Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

Departament d'Enginyeria Electrònica

EX 3 DIGITAL CIRCUITS AND SYSTEMS

Designing a serial multiplier

1.1 Cooperative group

TEAM NUMBER: _____
DUE DATE: _____ 1st review due date: _____
STUDY TIME:

Study time (in hours)	Group work	Classroom and laboratory sessions		Sessions out of classroom	
	Individual	Student 1			
		Student 2			
		Student 3			

STATEMENT:
My signature below indicates that I have (1) made equitable contribution to EX 3 as a member of the group, (2) read and fully agree with the contents (i.e., results, conclusions, analyses, simulations) of this document, and (3) acknowledged by name anyone outside this group who assisted this learning team or any individual member in the completion of this document.

Today's date: _____

Active members — Roles: (reporter, simulator, etc.)
(1) ____Angélica Rodríguez Sánchez_____ _____
(2) _____ _____
(3) _____ _____

Acknowledgement of individual(s) who assisted this group in completing this document:
(1) _____
(2) _____

1.2 Abstract

In this exercise we will perform a multiplier in VHDL. We will see each of the parts of this multiplier: a 4x4 serial multiplier, a clock divider, a timer, a binary to BCD converter of 8 bits, a quad mux4 and the BCD to 7 segment converter. We will study these parts step by step.

CONTENT

1.3 Problem solution

1.3.1 The structure of the top design

For do the multiplier, we will separate the project in different blocks. These blocks will be grouped in the block that we see in Fig. 1.

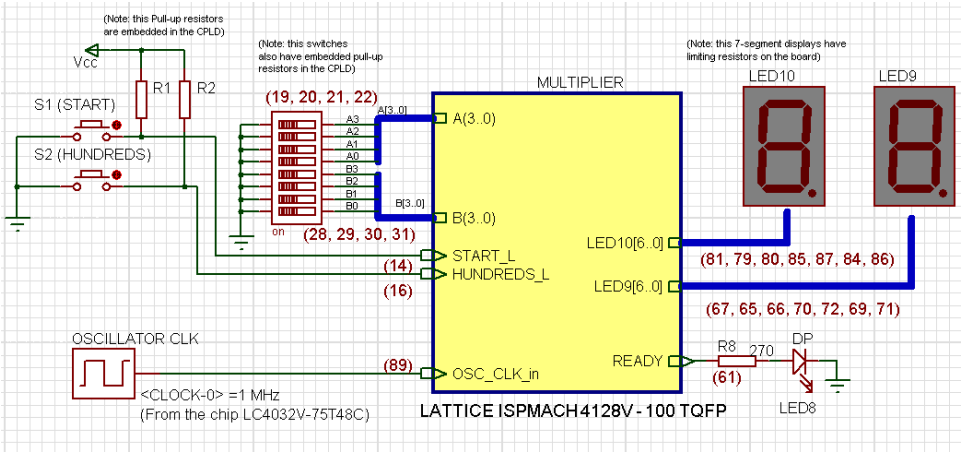

Fig. 1 Schematic of multiplier

In this schematic we can see a multiplier where LED10 show the operand A (the tens) and LED9 show the operand B (the units). Push button HUNDREDS shows the hundreds and start pulse triggers the operation. The result is displayed while READY is high (5 seconds), then the operand are shown again.

1.3.2 Multiplier

The main blocks that we will see for do the multiplier are one frequency divider, one unsigned 4 bits multiplier, one timer, one binary to BCD converter of 8 bits, two quad mux4 and two BCD to 7 segments converter. The design that we will do is in this Fig. 2.

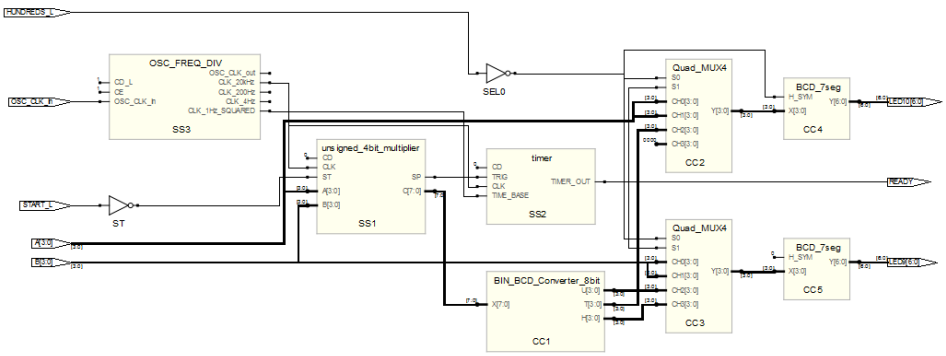

Fig. 2 Design of multiplier

For all this blocks we have a code in VHDL of the entire project in Fig. 3. This code group all the blocks that after break down one to one.

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;
USE IEEE.NUMERIC_STD.all;
ENTITY multiplier IS
        PORT (
                A,B                             : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);   --INPUTS
                START_L                         : IN    STD_LOGIC;
                HUNDREDS_L                      : IN    STD_LOGIC;
                OSC_CLK_in                      : IN    STD_LOGIC;
                LED9, LED10                     : OUT   STD_LOGIC_VECTOR(6 DOWNTO 0);   --OUTPUT
                READY                           : OUT   STD_LOGIC
                );
END multiplier;

ARCHITECTURE STRUCTURAL OF multiplier IS
        COMPONENT OSC_FREQ_DIV IS
        PORT (
                CE                              :IN  std_logic;
                OSC_CLK_in                      :IN  std_logic;
                CD_L                            :IN  std_logic;
                CLK_4Hz                         :OUT std_logic;
                CLK_200Hz                       :OUT std_logic;
                CLK_20kHz                       :OUT std_logic;
                CLK_1Hz_SQUARED                 :OUT std_logic;
                OSC_CLK_out                     :OUT std_logic
          );
        END COMPONENT;

        COMPONENT unsigned_4bit_multiplier IS
        PORT (
                A,B                             : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
                ST, CD, CLK                     : IN    STD_LOGIC;
                C                               : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                SP                              : OUT   STD_LOGIC
          );
        END COMPONENT;

        COMPONENT timer IS
        PORT (
                TRIG, TIME_BASE, CD, CLK        : IN STD_LOGIC;
                TIMER_OUT                               : OUT   STD_LOGIC
          );
        END COMPONENT;

        COMPONENT Bin_BCD_Converter_8bit IS
                PORT (
                        X               : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                        U, T, H         : OUT STD_LOGIC_VECTOR(3 DOWNTO 0)
          );
        END component;
        COMPONENT Quad_MUX4 IS
                PORT (
                CH0     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
                CH1     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
                CH2     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
                CH3     : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
                S0, S1  : IN    STD_LOGIC;
                Y       : OUT   STD_LOGIC_vector(3 downto 0)
          );
        END component;

        COMPONENT BCD_7seg IS
                PORT (
                        X       : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                        H_SYM   : IN    STD_LOGIC;
                        Y       : OUT STD_LOGIC_VECTOR(6 DOWNTO 0)
          );
        END component;

signal Trigger, ST, CLK_4Hz, CLK_200Hz, OSC_CLK_out, CLK_20kHz, CLK_1Hz_SQUARED, SEL0, SEL1      : STD_LOGIC;
signal C                                                                                        : STD_LOGIC_VECTOR(7 downto 0);
signal U, T, H, Digit2, Digit1                                                                  : STD_LOGIC_VECTOR(3 downto 0);



BEGIN

SS1: unsigned_4bit_multiplier
        port map (
                ST => ST,
                A => A,
                B => B,
                CD => '0',
                CLK => CLK_20kHz,
                C => C,
                SP => Trigger
                );
SS2: timer
        port map(
                TRIG => Trigger,
                TIME_BASE => CLK_1Hz_SQUARED,
                CD => '0',
                CLK => CLK_20kHz,
                TIMER_OUT => SEL1
                );

SS3: OSC_FREQ_DIV
        port map(
                CE              =>      '1',
                OSC_CLK_in      =>      OSC_CLK_in,
                CD_L            =>      '1',
                CLK_4Hz         =>      CLK_4Hz,
                CLK_200Hz       =>      CLK_200Hz,
                CLK_20kHz       =>      CLK_20kHz,
                CLK_1Hz_SQUARED =>      CLK_1Hz_SQUARED,
                OSC_CLK_out     => OSC_CLK_out
                );
CC1: Bin_BCD_Converter_8bit
        port map(
                X       =>      C,
                U       =>      U,
                T       =>      T,
                H       =>      H
                );
CC2: Quad_MUX4
        port map(
                CH0             =>      A,
                CH1             =>      A,
                CH2             =>      T,
                CH3             =>      "0000",
                S0              =>      SEL0,
                S1              =>      SEL1,
                Y               =>      Digit2
                );
CC3: Quad_MUX4
        port map(
                CH0             =>      B,
                CH1             =>      B,
                CH2             =>      U,
                CH3             =>      H,
                S0              =>      SEL0,
                S1              =>      SEL1,
                Y               =>      Digit1
                );
CC4: BCD_7seg
        port map(
                X               =>      Digit2,
                Y               =>      LED10,
                H_SYM           =>      SEL0
                );


CC5: BCD_7seg
        port map(
                X               =>      Digit1,
                Y               =>      LED9,
                H_SYM           =>      '0'
                );
ST <= NOT START_L;
READY <= SEL1;
SEL0 <= NOT HUNDREDS_L;

END STRUCTURAL;
```

Fig. 3 Code in VHDL of multiplier

Now we study all the parts of the multiplier step by step. The first block we let's see is the 4x4 unsigned serial multiplier.

1.3.2.1 The 4x4 unsigned serial multiplier

The first block that we study has for inputs the two numbers that we want multiplier and also it has the button ST (start) for do the operations.

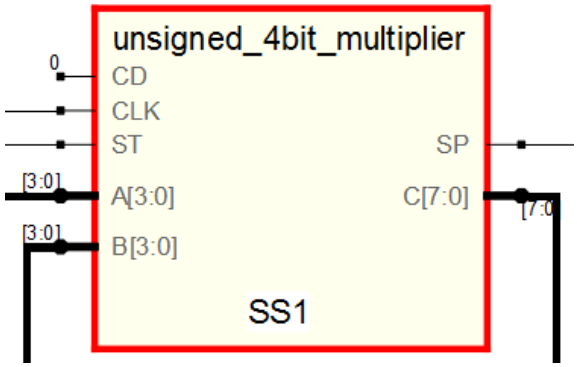

Fig. 4 Schematic of 4x4 unsigned serial multiplier

The code of this block is in Fig. 5 and we can see that this block has two others blocks within: the control unit multiplier and the datapath.

```
----------------------------------------------------------
-- Unsigned 4-bit serial multiplier
----------------------------------------------------------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY unsigned_4bit_multiplier IS
      PORT(     CD, CLK :       IN      std_logic;
                ST      :       IN      std_logic;
                A,B     :       IN      std_logic_vector(3 downto 0);
                C       :       OUT     std_logic_vector(7 downto 0);
                SP      :       OUT     std_logic
            );
END unsigned_4bit_multiplier;

ARCHITECTURE dedicated_processor OF unsigned_4bit_multiplier IS

        COMPONENT control_unit_multiplier IS
                PORT(   CD,CLK,ST       :       IN      std_logic;
                        BRZ, BR0        :       IN      std_logic;
                        LDW, LDC        :       OUT     std_logic;
                        RST             :       OUT     std_logic;
                        SA, SB          :       OUT     std_logic_vector(1 downto 0);
                        SP              :       OUT     std_logic
                );
        END COMPONENT;

        COMPONENT Datapath IS
                PORT(   CD,CLK          :       IN      std_logic;
                        SA, SB          :       IN      std_logic_vector(1 downto 0);
                        LDW, LDC        :       IN      std_logic;
                        RST             :       IN      std_logic;
                        A,B             :       IN      std_logic_vector(3 downto 0);
                        C               :       OUT     std_logic_vector(7 downto 0);
                        BRZ, BR0        :       OUT     std_logic

                );
        END COMPONENT;

-- The internal wires to interconnect the Datapath and the Control Unit
        SIGNAL BRZ, BR0 :       std_logic;      -- Status from the Datapath

        SIGNAL LDW, LDC :       std_logic;      -- Datapath control
        SIGNAL RST      :       std_logic;
        SIGNAL SA, SB   :       std_logic_vector(1 downto 0);

        BEGIN
-- Instantiation of components
```

```
        CU1     : control_unit_multiplier
        PORT MAP (
-- from component name  => to signal or port name
                CLK     => CLK,
                CD      => CD,
                ST      => ST,
                SP      => SP,
                LDW     => LDW,
                LDC     => LDC,
                RST     => RST,
                SA      => SA,
                SB      => SB,
                BRZ     => BRZ,
                BR0     => BR0
        );

        DP1     : Datapath
        PORT MAP (
-- from component name  => to signal or port name
                CLK     => CLK,
                CD      => CD,
                LDW     => LDW,
                LDC     => LDC,
                RST     => RST,
                SA      => SA,
                SB      => SB,
                BRZ     => BRZ,
                BR0     => BR0,
                A       => A,
                B       => B,
                C       => C
        );
END dedicated_processor ;
```

Fig. 5 Code un VHDL of 4x4 unsigned serial multiplier

In this Fig. 6 we can see the schematic inside of the unsigned 4 bit multiplier: we have the control unit multiplier and the datapath as we have seen before.

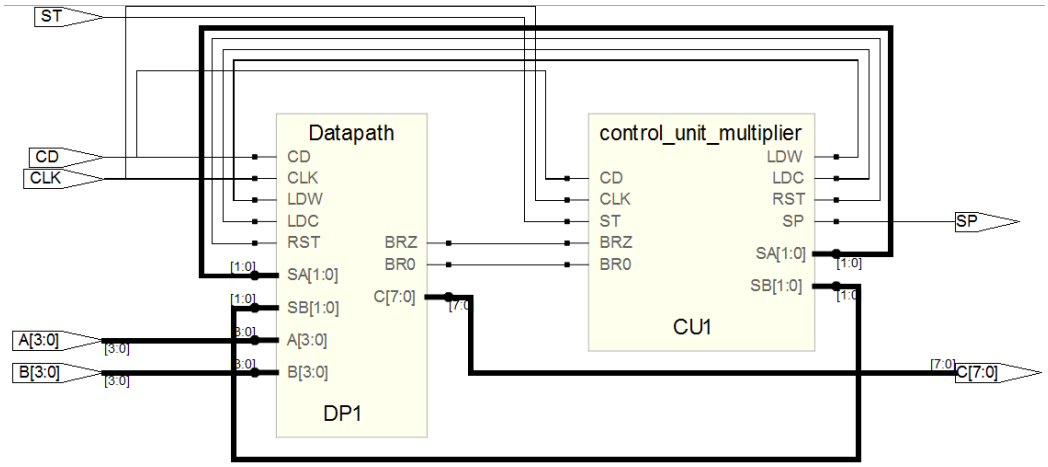

Fig. 6 Schematic inside 4x4 unsigned serial multiplier

Now we study these blocks, first the control unit multiplier and after tha datapath.

1.3.2.1.1 The Control Unit Multiplier

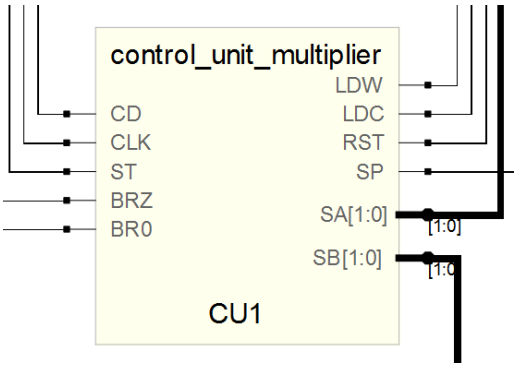

Fig. 7 Schematic of control unit multiplier

The code of the control unit multiplier is in Fig. 8

```
-------------------------------------------------------
-- Control Unit for the 4-bit unsigned multiplier
-------------------------------------------------------

LIBRARY IEEE;
   USE IEEE.STD_LOGIC_1164.ALL;
   USE IEEE.STD_LOGIC_ARITH.ALL;
   USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY control_unit_multiplier IS
        PORT(   CD,CLK,ST       :       IN      std_logic;
                BRZ, BR0        :       IN      std_logic;
                LDW, LDC        :       OUT     std_logic;
                RST             :       OUT     std_logic;
                SA, SB          :       OUT     std_logic_vector(1 downto 0);
                SP              :       OUT     std_logic
        );
END control_unit_multiplier;

ARCHITECTURE FSM_like OF control_unit_multiplier IS

        TYPE State_type IS (Idle, Load_Data, Flags, Load_Result, Add, Shift);
--------> This is important: specifying to the synthesiser the code for the states
        ATTRIBUTE syn_enum_encoding : STRING;
        ATTRIBUTE syn_enum_encoding OF State_type : TYPE IS "sequential";
                -- (It may be:"sequential" (binary); "gray"; "one-hot", etc.
        SIGNAL present_state, future_state      : State_type ;

                -- These two lines work OK, but the state machine is not recognised.
                -- and regular logic is synthesised (so it's not so good as the syn_encoding).
-- ATTRIBUTE enum_encoding : string;
-- ATTRIBUTE enum_encoding of State_type : TYPE IS "sequential";


-- Constants for special states (the first and the last state)
CONSTANT Reset  : State_type := Idle; -- The fisrt state. This is another name for the state Num0

BEGIN
----------------------- State Register: the only clocked block.
----------------------- The "memory" of the system (future events will depend on past events

state_register: PROCESS (CD,CLK)
        BEGIN

        IF CD = '1' THEN                        -- asynchronous reset of the FSM
                present_state <= Reset;
        ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
                present_state <= future_state;
        END IF;

 END PROCESS state_register;


 -------------------------- CC1: Combinational system for calculating next state

 CC_1: PROCESS (present_state, BR0, BRZ, ST)
          BEGIN


                CASE present_state is

                        when Idle =>

                                if ST='0' then

                                        future_state <= Idle;
                                else
                                        future_state <= Load_Data ;
                                end if;

                        when Load_Data =>

                                        future_state <= Flags ;

                        when Flags =>

                                if BRZ='1' then

                                        future_state <= Load_Result;

                                elsif ( BRZ='0' and BR0='1') then

                                         future_state <= Add;


                                elsif ( BRZ='0' and BR0='0') then

                                         future_state <= Shift;
                                end if;

                        when Load_Result =>

                                 future_state <=Idle;
                        when Add         =>

                                 future_state <=Shift;

                        when Shift       =>


                                   future_state <=Flags;
                    end case;
   END PROCESS CC_1;


   -------------------------- CS_2: combinational system for calculating extra outputs
   -------------------------- and outputing the present state (the actual count)

   CC_2: PROCESS (present_state)
            BEGIN

                  CASE present_state is
                          when Idle =>

                                  LDW <='0';
                                  LDC <='0';
                                  RST <='0';
                                  SA  <="00";
                                  SB  <="00";
                                  SP  <='0' ;

                          when Load_Data =>
                                  LDW <='0';
                                  LDC <='0';
                                  RST <='1';
                                  SA  <="11";
                                  SB  <="11";
                                  SP  <='0' ;
                          when Flags =>

                                  LDW <='0';
                                  LDC <='0';
                                  RST <='0';
                                  SA  <="00";
                                  SB  <="00";
                                  SP  <='0' ;
                          when Load_Result =>

                                  LDW <='0';
                                  LDC <='1';
                                  RST <='0';
                                  SA  <="00";
                                  SB  <="00";
                                  SP  <='1' ;
                          when Add         =>
```

```
                                        LDW <='1';
                                        LDC <='0';
                                        RST <='0';
                                        SA  <="00";
                                        SB  <="00";
                                        SP  <='0' ;
                        when Shift      =>
                                        LDW <='0';
                                        LDC <='0';
                                        RST <='0';
                                        SA  <="10";
                                        SB  <="01";
                                        SP  <='0' ;
                end case;

END PROCESS CC_2;

-- Place other logic if necessary

END FSM_like;
```

Fig. 8 Code in VHDL of control unit multiplier

1.3.2.1.2 The Datapath

The second block of the 4x4 unsigned serial multiplier is the datapath that we have in Fig. 9.

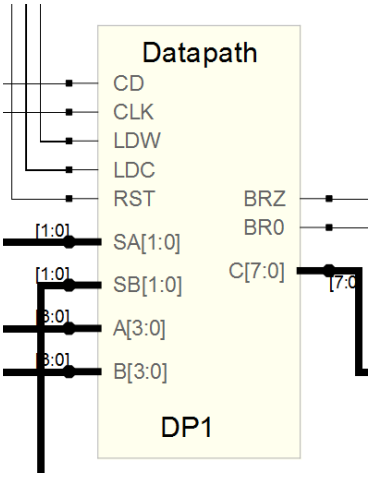

Fig. 9 Schematic of datapath

Below, for understand that we do, we have in Fig. 10 the flow diagram (while designing the datapath) for do the corresponding code.

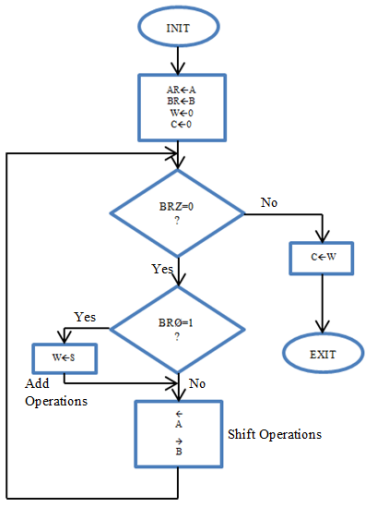

Fig. 10 Diagram of datapath

With this diagram we can do the code in VHDL that we have in Fig. 11.

```
-- 4-bit unsignel multiplier datapath
-- Structural and hiearchical design

LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;

ENTITY Datapath IS
        PORT(   CD,CLK          :       IN      std_logic;
                SA, SB          :       IN      std_logic_vector(1 downto 0);
                LDW, LDC        :       IN      std_logic;
                RST             :       IN      std_logic;
                A,B             :       IN      std_logic_vector(3 downto 0);
                C               :       OUT     std_logic_vector(7 downto 0);
                BRZ, BR0        :       OUT std_logic

        );
END Datapath;


ARCHITECTURE Structural OF Datapath IS
-- Components

        COMPONENT Shift_Data_Reg_4bits IS
                PORT(   CD,CLK  :       IN              std_logic;
                        X               :       IN              std_logic_vector(3 downto 0);
                        Y               :       OUT     std_logic_vector(3 downto 0);
                        S               :       IN              std_logic_vector(1 downto 0);
                        RSI,LSI         :       IN              std_logic

                );
        END COMPONENT;

        COMPONENT Shift_Data_Reg_8bits IS
                PORT(   CD,CLK  :       IN      std_logic;
                        X               :       IN      std_logic_vector(7 downto 0);
                        Y               :       OUT std_logic_vector(7 downto 0);
                        S               :       IN      std_logic_vector(1 downto 0);
                        RSI,LSI :       IN      std_logic

                );
        END COMPONENT;


        COMPONENT Data_Reg_8bit IS
                Port (  CLK     : IN  STD_LOGIC;
                        CD      : IN  STD_LOGIC;
                        RST     : IN  STD_LOGIC;
                        LD      : IN  STD_LOGIC;
                        Y       : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                        X       : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
                );
        END COMPONENT;


        COMPONENT Adder_8bit IS
                PORT (
                        A,B     : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
                        Ci      : IN    STD_LOGIC;
                        S       : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                        Co      : OUT   STD_LOGIC
                );

        END COMPONENT;

-- Internal wires
        SIGNAL S, W, AR :       STD_LOGIC_VECTOR(7 downto 0);
        SIGNAL BR       :       STD_LOGIC_VECTOR(3 downto 0);
        SIGNAL Co       :       STD_LOGIC;

BEGIN
-- Instantiation of components
RegW    : Data_Reg_8bit
        PORT MAP (
-- from component name  => to signal or port name
                CLK     => CLK,
                CD      => CD,
                RST     => RST,
                LD      => LDW,
                Y       => W,
                X       => S
        );

RegC    : Data_Reg_8bit
        PORT MAP (
-- from component name  => to signal or port name
                CLK     => CLK,
                CD      => CD,
                RST     => RST,
                LD      => LDC,
                Y       => C,
                X       => W
        );

RegA    : Shift_Data_Reg_8bits
        PORT MAP (
-- from component name  => to signal or port name
                CLK     => CLK,
                CD      => CD,
                Y       => AR,
                X (3 downto 0)  => A,
                X (7 downto 4)  => "0000",
                S       => SA,
                RSI     => '0',
                LSI     => '0'

        );

RegB    : Shift_Data_Reg_4bits
        PORT MAP (
-- from component name  => to signal or port name
                CLK     => CLK,
                CD      => CD,
                Y       => BR,
                X       => B,
                S       => SB,
                RSI     => '0',
                LSI     => '0'

        );

Adder8  : Adder_8bit
        PORT MAP (
-- from component name  => to signal or port name
                A       => AR,
                B       => W,
                S       => S,
                Ci      => '0',
                Co      => Co
        );

-- Extra logic in the circuit
BRZ <= NOT( BR(0) OR BR(1) OR BR(2) OR BR(3));
BR0 <= BR(0);


END Structural;
```

Fig. 11 Code in VHDL of datapath

Inside of this block we have the adder, the datareg 8 bit, the shift data register of 4 bit and the shift data register of 8 bit. We can see this in Fig. 12 and below we study this blocks step by step.

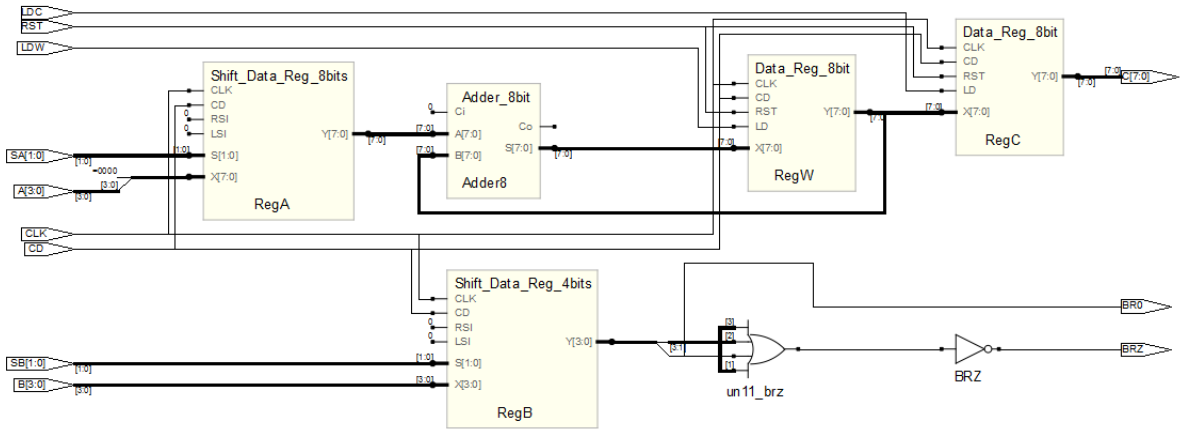

Fig. 12 Schematic inside the datapath

1.3.2.1.2.1 The Adder

We have study an adder previously and we know that an adder of 8 bits it compouned of four adder of two bits or, in this case, we have eight adders of 1 bit. Firt we see the 8 bit adder that we can see in Fig. 13.

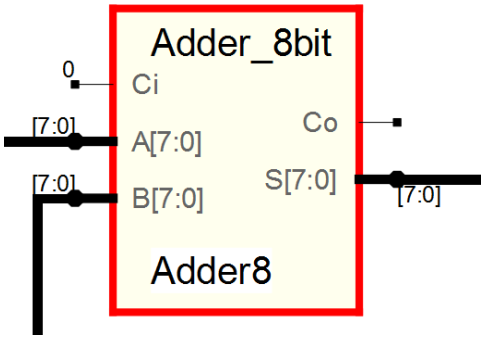

Fig. 13 Schematic of adder

The code of the 8 bit adder is in Fig. 14 and we can see the 8 adder that we need for do this part of the multiplier.

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;

ENTITY Adder_8bit IS
        PORT (  A,B     : IN    STD_LOGIC_VECTOR(7 DOWNTO 0);
                Ci      : IN    STD_LOGIC;
                S       : OUT   STD_LOGIC_VECTOR (7 DOWNTO 0);
                Co      : OUT   STD_LOGIC
             );
END Adder_8bit;

ARCHITECTURE estructura_en_cascada OF Adder_8bit IS
        COMPONENT One_bit_adder IS

                PORT (  Ai,Bi, Ci       : IN STD_LOGIC;
                        So, Co          : OUT STD_LOGIC
                     );
        END COMPONENT;

        SIGNAL  C1, C2, C3, C4,C5, C6, C7, C8   : STD_LOGIC;    -- The wires to connect 1-bit modules together
        SIGNAL  Y                               : STD_LOGIC_VECTOR (7 DOWNTO 0);

BEGIN
        adder_0 : One_bit_adder
                PORT MAP (
                        Ai              => A(0),
                        Bi              => B(0),
                        Ci              => Ci,
                        So              => Y(0),
                        Co              => C1
                );

        adder_1 : One_bit_adder
                PORT MAP (
                        Ai              => A(1),
                        Bi              => B(1),
                        Ci              => C1,
                        So              => Y(1),
                        Co              => C2
                );

        adder_2 : One_bit_adder
                PORT MAP (
                        Ai              => A(2),
                        Bi              => B(2),
                        Ci              => C2,
                        So              => Y(2),
                        Co              => C3
                );

        adder_3 : One_bit_adder
                PORT MAP (
                        Ai              => A(3),
                        Bi              => B(3),
                        Ci              => C3,
                        So              => Y(3),
                        Co              => C4
                );
        adder_4 : One_bit_adder
                PORT MAP (
                        Ai              => A(4),
                        Bi              => B(4),
                        Ci              => C4,
                        So              => Y(4),
                        Co              => C5
                );
        adder_5 : One_bit_adder
                PORT MAP (
                        Ai              => A(5),
                        Bi              => B(5),
                        Ci              => C5,
                        So              => Y(5),
                        Co              => C6
                );
        adder_6 : One_bit_adder
                PORT MAP (
                        Ai              => A(6),
                        Bi              => B(6),
                        Ci              => C6,
                        So              => Y(6),
                        Co              => C7
                );
        adder_7 : One_bit_adder
                PORT MAP (
                        Ai              => A(7),
                        Bi              => B(7),
                        Ci              => C7,
                        So              => Y(7),
                        Co              => C8
                );
S <= Y;
Co <= C8;

END estructura_en_cascada;
```

Fig. 14 Code in VHDL of adder

Inside of this adder we have the one bit adders that we speak previously.

1.3.2.1.2.1.1 One bit adder

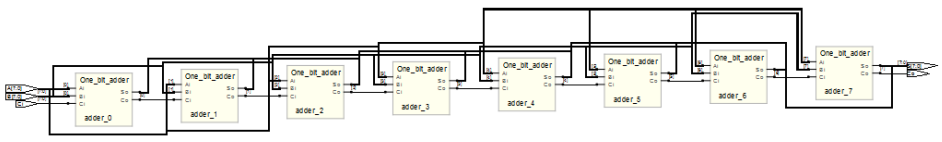

Fig. 15 Schematic of one bit adders

The code that we see in Fig. 16 of the one bit adder is exactly the code that we see in the previous chapter.

```
-------------------------------------------------------
-- A One-bit adder using logic equations (structural)
-------------------------------------------------------

LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;

ENTITY One_bit_adder IS

                PORT (
                        Ai,Bi, Ci       : IN STD_LOGIC;
                        So, Co          : OUT STD_LOGIC
                  );

END One_bit_adder;


ARCHITECTURE logic_equations OF One_bit_adder IS

BEGIN

So <= Ai XOR Bi XOR Ci;

Co <= (Ai AND Bi) OR (Ci AND (Ai OR Bi) );

END logic_equations ;
```

Fig. 16 Code in VHDL of one bit adder

1.3.2.1.2.2 The DataReg 8 bit

The next block we are study is the data register of 8 bits that we see in Fig. 17.

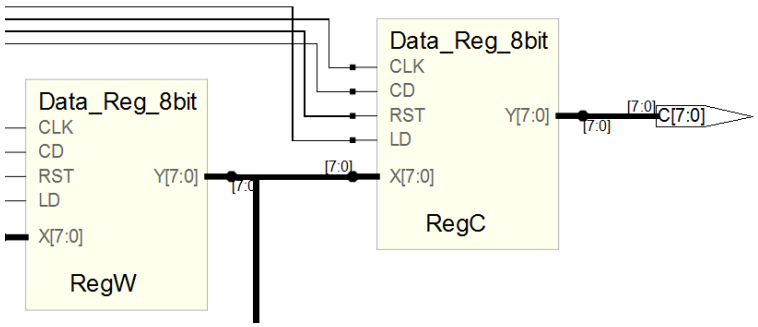

Fig. 17 Schematic of datareg 8 bit

The code of the datareg 8 bit is in Fig. 18 and we can see that if clear direct is '1' we have a all zeros, if it is '0' we pass to the future state, if RST is '1' we have a reset, and if LD is '1' we load a number X in future state.

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY Data_Reg_8bit IS
      Port (    CLK       : IN  STD_LOGIC;
                CD        : IN  STD_LOGIC;
                RST       : IN  STD_LOGIC;
                LD        : IN  STD_LOGIC;
                Y         : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                X         : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
END Data_Reg_8bit;

ARCHITECTURE FSM_like OF Data_Reg_8bit IS

CONSTANT Reset  : STD_LOGIC_VECTOR(7 DOWNTO 0) := "00000000";
SIGNAL present_state,future_state: STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN

State_Register: PROCESS (CD, CLK)
        BEGIN

        IF CD = '1' THEN                        -- reset counter ( an asynchronous reset which we call "Clear Direct
                present_state <= Reset;
        ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
                present_state <= future_state;
        END IF;

END PROCESS State_Register;

CC1: PROCESS (present_state, X, RST, LD)   -- All the block inputs in the sensitivity list
        BEGIN
        IF RST = '1' THEN               -- This is the signal with higher priority
                future_state <= Reset;

                ELSIF LD='1' THEN
                        future_state <= X;

                ELSE
                        future_state <= present_state;
        END IF;

END PROCESS CC1;

Y <= present_state;

END FSM_like;
```

Fig. 18 Code in VHDL of datareg 8 bit

1.3.2.1.2.3 The Shift DataReg 4 bit

The shift datareg that we have in Fig. 19 is very similar to datareg but now the numbers are moved to left or right.

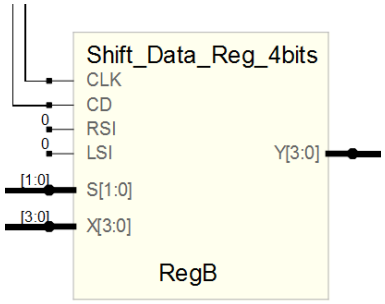

Fig. 19 Schematic of Shift DataReg 4 bit

This is the true table of this block, depending on the value of S1 and S0 in future state we have or present state, or a number loaded, or the number moved to right or left.

S1	S0	Y^+
0	0	Y
0	1	→
1	0	←
1	1	Load data

Fig. 20 True table of Shift DataReg 4 bit

In Fig. 21 we have the code of the Shift DataReg 4 bit and here we can see how to do this true table in code VHDL.

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY Shift_Data_Reg_4bits IS
      Port (     CLK        : IN  STD_LOGIC;
                 CD         : IN  STD_LOGIC;
                 S          : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
                 RSI        : IN  STD_LOGIC;
                 LSI        : IN  STD_LOGIC;
                 Y          : OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
                 X          : IN STD_LOGIC_VECTOR(3 DOWNTO 0)
                              );
END Shift_Data_Reg_4bits;
ARCHITECTURE FSM_like OF Shift_Data_Reg_4bits IS
CONSTANT Reset          : STD_LOGIC_VECTOR(3 DOWNTO 0) := "0000";
SIGNAL present_state,future_state: STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
State_Register: PROCESS (CD, CLK)
        BEGIN

        IF CD = '1' THEN                        -- reset counter ( an asynchronous reset which we call "Clear Direct"
                present_state <= Reset;
        ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
                present_state <= future_state;
        END IF;
END PROCESS State_Register;

CC1: PROCESS (present_state, X, S, RSI, LSI)
        BEGIN
        IF S = "11" THEN
                future_state <= X;
        ELSIF S="01" THEN
                future_state(0) <= present_state(1);
                future_state(1) <= present_state(2);
                future_state(2) <= present_state(3);
                future_state(3) <= RSI;
        ELSIF S="10" THEN
                future_state(3) <= present_state(2);
                future_state(2) <= present_state(1);
                future_state(1) <= present_state(0);
                future_state(0) <= LSI;
        ELSE
                future_state <= present_state;
        END IF;
END PROCESS CC1;
Y <= present_state;
END FSM_like;
```

Fig. 21 Code in VHDL of Shift DataReg 4 bit

~~1.3.2.1.2.4 The Shift DataReg 8 bit~~

This block is the same that the Sift DataReg 4 bit, but with 8 bits.

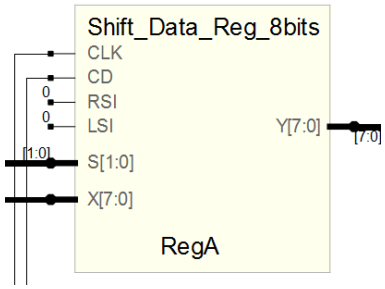

Fig. 22 Schematic of Shift DataReg 8 bit

The code of the Shift DataReg 8 bit is in Fig. 23. Is exactly to the previous code but now we have more bits.

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;
ENTITY Shift_Data_Reg_8bits IS
      Port (    CLK         : IN  STD_LOGIC;
                CD          : IN  STD_LOGIC;
                S           : IN  STD_LOGIC_VECTOR(1 DOWNTO 0);
                RSI         : IN  STD_LOGIC;
                LSI         : IN  STD_LOGIC;
                Y           : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
                X           : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
                              );
END Shift_Data_Reg_8bits;
ARCHITECTURE FSM_like OF Shift_Data_Reg_8bits IS
CONSTANT Reset          : STD_LOGIC_VECTOR(7 DOWNTO 0) := "00000000";
SIGNAL present_state,future_state: STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
State_Register: PROCESS (CD, CLK)
        BEGIN
        IF CD = '1' THEN                        -- reset counter ( an asynchronous reset which we call "Clear Direct"
                present_state <= Reset;
        ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
                present_state <= future_state;
        END IF;
END PROCESS State_Register;
CC1: PROCESS (present_state, X, S, RSI, LSI)
        BEGIN
        IF S = "11" THEN
                future_state <= X;
        ELSIF S="01" THEN
                future_state(0) <= present_state(1);
                future_state(1) <= present_state(2);
                future_state(2) <= present_state(3);
                future_state(3) <= present_state(4);
                future_state(4) <= present_state(5);
                future_state(5) <= present_state(6);
                future_state(6) <= present_state(7);
                future_state(7) <= RSI;
        ELSIF S="10" THEN
                future_state(7) <= present_state(6);
                future_state(6) <= present_state(5);
                future_state(5) <= present_state(4);
                future_state(4) <= present_state(3);
                future_state(3) <= present_state(2);
                future_state(2) <= present_state(1);
                future_state(1) <= present_state(0);
                future_state(0) <= LSI;
        ELSE
                future state <= present state;


        END IF;
END PROCESS CC1;
Y <= present_state;
END FSM_like;
```

Fig. 23 Code in VHDL of Shift DataReg 8 bit

1.3.2.2 The clock divider

The next block is the clock divider that we can see in Fig. 24.

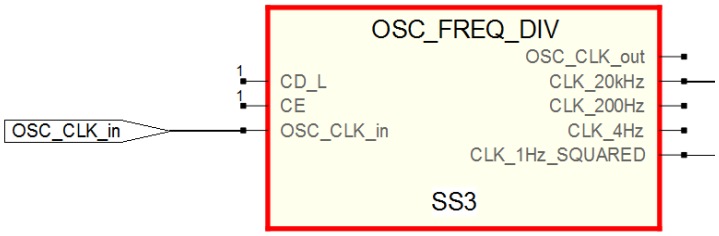

Fig. 24 Schematic of clock divider

We have alredy study this block in another excescise and we know that this block have 5 blocks inside. In Fig. 25 we can see the code of the clock divider where we can see the others blocks.

```
LIBRARY IEEE;
   USE IEEE.STD_LOGIC_1164.ALL;
   USE IEEE.STD_LOGIC_ARITH.ALL;
   USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY OSC_FREQ_DIV IS
      PORT(     CD_L,CE                                 :       IN  std_logic;
                OSC_CLK_in                              :       IN  std_logic;
                OSC_CLK_out                             :       OUT std_logic;
                CLK_20kHz                               :       OUT std_logic;
                CLK_200Hz                               :       OUT std_logic;
                CLK_4Hz                                 :       OUT std_logic;
                CLK_1Hz_SQUARED         :       OUT std_logic
--              DISPLAYS                                :       OUT std_logic_vector(14 downto 0;
-- This DISPLAYS vector is an extra output only to blank unused LED's and segments
-- In this example, only the decimal point in LED9 is used
            );
END OSC_FREQ_DIV;

ARCHITECTURE schematic OF OSC_FREQ_DIV IS
-- Components

        COMPONENT Freq_Div_50 IS
                PORT(   CD,CLK,CE       :       IN  std_logic;
                        TC50            :       OUT std_logic
                );
        END COMPONENT;


        COMPONENT Freq_Div_100 IS
                PORT(   CD,CLK,CE       :       IN  std_logic;
                        TC100           :       OUT std_logic
                );
        END COMPONENT;

        COMPONENT Freq_Div_2 IS
                PORT(   CD,CLK,CE                       :       IN  std_logic;
                                TC2                                     :       OUT std_logic
                );
        END COMPONENT;

        COMPONENT T_Flip_Flop IS
                PORT(   CLK     : IN    STD_LOGIC;
                        CD                      : IN    STD_LOGIC;
                        T                       : IN    STD_LOGIC;
                        Q                       : OUT   STD_LOGIC
                );
        END COMPONENT;

      -- Signals for connecting components together (just the internal wires)
              SIGNAL CD, OSC_CLK                :       std_logic;
              SIGNAL CE2, CE3, CE4, CE5         :       std_logic;

              BEGIN
      -- Instantiation of components

              SS1     : Freq_Div_50
              PORT MAP (
      -- from component name  => to signal or port name
                              CLK             => OSC_CLK,
                              CD              => CD,
                              CE              => CE,
                              TC50    => CE2
                      );

              SS2     : Freq_Div_100
                      PORT MAP (
      -- from component name  => to signal or port name
                              CLK             => OSC_CLK,
                              CD              => CD,
                              CE              => CE2,
                              TC100   => CE3
                      );

              SS3     : Freq_Div_50
                      PORT MAP (
      -- from component name  => to signal or port name
                              CLK             => OSC_CLK,
                              CD              => CD,
                              CE              => CE3,
                              TC50    => CE4
                      );

              SS4     : Freq_Div_2
                      PORT MAP (
      -- from component name  => to signal or port name
                              CLK             => OSC_CLK,
                              CD              => CD,
                              CE              => CE4,
                              TC2     => CE5
                      );

              SS5     : T_Flip_Flop
              PORT MAP (
              -- from component name  => to signal or port name
                              CLK     => OSC_CLK,
                              CD      => CD,
                              T       => CE5,
                              Q       => CLK_1Hz_SQUARED
                      );
      --
      -- connections and logic between components

              -- The circuit's signals that have to be connected to input ports
              OSC_CLK <= OSC_CLK_in;
              CD      <= NOT(CD_L);

              -- The output ports that have to be connected to signals
              OSC_CLK_out     <= OSC_CLK;
              CLK_20kHz       <= CE2;
              CLK_200Hz       <= CE3;
              CLK_4Hz         <= CE4;

      --      Displays OFF when high. In this fashion, only the SQUARED LED will be ON/OFF
      --      DISPLAYS <= "111111111111111";

      END schematic ;
```

Fig. 25 Code in VHDL of the clock divider

Inside the clock divider we have two frequency dividers by 50, a frequency divider by 100, a frequency divider by 2 and one T_Flip_Flop that we study now step by step that we can see in Fig. 26.

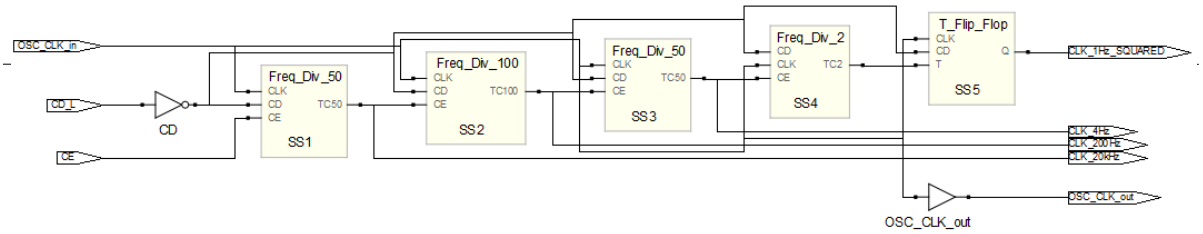

Fig. 26 Schematic inside the clock divider

1.3.2.2.1 The frequency divider by 50

In this part we perform a frequency divider by 50.

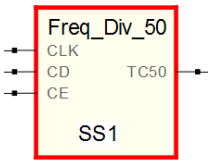

Fig. 27 Schematic of frequency divider by 50

At first we will see the ideal case, as in Active-HDL we didn't observe any delay. In Fig. 28 we have the code in VHDL of the ideal case.

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY Freq_Div_50 IS
      Port (     CLK      : IN    STD_LOGIC;
                 CD                          : IN    STD_LOGIC;
                 CE                          : IN    STD_LOGIC;
                 --- Q                       : OUT   STD_LOGIC_VECTOR(23 DOWNTO 0);
                 TC50            : OUT   STD_LOGIC

        );

END Freq_Div_50;

                    -- Internal desciption in FSM style

ARCHITECTURE FSM_like OF Freq_Div_50 IS
-- This is the real and necessary Max_Count:
-- CONSTANT Max_Count    : STD_LOGIC_VECTOR(23 DOWNTO 0) := "110000000001000111101011"; -- terminal_count after 10 states 12587500

-- This the the Max_Count modified to speed simulations.
-- So you must comment this line and activate the above one if you plan to use the real UP2 board
CONSTANT Max_Count      : STD_LOGIC_VECTOR(5 DOWNTO 0) := "110001";


CONSTANT Reset          : STD_LOGIC_VECTOR(5 DOWNTO 0) := "000000"; -- 1100 0000 0001 0001 1110 1100 --1100 0000 0001 0001 1110 1011

-- Internal wires

SIGNAL present_state,future_state: STD_LOGIC_VECTOR(5 downto 0) := Reset;

BEGIN
------------------------------ the only clocked block : the state register
state_register: PROCESS (CD, CLK)
        BEGIN

        IF CD = '1' THEN                        -- reset counter
                present_state <= Reset;
        ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
                present_state <= future_state;
        END IF;

END PROCESS state_register;

--------------------------- ESS state_registercombinational system for calculating next state
CS_1: PROCESS (present_state, CE)
        BEGIN
        IF CE = '1' THEN
                IF(present_state < Max_Count ) THEN
                        future_state <= present_state + 1 ;
                ELSE
                        future_state <= Reset;
                END IF;
        ELSE
                future_state <= present_state;  -- count disable
        END IF;

END PROCESS CS_1;

--------------------------- CS_2: combinational system for calculating extra outputs
---------------------------- and outputing the present state (the actual count)

TC50 <= '1' WHEN ((present_state = Max_Count)AND CE = '1') ELSE '0';  --terminal count
-- Q <= present_state; In this application, there is no need to output the count value Q


END FSM_like;
```

Fig. 28 Code in VHDL of frequency divider by 50

As we shall see in *Fig. 29*, we speak of an ideal system. If we do a zoom we can see no delay from CLK to TC10.

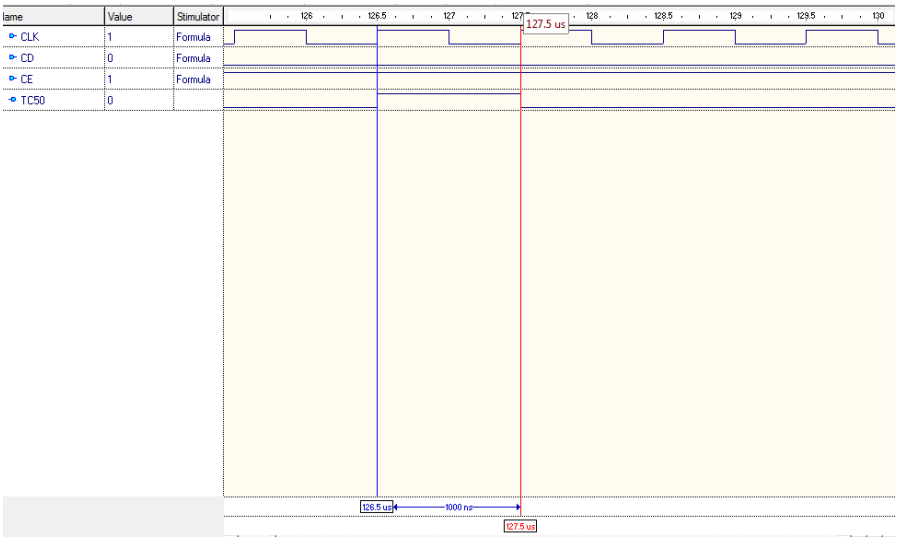

Fig. 29 Simulation ideal frequency divider by 50

And now, in *Fig. 30*, we let's see the case in which we observe this delay. For this we need create the .vho and the .sdf files. Once created these files we see the next simulation.

Fig. 30 Simulation real frequency divider by 50

Now we can see clearly that a delay exist from the clock give a pulse until the output reply to this pulse. In this case we have a delay of 8000 ps.

1.3.2.2.2 The frequency divider by 100

Now we will perform a frequency divider by 100 that we see in Fig. 31.

Fig. 31 Schematic of frequency divider by 100

We'll see the same that the previous case but with TC100 pulses every 100 clock's. In we see the VHDL code of the divisor.

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY Freq_Div_100 IS
      Port (    CLK       : IN    STD_LOGIC;
                CD                                : IN    STD_LOGIC;
                CE                                : IN    STD_LOGIC;
                -- Q                              : OUT   STD_LOGIC_VECTOR(23 DOWNTO 0);
                TC100           : OUT   STD_LOGIC

            );

END Freq_Div_100;

                        -- Internal desciption in FSM style

ARCHITECTURE FSM_like OF Freq_Div_100 IS
-- This is the real and necessary Max_Count:
-- CONSTANT Max_Count    : STD_LOGIC_VECTOR(23 DOWNTO 0) := "110000000001000111101011"; -- terminal_count after 10 states 12587500

-- This the the Max_Count modified to speed simulations.
-- So you must comment this line and activate the above one if you plan to use the real UP2 board
CONSTANT Max_Count      : STD_LOGIC_VECTOR(6 DOWNTO 0) := "1100100";


CONSTANT Reset          : STD_LOGIC_VECTOR(6 DOWNTO 0) := "0000000"; -- 1100 0000 0001 0001 1110 1100 --1100 0000 0001 0001 1110 1011

-- Internal wires

SIGNAL present_state,future_state: STD_LOGIC_VECTOR(6 downto 0) := Reset;

BEGIN
------------------------------ the only clocked block : the state register
state_register: PROCESS (CD, CLK)
        BEGIN

        IF CD = '1' THEN                        -- reset counter
                present_state <= Reset;
        ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
                present_state <= future_state;
        END IF;

END PROCESS state_register;



------------------------- ESS state_registercombinational system for calculating next state
CS_1: PROCESS (present_state, CE)
        BEGIN
        IF CE = '1' THEN
                IF(present_state < Max_Count ) THEN
                        future_state <= present_state + 1 ;
                ELSE
                        future_state <= Reset;
                END IF;
        ELSE
                future_state <= present_state;  -- count disable
        END IF;

END PROCESS CS_1;

------------------------- CS_2: combinational system for calculating extra outputs
---------------------------- and outputing the present state (the actual count)

TC100 <= '1' WHEN ((present_state = Max_Count)AND CE = '1') ELSE '0';  --terminal count
-- Q <= present_state; In this application, there is no need to output the count value Q

END FSM_like;
```

Fig. 32 Code in VHDL of frequency divider by 100

In the simulation we have again two cases, first is the ideal case where we don't have delay and the second is the real case. In *Fig. 33* we have the ideal simulation of the divider.

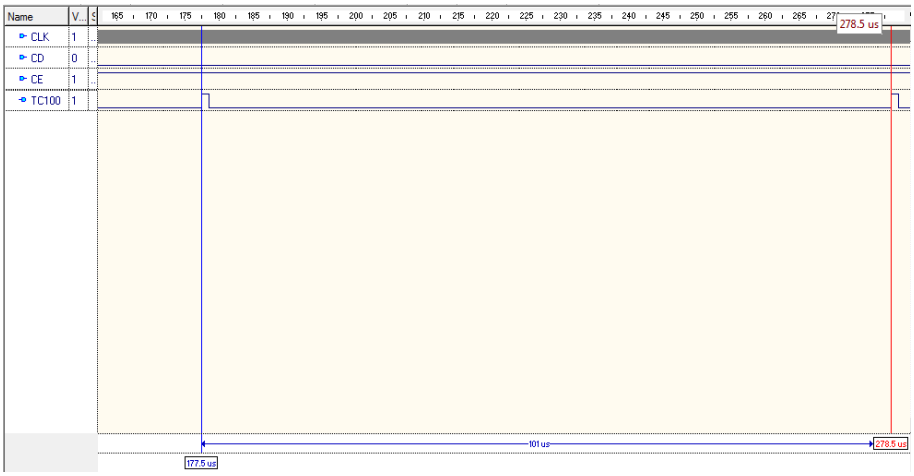

Fig. 33 Simulation ideal frequency divider by 100

Here we can see how often we have a pulse in TC100, it repeats every 100 clock's when CE is '1' and we are not resetting.

Now we see the real case and we will have the delay between the input and output.

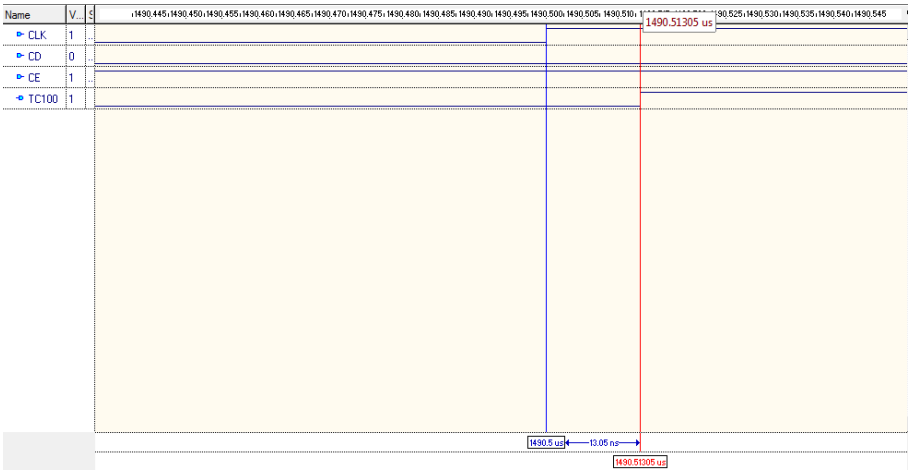

Fig. 34 Simulation real frequency divider by 100

In this *Fig. 34* we can appreciate a delay between CLK and TC100 of 13,05 ns.

1.3.2.2.3 The frequency divider by 2

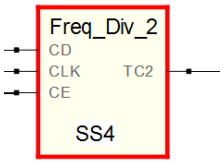

Fig. 35 Schematic of frequency divider by 2

The code of this block is very similar with the frequency divider by 50 and 100.

```
LIBRARY IEEE;
   USE IEEE.STD_LOGIC_1164.ALL;
   USE IEEE.STD_LOGIC_ARITH.ALL;
   USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY Freq_Div_2 IS
      PORT(CD,CLK,CE            :       IN  std_logic;
           TC2                  :       OUT std_logic
          );
END Freq_Div_2;

ARCHITECTURE FSM_like OF Freq_Div_2 IS

CONSTANT Max_Count      : STD_LOGIC := '1';
CONSTANT Reset          : STD_LOGIC := '0';

-- Internal state machine wires -
SIGNAL present_state,future_state: STD_LOGIC := Reset;
-- Remember that this idea of ":= Reset," initalising the wires, has sense only for the simulator, not
-- for a real synthesised flatened circuit, in which CD is used to reset the flip-flops

BEGIN
------------------------------ the only clocked block : the state register
state_register: PROCESS (CD, CLK)
        BEGIN

        IF CD = '1' THEN                                        -- reset counter
                present_state <= Reset;
        ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
                present_state <= future_state;
        END IF;

END PROCESS state_register;

------------------------- CC1: combinational circuit for calculating next state
CC1: PROCESS (present_state, CE)
        BEGIN
        IF CE = '1' THEN
                IF(present_state =  Reset ) THEN
                        future_state <=  Max_Count;  --(only 2 states, no need to add )
                ELSE
                        future_state <= Reset;
                END IF;
        ELSE
                future_state <= present_state;  -- count disable
        END IF;

END PROCESS CC1;

------------------------- CC2: combinational circuit for calculating extra outputs

TC2 <= '1' WHEN ((present_state = Max_count)AND CE = '1') ELSE '0';  --terminal count

END FSM_like;
```

Fig. 36 Schematic of multiplier

In the *Fig. 37* we have the simulation.

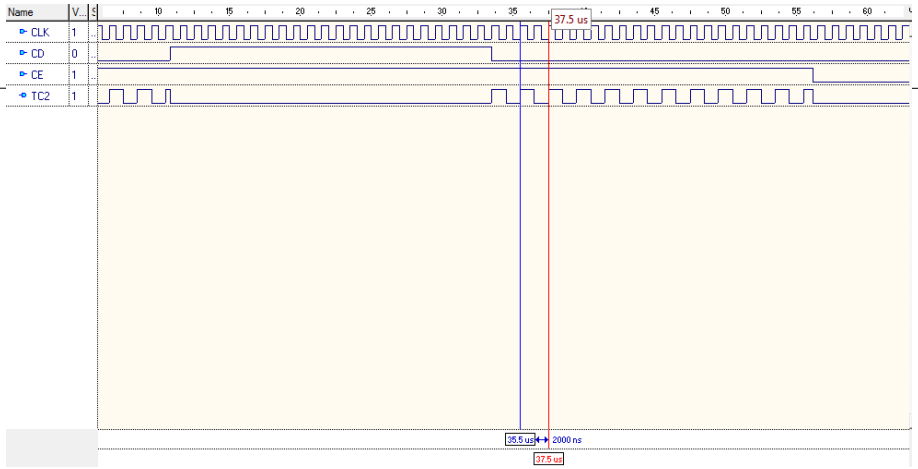

Fig. 37 Simulation frequency divider by 2

In the simulation we check that every two clock's we have a pulse in output. To finish we are doing a T-Flip-Flop.

1.3.2.2.4 The T_Flip_Flop

Finally we are doing a T-Flip-Flop that we can see in Fig. 38

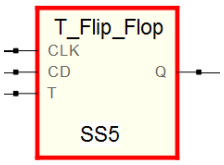

Fig. 38 Schematic of T-Flip-Flop

The code of this block is in Fig. 39

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;

ENTITY T_Flip_Flop IS
      Port (     CLK      : IN    STD_LOGIC;
                 CD       : IN    STD_LOGIC;
                 T        : IN    STD_LOGIC;
                 Q        : OUT   STD_LOGIC
            );

END T_Flip_Flop;

                        -- Internal desciption in FSM style

ARCHITECTURE FSM_like OF T_Flip_Flop IS

CONSTANT Reset  : STD_LOGIC := '0';

-- Internal wires

SIGNAL present_state,future_state: STD_LOGIC := Reset;
  -- This thing of initialising these signals to the "Reset" state,
  -- is only an issue for the functional simulator. Once the circuit
  -- is synthesised, this thing is completely irrelevant.

BEGIN
-------------------------------- the only clocked block : the state register
state_register: PROCESS (CD, CLK)
        BEGIN

        IF CD = '1' THEN                        -- reset counter
                present_state <= Reset;
        ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
                present_state <= future_state;
        END IF;

END PROCESS state_register;

-------------------------- next state logic
-- A T flip flop invert the output when T = 1, and do nothing when T = 0

CS_1: PROCESS (present_state, T)
        BEGIN
        IF T = '1' THEN
                        future_state <= NOT (present_state);
        ELSE
                future_state <= present_state;
        END IF;

END PROCESS CS_1;

------------------------- CS_2: combinational system for calculating extra outputs
-- Very simple in this case, a buffer.
Q <= present_state;


END FSM_like;
```

Fig. 39 Code in VHDL of T-Flip-Flop

We see the simulation in the *Fig. 40* and we check the proper operation.

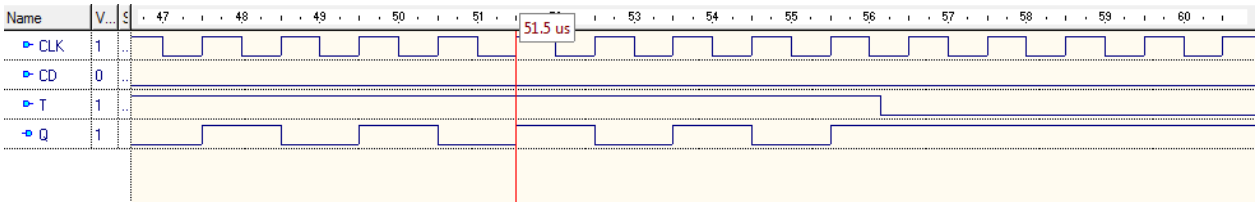

Fig. 40 Simulation T-FF

1.3.2.3 The Timer

Now we are study the timer that we see in Fig. 41.

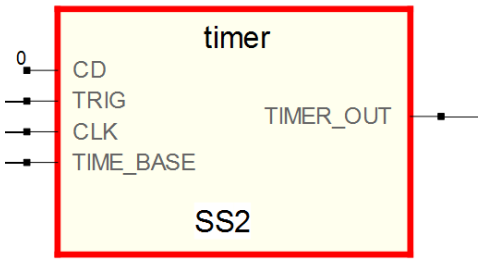

Fig. 41 Schematic of Timer

In Fig. 42 we have the code of this timer and we can see that this block has two blocks inside: the control timer and a counter.

```
LIBRARY IEEE;
   USE IEEE.STD_LOGIC_1164.ALL;
   USE IEEE.STD_LOGIC_ARITH.ALL;
   USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY timer IS
      PORT(
                        CD                              :       IN  std_logic;
                        TRIG                            :       IN  std_logic;
                        CLK                             :       IN std_logic;
                        TIME_BASE                       :       IN std_logic;
                        TIMER_OUT                       :       OUT std_logic
            );
END timer;

ARCHITECTURE schematic OF timer IS
-- Components
      COMPONENT control_unit_timer IS
      PORT(
                TRIG,TC5,CD,CLK                 :       IN  std_logic;
                Enable,Clear                    :       OUT std_logic
                );

        END COMPONENT;

        COMPONENT counter IS

      PORT(
                CD,CLK,CE                       :       IN  std_logic;
                Q                               :       OUT std_logic_vector (2 downto 0);
                TC8                             :       OUT std_logic
                );

        END COMPONENT;

-- Signals for connecting components together (just the internal wires)

        SIGNAL CE, Clear, TC5, Q0, Q1, Q2, Z, TC8       :       std_logic;

        BEGIN

-- Instantiation of components
```

```
    FSM     : control_unit_timer
PORT MAP (
                        -- from component name  => to signal or port name
                                        TRIG            => TRIG,
                                        CLK             => CLK,
                                        CD              => CD,
                                        Clear           => Clear,
                                        Enable          => CE,
                                        TC5             => TC5
                );

    three_bit_Counter       : counter
PORT MAP (
                                        CLK     => TIME_BASE,
                                        CD      => Clear,
                                        CE      => CE,
                                        TC8     => TC8,
                                        Q(0)    => Q0,
                                        Q(1)    => Q1,
                                        Q(2)    => Q2
                );
--Detector '5'
Z   <=   NOT(Q1);

TC5 <= (Q0 AND Z AND Q2);

-- The output ports that have to be connected to signals

TIMER_OUT       <= CE;

END schematic ;
```

Fig. 42 Code in VHDL of Timer

Inside this timer we have two blocks: a counter and a control unit timer that we can see in Fig. 43.

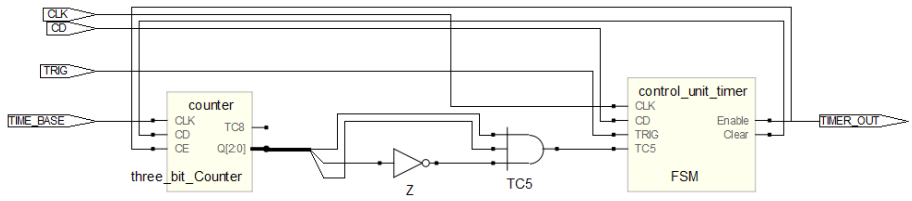

Fig. 43 Schematic inside of Timer

1.3.2.3.1 The Counter

We will perform a counter to eight as show in Fig. 44.

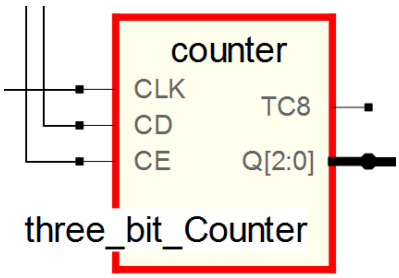

Fig. 44 Schematic of counter

In the code of Fig. 45 we can see that the counter goes from 0 to 7 depending the value of CE.

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;


ENTITY counter IS
      Port (    CLK     : IN    STD_LOGIC;
                CD              : IN    STD_LOGIC;
                CE              : IN    STD_LOGIC;
                Q               : OUT   STD_LOGIC_VECTOR(2 DOWNTO  0);
                TC8     : OUT   STD_LOGIC
        );

END counter;

                        -- Internal desciption in FSM style

ARCHITECTURE FSM_like OF counter IS

-- Internal wires
                                -- State signals declaration

        TYPE State_type IS (Num0, Num1, Num2, Num3, Num4, Num5, Num6, Num7) ;
--------> This is important: specifying to the synthesiser the code for the states
        ATTRIBUTE syn_enum_encoding : STRING;
        ATTRIBUTE syn_enum_encoding OF State_type : TYPE IS "sequential";
                -- (It may be:"sequential" (binary); "gray"; "one-hot", etc.
        SIGNAL present_state, future_state      : State_type ;


-- Constants for special states (the first and the last state)
CONSTANT Reset          : State_type := Num0; -- The fisrt state. This is another name for the state Num0
CONSTANT Max_count      : State_type := Num7; -- The last state. This is anather way to name the state Num7

BEGIN

---------------------- State Register: the only clocked block.

state_register: PROCESS (CD,CLK)
        BEGIN

        IF CD = '1' THEN                        -- reset counter
                present_state <= Reset;
        ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
                present_state <= future_state;
        END IF;

END PROCESS state_register;

-------------------------- CC1: Combinational system for calculating next state

CC_1: PROCESS (present_state, CE)
        BEGIN
                IF CE = '0' THEN
                        future_state <= present_state;  -- count disable
                ELSE                                    -- just a simple state up count
                        CASE present_state IS
                                WHEN Num0 =>
                                        future_state <= Num1 ;
                                WHEN Num1 =>
                                        future_state <= Num2 ;
                                WHEN Num2 =>
                                        future_state <= Num3 ;
                                WHEN Num3 =>
                                        future_state <= Num4 ;
                                WHEN Num4 =>
                                        future_state <= Num5 ;
                                WHEN Num5 =>
                                        future_state <= Num6 ;
                                WHEN Num6 =>
                                        future_state <= Num7 ;
                                WHEN Num7 =>
                                        future_state <= Num0 ;
                        END CASE;
                END IF;
END PROCESS CC_1;


CC_2: PROCESS (present_state, CE)
          BEGIN
-- The terminal count output
                IF ((present_state = Max_count) AND (CE = '1') ) THEN
                        TC8 <= '1';
                ELSE
                        TC8 <= '0';
                END IF;

-- And now just copying the present state to the output:
                CASE present_state IS
                                WHEN Num0 =>
                                        Q <= "000";
                                WHEN Num1 =>
                                        Q <= "001";
                                WHEN Num2 =>
                                        Q <= "010";
                                WHEN Num3 =>
                                        Q <= "011";
                                WHEN Num4 =>
                                        Q <= "100";
                                WHEN Num5 =>
                                        Q <= "101";
                                WHEN Num6 =>
                                        Q <= "110";
                                WHEN Num7 =>
                                        Q <= "111";

                END CASE ;


END PROCESS CC_2;

END FSM_like;
```

Fig. 45 Code in VHDL of counter

1.3.2.3.2 The Control Unit Timer

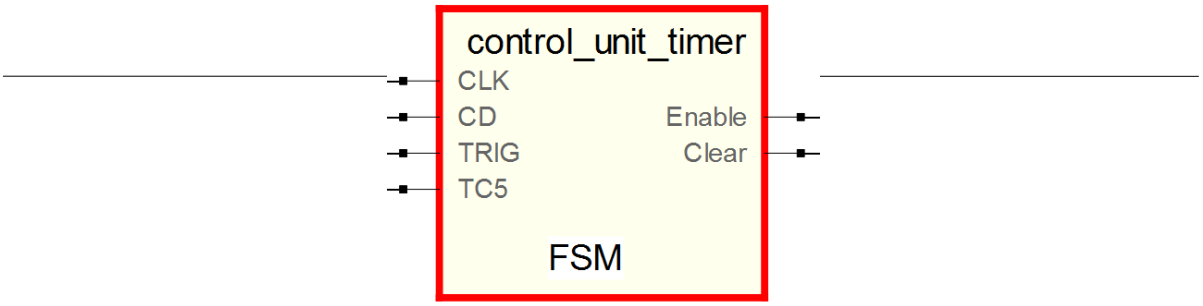

Fig. 46 Schematic of control unit timer

In Fig. 46 we have the control unit timer. In the code of the control unit timer (Fig. 47) we can see that depending the inputs (if we are in case idle, clear_s or count) we will have different values in outputs enable and clear.

```
LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;
USE  IEEE.STD_LOGIC_ARITH.all;
USE  IEEE.STD_LOGIC_UNSIGNED.all;


ENTITY control_unit_timer IS
      Port (     CLK             : IN    STD_LOGIC;
                 CD              : IN    STD_LOGIC;
                 TRIG            : IN    STD_LOGIC;
                 TC5             : IN    STD_LOGIC;
                 Enable          : OUT   STD_LOGIC;
                 Clear           : OUT   STD_LOGIC

      );

END control_unit_timer;

                        -- Internal desciption in FSM style

ARCHITECTURE FSM_like OF control_unit_timer IS

-- Internal wires

       TYPE State_type IS (Idle, Clear_s, Count);
--------> This is important: specifying to the synthesiser the code for the states
       ATTRIBUTE syn_enum_encoding : STRING;
       ATTRIBUTE syn_enum_encoding OF State_type : TYPE IS "sequential";
                 -- (It may be:"sequential" (binary); "gray"; "one-hot", etc.
       SIGNAL present_state, future_state      : State_type ;


-- Constants for special states (the first and the last state)
CONSTANT Reset          : State_type := Idle; -- The fisrt state. This is another name for the state Idle
CONSTANT Max_count      : State_type := Count; -- The last state. This is anather way to name the state Count

BEGIN

----------------------- State Register: the only clocked block.
----------------------- The "memory" of the system (future events will depend on past events
state_register: PROCESS (CD,CLK)
       BEGIN

       IF CD='1' THEN                  -- reset counter
            present_state <= Reset;
       ELSIF (CLK'EVENT and CLK = '1') THEN    -- Synchronous register (D-type flip-flop)
            present_state <= future_state;
       END IF;

END PROCESS state register;
-------------------------- CC1: Combinational system for calculating next state
CC_1: PROCESS (present_state, TRIG, TC5)
       BEGIN
                       CASE present_state IS
                              WHEN Idle =>
                                     IF TRIG = '0' THEN
                                     future_state <= present_state;
                                     ELSIF (TRIG = '1') THEN
                                     future_state <= Clear_s ;
                                     END IF;
                              WHEN Clear_s =>
                                     future_state <= Count ;
                              WHEN Count  =>
                                     IF TC5='0' THEN
                                     future_state <= present_state;
                                     ELSE
                                     future_state <= Idle ;
                                     END IF;

                       END CASE;

END PROCESS CC_1;


CC_2: PROCESS (present_state)
       BEGIN
-- The terminal count output
-- And now just copying the present state to the output:
                CASE present_state IS
                              WHEN Idle =>
                                     Enable <= '0';
                                     Clear <= '0' ;
                              WHEN Clear_s =>
                                     Enable <= '0';
                                     Clear <= '1';
                              WHEN Count =>
                                     Enable <= '1';
                                     Clear <= '0';
                END CASE ;

END PROCESS CC_2;

END FSM_like;
```

Fig. 47 Code in VHDL of Control Unit Timer

1.3.2.4 The Binary to BCD converter of 8 bit

This block has for input a binary number of 8 bits and the outputs are this number converted to BCD in units, tens and hundreds. For do this we need another block, it is the binary to BCD type 74185.

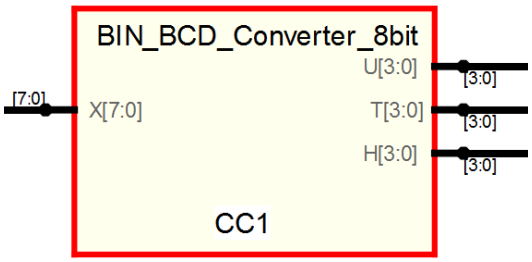

Fig. 48 Schematic of Binary to BCD converter

The code of the binary to BCD converter of 8 bit is in this Fig. 49

```
-------------------------------------------------------
-- A sample design for building circuits cascading
-- basic blocks
-------------------------------------------------------
-- A BIN_BCD_Converter_8bit
-------------------------------------------------------

LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;

ENTITY BIN_BCD_Converter_8bit IS

        PORT (
                X               : IN    STD_LOGIC_VECTOR (7 DOWNTO 0);
                U               : OUT   STD_LOGIC_VECTOR (3 DOWNTO 0);
                T               : OUT   STD_LOGIC_VECTOR (3 DOWNTO 0);
                H               : OUT   STD_LOGIC_VECTOR (3 DOWNTO 0)
                );

END BIN_BCD_Converter_8bit;

ARCHITECTURE estructura_en_cascada OF BIN_BCD_Converter_8bit IS

-- The elemental component to be used:

        COMPONENT Bin_BCD_type74185 IS

                port (
                        A,B,C,D,E       :       in STD_LOGIC;
                        Y               :       out STD_LOGIC_VECTOR(6 DOWNTO 1)
                          );


        END COMPONENT;

-- Signals

        SIGNAL  K               : STD_LOGIC_VECTOR (8 DOWNTO 0);

BEGIN

-- Instantiation of up to 3 basic 1-BIN-BCD:

        adder_0 : Bin_BCD_type74185
                PORT MAP (
                                -- from component name  => to signal or port name
                        A               => X(3),
                        B               => X(4),
                        C               => X(5),
                        D               => X(6),
                        E               => X(7),
                        Y(1)    => K(0),
                        Y(2)    => K(1),
                        Y(3)    => K(2),
                        Y(4)    => K(3),
                        Y(5)    => K(4),
                        Y(6)    => K(5)
                );
        adder_1 : Bin_BCD_type74185
                PORT MAP (
                                -- from component name  => to signal or port name
                        A               => X(1),
                        B               => X(2),
                        C               => K(0),
                        D               => K(1),
                        E               => K(2),
                        Y(1)    => U(1),
                        Y(2)    => U(2),
                        Y(3)    => U(3),
                        Y(4)    => T(0),
                        Y(5)    => K(6),
                        Y(6)    => K(7)
                );
        adder_2 : Bin_BCD_type74185
                PORT MAP (
                                -- from component name  => to signal or port name
                        A               => K(6),
                        B               => K(3),
                        C               => K(4),
                        D               => K(5),
                        E               => '0',
                        Y(1)    => T(1),
                        Y(2)    => T(2),
                        Y(3)    => T(3),
                        Y(4)    => H(0),
                        Y(5)    => H(1),
                        Y(6)    => K(8)
                );


U(0)<= X(0);
H(2)<='0';
H(3)<='0';

END estructura_en_cascada;
```

Fig. 49 Code in VHDL of binary to BCD converter

The circuit inside the binary to BCD converter is in Fig. 50. In this figure we can see the next block: the binary to BCD type 74185.

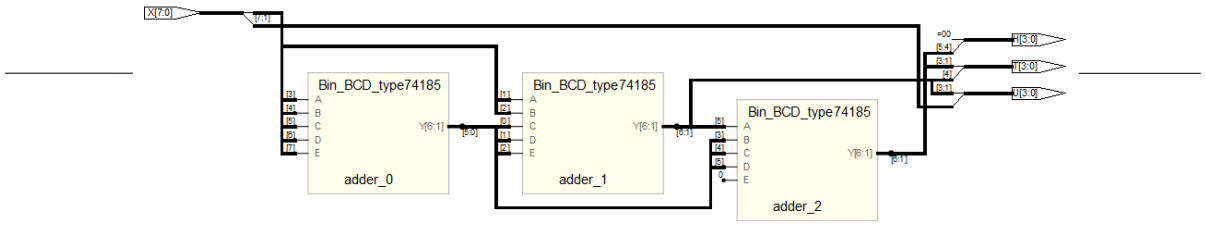

Fig. 50 Schematic inside the binary to BCD converter

1.3.2.4.1 The Binary to BCD type 74185

The code of this block is in Fig. 51. We can see the true table of this converter, the input of each block has 5 bits and in the outputs we have 6 bits.

```
library ieee;
use ieee.std_logic_1164.all;

entity Bin_BCD_type74185 is

                port (
                        A,B,C,D,E       :       in STD_LOGIC;
                        Y               :       out STD_LOGIC_VECTOR(6 DOWNTO 1)
                          );

end Bin_BCD_type74185;

        -- EL vector d'entrada és: V_ENT(4) = E; V_ENT(3) = D; V_ENT(2) = C; V_ENT(1) = B; V_ENT(0)=A

        -- EL vector de sortida és V_SOR(5) = Y(1); V_SOR(4) = Y(2); V_SOR(3) = Y(3); V_SOR(2) = Y(4); V_SOR(1) = Y(5); V_SOR(0) = Y(6)

architecture Arch_Taula_Veritat of Bin_BCD_type74185 is

                SIGNAL  V_ENT           : STD_LOGIC_VECTOR (4 DOWNTO 0);
                SIGNAL  V_SOR           : STD_LOGIC_VECTOR (5 DOWNTO 0);

begin

                        with V_ENT select
                        --                              Sortides        Entrades
                        --                               --------        -------
                                        V_SOR <= "000000" when "00000",
                                                 "000001" when "00001",
                                                 "000010" when "00010",
                                                 "000011" when "00011",
                                                 "000100" when "00100",
                                                 "001000" when "00101",
                                                 "001001" when "00110",
                                                 "001010" when "00111",
                                                 "001011" when "01000",
                                                 "001100" when "01001",
                                                 "010000" when "01010",
                                                 "010001" when "01011",
                                                 "010010" when "01100",
                                                 "010011" when "01101",
                                                 "010100" when "01110",
                                                 "011000" when "01111",
                                                 "011001" when "10000",
                                                 "011010" when "10001",
                                                 "011011" when "10010",
                                                 "011100" when "10011",
                                                 "100000" when "10100",
                                                 "100001" when "10101",
                                                 "100010" when "10110",
                                                 "100011" when "10111",
                                                 "100100" when "11000",
                                                 "101000" when "11001",
                                                 "101001" when "11010",
                                                 "101010" when "11011",
                                                 "101011" when "11100",
                                                 "101000" when "11101",
                                                 "110000" when "11110",
                                                 "110001" when others;

                V_ENT(4) <= E;
                V_ENT(3) <= D;
                V_ENT(2) <= C;
                V_ENT(1) <= B;
                V_ENT(0) <= A;


                Y(6) <= V_SOR(5);
                Y(5) <= V_SOR(4);
                Y(4) <= V_SOR(3);
                Y(3) <= V_SOR(2);
                Y(2) <= V_SOR(1);
                Y(1) <= V_SOR(0);

end Arch_Taula_Veritat;
```

Fig. 51 Code in VHDL of binary to BCD type 74185

1.3.2.5 The Quad_Mux4

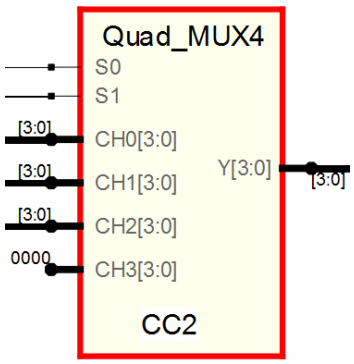

Fig. 52 Schematic of Quad Mux4

The code of the Quad_Mux4 is in Fig. 53 and we can see a multiplexor that depending the values of S1 and S0, we have I output CH0, CH1, CH2 or CH3.

```
library ieee;
use ieee.std_logic_1164.all;

entity Quad_MUX4 is
PORT(
        CH0         :IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        CH1         :IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        CH2         :IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        CH3         :IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        Y           :OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
        S0,S1       :IN STD_LOGIC
    );


end Quad_MUX4;

architecture archMUX of Quad_MUX4 is
begin

Y     <= CH0  WHEN S0='0' AND S1='0' ELSE
         CH1  WHEN S0='1' AND S1='0' ELSE
         CH2  WHEN S0='0' AND S1='1' ELSE
         CH3  WHEN S0='1' AND S1='1';


end archMUX;
```

Fig. 53 Code in VHDL of Quad Mux4

1.3.2.6 The BCD to 7 segments

Finally, to see the result we are going to convert of BCD to 7 segments.

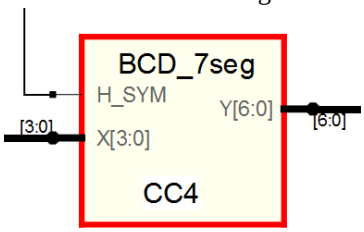

Fig. 54 Schematic of BCD to 7seg

The code of the BCD to 7 segments is in Fig. 55.

```
-- A sample design in which has to be programmed the functionality of: RBI_L, RBO_L i LT_L
-- Make a project for this file.
-- Add the functionality of the LT_L signal and synthesise again
-- Add the functionality of the RBI_L signal and synthesise again
-- Finaly, add the output RBO_L and synthesise again

LIBRARY ieee;
USE  IEEE.STD_LOGIC_1164.all;

ENTITY BCD_7seg IS

                PORT (
                        X       : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
                        H_SYM   : IN STD_LOGIC;
                        Y       : OUT STD_LOGIC_VECTOR (6 DOWNTO 0)
                          );

END BCD_7seg;

        -- EL vector d'entrada és: V_ENT(3) = D; V_ENT(2) = C; V_ENT(1) = B; V_ENT(0) = A;

        -- EL vector de sortida és V_SOR(6) = a; V_SOR(5) = b; V_SOR(4) = c; V_SOR(3) = d;
        --                         V_SOR(2) = e; V_SOR(2) = f; V_SOR(0) = g;


        -- Descripció de l'arquitectura del bloc a partir de la TdV

ARCHITECTURE Arch_Taula_Veritat OF BCD_7seg IS
BEGIN
        PROCESS (H_SYM, X)
                BEGIN
                        IF H_SYM = '0' THEN
                                CASE X IS
                                        -- DCBA             abcdefg
                                        -- ----             -------
                        --              El NOT representa sortides actives a nivell baix
                                                WHEN "0000" =>                  -- 0
                                                        Y <= ("1111110");
                                                WHEN "0001" =>                  -- 1
                                                        Y <= ("0110000");
                                                WHEN "0010" =>                  -- 2
                                                        Y <= ("1101101");
                                                WHEN "0011" =>                  -- 3
                                                        Y <= ("1111001");
                                                WHEN "0100" =>                  -- 4
                                                        Y <= ("0110011");
                                                WHEN "0101" =>                  -- 5
                                                        Y <= ("1011011");
                                                WHEN "0110" =>                  -- 6
                                                        Y <= ("1011111");
                                                WHEN "0111" =>                  -- 7
                                                        Y <= ("1110000");
                                                WHEN "1000" =>                  -- 8
                                                        Y <= ("1111111");
                                                WHEN "1001" =>                  -- 9
                                                        Y <= ("1110011");
                                                WHEN "1010" =>                  -- A
                                                        Y <= ("1110111");
                                                WHEN "1011" =>                  -- B
                                                        Y <= ("0011111");
                                                WHEN "1100" =>                  -- C
                                                        Y <= ("1001110");
                                                WHEN "1101" =>                  -- D
                                                        Y <= ("0111101");
                                                WHEN "1110" =>                  -- E
                                                        Y <= ("1001111");
                                                WHEN others =>                  -- F
                                                        Y <= ("1000111");
                                        END CASE;
                                ELSE Y <= ("0110111");
                                END IF;
        END PROCESS;

END Arch_Taula_Veritat;
```

Fig. 55 Schematic of multiplier

1.3.3 Total design

For show the result of this design, we can see in these figures the real result. In Fig. 56 we can see that we introduce a 3 and a 7 (the buttons we can see at the bottom); in Fig. 57 we pulse start and we can see the units and the tens of the result (and a led lights); and finally in Fig. 58 we pulse the other button for show the hundreds while the led is lit.

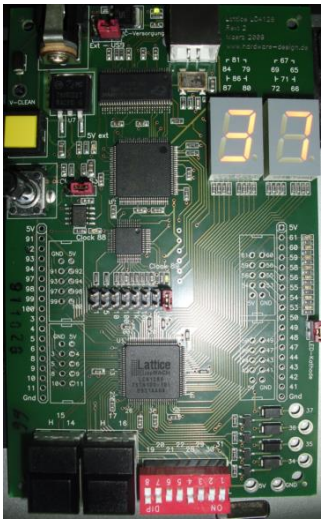

Fig. 56 Multiplier

Fig. 57 Multiplier

Fig. 58 Multiplier