Practice 8. Design of a BPSK modulator with VHDL

8.1. Objectives

To design a binary phase shift keying modulator (BPSK) by using VHDL and its implementation in a FPGA. To learn to establish communications with peripherals, in this case the DAC LTC 2624.

8.2. Digital-to-analog converter LTC 2624

The Spartan-3A/3AN Starter Kit Board has a serial digital-to-analog converter (DAC) of four channels and SPI-compatible (*Serial Peripheral Interface*). The four outputs from the DAC are accessible from the connector J21, which is located, together with the DAC, immediately below the Ethernet RJ-45 connector of the board, such as we can see in the figure.

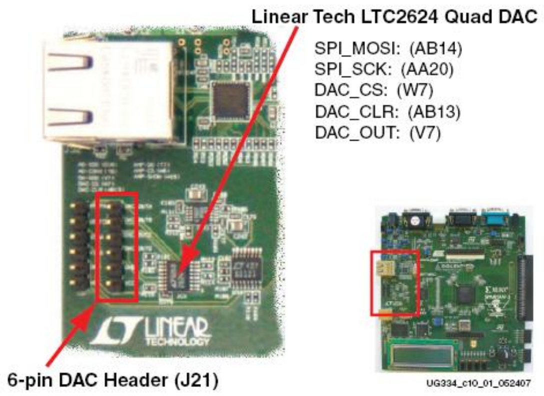

Figure 8.1. Location of the DAC LTC 2624 and the connector J21 in the development board

The SPI is a synchronous *full-duplex* character-oriented bus which only employs five communication wires. In the figure is shown the connection of the FPGA with the SPI bus interface and that of the latter with the DAC.

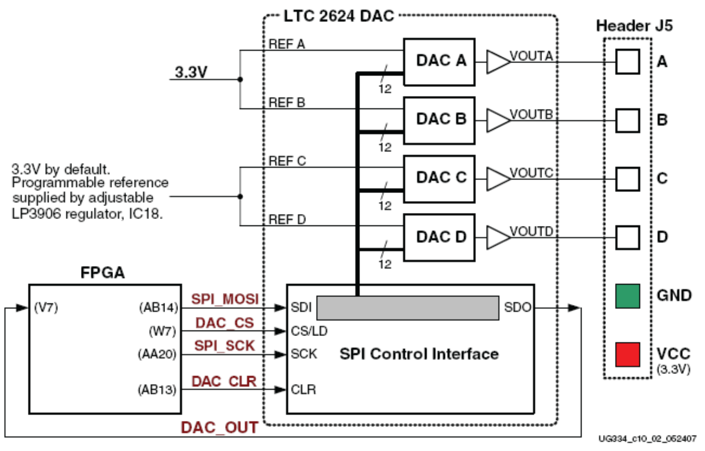

Figure 8.2. Connections schematics of the digital-to-analog converter

The next table lists the interface signals between the FPGA and the DAC, where the FPGA is the *master* of the communication and the DAC the *slave*. The SPI_MOSI, SPI_SCK and DAC_OUT signals are shared with other devices on the SPI bus, therefore the active-low DAC_CS signal is used for selecting the DAC as slave of the communication. The DAC_CLR signal is the active-low reset input to the DAC.

Signal	FPGA Pin	Direction	Description
SPI_MOSI	AB14	FPGA→DAC	Serial data: Master Output, Slave Input
DAC_CS	W7	FPGA→DAC	Active-Low chip-select. Digital-to-analog conversion starts when this signal returns High.
SPI_SCK	AA20	FPGA→DAC	Clock
DAC_CLR	AB13	FPGA→DAC	Asynchronous, active-Low reset input
DAC_OUT	V7	FPGA←DAC	Serial data from the DAC

Table 8.1. DAC interface signals

The next figure shows a detailed example of the SPI bus timing. Each bit is transmitted or received relative to the SPI_SCK clock signal. After driving the DAC_CS slave select signal Low, the FPGA transmits data on the SPI_MOSI signal, MSB first. The LTC2624 captures input data (SPI_MOSI) on the rising edge of SPI_SCK; the data must be valid for at least 4 ns relative to the rising clock edge. The LTC2624 DAC transmits its data on the DAC_OUT signal on the falling edge of SPI_SCK. The FPGA captures this data on the next rising SPI_SCK edge. The FPGA must read the first DAC_OUT value on the first rising SPI_SCK edge after DAC_CS goes Low. Otherwise, bit 31 is missed. After transmitting all 32 data bits, the FPGA completes the SPI bus transaction by returning the DAC_CS slave select signal High. The High-going edge starts the actual digital-to-analog conversion process within the DAC.

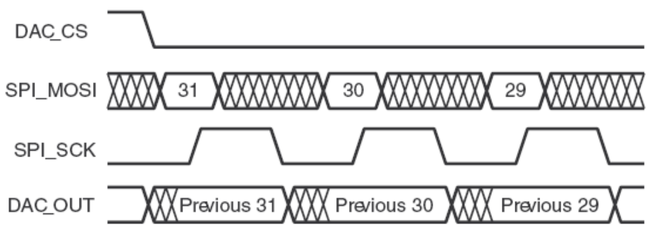

Figure 8.3. SPI communication waveforms

Figure 8.4 shows the communications protocol required to interface with the LTC2624 DAC. Inside the DAC, the SPI interface is formed by a 32-bit shift register. Each 32-bit command word consists of a command and an address, followed by a data value. As a new command enters the DAC, the previous 32-bit command word is echoed back to the master. The response from the DAC can be ignored although it is useful to confirm correct communication.

The FPGA first sends eight dummy or don't care bits, followed by a four-bit command. The most commonly used command with the board is $c_3c_2c_1c_0$ =

"0011", which immediately updates the selected DAC output with the specified data value. Following the command, the FPGA selects one or all the DAC output channels via a four-bit address field. Following the address field, the FPGA sends a 12-bit unsigned data value that the DAC converts to an analog value on the selected output(s). Finally, four additional dummy or don't care bits pad the 32-bit command word.

As shown in Figure 8.2, each DAC output level is the analog equivalent of a 12-bit unsigned digital value, *DATA*[11:0], written by the FPGA to the DAC via the SPI interface. The voltage on a specific output is generally described by the next equation:

$$V_{OUT} = \frac{DATA[11:0]}{4,096} V_{REF}$$

The reference voltage, V_{REF}, is different between the four DAC outputs. Channels A and B use a 3.3V reference voltage. Channels C and D have a separate reference voltage, nominally also 3.3V, supplied by the LP3906 regulator designated as IC18.

According to the previous equation, the DAC only works with positive values (unsigned integers), feature that we will take into account when we send to the DAC the digital word whose value has to be represented.

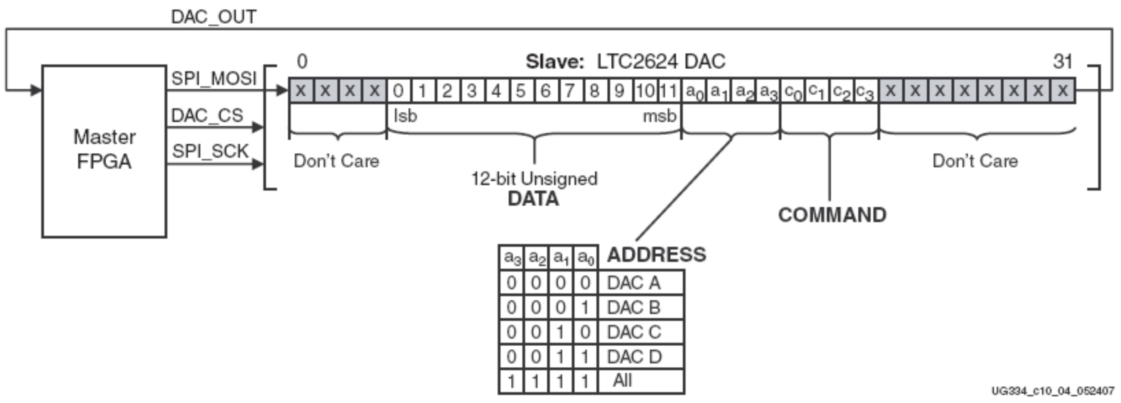

Figure 8.4. SPI communication protocol to DAC LTC2624

8.3. Practical development

In this practice, we are going to design, by using VHDL, a BPSK modulator which is constituted by a random data generator, the BPSK modulator itself and a DAC interface device (see figure 8.5). As we can see in the figure, the data generator has two inputs (*clk* and *reset*) and two outputs (*data* and *sync*), whereas the BPSK modulator has three inputs (*clk*, *reset* and *serial_data*) and three outputs (*clk_data*, *clk_spi* and *data*). Moreover, the output *clk_data* is fed back to the clock input of the data generator, whereas the output of the latter is connected to the serial data input of the BPSK modulator.

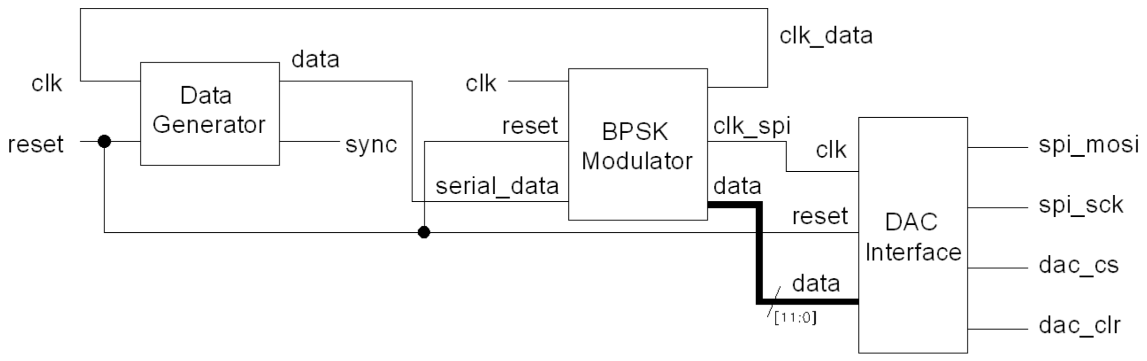

Figure 8.5. Block diagram of the BPSK system

Given that the output of the BPSK modulator has to be analog, a DAC interface is included to establish the communication with this device by means of the SPI bus. The BPSK modulator supplies the clock reference signal for the SPI bus, as well as the digital data word which must be converted to analog by the DAC. Therefore, the BPSK modulator is in charge of controlling the synchronism of all the system components, generating the clock signals for the SPI bus, as well as those for the data generator and the modulator itself.

8.3.1. DAC interface

The device which works as DAC interface has three inputs (*reset*, *clk* and *data*) and four outputs corresponding to the communication wires of the SPI bus (*spi_mosi*, *spi_sck*, *dac_cs*, *dac_clr*). Next we are going to design this interface with the DAC, which will allow us to represent an analog signal through channel DAC A.

1. Design a VHDL code for the connection interface to the DAC through the SPI bus. A possible example for this code would be the next one:

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

entity com_dac is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           dac_cs : out  STD_LOGIC;
           dac_clr : out STD_LOGIC;
           spi_mosi : out  STD_LOGIC;
           spi_sck : out  STD_LOGIC;
           data : in  STD_LOGIC_VECTOR(11 downto 0);
           count_out : out std_logic_vector(6 downto 0));
end com_dac;

architecture Behavioral of com_dac is
signal memory_dac : std_logic_vector(31 downto 0) := (others => '0');
begin
process(clk,reset)
```

```
variable count : natural range 0 to 100 := 0;
begin
     if reset = '1' then
          -- Sets the command by default
          memory_dac(23 downto 20) <= "0011";
          -- The address points out to DAC A by default
          memory_dac(19 downto 16) <= "0000";
          count := 0;
          dac_cs <= '1';
     elsif clk'event and clk = '1' then
          count := count + 1;
          case count is
          when 1 =>   dac_cs <= '0';
                      memory_dac(15 downto 4) <= data; -- Load the data
                      -- Points out to DAC A
                      memory_dac(19 downto 16) <= "0000";
                      spi_mosi <= memory_dac(31);
          when 33 =>  dac_cs <= '1';
          when 64 =>  count := 0;
          when others =>
                      spi_mosi <= memory_dac(31-((count-1) mod 32));
          end case;
     end if;
     count_out <= std_logic_vector(conv_unsigned(count,7));
end process;
spi_sck <= not(clk);
dac_clr <= not(reset);
end Behavioral;
```

Observe as the controller of the DAC interface is basically constituted by a state machine based on the variable *count*. Every time that there is a rising edge of the signal *clk*, this variable is increased and several conditions are checked. If *count* is equal to one, the data are loaded to the memory of the interface (*memory_dac*) and the latter is prepared for the transmission of the data towards channel DAC A through the SPI bus. Once the 32 data bits are transmitted, the signal *dac_cs* is set to high logic level in order to indicate to the DAC that initiates the conversion, re-initiating all the process again. This process is repeated endlessly, unless this is interrupted by a reset signal, which would initialize the interface. We can see as the *spi_sck* and *dac_clr* signals are obtained by simply inverting the *clk* and *reset* signals, respectively.

2. Now, we are going to carry out a simulation by using the ISE Simulator in order to check the correct behaviour of the interface with the DAC. Observe as a signal of type *std_logic_vector* called *count_out* has been included with the unique mission of making easier the analysis of the different signals of the device.
3. Prove in the board the correct behaviour of the DAC interface, using as data inputs the switchers SW0 to SW3 of the development board (they are used to represent the most significant bits). For the *reset* signal, use any push button and, for the *clk* input, make use of the on-board clock oscillator. A possible configuration file could be the following:

```
NET "reset" CLOCK_DEDICATED_ROUTE = FALSE;
NET "data<11>" LOC = T9;
NET "data<10>" LOC = U8;
NET "data<9>" LOC = U10;
NET "data<8>" LOC = V8;
NET "clk" LOC = E12;
NET "reset" LOC = T15;
NET "spi_mosi" LOC = AB14;
NET "spi_sck" LOC = AA20;
NET "dac_cs" LOC = W7;
NET "dac_clr" LOC = AB13;
```

8.3.2. Data generator

For the design of the pseudo-random data generator we can use that implemented in the previous practice. This data generator requires from the definition of several constants and a component which works as register. Next we show the VHDL codes which are necessary in both cases.

1. Create a package called *constants* which contains all the constants that we are going to use during this practice. The VHDL code would be the next one:

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

package constants is
     constant N : positive := 4;
     constant M : positive := 32;
     constant nbits : positive := 12;
     constant ndec : positive := 10;
     subtype word is signed(nbits-1 downto 0);
     type table is array (M-1 downto 0) of word;
     constant Pi : real := 3.1415927;
     constant delta_phi : real := 2.0*Pi/real(M);
     function and_vector (vector : in std_logic_vector(0 to N-1))
return std_logic;
end constants;

package body constants is
     function and_vector (vector : in std_logic_vector(0 to N-1))
          return std_logic is
     variable result : std_logic;
     begin
          result := vector(0);
          for I in 1 to N-1 loop
              result := vector(I) and result;
          end loop;
          return result;
     end and_vector;
end constants;
```

Observe as, apart from the constant *N* which is referred to the length (number of registers) of the data generator and the *and_vector* function which determines the *and-logic* function of a data vector, are also defined a constant *M* for the number of positions of a table which is going to contain the sine wave values (see section 8.3.3), the number of bits (*nbits*) of each *word* of the table, as well as the number of bits used as decimals (*ndec*), and two real constants (*Pi* and *delta_phi*). The two last ones are used by a function which initializes the table and they are referred to the irrational number π and the phase increment $\Delta\varphi$ between consecutive positions in the table which is given by $2\pi/M$. In the section 8.3.3 we will address these aspects in more detail. The library *MATH_REAL* of *IEEE* is included to work with real numbers.

2. Create a device which works as register to be used by the data generator. A possible VHDL code would be the next one:

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

entity register is
    Port ( clk,preset,D : in  STD_LOGIC;
           Q : out  STD_LOGIC);
end register;

architecture Behavioral of register is

begin
     process(clk,preset)
     begin
          if preset='1' then
              Q <= '1';
          elsif clk'event and clk='1' then
              Q <= D;
          end if;
     end process;
end Behavioral;
```

Observe as the register responds to the changes in the input signal *clk* (clock signal) and *preset* (active-high signal which forces a *set* putting the register to logic level ‘1’). Whenever a rising edge occurs in the clock signal *clk*, the data at the *D* input of the register will be transferred to its output *Q*.

3. In figure 8.6 is shown the internal structure of the data generator which is constituted by four register, therefore generating a pseudo-random sequence of length $2^4 - 1 = 15$.

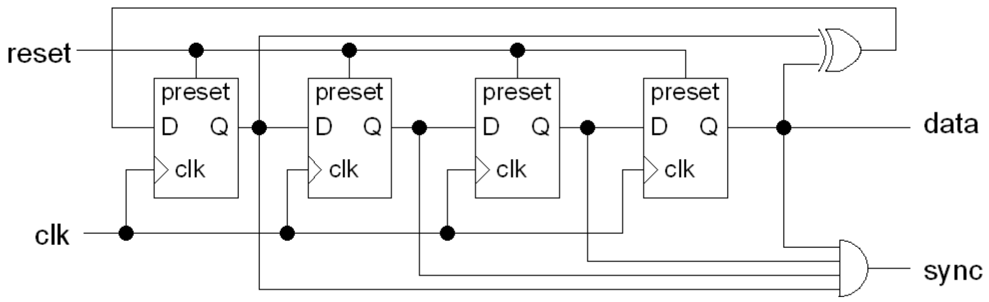

Figure 8.6. Internal structure of the pseudo-random data generator

A possible VHDL code which allows us to implement the pseudo-random data generator of figure 8.6 is the next one:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.constants.ALL;

entity data_gen is
     generic (Nreg : positive := N);
     port  (clk,reset : in  STD_LOGIC;
            data,sync : out  STD_LOGIC);
end data_gen;


architecture Behavioral of data_gen is

component register is
     port (clk,preset,D : in std_logic;
               Q : out std_logic);
end component register;
signal sig_xor : std_logic;
signal Q_int : std_logic_vector(0 to Nreg-1);
begin
     Data_generator: for I in 0 to Nreg-1 generate
        Reg00: if (I=0) generate
          Reg0: Register port map (clk,reset,sig_xor,Q_int(0));
        end generate;
        Regs: if I>0 generate
          Reg: Register port map (clk,reset,Q_int(I-1),Q_int(I));
        end generate;
     end generate;
     data <= Q_int(Nreg-1);
     sig_xor <= Q_int(0) xor Q_int(Nreg-1);
     sync <= and_vector(Q_int);
end Behavioral;
```

8.3.3. BPSK modulator

In this section we are going to proceed to design the BPSK modulator. First of all, we have to take into account that, considering the way in which the *DAC interface* has been defined, this device requires from 64 clock cycles in order to carry out the transmission of the digital data to be represented by the DAC in an analog way. Therefore, the clock to be used by this device will be the fastest one, that is, the on-board 50 MHz clock oscillator. During these 64 clock cycles, the data supplied by the DAC is not allowed to change, therefore the modulator clock which controls the addressing to the sine wave table has to oscillate at a frequency 64 times slower than that of the basis clock. Moreover, the data supplied by the generator, which modulates the sine wave, must be maintained unalterable during at least a complete cycle of the sine wave, which is constituted by *M* samples, hence the data clock has to oscillate at a frequency *M* times slower than that of the table addressing. Taking into account everything that has previously been mentioned, it is possible to define the rhythm of the different clock signals generated by the *BPSK modulator*, but it is also necessary to create the table which contains the values of the different samples of the sine wave. Thus, we are going to create a package, which we are going to name *real2bit*, where all the functions necessary to generate the table are defined.

1. Add the package *real2bit* to the system design:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.MATH_REAL.ALL;
use work.constants.all;

package real2bit is
      subtype double is signed(2*nbits-1 downto 0);
      function truncate (a: real; numdec : natural := ndec) return
signed;
      function extract (a: double; numdec : natural := ndec) return
signed;
      function initialize_table return table;
      constant table_wave : table := initialize_table;
end real2bit;


package body real2bit is

      function truncate (a: real; numdec : natural := ndec)
      return signed is
            variable result: signed(nbits-1 downto 0);
            variable tmp, comp : real := 0.0;
            variable comp_int, sign : integer := 0;
      begin
            -- numdec indicates the number of bits referred to decimals
            tmp := abs(a*(2.0**numdec));
            if a < 0.0 then
                sign:= -1;
            else
                sign:= 1;
            end if;
            for I in nbits-2 downto 0 loop
                comp := comp + 2.0**I;
                comp_int := comp_int + 2**I;
                if tmp < comp then
                    comp := comp - 2.0**I;
                    comp_int := comp_int - 2**I;
                end if;
            end loop;
            result := conv_signed(comp_int*sign,nbits);
            return result;
      end truncate;

      function extract (a: double; numdec : natural := ndec) return
      signed is
            variable result : signed(nbits-1 downto 0);
      begin
            result := signed(a(numdec+nbits-1 downto numdec));
            return result;
      end function extract;

      function initialize_table return table is
      variable result : table;
      begin
            for I in 0 to M-1 loop
                result (I) := truncate(2.0*sin(delta_phi*real(I)));
            end loop;
            return result;
      end function initialize_table;
end real2bit;
```

In this package, two functions has been defined which allow us to work with real numbers, *truncate* and *extract* to be precise, as well as a function to initialize the table with the different samples of the sine wave. The function *truncate* allows us to convert a real number to a binary number of *nbits* bits in two's complement notation (type *signed*), where the least significant bits are referred to the decimal part of the represented number. For example, if we pass to the function *truncate* the value 1.5, according to the definitions in the package *constants* for *nbits* = 12 and *ndec* = 7 (value by default for *numdec* if we do not pass any value as second argument of the function), this function returns "00001.1000000" (the decimal point has been included for a better understanding of the result). On the contrary, *truncate*(-1.5) returns "11110.1000000".

The function *extract* allows us to obtain a new result of type *word* (defined in the package *constants*) from a value of type *double*, result that is obtained by multiplying two values of type *word*. This function is not going to be used in this practice but it is useful when we multiply two values of type *word* and we are interested in recovering the result keeping the same format as that of the operands. As it is known, when a binary data word of *nbits* bits is multiplied by other *nbits*-binary word too, the result will be of 2*nbits* bits, with a shift of the decimal point to the bit position 2*ndec*. The function *extract* only recovers the central part of the result obtained after a multiplication, in order to obtain a value of type *word* again.

Finally, we have defined a function called *initialize_table*, which saves the sample values of the sine wave in an *array* of integers with sign (*signed*) along *M* consecutive positions. This array is defined as the type *table* in the package *constants*. The resulting table, named *table_wave*, is a constant which can be indexed in order to extract its values along the time.

2. Taking into account the previously mentioned, we are going to proceed to design the BPSK modulator. A possible VHDL code would be the next one:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use work.constants.all;
use work.real2bit.all;

entity bpsk is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           serial_data : in  STD_LOGIC;
           clk_data : out  STD_LOGIC;
           clk_spi : out  STD_LOGIC;
           clk_bpsk : out STD_LOGIC;
           data : out  STD_LOGIC_VECTOR (11 downto 0));
end bpsk;

architecture Behavioral of bpsk is
signal pointer : natural range 0 to (M-1) := M-1;
signal value : word := (others => '0');
signal clk_bpsk : std_logic := '0';
begin
process(reset,clk,clk_bpsk)
variable count : natural range 0 to (64*M-1) := 0;
begin
     if reset = '1' then
          clk_bpsk <= '0';
          clk_data <= '0';
          count := 0;
     elsif clk'event and clk = '1' then
          if count = 0 then
              clk_bpsk <= '1';
              clk_data <= '1';
          elsif count mod 64 = 0 then
              clk_bpsk <= '1';
          else
              clk_bpsk <= '0';
              clk_data <= '0';
          end if;
          count := (count + 1) mod (64*M);
     end if;
end process;
process(reset,clk_bpsk)
begin
     if reset = '1' then
          pointer <= M-1;
     elsif clk_bpsk'event and clk_bpsk = '1' then
          pointer <= (pointer + 1) mod M;
     end if;
end process;
clk_spi <= clk;
value <= -table_wave(pointer) when serial_data = '1' else
table_wave(pointer);
-- Use the next line for simulation
data <= std_logic_vector(value);
-- Use the next line for synthesis and comment the previous one
--data <= value + conv_signed(2**(nbits-1),nbits);
clk_bpsk <= clk_bpsk;
end Behavioral;
```

Observe as there exists a first process which generates the clock signals (*clk_data*, *clk_bpsk*) according to the previously mentioned considerations. Moreover, the clock signal for the DAC interface, *clk_spi*, is directly the input signal *clk*, which coincides with the output of the on-board 50 MHz clock oscillator.
A second process modifies the value of a *pointer*, which indexes the table that contains the samples of the sine wave, *table_wave*, at each rising edge of the signal *clk_bpsk*. Given that the samples are sequentially positioned in the table, at each leap of the pointer a new sample will be represented along the time, hence, after *M* jumps, the representation of the sine wave will be completed.
Finally, we have several code lines dedicated to the transmission of the digital word to be represented by the DAC. In first place, we have created an internal signal called *value* to which is assigned the table value if the data digit supplied by the data generator (*serial_data*) is '0' or this same value but negated if the data digit is '1'. This is basically a BPSK modulation. After that, this value is sent to the data output of the

modulator through the port *data*. We have included two code lines, both excluding each other, one of them for simulation, where the data value is directly sent, and another one for the synthesis where a certain quantity is added so as to make the value sent to the DAC interface positive. Take into account that the DAC only works with unsigned integers (type *unsigned*). What has been done is to add a constant value which corresponds to the half of the output range of the DAC ($V_{REF}/2$). Thus, the wave at the output is represented centred round this value.

3. By using the ISE Simulator, check the correct behaviour of the BPSK modulator. Observe as we have included a signal *clk_bpsk* as output wire of the block (by assigning to it the value *clk_bpsk*), which has been done only to ease the analysis of the device, since this signal actually only works internally inside the block.
4. Next it is shown a possible VHDL code which integrates all the blocks previously designed:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.constants.all;

entity system is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           dac_cs : out  STD_LOGIC;
           spi_mosi : out  STD_LOGIC;
           spi_sck : out  STD_LOGIC;
           dac_clr : out STD_LOGIC;
           data : out  STD_LOGIC);
end system;

architecture Behavioral of system is
component bpsk is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           serial_data : in  STD_LOGIC;
           clk_data : out  STD_LOGIC;
           clk_spi : out  STD_LOGIC;
           clk_bpsk : out STD_LOGIC;
           data : out  STD_LOGIC_VECTOR (11 downto 0));
end component bpsk;
component data_gen is
     generic (Nreg : positive := N);
     port (clk,reset : in  STD_LOGIC;
                data,sync : out  STD_LOGIC);
end component data_gen;
component com_dac is
    Port ( clk : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           dac_cs : out  STD_LOGIC;
           dac_clr : out STD_LOGIC;
           spi_mosi : out  STD_LOGIC;
           spi_sck : out  STD_LOGIC;
           data : in  STD_LOGIC_VECTOR(11 downto 0);
           count_out : out std_logic_vector(6 downto 0));
end component com_dac;
signal clk_data, clk_spi, serial_data : std_logic;
signal data_int : std_logic_vector (11 downto 0);
begin
     Data_gen0: data_gen port map (clk_data,reset,serial_data);
     Modulador: bpsk port map
          (clk,reset,serial_data,clk_data,clk_spi,
              data => data_int);
     DAC_interface: com_dac port map
          (clk_spi,reset,dac_cs,dac_clr,spi_mosi,spi_sck,data_int);
     data <= serial_data;
end Behavioral;
```

Observe as there are several signals such as *clk_bpsk*, *sync* and *count_out* which are not used by the block *system*, since they are only added in order to ease the analysis of the different devices.

5. Implement into the FPGA the design of the BPSK modulator. Observe the output signals by using a scope/logic analyzer. Next it is shown a possible configuration for the I/O pins in the FPGA:

```
NET "reset" CLOCK_DEDICATED_ROUTE = FALSE;
NET "dac_clr" LOC = AB13;
NET "dac_cs" LOC = W7;
NET "data" LOC = A13;
NET "clk" LOC = E12;
NET "reset" LOC = T15;
NET "spi_mosi" LOC = AB14;
NET "spi_sck" LOC = AA20;
```