signal (simulating the work of a 4-digit DAC). It is seen that the signal in the figures below is composed of steps. This is due to low DAC (4 bits). Increasing the DAC bit to 12-24 bits, or using a low frequency filter at the DAC output will solve this problem.

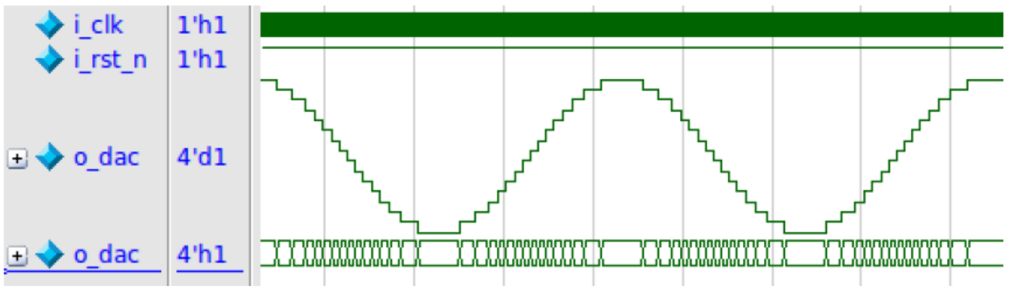


FIGURE 2.29: Timing signal diagram for the sequence of the digital sinewave signal generator

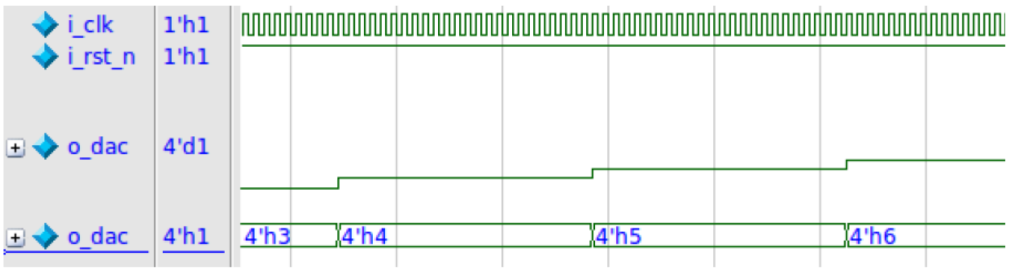


FIGURE 2.30: Timing signal diagram for the sequenece of the digital sinewave generator (Zoom)

2.2.12 Verilog Testbench (initial processes, cycles, delays)

The above discussed how to use the Verilog language for describing digital circuits. Hoit ver, Verilog can also be used to check the work of the schemes. It is a question of developing so-called testbenches, testbenches (testbench) that generate digital signals that are fed to digital circuit inputs and check whether the correct signals appear on the outputs of the digital circuit. Testbenches are ordinary programs and are not synthesized in a digital circuit. Test files do not need to be added to project files in Quartus Prime. The work of the digital diagram described in the Verilog language and its testbench can be verified in a special program called a simulator.

There are free simulators (Icarus, Verilator) and commercial (ModelSim, Incisive, VCS). In laboratory work, it will be used the ModelSim Intel FPGA Starter Edition simulator included in the Quartus Prime Lite Edition suite that you installed in the initial laboratory work. You can use the ModelSim Intel FPGA Starter Edition for free. The free version has some limitations, but for simple projects of our labs this is not fundamentally. The instructions for using ModelSim are given in section 2.3.1 of this laboratory work. All Verilog language designs can be divided into two categories: synthesized and not synthesized. In a digital circuit, you can transform (synthesize) only Verilog code consisting of synthesized designs. Therefore, synthesized structures are used to describe a digital circuit, and not synthesized designs are used to create testbench (although synthesized designs can also be used when describing testbench).

Previously, in laboratory work is considered predominantly synthesized structures (with the exception of the procedural **initial** block). Next, when describing Verilog constructs it will be assumed that are synthesized, unless otherwise indicated directly. It is important to understand that for the correct synthesis of the digital circuitry during its description it is necessary to apply the correct code templates for various components of the digital circuits (see section 2.2.10) and only synthesized Verilog language constructs. It is important to always imagine what scheme your Verilog code is synthesized. On the other hand, the described restrictions do not apply when writing testbench. TestBenches in Verilog have a program logic in which the instructions are executed sequentially and it is possible to call functions described in the Verilog language (if desired, it is possible to call functions from libraries written in C using PLI technology).

So testbenches in Verilog are very similar to ordinary programming. Hover, there is one difference from ordinary programs - this is a concept of time. Tetsbenchni simulate the work of the digital circuit described in Verilog over a period of time, generate input signals for time-varying circuits, etc. To implement such a functionality, the concept of delays and expectations for events is used. An example testbench for checking the operation of the sinusoidal generator from the previous section is listed in Listing 2.5.

Listing 2.5 - Testbench for the verilog sinusoidal signal generator

```
`timescale 1ns / 1ps

module testbench;

parameter PERIOD = 20;

reg         i_clk, i_rst_n;
wire [3:0]  o_dac;

sin_gen  gen_inst(.i_clk (i_clk),
                  .i_rst_n (i_rst_n),
                  .o_dac (o_dac)
                  );

initial begin
    i_clk = 0;
    forever #(PERIOD/2) i_clk = ~i_clk;
end

initial begin
    i_rst_n = 1'b0;

    @(negedge i_clk) i_rst_n = 1;

    repeat (10000) @(negedge i_clk);

    $finish;
end

endmodule
```

Testbench in the Verilog language describes inside a module that does not have I/O ports. In our case, this is a module with a name initial. Inside testbench, as in any Verilog modules, you can create wire type conductors and variable type **reg**. The conductors of the type wire connect to the source ports of the sample module of the digital circuitry which is being verified and the mentioned outputs create certain signals on the conductors, the change of which can be vied in the form of time charts. The **reg** type variables connect the instance of the verified digital circuit module to the input ports. Then in testbench at certain points of time write new values in variables of type **reg** and these values are transmitted to the instance of the module of the verified digital circuit. In the testbench you need to create an instance of the digital circuit module, whose work will be checked. It can be assumed that the module in the Verilog language - is a diagram of the digital circuit. But the instance of the module corresponds to a specific digital circuit drawn by this drawing. Optionally, you can create multiple instances of one module. If you give analogies to C ++ and other languages of object-oriented programming, the description of the module Verilog is similar to the description of the C ++ class. But the instance of the module is similar to the instance of the class.

Example of creating an instance of the module in Verilog language:

```
sin_gen  gen_inst(.i_clk (i_clk),
                  .i_rst_n (i_rst_n),
                  .o_dac (o_dac)
                  );
```

In this example, an instance of gen_inst is created for the sin_gen module. Module sin_gen about writing in listing 2.4. From Listing 2.4 it is seen that the module sin_gen has input ports **i_clk** and **i_rst_n** and the output port **o_dac**. When creating an instance of the module, the module name is first written, then the name of the instance of the module, and then in the parentheses, connect the testbench signals to the input and output ports of the instance of the module.

Connecting to the instances of the instance of the module is as follows. Write the port name of the instance of the module starting at the point, and then in the round brackets write the name of the testbench signal (type **wire** as a **reg**), which connects to this port. This way, through a comma it is necessary to connect the necessary ports of the module. If you do not plan to use a specific module port, you can not connect it. It is recommended to describe all ports of the module, but for those ports that are not used, do not send signals in parentheses. Input signals for an instance of a verifiable module are described in the **initial** procedural blocks. The **initial** blocks are not synthesized and used only in testbenches.

When a CAD like Quartus Prime encounters in the Verilog **initial** block file, such code block is simply ignored and not synthesized. The only exception when the **initial** block is taken into account by the synthesizer is the initialization of the ROM of memory for the FPGA (see section 2.2.9). The **initial** code blocks begin to run immediately after the start of the simulation (simulation is called modeling in Modelsim). And the simulation starts at zero time. You can use delays to execute a certain instruction in the **initial** block at zero time. The instructions in the **initial** block of the code are executed consecutively one after another (from this rule there can be exceptions which for simplicity it will not consider yet).

The execution of the **initial** block ends after all its instructions are completed. However, if the **initial** block contains, for example, an infinite loop, such **initial** primary block will be executed until the end of the simulation. There may be several **initial** blocks in the testbench. The simulator performs the **initial** blocks at the same time. In other words, several **initial** blocks work, which several streams in a multi-threaded program written in C language. Typically, testbenches in Verilog include more than one **initial** code block. So it can be seen that multithreading is typical for testbenches in the Verilog language.

The delay in time between executing instructions in the initial block of a code can be realized using the operator #. After the operator # indicate how many units of the model time it is necessary to delay execution of the following instruction. In other words, when the simulator meets operator #, it will delay the execution of all subsequent instructions for the number of time units specified after the # operator. Upon completion of the delay, instructions continue to be executed. The delay in the # operator does not necessarily have to be determined by a constant. This can be a variable, or a verilog parameter. Similar delays are used only during simulation in testbench and not synthesized. The dimension of the model time units in which the delays are measured is indicated by the first argument of the design **`timescale**, which is determined at the very beginning of the Verilog testbench.
The **`timescale** design has the following syntax:
`timescale 1 ns / **1** ps
Where the first argument of **`timescale** (in this case **1** ns) - is the dimension of units of model time, in which the delays in the operator # are determined. For example, design #6 for this **`timescale** format means a delay of 6 nanoseconds. The second **`timescale** argument (in this case 1ps) - is the accuracy of the delay definition. In other words, in this example, the smallest step of the delay change is 1 picosecond.

Another way to handle Verilog testbench delays is to wait for the event with the @ operator. The work of the operator @ is described in section 1.2.2 of the previous laboratory work. When the simulator meets Verilog code @ operator, execution the following operators stops until the event specified after the @ operator occurs. For example, the design @(**negedge** i_clk) blocks the execution of all subsequent Verilog operators until the i_clk signal is closest to the front. Consider the **initial** block of code from Listing 2.5, which creates a synchronization clock for an instance of the sinusoidal signal generator module.

```
initial begin
    i_clk = 0;
    forever #(PERIOD/2) i_clk = ~i_clk;
end
```

At zero moment of time, at the beginning of the simulation, the execution of the initial block begins, which leads to the operation i_clk = **0**. Next, the simulator enters an infinite loop **forever**, the iteration of which will run until the completion of the simulation. At the beginning of each iteration of an infinite loop, the simulator encounters a delay equal to half the value of the variable **PERIOD**. Upon completion of the Verilog execution delay, the code continues and inverts the i_clk variable: i_clk = ∼i_clk. After that, the simulator goes on to the next iteration of the infinite loop and the delay is restarted. The described code thus leads to the fact that i_clk is inverted every half-period.
The **PERIOD** constant is determined as follows:
parameter PERIOD = **20** ;
Parameters in the Verilog language, where will be considering in the following laboratory work. Meanwhile, parameters can be considered as a method for determining named constants.
Let us consider one more **initial** block of process with listing 2.5.

```
initial begin
    i_rst_n = 1'b0;

    @(negedge i_clk) i_rst_n = 1;

    repeat (10000) @(negedge i_clk);

    $finish;
end
```

In the given initial block at the beginning of the simulation, at the zero point of time, the variable i_rst_n will be log.0. Next is the i_clk closest front edge and after that event, i_rst_n will write log.1. In such a way, a signal is generated. By the end of the **repeat** cycle, the number of characters is displayed (in a given number of **10000** characters). For each iteration, the i_clk back edge is expected. Thus, the delay of **10000** i_clk sync signal periods is realized. After completing the **repeat** cycle, the system function $finish end is called, which leads to the completion of the simulation and the stop of the execution of all **always** the **initial** code blocks. By the way, with the exception of the exception, Verilog system functions are not synthesized (that is, they are used only in simulation). The result of simulation of the testbench generator of the sinusoidal signal is shown in Fig.2.29-2.30.

2.2.13 Digital Sinewave Generator with Controlled Frequency

In one of the previous sections, it was considered an application design for a simplest digital frequency generator for a sinusoidal signal. However, a digital generator with the ability to change the sinusoidal frequency during the work is of great interest. That is it the circuit known as the **Numerically Controlled Oscillator** (NCO). NCO is the basis of the modern digital analog signal generator, with the help of NCO, you can create a radio frequency transmitter (FM) or amplitude (AM) modulation and more advanced radio systems, a digital **phase-locked loop generator** (PLL) and many other interesting and useful digital systems. The block diagram of NCO is shown in Figure 2.31.

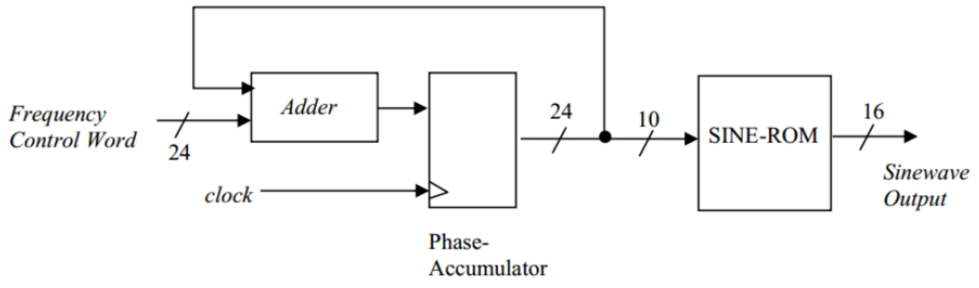


FIGURE 2.31: Block diagram of a digital frequency generator (NCO) for a sinusoidal signal generation

The source code of NCO in Verilog, implementing the hierarchical top-level design from Figure 2.31, is listed in Listing 2.6. The result of the synthesis of this source code is shown in Fig. 2.32. **Listing 2.6** - Descibes the source code of the digital generator for a sinusoidal signal with controlled frequency in the Verilog language.

```
module nco(i_clk, i_rst_n, i_ctrl_code, o_dac);

input           i_clk;
input           i_rst_n;
input   [31:0]  i_ctrl_code;
output  [3:0]   o_dac;

reg     [3:0]   sin_table_rom[1023:0];
reg     [31:0]  phase;
reg     [3:0]   dac_data;
reg     [31:0]  phase_step;

assign  o_dac = dac_data;

initial $readmemh("sin_table_4bit.hex", sin_table_rom);

always @(posedge i_clk)
    dac_data <= sin_table_rom[phase[31:22]];

always @(posedge i_clk, negedge i_rst_n) begin
    if(~i_rst_n) begin
        phase_step <= 0;
    end else begin
        phase_step <= i_ctrl_code;
    end
end

always @(posedge i_clk, negedge i_rst_n) begin
    if(~i_rst_n) begin
        phase <= 0;
    end else begin
        phase <= phase + phase_step;
    end
end

endmodule
```

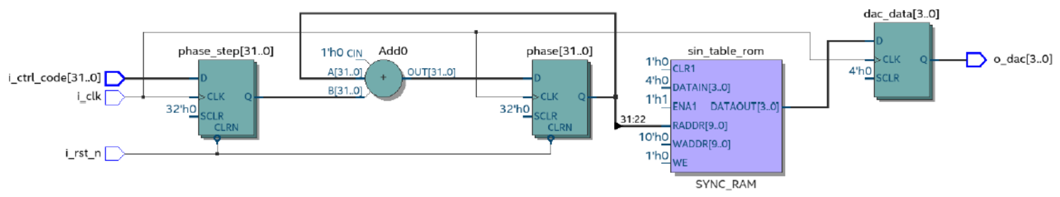


FIGURE 2.32: Result of the synthesis source code from listing 2.6

If comparing the structure of the usual digital generator of the sinusoid (Fig. 2.25) and its source code (listing 2.4) with the structure (Fig. 2.31) and the source code NCO (Listing 2.6), can be concluded that the schemes are very similar (both circuits are based on the counter and permanent memory), but there are following differences:

1. High bit count in the NCO scheme (32 bits);
2. In the scheme of the ordinary sine wave generator on each active front of the synchronization signal, the content of the counter increases by 1. However, in the NCO scheme for each active front of the signal synchronization, the content of the counter increases to an arbitrary value, which is fed to one of the inputs NCO and has a bit length of 32 bits (that is can be a very large number);
3. The bus density of the permanent memory address in both circuits is the same (10 bits), but in the NCO scheme only the older 10 bits of the 32-bit counter are used to form the permanent memory address.

Consider the principle of the NCO. From the logic of the usual digital generator of the sinusoidal signal (see 2.2.11), it is evident that the period of the sinusoidal signal in such a generator is determined by the period of overflow of the counter, which specifies the address of the cells of the permanent memory. The first point of the sinusoidal period corresponds to the zero value of the ROM address, and the last point of the sinusoidal period corresponds to the largest value of the N-bit ROM address ($2^N - 1$). If the ROM address accepts the maximum value, on the next active front of the synchronization signal, the contents of the counter is overwritten and the formation of the sinusoidal signal period begins from the beginning. So the frequency of the sinusoid (the number of sinusoidal periods per second) is determined by the frequency of overflow of the counter of the address of the permanent memory (the number of overflows of the counter in a second). If for each active phase of the synchronization signal, the content of the phase counter, forming the ROM address, is increased not by the unit but by the **phase_step** value, this will allow to control the frequency of overflow of the counter and, consequently the frequency of the sinusoidal signal. For example, to overflow a 3-digit phase counter (which can store 8 values from 0 to 7), it is necessary to add 8 times the unit to the zero content of the counter (after the addition of the 8th unit in the 3-digit counter will again be 0), or 4 times add value 2, or 2 times add value 4. This is also the key idea of the NCO. Such a counter in the NCO, to which a certain number is added for each active front of the synchronization signal, is called **phase battery**.
Taking into account the above statement, the following expression can be written:

$$\frac{2^N}{\textbf{phase\_step}} = \frac{T}{T_{in}} \tag{2.14}$$

The formula (2.14) can be rewritten as follows:

$$\frac{2^N}{\textbf{phase\_step}} = \frac{F_{in}}{F} \tag{2.15}$$

T - period of sinusoidal signal, sec;
F - frequency of sinusoidal signal, Hz;
T_{in} - is the period of the synchronization signal applied to the clock input of the counter, sec;
F_{in} - frequency of the synchronization signal applied to the clock input of the counter, Hz;
$2N$ - number of values that can be taken by N-digit. counter (from 0 to 2^N-1);
phase_step - number added to the content of the counter on each active front of the signal synchronization;
From the formula (2.15) you can calculate the value of **phase_step** to obtain the required frequency of a sinusoidal signal at a known clock frequency F_{in}:

$$\textbf{phase\_step} = 2^N \cdot \frac{F}{F_{in}} \tag{2.16}$$

It can also be seen from formula (2.15) that the smallest change in frequency will be:

$$F = \frac{F_{in}}{2^N} \tag{2.17}$$

In other words, the smallest step of changing the frequency of a sinusoidal signal is equal to the smallest value of the sinusoidal signal frequency (at **phase_step** = 1). For example, when using a 32-bit phase counter and 50 MHz of input signal synchronization, which is fed to the clock input of the counter, it is possible to obtain the accuracy of the minimum output frequency resolution of the sinusoidal signal is equal to 50 000 000/2^{32} = 0.01 Hz. Another idea important for understanding the NCO's work is that only the highest levels of the **phase** counter are used as the permanent memory address. This is necessary for the use of memory of adequate size, because if for the formation of the address to use all 32 digits phase phase, would require a memory size of 2^{32} depth. Using only the highest digits of the counter as a permanent memory address can be limited to a smaller amount of memory while maintaining a high accuracy of frequency formation. For example, in the case of using the address of the older 10 digits of a 32-bit counter, 1024-cell memory and 10-bit address entry will be required. In this case, the accuracy of determining the frequency of the sinusoidal signal remains equal to 0.01 Hz. The structure of the phase battery (counter) in the NCO is shown in Figure 2.33.

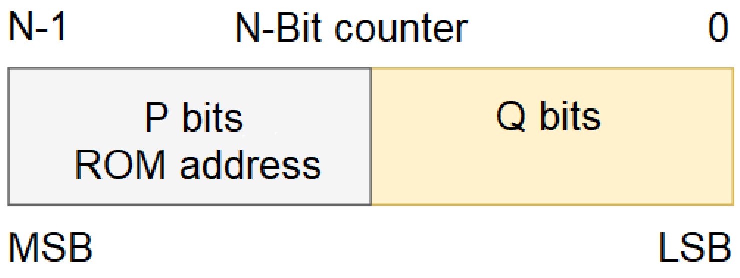


FIGURE 2.33: Structure of the phase battery (counter) in the NCO

Figure 2.33 shows that the N-bit counter consists of two parts. P Bits specify the number of the permanent memory cell. Q bits determine the frequency with which the value of the higher P digits increases. In general, N = P + Q. It is known from the principle of the counter that the content of the older P-discharges will increase by 1 if there are units in the lower Q bits and an active front of the synchronization signal is received. In other words, the incremental frequency of older P bits (the frequency of forming new ROM values) will depend on the frequency of overflow of the younger Q bits. With the use of this approach you can create sinusoidal signals whose frequency does not exceed half the frequency of the input clock signal. The justification for such a limitation is a volumetric theory, and now it is better to perceive this assertion. Relevant issues of digitization and generation of analog signals are described in detail in the third section of the book [2.4]

2.3 Practical part

2.3.1 Simulation of projects in ModelSim

As noted in Section 2.2.12, the operation of a digital circuit and a test stand (testbench) can be modeled in the simulation program. In these labs, the Modelsim simulator is used. The capabilities of commercial simulators in the field of simulation and analysis of digital circuits are very broad, so now only the basic steps to get the simplest result will be described.
To implement the simulation you need to complete the following steps:

1. Perform compilation of Verilog files containing modules for simulation;
2. Download the compiled modules in Modelsim;
3. Add the signals you need to watch on the timeline;
4. Run simulation testbench on simulation;

Consider further the need to compile Verilog files. The first Verilog simulators worked as interpreters, reading and executing instructions directly from the Verilog file. However, this is a very unproductive approach that does not allow get a high speed simulation. Modern commercial simulators convert Verilog to binary code executed directly on the processor. This process is called compilation. The steps above, which are required to run the simulation, can be performed, if desired, from the Modelsim graphic interface. However, it's much easier to write a short command script once, and then just run it for execution, instead of repeating the same kind of actions over and over again. The TeamMap scripts for Modelsim are written in TCL and stored in *.do file extensions. The command file sim.do for modeling the digital generator of the sinusoidal signal (Listing 2.4) and the corresponding testbench (Listing 2.5) is listed in Listing 2.7. It is very easy to run a sim.do command file on the simulator. It is enough being in the directory where Verilog files are stored, so that, execute x terminal (console) command **vsim** -do sim.do

Listing 2.7 - The source code for the sim.do batch file for the Modelsim simulator

```
vlib work

vlog  sin_gen.v sin_gen_tb.v

vsim -novopt work.testbench

add wave /testbench/i_clk
add wave /testbench/i_rst_n
add  wave  -format  Analog-Step  -height  84  -max  15.0  -radix  unsigned
/testbench/o_dac

run -all
```

You can also run the sim.do command file for execution from the Modelsim graphic interface. To do this, select the item in the main menu Tools → Tcl → Execute Macro. Then in the window that opens, you must select the file sim.do and click on "Open". After the system function is called in the Verilog code $finish, the simulation will end and a window will appear asking "Are you sure you want to finish?". You need to click on No button and look at the time charts of signal changes in the Wave window. Analyze a more detailed batch file simulation from listing 2.7.

- The **vlib** work command creates a library with the name of work where the Verilog files will be compiled.
- The command **vlog** sin_gen.v sin_gen_tb.v opens the compilation of Verilog files to the previously created library.
- The **vsim** command -novopt work.testbench opens the simulation module called testbench (see Listing 2.5) contained in the work library. Execution of this command leads to the fact that the simulator Modelsim from the work library loads the compiled testbench module and other modules, instances which are created in the testbench. The -novopt key in the simulator does not optimize (during optimization, Modelsim often removes/optimizes some of the signals that may be interesting for analysis, so it's best to perform a simulation without optimization first).
- The **add wave** /testbench/i_clk command adds the signal i_clk contained in the testbench module in the **wave** timeline window.
- Command **add** wave -format Analog-Step -height **84** -max **15.0** -radix unsigned / testbench/o_dac adds the o_dac bus to the timetable window, but indicates that it is necessary to display the digital codes on the o_dac as an analog signal (DAC modeling).
- The command **run** -all starts the simulation.

Modeling results in Modelsim after starting the batch file from listing 2.7 are shown in Figure 2.29.

2.3.2 Custom memory synthesis in Quartus Prime (Parameterized functions)

By Verilog template code from Listing 2.3, a ROM based on block memory is synthesized by default and initialized by the contents of the initialization file. If necessary, you can configure the Quartus Prime to implement memory based on truth cards and flip-flops (such a need arises very rarely). To do this, go to the settings of the project (Settings), on the Compiler Settings tab, click Advanced Settings (Synthesis), and in the window that opens, put the "Auto RAM Replacement", "Infer RAMs from Raw Logic", "Auto ROM Replacement", "Infer ROMs from Raw Logic". At the end, press the "On" and "Apply" buttons. In the FPGA of MAX10 series chips, the default memory can be synthesized on the basis of triggers and truth cards without the use of blocked memory resources. To fix this shortcoming, go to the FPGA Chip tab (Assignments → Device), then click on the "Device and Pin Options" button, in the window that opens, select the Configuration tab and select Configuration Mode in the "Single Uncompressed Image with Memory Initiation (512 Kbits UFM)", click the Ok button.

2.3.3 Digital-to-analog converter in debug boards with VGA

The description and calculation of a digital-to-analog converter with weight resistors is given in section 2.2.7 of this laboratory work. The DE0-CV and DE10-Lite debugging boards used for laboratory work contain the VGA interface used to display the image on a computer monitor. Detailed VGA interface, it will be considered in one of the following laboratory work. Now it's important to understand that in the VGA interface, the color of each pixel is determined in **RGB** format as a mixture of red, green and blue. The intensity of each color is determined by the voltage on the corresponding output of the VGA interface. On the output **R** there is a voltage that determines the intensity of the red color. At the output **G** there is a voltage that determines the intensity of the green color. At output **B** there is a voltage that determines the intensity of the blue color. Voltage 0 Volts corresponds to a lack of color. The voltage of 0.7 volts corresponds to the maximum brightness of the color. Outputs of the VGA interface from the source of the video signal (in our case, the debug board) are shown in Figure 2.34. As can be seen from the figure, outputs 5, 6, 7, 8, 10 are connected to the "ground" (GND). In this way, the voltage determining the intensity of the red component of the pixel color is measured between output 1 (output R) and output 6 (GND) of the VGA interface.

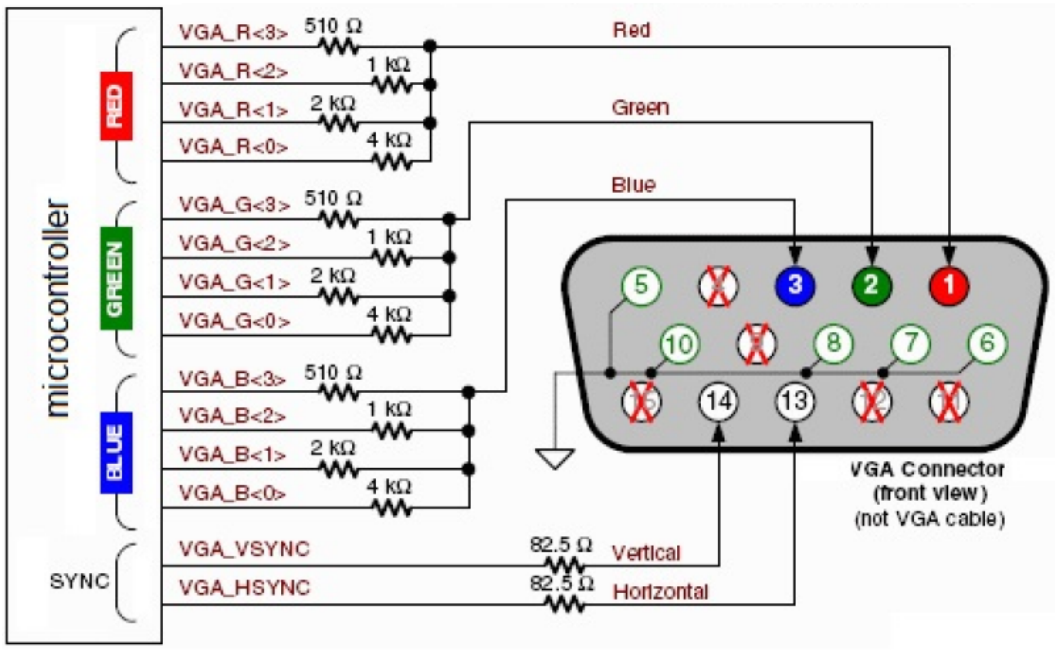


FIGURE 2.34: 4-bit VGA DAC and VGA interface outputs

For each component of the selected pixel color (**R**, **G**, or **B**) in the VGA source of the video signal there is a DAC that converts the digital code of the intensity of a certain color into a voltage. In other words, each VGA video source contains 3 DACs: for red, green, and blue color constituents (or one three channel DAC). The debug boards DE0-CV and DE10-Lite contain 4-bit VGA digital-to-analog converters. This means that the intensity of each component of the color of the pixel (**R**, **G**, or **B**) can take only 16 values. In modern VGA DACs, like ADV7123, have 10-bit tires to determine the intensity of the **R**, **G**, and **B**) components of the pixel color. Figure 2.34 shows the implementation of 4-bit DACs for R, G and B channels and the exit of the outputs of the specified DACs to the corresponding output VGA connector source video signal.

In Fig. 2.34, the digital codes on the inputs of the DAC from a microcontroller, and in our laboratory work, instead of the microcontroller, a FPGA chip can be used. The digital input for a DAC, which determines the intensity of the red component of the color of the pixel, is often called VGA_R. Similarly, digital inputs that form green and blue pixel color components are often called VGA_G and VGA_B. If the DAC input (for example, the DAC input of the red channel VGA_R) contains the maximum digital code, in idle mode (without loading the load) the output voltage will be logic 1 (HIGH) (3.3 volts). But if the output of the DAC is to connect a load of 75 ohms (standard load outputs **R**, **G**, and **B**), the maximum voltage output will be 0.7 Volt. In examples of analog signal generators in this laboratory work will use the DAC red channel of the VGA interface. Accordingly, the digital codes must be fed to the VGA_R input of the DAC, and the voltage between the outputs 1 and 6 of the VGA interface (see Fig. 2.34, output 1 - output of the red channel VGA, output 6 -GND). The full block diagram of the DE10-Lite board is shown in the Fig. 2.35.

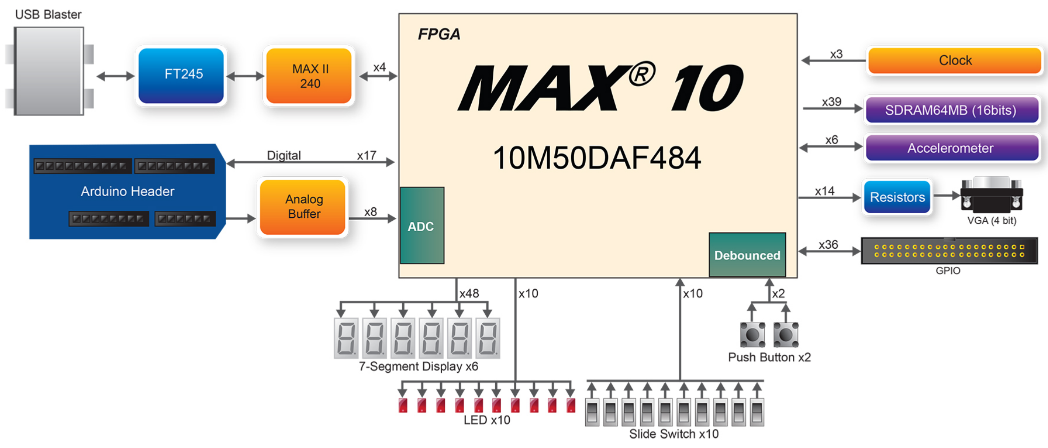


FIGURE 2.35: DE10-Lite board full block diagram

2.3.4 Tasks for laboratory work

The source code can be found at the link [2.5]

Appendix A

Frequently Asked Questions

A.1 What kinds of programmable logic devices are available today?

Which companies supply programmable logic:

1. Altera
2. Xilinx
3. Vantis (formerly AMD's programmable logic division, now part of Lattice)
4. Lattice Semiconductor
5. Actel
6. Lucent Technologies
7. Cypress Semiconductor
8. Atmel
9. QuickLogic

If you want to completely hide the links, you can use:
`\hypersetup{allcolors=.}`, or even better:
`\hypersetup{hidelinks}`.
If you want to have obvious links in the PDF but not the printed text, use:
`\hypersetup{colorlinks=false}`.