**People's Democratic Republic of Algeria**
**Ministry of Higher Education and Scientific Research**

**Institute of Electrical and Electronic Engineering**

**EE222 Digital Systems**

---

# LABORATORY REPORT #1

## Design, Simulation and Implementation of BCD Adders and ALU Using behavioral Modeling

---

**Done by:**

AMMOUR Fatma
BOUTITE Ramzi
BOULEGROUN Amin

**Group:**

L2 Group 08

**Instructor:**

Y.AZZOUGUI

**Date:**
19/02/2023

# Introduction

Arithmetic and logical operations are a fundamental building block of processors and controllers. Therefore, it is important to understand how to implement them using VHDL, and how to simulate and test their functionality.

# Objectives

The objective of this lab work is to design a one-digit and two-digit BCD adder and an arithmetic and logical unit (ALU) using sequential VHDL programming.

# Part I: One-Digit BCD Adder

## 1.1 Problem Statement:

A 1-digit BCD adders outputs the sum of the two BCD digits input, and also considers a carry in and out bit. The problem requires implementing the adder using the IF-ELSE statement. The design requires 2 BCD digits (4bits $\times$ 2) and a carry-in bit as inputs, and the output is 1 BCD digit and a carry-out bit.

## 1.2 Design approach:

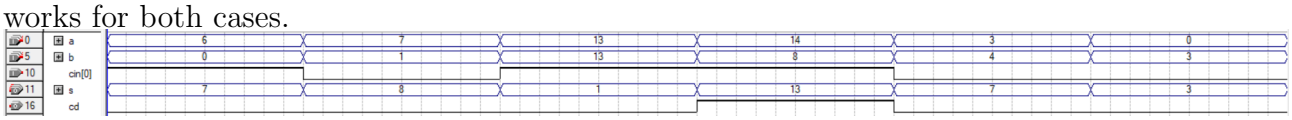
Constructing a BCD adder is based on 2 conditions. If the sum is less or equal to 9, then we perform the addition in a normal fashion. Else, the sum is greater than 9, so we adjust by adding 6 to the sum to complement the base 16 digit into 2 digits of base 10. The second digit can only be 1, so we represent it using the carry-out bit.
The codes will utilize the numeric_std in order to be able to perform different operations.
Notice the concatenation of 0 to the beginning of the first input, this is to allocate the extra bit needed for the carry-out.

## 1.3 VHDL code:

```
1  library ieee;
2  use  ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity part1 is
6          port(
7                  a,b : in unsigned(3 downto 0);
8                  cin : in unsigned(0 downto 0);
9                  s: out unsigned (3 downto 0);
10                 cd : out std_logic
11         );
12 end part1;
13
14 architecture arch of part1 is
15         signal o : unsigned(4 downto 0);
16         signal result : unsigned (4 downto 0);
17         begin
18         process (a,b,cin)
19                 begin
20                 o <= ('0'& a)+b+cin;
21                 if (o>"1001") then result <= o + "110";
22                 elsif (o<="1001")then result <= o;
23                 end if;
24         end process;
25         cd <= result(4);
26         s <= result (3 downto 0);
27 end arch;
```

## 1.4 Simulation result:

For the simulation, we avoided including all cases, since we only need to confirm that it works for both cases.




From the simulation, we notice that it works just as planned, except for when the entered values are out range of a BCD digit. The problem can be solved by adding a condition to confirm that the input digit is less than 10. However for normal cases it should not be a problem.

# Part II: Two-Digit BCD Adder

## 2.1 Problem statement:

A 2-digit BCD adder outputs the sum of 2 × 2 BCD digits input and a carry-in bit. The output is a 2 digit BCD, with the carry-out bit. A 2 digit BCD contains 4 × 2 bits per pair.

## 2.2 Design approach:

Using the same principle from 1-digit BCD adder, we can combine two of them to make our adder. The first adder will calculate the first digit output from our 2 input digits, then the second digit using the 2nd 2 inputs. While passing the carry-out of first adder into the carry-in of the second adder. And taking the extra resulting third digit as the final carry-out.

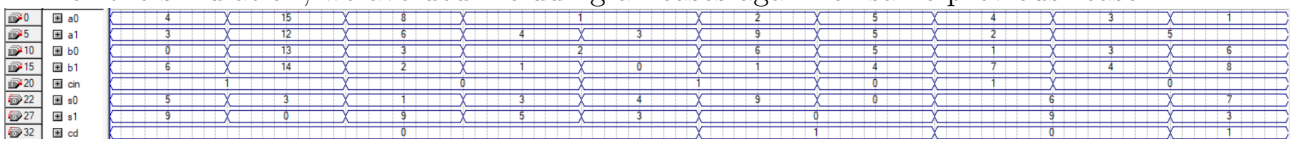
## 2.3 VHDL code:

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

entity part2 is
port  (
         a1,a0,b1,b0 : in unsigned(3 downto 0);
         cin : in unsigned(0 downto 0);
         s1,s0 : out unsigned(3 downto 0);
         cd : out unsigned(0 downto 0));
end part2;

architecture arch of part2 is
signal seven_links : unsigned(0 downto 0);
signal result1,result0 : unsigned (4 downto 0);
signal o0,o1 : unsigned(4 downto 0);

begin
p0 : process (a0,b0,cin)
begin
        o0 <= ('0'& a0)+b0+cin;
        if (o0>"1001") then result0 <= o0 + "110";
        elsif (o0<="1001")then result0 <= o0;
        end if;
end process;
s0 <= result0 (3 downto 0);
seven_links <= result0(4 downto 4);
p1 : process (a1,b1,seven_links)
begin
        o1 <= ('0'& a1)+b1+seven_links;
        if (o1>"1001") then result1 <= o1 + "110";
        elsif (o1<="1001")then result1 <= o1;
        end if;
end process;
cd <= result1(4 downto 4);
```

```
36 s1 <= result1 (3 downto 0);
37 end arch;
```

## 2.4 Simulation result:

For the simulation, we avoided including all cases again for same previous reason.




Since we continued using the same principle from the earlier, the "problem" is still relevant here and can be solved with the same solution.

# Part III: The Arithmetic and logic Unit (ALU):

## 3.1 Problem statement:

An ALU that can compute various arithmetic and logical operations. It accepts two 4 bits inputs, 4 bits opCode input, and outputs a 4bit result with 3 different flags (negative, zero and carry flag). The output is also passed through a hex to 7-seg decoder.

## 3.2 Design approach:

Using CASE-WHEN, we can check for the opCode and return the appropriate operation result to Y. For the hex to 7-seg decoder, we assign the LEDs activation order for each value possible.

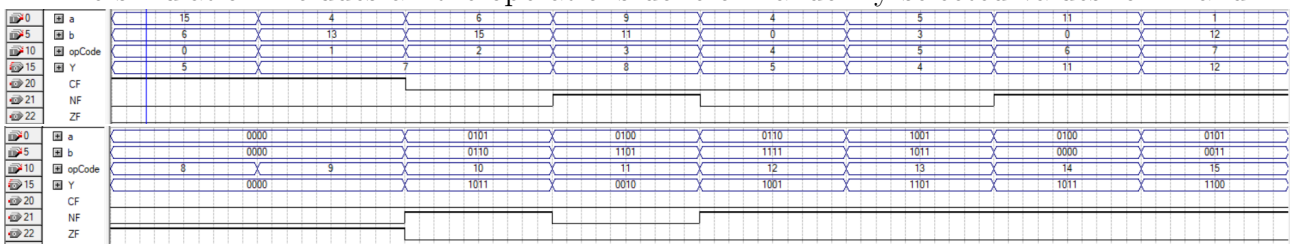
## 3.3 VHDL code:

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity ex3 is
6  port  (
7          a,b : in unsigned(3 downto 0);
8          Y : out unsigned (3 downto 0);
9          NF,CF,ZF : out std_logic;
10         opCode : in std_logic_vector( 3 downto 0));
11 end ex3;
12
13 architecture arch of ex3 is
14 signal result: unsigned(4 downto 0);
15 begin
16 process (a,b,opCode)
17 begin
18         case opCode is --operations case
19                 when "0000" => result <= ("0"&a) + b;
20                 when "0001" => result <= ("0"&a) - b;
21                 when "0010" => result <= ("0"&a) + 1;
22                 when "0011" => result <= ("0"&a) - 1;
23                 when "0100" => result <= ("0"&a) + 1;
24                 when "0101" => result <= ("0"&a) - 1;
25                 when "0110" => result <= ("0"&a);
26                 when "0111" => result <= ("0"&b);
27                 when "1000" => result <= "0"&(a AND b);
28                 when "1001" => result <= "0"&(a OR b);
29                 when "1010" => result <= "0"&(a NAND b);
30                 when "1011" => result <= "0"&(a NOR b);
31                 when "1100" => result <= "0"&(a XOR b);
32                 when "1101" => result <= "0"&(a XNOR b);
33                 when "1110" => result <= "0"&(NOT a);
34                 when "1111" => result <= "0"&(NOT b);
```

```
        end case;

        case result(3 downto 0) is --hex to 7-seg decoder case
                when x"0" => seg7 <= "1000000";
                when x"1" => seg7 <= "1111001";
                when x"2" => seg7 <= "0100100";
                when x"3" => seg7 <= "0110000";
                when x"4" => seg7 <= "0011001";
                when x"5" => seg7 <= "0010010";
                when x"6" => seg7 <= "0000010";
                when x"7" => seg7 <= "1111000";
                when x"8" => seg7 <= "0000000";
                when x"8" => seg7 <= "0010000";
                when x"9" => seg7 <= "0000011";
                when x"A" => seg7 <= "0001000";
                when x"B" => seg7 <= "0000011";
                when x"C" => seg7 <= "1001110";
                when x"D" => seg7 <= "0100001";
                when x"E" => seg7 <= "0000110";
                when x"F" => seg7 <= "0001110";

        end case;
end process;
Y<= result(3 downto 0);
ZF<= not (result (3) or result(2) or result(1) or result(0));
NF<= result(3);
CF<= result(4);
end arch;
```

## 3.4 Simulation result:

The simulation includes all the operations done on randomly selected values for A and B.




We remark that even though the NF was design for subtraction except that it is activated in the other cases, this is because most of the other cases are logical operations that can invert zeros to ones, thus inverting the "sign bit" from positive 0 to negative 1.

# Conclusion

In conclusion, this lab work focused on designing and implementing various adders and an ALU using VHDL. The one-digit and two-digit BCD adders were designed and simulated successfully. The ALU was designed to perform various arithmetic and logical operations based on the opcode input with the output hex to 7-seg decoder and was also successful.