



Configurable Space Microsystems Innovations & Applications Center

Lab 14 – Tutorial

Two Function Calculator with Special Operations with Xilinx ISE 10.1 and Digilent Spartan 3E Starter Kit Board

Introduction

At this time, the student should be able to deal with sequence detectors, counters, seven segment displays and frequency dividers (among others). The two function calculator with special operations is a good laboratory to implement a system composed of several of these components exploring new implementation alternatives using VHDL and the Digilent Spartan 3E Starter Kit Board resources.

Objective

In this tutorial, the designer will have to implement a two-function calculator with special operations. The calculator should accept two parallel-input numbers, **A** and **B**, (2 bits each) and multiplies them if the Operation input equals '1' or adds them if '0'. Then, the system has to count up cyclically from 0 to 99 in steps equals to the result of the operation synchronously with the system clock, displaying the result on a two-digit display. The counter stops if the serial sequence "11101" is entered by a serial input *rx*, and restarts if a second sequence "11101" is detected.

Process

1. Design a unit able to add and multiply two 2-bits numbers.
2. Design a serial receiver that identifies a specific sequence of bits.
3. Design a cyclic upward counter with a variable step size determined by a digital input.
4. Design a synchronous 7-segment display controller that shows the output of the upward counter.

Implementation

1. The unit that will add and multiply two 2-bit numbers should have a selector input named *Operation* that is type *std_logic*. When this input equals '0' the unit adds the two 2-bit inputs, otherwise the system multiplies them. The following is a possible code to implement such a unit:

EXAMPLE ADDER MULTIPLIER CODE

```
-- This entity behaves as an adder/multiplier with 2 inputs circuit.
-- Every input has a length of 2 bits
entity add_mult is
PORT (      A : in integer range 0 to 3;
        B : in integer range 0 to 3;
        Operation : in std_logic;
        F : out integer range 0 to 9);

end add_mult;

architecture Behavioral of add_mult is

begin

process (Operation,A,B)
begin
    case Operation is -- If Operations equals '0' the system adds A+B otherwise multiplies A*B
        when '0' => F <= A + B;
        when others => F <= A * B;
    end case;
end process;
end Behavioral;
```

The behavioral simulation using ISE Simulator [1] of the system above is shown in Fig.1. When the operation signal is equal to '1' the inputs 'a' and 'b' are multiplied otherwise are added.

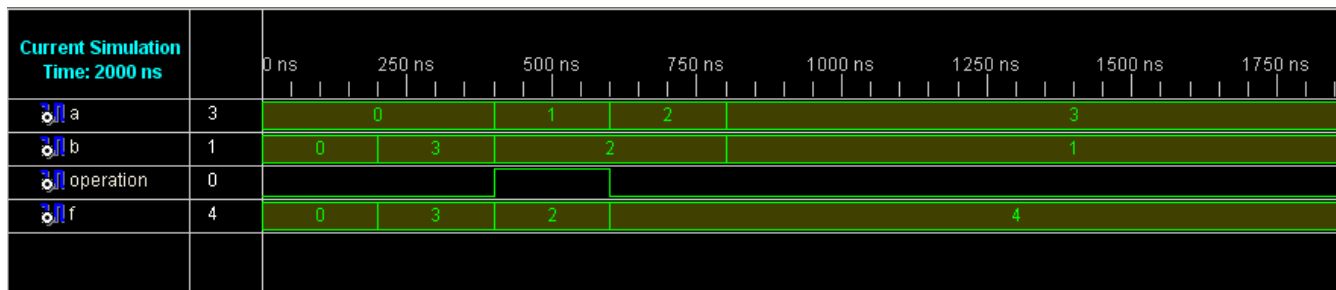


Fig. 1. Behavioral simulation of the *add_mult.vhd* entity.

2. Another important part in this design is the serial receiver that generates a positive pulse every time it identifies the sequence “11101”. A shift register can be used for this purposes and the following code shows an alternative to implement the serial receiver unit:

EXAMPLE SERIAL RECEIVER UNIT CODE

```
-- This entity behaves as a serial receiver with a 5 bit buffer
entity rcvr is
-- The generic variable determines the divisor of the receiver clock.
-- For example, if N = 99 then the clock will be divided by 100
generic ( N : integer := 1);
PORT ( rx,reset,clock : in std_logic; -- rx is the serial input
        enable : out std_logic); -- enable equals '1' when the number “11101” is loaded
-- in the register rsr.

end rcvr;
```

architecture Behavioral of rcvr is

```

signal  rsr      :      std_logic_vector(4 downto 0);
signal  newclk   :      std_logic;

begin

divider0 :      divider generic map      (N=> N)
                  port map              (clock,reset,newclk); -- Instance of the frequency divider

process (reset,newclk)
-- In this process every bit that is received by the serial input rx is going to be loaded
-- in the shift register rsr, shifting the bits to the right
begin
    if reset = '1' then
        rsr <= "00000";
    elsif newclk'event and newclk = '1' then
        rsr(0) <= rx;
        rsr(4 downto 1) <= rsr(3 downto 0);
    end if;
end process;

process (reset,newclk,rsr)
-- In this proces if the number "11101" is loaded in the register rsr
-- the receiver will generate a positive pulse in the output enable.
begin
    if reset = '1' then
        enable <= '0';
    elsif newclk'event and newclk = '1' then
        if rsr = "11101" then
            enable <= '1';
        else
            enable <= '0';
        end if;
    end if;
end process;

end Behavioral;

```

The behavioral simulation using ISE Simulator [1] of this unit is presented in Fig. 2. The signals *rx* and *enable* correspond to the *serial* input and the *enable* output respectively. Notice that when the system receives the sequence “11101” in the register *rsr* generates a positive pulse in the *enable* output.

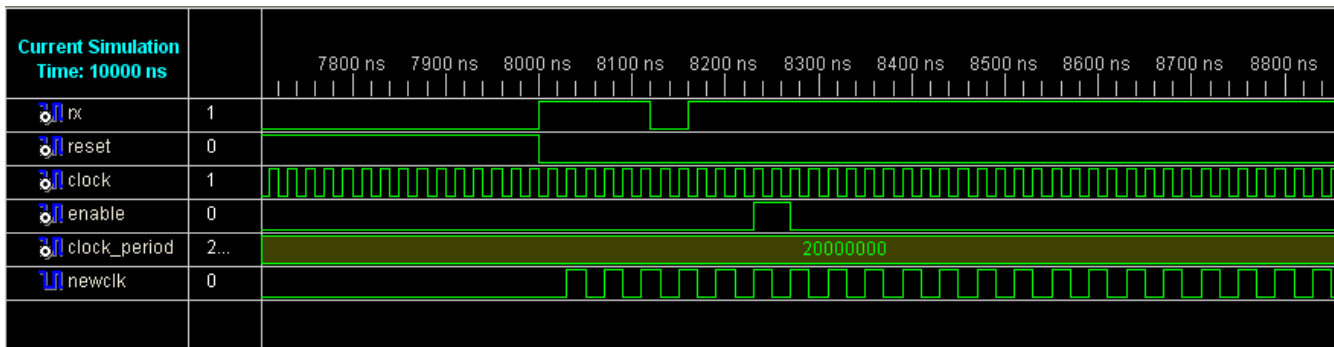


Fig.2. Behavioral simulation of the *rcvr.vhd* entity.

Note that the system implements an instance of a frequency divider that determines the clock of the serial data transmission, therefore the frequency divisor should be chosen in such a way that synchronize the communication. For example if the data are transmitted by a RS 232 (9,600 Hz) standard the 50 Mhz clock should be divided by 5208 in order to get an approximate frequency of 9,600 Hz. The following is an example of a frequency divider code, is the same that was used in the Tutorial No. 9.

EXAMPLE FREQUENCY DIVIDER CODE

```
-- This entity behaves as a frequency divider
entity divider is
generic (N : INTEGER := 100); -- The Generic variable N determines the frequency divisor
    -- For example if N = 1, then the frequency will be divided by 2 (N+1)
PORT ( clk,reset : in std_logic;
      Newclk : buffer std_logic);-- This output is the clock with the divided frequency
end divider;

architecture Behavioral of divider is

SIGNAL step : integer range 0 to N;

begin
    process(clk,reset)
    begin
        if reset = '1' then -- Asynchronous reset
            newclk <= '0';
            step <= 0;
        elsif clk'event and clk = '1' then
            if step = N then ---- Every time step reach the value of N, the system generates
                -- a high pulse in the output newclk and step go back to 0.
                    step <= 0;
                    newclk <= '1';
            else
                step <= step + 1;
                newclk <= '0';
            end if;
        end if;
    end process;
end Behavioral;
```

Note also the *GENERIC* statement [2] in the second line. This is a powerful tool when several instances of an entity are needed. The line `GENERIC(N : INTEGER := 100);` defines an integer variable `N` equal to 100, that determines the clock division by 101 (`N+1`). Notice that if you need to divide the clock by a different value, only have to change the value of `N`. Furthermore, this value could be changed in a new instance as follows [2]:

```
hz100 : divcount generic map (N => 499999)
    port map (reset,direction,clock,count100,clock100);
```

In this example, we have an instance of the entity *divcount.vhd*, but before we proceed with the *port map* [3] command we add a line with the command *generic map* in order to assign a new value to the parameter `N` (in this case 499999). It is worthy to say that you can use more than one parameter if it is necessary.

3. The third unit to be implemented is the upward counter from 0 to 99 with step size determined by an input called *step*. Furthermore, based on the system requirements the counter should stop or start if a serial sequence “11101” is detected, and as is described in the previous step, the serial receiver send a positive pulse by the output *enable* every time it detects the mentioned sequence.

Because the system is cyclic it is necessary to implement two different counters for the units and the tents. The units increase with a step size determined by the input *step*, the maximum value of which is $11_b \times 11_b = 1001_b = 9_d$. However, the tents always increase by one step when the units complete a count between 0 and 9, it means that the counter entities that are used for each counter are different. The following are the code examples related to each counter (tents and units)

EXAMPLE UNITS COUNTER CODE

```
-- This entity behaves as a cyclic upward counter, and the step size of the counter
-- is determined by the input step. Furthermore, every time the count completes a cycle
-- generates a positive pulse.
entity counteroutclk is
-- The generic variable N determines the upper bound of the count
GENERIC (N : INTEGER := 9);
PORT (reset,clock : in std_logic;
      step : in integer range 0 to 9; -- This input determines the step size of the counter
      enable : in std_logic; -- This input starts and stops the counter with a positive pulse.
      -- It depends on either the counter is started or stopped.
      count : buffer integer range 0 to N;
      out_clock : out std_logic); -- This output generates a positive pulse when the counter
      -- completes a cycle.
end counteroutclk;

architecture Behavioral of counteroutclk is

signal clock_enable : std_logic;

begin

process (enable,reset)
begin
    if reset = '1' then -- Asynchronous reset
        clock_enable <= '0';
    elsif enable'event and enable = '1' then -- The enable input starts or stop the count
        -- everytime it receives a positive pulse, thorough
        -- the signal clock_enable
        clock_enable <= not clock_enable;
    end if;
end process;

process (clock, reset)
begin
    if reset='1' then -- Asynchronous reset
        count <= 0;
        out_clock <= '0';
    elsif clock='1' and clock'event then
        if clock_enable = '1' then
            if count + step > N then -- If the count is going to overflow by the step, the
```

```

                                -- count is constrained to remain in range
                                count <= count + step - 10;
                                out_clock <= '1';
else
                                count <= count + step;    -- Otherwise the count keeps going
                                out_clock <= '0';
end if;
end if;
end process;

end Behavioral;

```

EXAMPLE TENTS COUNTER CODE

```

-- This entity behaves as a cyclic upward counter with step size 1.
entity counter is
generic (N : integer := 9);
PORT ( reset,clock : in std_logic;
       enable : in std_logic;    -- This input starts and stops the counter with a positive pulse.
       -- It depends on either the counter is started or stopped.
       count : buffer integer range 0 to N);
end counter;

architecture Behavioral of counter is

signal  clock_enable    : std_logic;

begin

process (enable,reset)
begin
    if reset = '1' then
        clock_enable <= '0';
    elsif enable'event and enable = '1' then    -- The enable input starts or stop the count
                                                -- every time it receives a positive pulse,
                                                -- thorough the signal clock_enable
        clock_enable <= not clock_enable;
    end if;
end process;

process (clock, reset)
begin
    if reset='1' then
        count <= 0;
    elsif clock='1' and clock'event then
        if clock_enable = '1' then
            if count = N then
                count <= 0;
            else
                count <= count + 1;
            end if;
        end if;
    end if;
end process;

end Behavioral;

```

Note that the code for the tents counter doesn't have neither the *step* input nor the *out_clock* output because as mentioned previously it increases by step sizes equal to one and doesn't stimulate another counter to increase with an *out_clock* output.

4. Using *component* and *port map* commands [2], and adding a frequency divider as the one described before is possible to construct the system in Fig. 3:

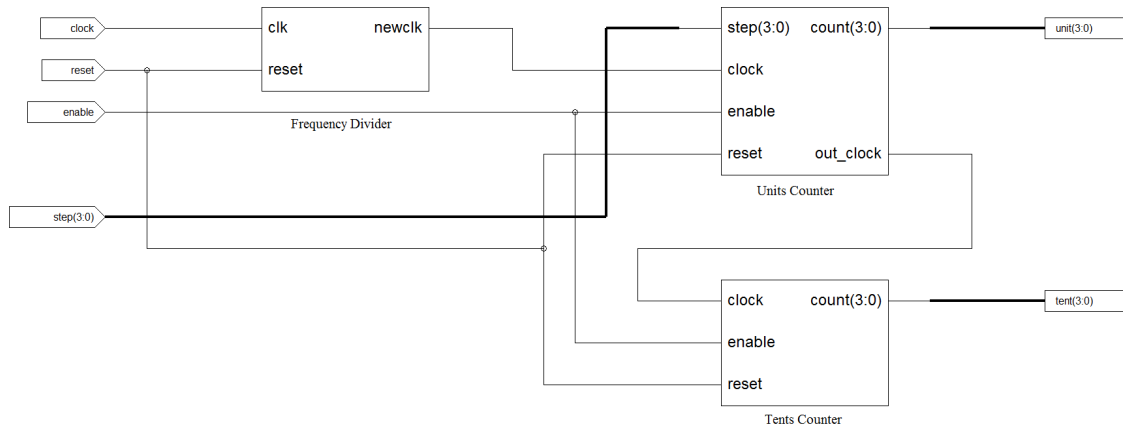


Fig. 3. RTL diagram of the entity *ClockCounter.vhd*.

The whole system forms an upward counter from 0 to 99 with a variable step size between 0 and 9 that are minimum and maximum values of the addition and product of the adder-multiplier unit designed in the step 1. Furthermore the *enable* input starts or stops the counter when receives a positive pulse, in this case from the serial receiver enable output designed in the step 2. The entity showed in Fig. 3 can be saved as *ClockCounter.vhd*.

The behavioral simulation using ISE Simulator [1] of the *ClockCounter.vhd* entity is showed in Fig. 4. The tens and units are represented by the outputs *unit* and *tent*. Note how the first pulse of the enable signal starts the counter and the second one stops it.

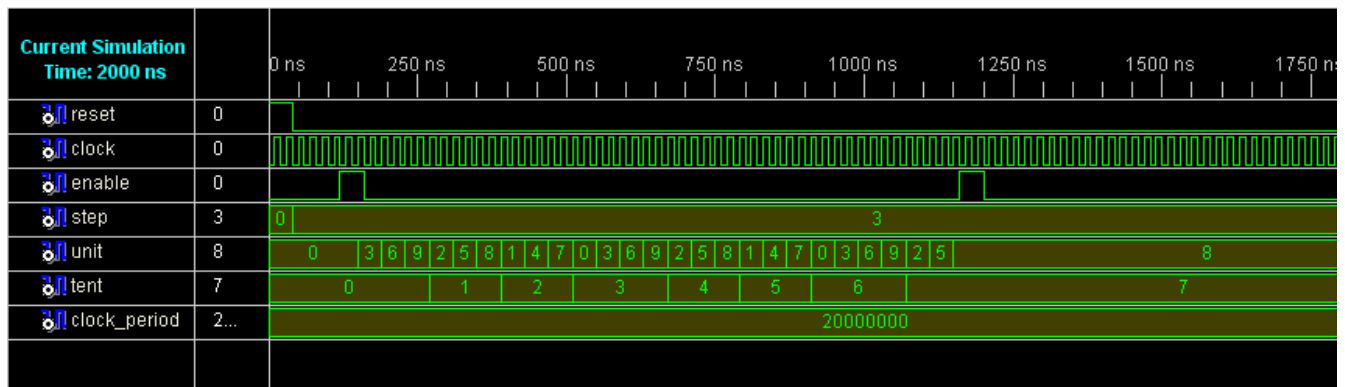


Fig. 4. Behavioral simulation of the *ClockCounter.vhd* entity.

5. The next step is showing the count using the PmodSSD peripheral module [3], which has a double 7 segment display. Because both segments are controlled simultaneously, to enable a single segment it is necessary to alternate the stimuli of the cathodes C1 and C2 at a convenient frequency such that the human eye can't perceive the blinking and the digits appear

continuously illuminated (see Fig. 5). The recommended frequency range for this purpose is between 60 Hz and 1 KHz. Therefore, an instance of the frequency divider used in the step 2 is needed in order to get the desired frequency, in this case 60Hz.

6. In order to alternate the stimuli of the cathodes C1 and C2 we need a system with a pulse train output with duty cycle of 50%. This is possible implementing the entity *cathode.vhd* as follows:

EXAMPLE CATHODE CONTROL CODE:

```
-- This entity behaves as a clock generator with duty cycle of 50%
entity cathodectrl is
generic (N : INTEGER := 3);
PORT (reset,clock : in std_logic;
      cathode : buffer std_logic);
end cathodectrl;

architecture Behavioral of cathodectrl is

begin

process (clock, reset)
begin
if reset='1' then -- Asynchronous reset
cathode <= '0';
elsif clock='1' and clock'event then -- After a positive edge the cathode is toggled
cathode <= not cathode;
end if;
end process;

end Behavioral;
```

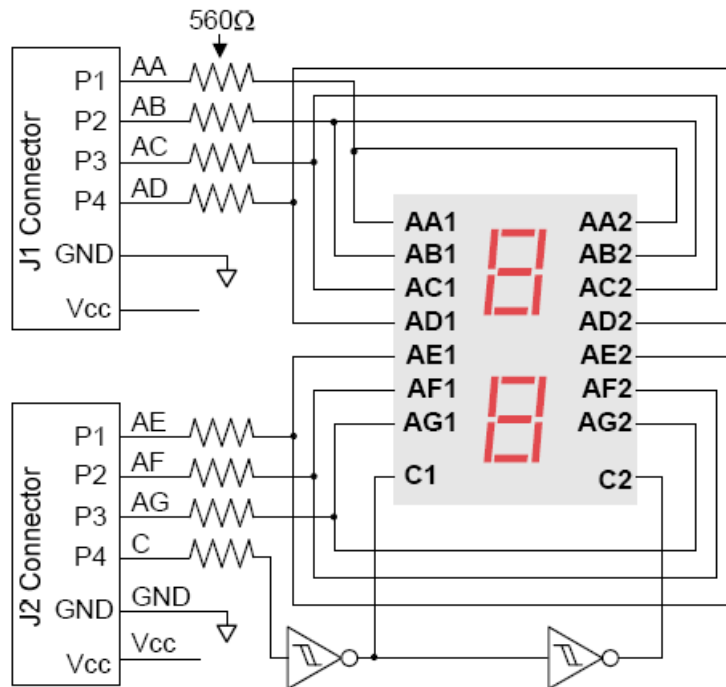


Fig. 5. Connections between the J1 and J2 headers and the PmodSSD peripheral module [3].

7. After the synchronization problem has been solved, the next step is converting the BCD outputs in 7 segment codes. The following code shows an alternative of a BCD to 7 segment converter [2]:

EXAMPLE BCD TO 7 SEGMENT CONVERTER CODE

```
-- This entity behaves as a BCD to 7 segment converter
entity BCDseven is
    PORT( HEX      : in      integer range 0 to 9;
          LED      : out     std_logic_vector(6 downto 0));
end BCDseven;

architecture Behavioral of BCDseven is

begin

    with HEX SElect    -- The possible values for the a,b,c,d,e,f,g leds in
        -- a seven segment displays are tabulated for each
        -- integer value at the input HEX
    LED<= "0111111" when 0,
          "0000110" when 1,
          "1011011" when 2,
          "1001111" when 3,
          "1100110" when 4,
          "1101101" when 5,
          "1111101" when 6,
          "0000111" when 7,
          "1111111" when 8,
          "1101111" when 9,
          "-----" when others;

end Behavioral;
```

8. The next step consists in multiplexing the outputs of the counter in order to alternate units and tents before they drive the BCD to 7 segment converter. For this, a quadruple 2 inputs multiplexer [2] is implemented as follows:

```
-- This entity behaves as a quadruple 2 inputs multiplexer
entity mux4 is
    PORT( A,B      : in      integer range 0 to 9;
          S        : in      std_logic;
          F        : out     integer range 0 to 9);
end mux4;

architecture Behavioral of mux4 is

begin

    with S Select
    F    <=    A when '0',
           B when others;

end Behavioral;
```

9. Now, joining the *cathodectrl.vhd* and the *divider.vhd* entities in a single entity called *contdisplay.vhd* and connecting them with the *mux04.vhd* (quad 2 inputs multiplexer) and the *BCD7.vhd* (BCT to 7 segment converter) it gives a new entity that can be called *DisplayCtrl.vhd* that is showed in Fig. 6 using an RTL diagram generated using Xilinx ISE 10.1 [1]. This entity should be built using *component* and *port map* commands.

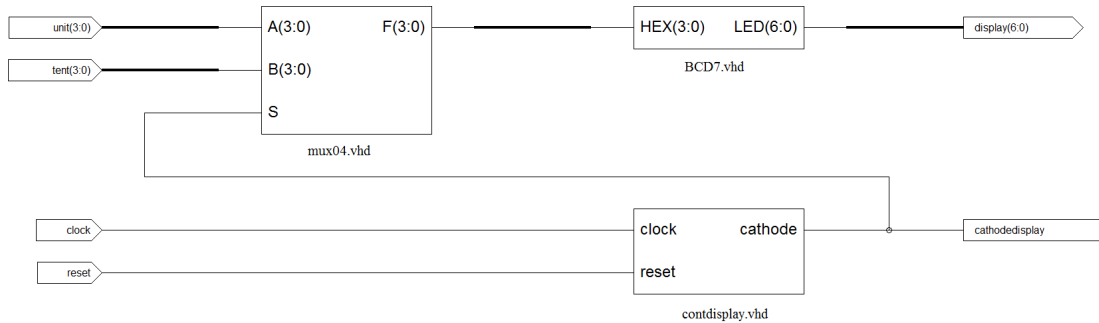


Fig. 6. RTL diagram of the *DisplayCtrl.vhd* entity.

The behavioral simulation of this system using ISE Simulator [1] is illustrated in Fig. 7. In this simulation the system shows a 24 number in the 7 segment display. Note how the *cathodedisplay* output alternates with a duty cycle of 50% switching, at the same time, the tents and units that are showed through the *display* signal.

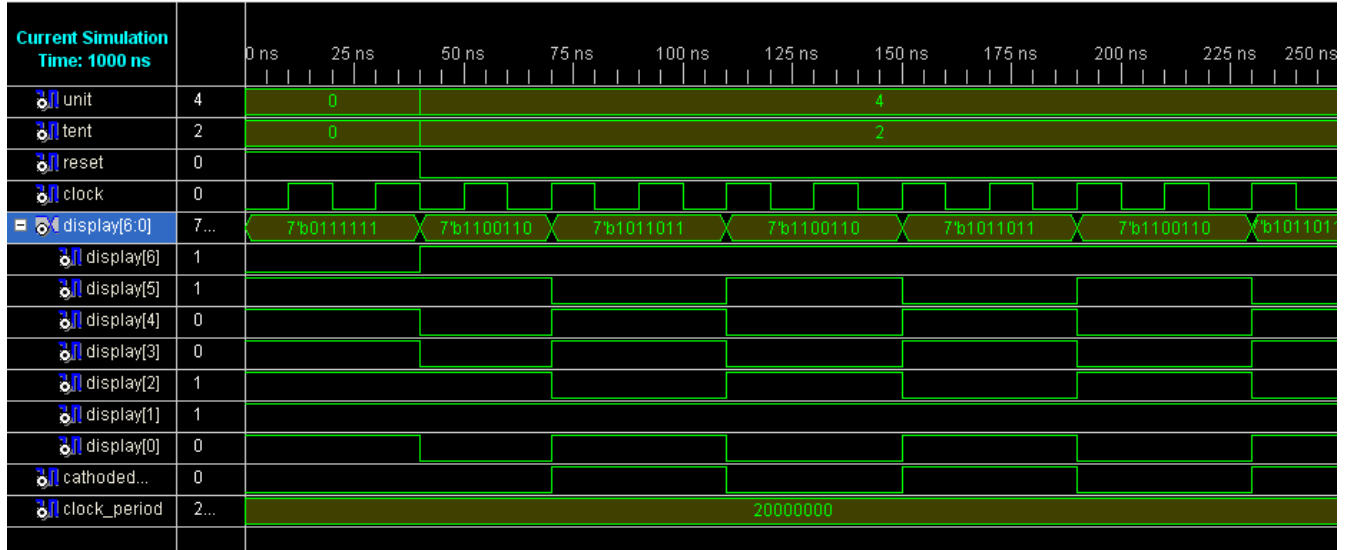


Fig.7. Behavioral simulation of the *DisplayCtrl.vhd* display.

10. At this time there are four fundamental entities in the system: *add_mult.vhd*, *rcvr.vhd*, *ClockCounter.vhd* and *DisplayControl.vhd*. Every unit carries out a specific task, now it is possible to connect the system through *component* and *port map* commands as is showed in Fig. 8.

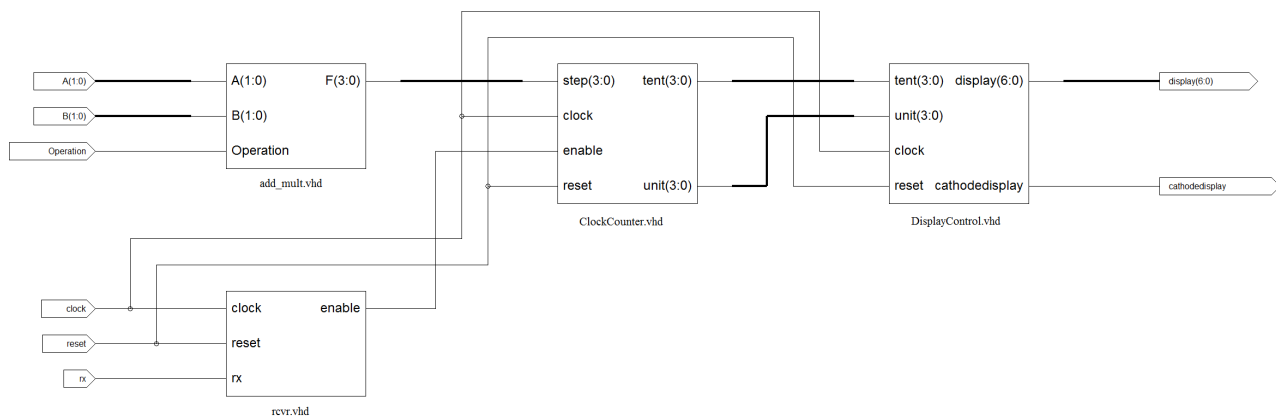


Fig. 8. General connections between the four fundamental modules *add_mult.vhd*, *rcvr.vhd*, *ClockCounter.vhd* and *DisplayControl.vhd* using Xilinx ISE 10.1 [1].

So, this is the final system that implements the two function calculator with special operations. This system can be saved as *Calculator.vhd*.

11. Notice that the system has two 2-bit inputs A and B that are going to be assigned to the pins associated to the slide switches SW3, SW2, SW1 and SW0 (pins N17, H18, L14 and L13 respectively) that are showed in Fig. 9. The clock input is associated to the 50 Mhz crystal pin C9 and the *reset* and *rx* (that is the serial input in the sequence detector) inputs are associated to the pushbuttons BTN_SOUTH (pin K17) and BTN_WEST (D18) that are showed in Fig. 9. For the inputs *Operation*, *reset* and *rx* the pushbuttons K17 and D18 and the rotary pushbutton V16 are used and each of them have a pulldown resistor associated in the text constrain file in order to overcome possible noise issues. It is worthy to emphasize that it is necessary to associate this pulldown resistors explicitly.

PIN ASSIGNMENT FOR THE CALCULATOR WITH SPECIAL OPERATIONS

```

NET "A<0>" LOC = "H18" ;
NET "A<1>" LOC = "N17" ;
NET "B<0>" LOC = "L13" ;
NET "B<1>" LOC = "L14" ;
NET "cathodedisplay" LOC = "F7" ;
NET "clock" LOC = "C9" ;
NET "display<0>" LOC = "B4" ;
NET "display<1>" LOC = "A4" ;
NET "display<2>" LOC = "D5" ;
NET "display<3>" LOC = "C5" ;
NET "display<4>" LOC = "A6" ;
NET "display<5>" LOC = "B6" ;
NET "display<6>" LOC = "E7" ;
NET "Operation" LOC = "V16" | IOSTANDARD = LVTTTL | PULLDOWN ;
NET "reset" LOC = "K17" | IOSTANDARD = LVTTTL | PULLDOWN ;
NET "rx" LOC = "D18" | IOSTANDARD = LVTTTL | PULLDOWN ;

```

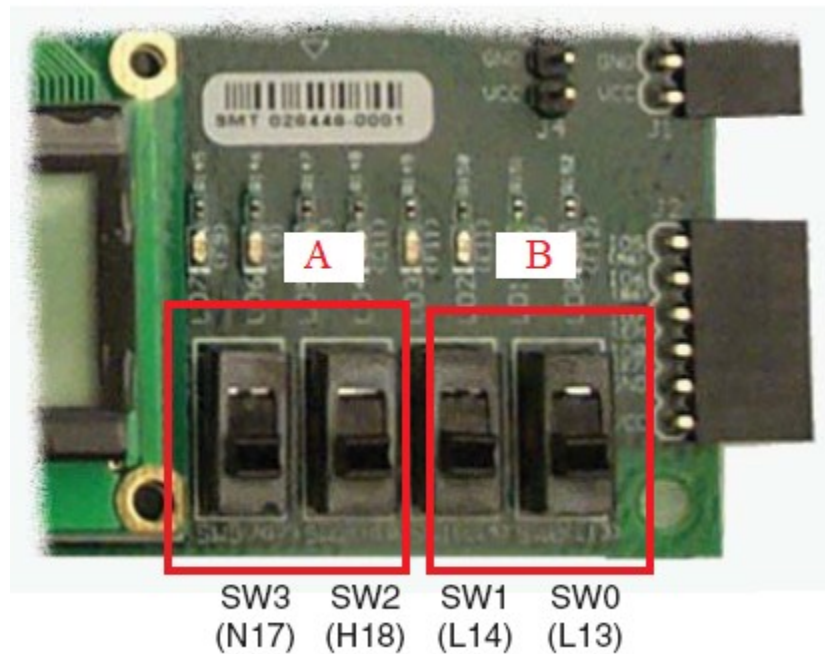


Fig. 9. The switches set used for the 2 bit inputs A and B of the *Calculator.vhd* entity. This are included in the Digilent Spartan 3E Starter Kit Board [4].

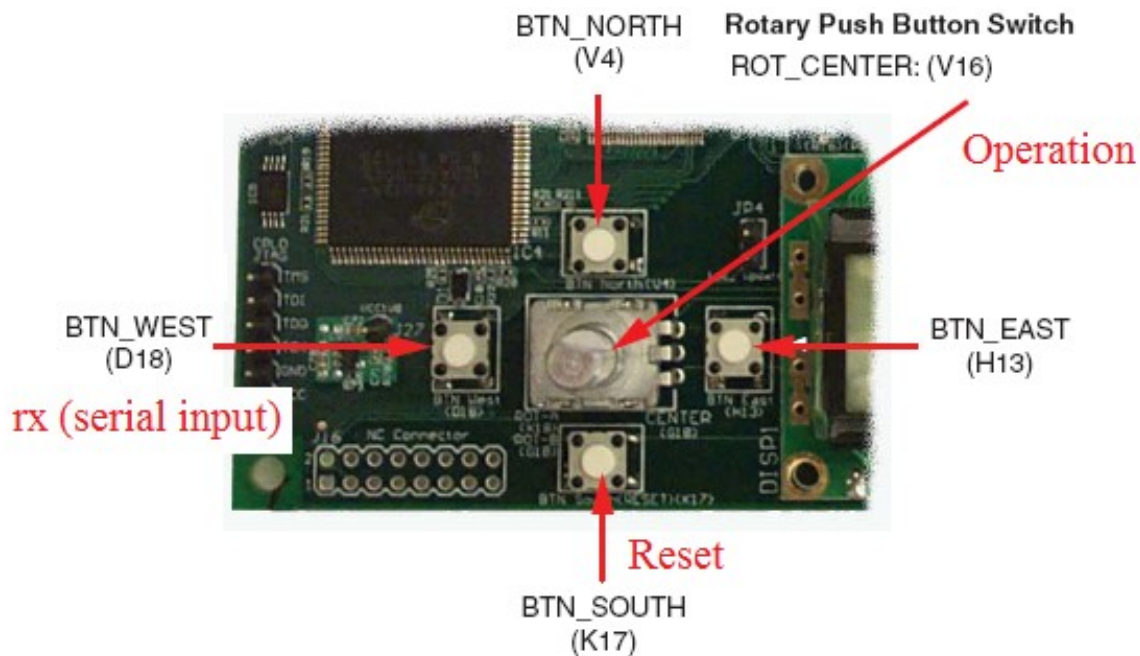


Fig. 10. The pushbuttons set used for the *Operation*, *reset* and *rx* inputs respectively [4].

12. At the same time it is very illustrative to introduce the sequence by alternative methods rather than design an elaborated serial transmitter. The proposal in this tutorial consists in using the pushbutton *BTN_WEST* (pin D18) to introduce the sequence “11101” at a frequency of 1Hz, it means that the user can introduce the sequence expending about a second by binary digit. For example to introduce the sequence “11101” the user should keep the pushbutton pressed by three seconds, after that release the button by one second and afterwards press the button again for a second in order to enable or disable the counter. Finally the input *Operation*

is associated to the rotary pushbutton ROT_CENTER (pinV16). In this case, if you keep pressed the button then the *add_mult* entity multiplies the 2 bit inputs, A and B and if you release it the inputs are added.

13. Lastly, the output *display* is assigned to the pins associated to the J1 and J2 headers as indicated in Fig. 11. The output *cathodedisplay* is assigned to the pin F7.

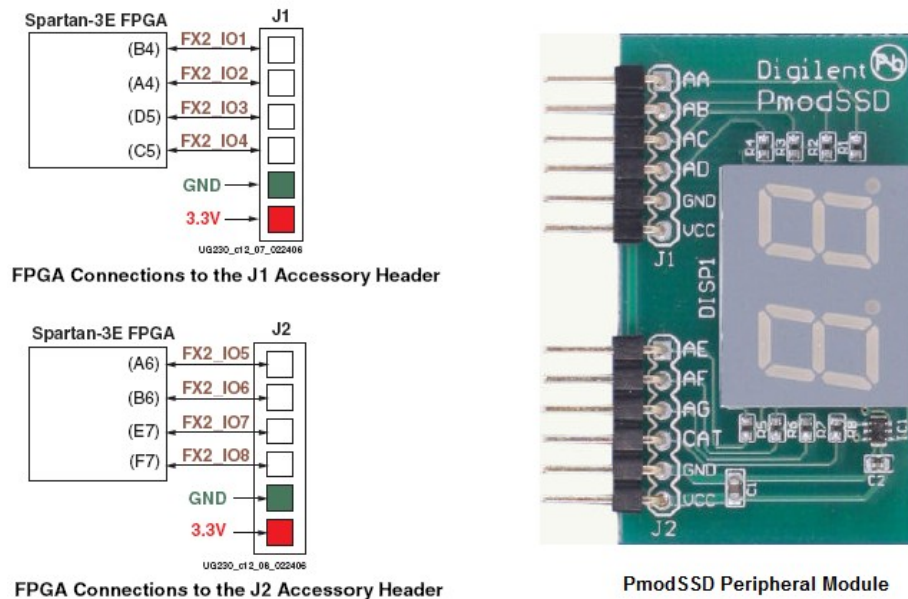


Fig. 10. Correspondence between the pins in the J1 and J2 headers and the PmodSSD pins [3],[4].

References

- [1] ISE In-Depth Tutorial. Xilinx Inc. Copyright 1995-2009. Available: <http://www.xilinx.com/itp/xilinx10/books/manuals.pdf>
- [2] S. Brown, "Fundamentals of Digital Logic with VHDL Design" Ed. Singapore: McGraw-Hill. 2000, pp 828.
- [3] *Digilent PmodSSD Peripheral Module Board Reference Manual*. Digilent Inc. Revision 06/07/05. pp. 1-2. Available: http://www.digilentinc.com/Data/Products/PMOD-SSD/Pmod%20SSD_rm.pdf
- [4] *Spartan-3E Starter Kit Board User Guide*. Xilinx Inc. Revision UG230 March 9, 2006. pp. 15-122. Available: http://www.digilentinc.com/Data/Products/S3EBOARD/S3EStarter_ug230.pdf

About the author:

Jose Marcio Luna Castaneda
M.S. in Electrical Engineering Student
The University of New Mexico
Office: ECE Building Room 223, Phone: 505-277-1372
Areas of Interest: Multi-agents, robotics, hybrid systems, computational intelligence.
e-mail: jmarcio@ece.unm.edu

Updated By:
Brian Zufelt
Undergraduate student at the University of New Mexico