****

**Faculty of Automation and Computer Science**

* **PROJECT-**

**Pocket calculator with**

**elementary operations**

**Student: Ungureanu G. Florin - Catalin**

**Advisor :** Timar Mihai

Table of contents

1. Specification .........................................................................2
2. Block diagram ......................................................................5
3. Used Components................................................................6

3.1 Adder ...............................................................................6

3.2 Subtractor .......................................................................7

3.3 Multiplier ........................................................................7

3.4 Division ............................................................................ 9

3.5 7Segment Display ..................................................11

3.6 Double Dabble ……………………………………………………15

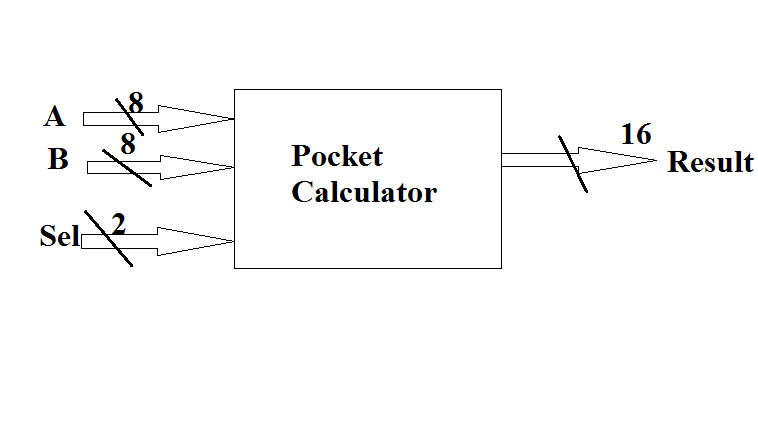
3.7 Sign Magnitude to 2’s c ....................................... 17

1. Inputs and outputs................................................................20
2. Instructions .................................................................20
3. Used solution justification ............................................22
4. Future development ..................................................23

Pocket Calculator

This mini-project was written in synthesizable VHDL . It’s simulation and functionality are presented through the Active – HDL simulator and implemented on Nexys 3 FPGA . The aim of Pocket Calculator is to implement the operation of binary addition , subtraction , multiplication and division with signed number represented on 8 bits .

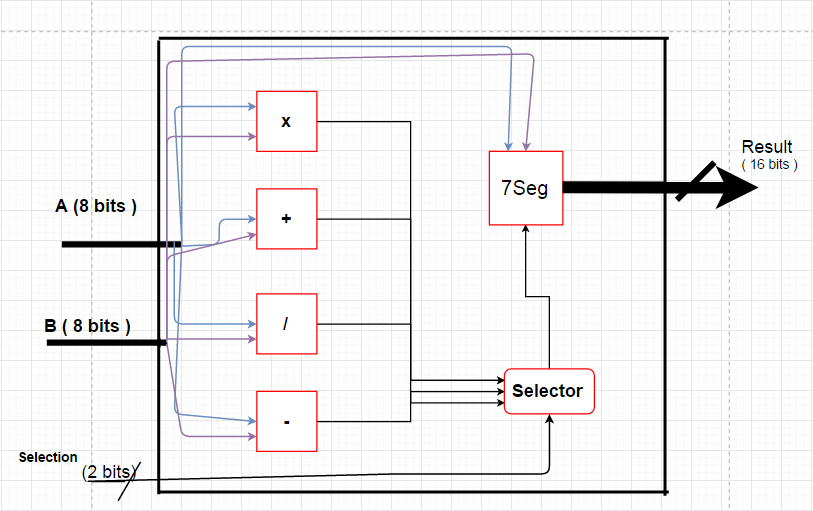
Black Box



These two operands on 8 bits are A and B , and are introduced liniarly through buttons from the FPGA .

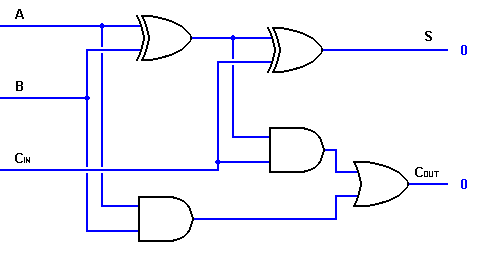
The selectios are : 00 for multiplication , 01 addition , 10 subtraction ,11 division

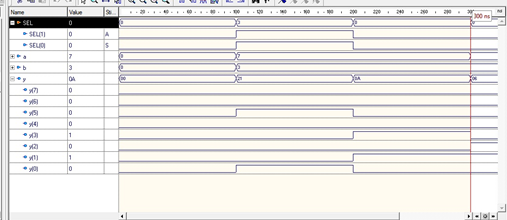
Block diagram



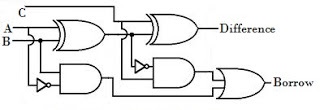
3. Used Components

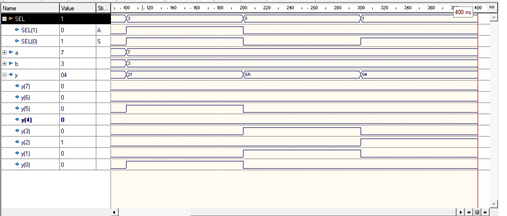
* 1. Adder





* 1. Subtractor

****



* 1. Booth's multiplication algorithm

Booth's multiplication algorithm is a multiplication algorithm that multiplies two signed binary numbers in two's complement notation. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth's algorithm is of interest in the study of computer architecture.

Booth's algorithm examines adjacent pairs of bits of the N-bit multiplier Y in signed two's complement representation, including an implicit bit below the least significant bit, y−1 = 0. For each bit yi, for i running from 0 to N − 1, the bits yi and yi−1 are considered. Where these two bits are equal, the product accumulator P is left unchanged. Where yi = 0 and yi−1 = 1, the multiplicand times 2i is added to P; and where yi = 1 and yi−1 = 0, the multiplicand times 2i is subtracted from P. The final value of P is the signed product.

The representations of the multiplicand and product are not specified; typically, these are both also in two's complement representation, like the multiplier, but any number system that supports addition and subtraction will work as well. As stated here, the order of the steps is not determined. Typically, it proceeds from LSB to MSB, starting at i = 0; the multiplication by 2i is then typically replaced by incremental shifting of the P accumulator to the right between steps; low bits can be shifted out, and subsequent additions and subtractions can then be done just on the highest N bits of P.[1] There are many variations and optimizations on these details.....

The algorithm is often described as converting strings of 1s in the multiplier to a high-order +1 and a low-order −1 at the ends of the string. When a string runs through the MSB, there is no high-order +1, and the net effect is interpretation as a negative of the appropriate value.

entity Multiplier is

port(

A : in std\_logic\_vector(7 downto 0);

B : in std\_logic\_vector(7 downto 0);

C : out std\_logic\_vector(10 downto 0)

);

end Multiplier;

architecture Behavioral of Multiplier is

begin

process(A,B)

variable ADD,SUB,PRO,AUX : std\_logic\_vector(16 downto 0);

begin

ADD := (others => '0');

SUB := (others => '0');

PRO := (others => '0');

ADD(16 downto 9) := A;

--if A(7) = '0' then

SUB(16 downto 9) := not(A)+1;

--else

-- SUB(16 downto 9) := not(A)-1;

--end if;

PRO(8 downto 1) := B;

for i in 0 to 7 loop

case PRO(1 downto 0) is

when "00" => AUX := PRO;

when "01" => AUX := PRO + ADD;

when "10" => AUX := PRO + SUB;

when "11" => AUX := PRO;

when others => null;

end case;

PRO := AUX (16) & AUX(16 downto 1);

end loop;

C <= PRO(11 downto 1);

end

process;

end Behavioral;

* 1. Divider

Division is done by repeated subtraction which is easy to implement and understand but inefficient .

entity Divider is

port(

A : in std\_logic\_vector(7 downto 0);--Numerator

B : in std\_logic\_vector(7 downto 0);--Denominator

C : out std\_logic\_vector(10 downto 0));--Easier to use in main by using 11 bits, even if the integer division can not use more than the numerator

end Divider;

architecture Behavioral of Divider is

begin

process(A,B)

Variable Numerator,Denominator,Quotient : std\_logic\_vector(7 downto 0);

begin

Quotient := "00000000";

Numerator := A;

Denominator := B;

for i in 127 downto 0 loop

if Numerator(6 downto 0) >= Denominator(6 downto 0) then -- If the subtraction makes sense then the qutioent should be incremented

Numerator(6 downto 0) := Numerator(6 downto 0) - Denominator(6 downto 0);

Quotient := Quotient + '1';

end if;

end loop;

if ((A(7) xor B(7)) = '0') then -- Again, normalizing the result depending on sign

C(9 downto 7) <= "000";

C(6 downto 0) <= Quotient(6 downto 0);

else

C(9 downto 7) <= "111";

C(6 downto 0) <= Quotient(6 downto 0);

end if;

C(10) <= A(7) xor B(7); -- The sign of the division

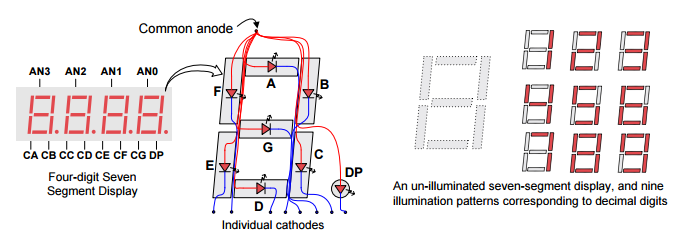
end

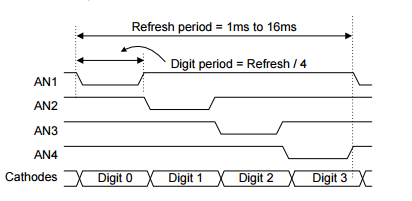
process;

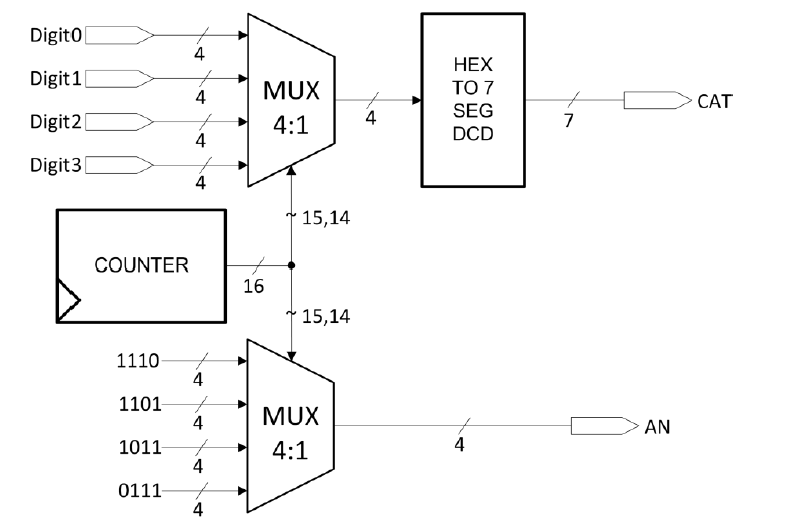
end Behavioral;

* 1. Seven Segment display

A scanning display controller circuit can be used to show a four-digit number on this display. This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession, at an update rate that is faster than the human eye can detect. Each digit is illuminated just one-quarter of the time, but because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears continuously illuminated. If the update or “refresh” rate is slowed to around 45 hertz, most people will begin to see the display flicker. In order for each of the four digits to appear bright and continuously illuminated, all four digits should be driven once every 1 to 16ms, for a refresh frequency of 1KHz to 60Hz







entity Seg7 is

port(

digit0: in std\_logic\_vector(3 downto 0);

digit1: in std\_logic\_vector(3 downto 0);

digit2: in std\_logic\_vector(3 downto 0);

digit3: in std\_logic\_vector(3 downto 0);

CLK : in std\_logic;

seg: out std\_logic\_vector(6 downto 0);

ano : out std\_logic\_vector(3 downto 0)

);

end Seg7;

architecture Behavioral of Seg7 is

signal cnt : std\_logic\_vector(15 downto 0);

signal aux : std\_logic\_vector(3 downto 0);

begin

process(CLK)

begin

if rising\_edge( CLK ) then

cnt <= cnt + 1;

end if;

end

process;

process(cnt( 15 ) , cnt( 14 ))

begin

case cnt( 15 downto 14 ) is

when "00" => ano <= "1110";

when "01" => ano <= "1101";

when "10" => ano <= "1011";

when "11" => ano <= "0111";

when others => ano <= "0000";

end case;

case cnt( 15 downto 14 ) is

when "00" => aux <= digit0;

when "01" => aux <= digit1;

when "10" => aux <= digit2;

when "11" => aux <= digit3;

when others => aux <= "1111";

end case;

end

process;

process(aux)

begin

case aux is

when "0000" => seg <="1000000"; --0

when "0001" => seg <="1111001"; --1

when "0010" => seg <="0100100"; --2

when "0011" => seg <="0110000"; --3

when "0100" => seg <="0011001"; --4

when "0101" => seg <="0010010"; --5

when "0110" => seg <="0000010"; --6

when "0111" => seg <="1111000"; --7

when "1000" => seg <="0000000"; --8

when "1001" => seg <="0010000"; --9

when "1010" => seg <="0111111"; -- minus

when others => seg <="1111111"; -- nothing

end case;

end

* 1. Binary to BCD converter ( Double Dabble Alghorithm )

The double dabble algorithm is used to convert binary numbers into binary-coded decimal (BCD) notation. It is also known as the Shift-and-add-3 algorithm, and can be implemented using a small number of gates in computer hardware, but at the expense of high latency.

The algorithm operates as follows:

Suppose the original number to be converted is stored in a register that is *n* bits wide. Reserve a scratch space wide enough to hold both the original number and its BCD representation; *(n* + 4)×*ceil*(*n*/3) bits will be enough. It takes a maximum of 4 bits in binary to store each decimal digit.

Then partition the scratch space into BCD digits (on the left) and the original register (on the right). For example, if the original number to be converted is eight bits wide, the scratch space would be partitioned as follows:

100s Tens Ones Original

0010 0100 0011 11110011

**entity bin2bcd\_9bit is**

**Port ( binIN : in STD\_LOGIC\_VECTOR (10 downto 0); -- Numbe that ahs to be converted**

**sign : out STD\_LOGIC; -- The sign of the number**

**ones : out STD\_LOGIC\_VECTOR (3 downto 0); -- The digits for ones**

**tens : out STD\_LOGIC\_VECTOR (3 downto 0); -- The digits for tens**

**hundreds : out STD\_LOGIC\_VECTOR (3 downto 0) -- The digits for hundreds**

**);**

**end bin2bcd\_9bit;**

**architecture Behavioral of bin2bcd\_9bit is**

**begin**

**process(binIN)**

**variable temp : STD\_LOGIC\_VECTOR (9 downto 0); -- Will keep the number during the algorithm**

**variable bcd : UNSIGNED (11 downto 0) := (others => '0'); -- Will contain the partial result during the algorithm**

**begin**

**bcd := (others => '0');**

**if binIN(10) = '0' then -- If the value is positive, load it directly , otherwise change it to it's negative counterpart**

**temp(9 downto 0) := binIN(9 downto 0);**

**else**

**temp(9 downto 0) := (not binIN(9 downto 0)) + "0000000001"; -- Normal 2C conversion**

**end if;**

**for i in 0 to 8 loop**

**if bcd(3 downto 0) > 4 then -- the check for ones**

**bcd(3 downto 0) := bcd(3 downto 0) + 3;**

**end if;**

**if bcd(7 downto 4) > 4 then -- the check for tens**

**bcd(7 downto 4) := bcd(7 downto 4) + 3;**

**end if;**

**if bcd(11 downto 8) > 4 then -- the check for hundreds**

**bcd(11 downto 8) := bcd(11 downto 8) + 3;**

**end if;**

**bcd := bcd(10 downto 0) & temp(8); -- Shift BCD to the left and fill with the 9th bit of temp**

**temp := temp(8 downto 0) & '0'; -- Shift temp to the left**

**end loop;**

**-- set outputs**

**ones <= STD\_LOGIC\_VECTOR(bcd(3 downto 0));**

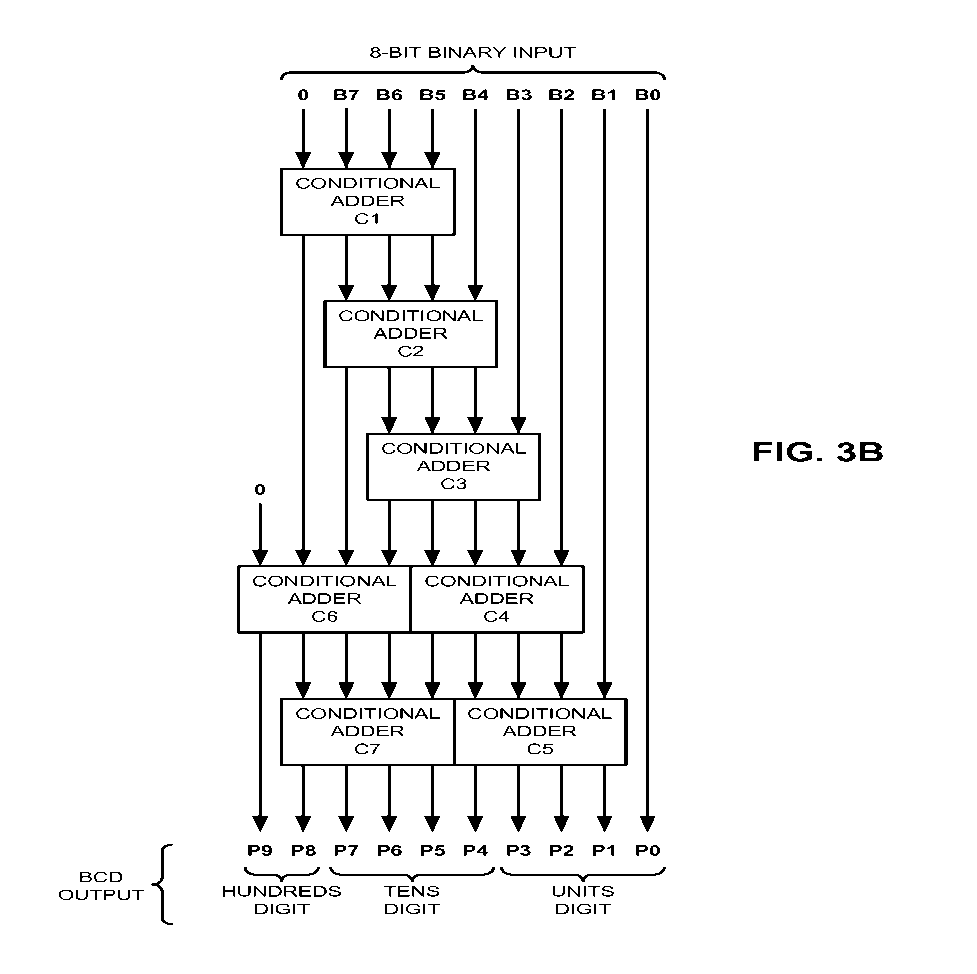
**tens <= STD\_LOGIC\_VECTOR(bcd(7 downto 4));**

**hundreds <= STD\_LOGIC\_VECTOR(bcd(11 downto 8));**

**sign <= binIN(10); -- Sign will always be on the 10th bit**

**end process bcd1;**

**end Behavioral;**



* 1. Signed magnitude to 2’s complement

Converter

entity SigMagTo2C is

port(

A : in std\_logic\_vector(7 downto 0);

B : out std\_logic\_vector(7 downto 0)

);

end SigMagTo2C;

architecture Behavioral of SigMagTo2C is

begin

process

begin

if A(7) = '0' then

B <= A;

else

B(6 downto 0) <= ((not A(6 downto 0)) + "0000001");--Normal conversion algorithm

B(7) <= '1';

end if;

end process;

end Behavioral;

1. Inputs and Outputs

Number: in std\_logic\_vector

Buttons: in std\_logic\_vector

Clock: in std\_logic;

Segments: out std\_logic\_vector

Anodes : out std\_logic\_vector

1. Instructions

#Switches

NET "Number[0]" LOC = "T10" ;

NET "Number[1]" LOC = "T9" ;

NET "Number[2]" LOC = "V9" ;

NET "Number[3]" LOC = "M8" ;

NET "Number[4]" LOC = "N8" ;

NET "Number[5]" LOC = "U8" ;

NET "Number[6]" LOC = "V8" ;

NET "Number[7]" LOC = "T5" ;

#Buttons

NET "Buttons[0]" LOC = "C4" ;

NET "Buttons[1]" LOC = "D9" ;

NET "Buttons[2]" LOC = "A8" ;

NET "Buttons[3]" LOC = "C9" ;

NET "Buttons[4]" LOC = "B8" ;

NET "Buttons[4]" CLOCK\_DEDICATED\_ROUTE = FALSE;

#Anodes

NET "Anodes[0]" LOC = "N16" ;

NET "Anodes[1]" LOC = "N15" ;

NET "Anodes[2]" LOC = "P18" ;

NET "Anodes[3]" LOC = "P17" ;

#7Seg

NET "Segments[0]" LOC = "T17" ;

NET "Segments[1]" LOC = "T18" ;

NET "Segments[2]" LOC = "U17" ;

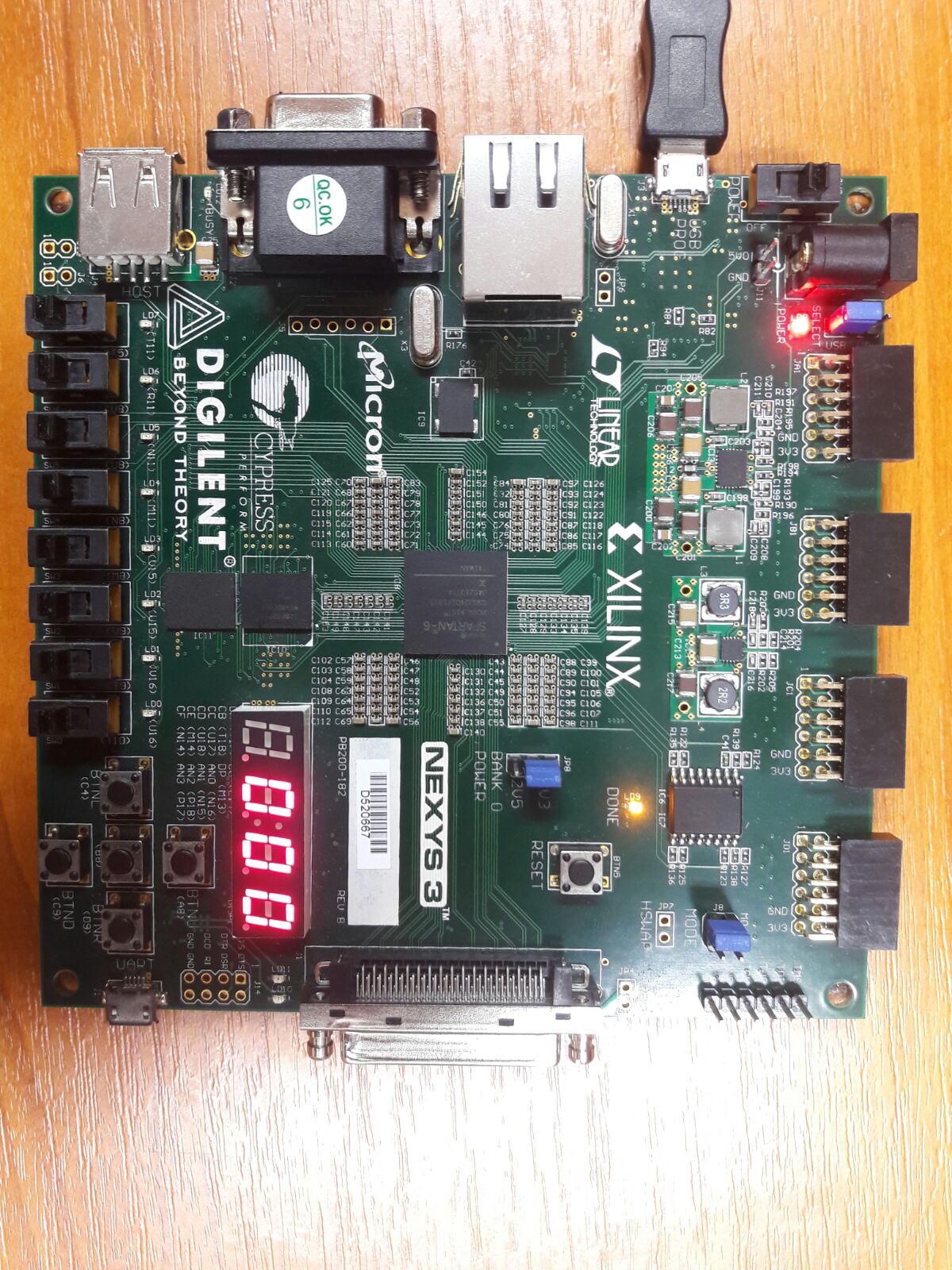
NET "Segments[3]" LOC = "U18" ;

NET "Segments[4]" LOC = "M14" ;

NET "Segments[5]" LOC = "N14" ;

NET "Segments[6]" LOC = "L14" ;

#NET "seg[7]" LOC = "M13" ;

NET Clock LOC = "V10";

7.Justification of the solution

At a first glance my solution might seem a little bit disproportionate. As I combine both complicated and simple , efficient and less efficient algorithms to get it work .

The multiple conversions are due to the fact that different signed numbering system has their advantage over different operation .

Is easier to do addition/subtraction in 2’s complement and for multiplication/ division , signed magnitude is more appropriate .

I used Booth’s multiplication algorithm because it is relatively efficient and easy to implement . The division algorithm is not efficient and consume a lot of memory but is the most easiest to understand and implement .

8.Future development

There is still room of improvement for this project , especially for the divider .

1. All leds lit up if one try to divide by 0 as a ERROR signal
2. The alghorithm for division should be replaced with one more efficient and less trouble maker as this one .
3. One can extend the dimension of the operands .
4. Other operations / options should also be taken in consideration .
5. More generally speaking , one can create balance in terms of complexity between the components .