# Laboratory exercise 1
# Arithmetic and logic unit

**1 Goals:**
to achieve knowledges and practical experience in ALU design for modern computers, to get programming and debugging exercise in VHDL language.

**2 Theoretical information**

ALU is intended for calculating both arithmetical functions (addition, subtraction) and logic functions (bit-wise AND, OR, NOT, XOR) of data represented by *n*-bit wide fixed point codes, say *A* and *B*. In most of cases the data are represented by twos complement binary codes. The carry bit $C_0$ serves as a special operand. Except *n*-bit result *Y*, the ALU results are the result flags like the carry bit from the most significant bit (MSB) $C_n$, overflow flag *V*, zero flag *Z* and sign flag *S*.

The control code *F* gives the ALU operation type, which coding usually depends on the insrtuction opcode set.

3. **ALU design example**

Consider the example of 4-bit ALU named as LSM, which implements $Y=A+B+C_0$ by $F$=00, $A$-$B$-$C_0$ by $F$=01, bit-wise AND by $F$=10, and bit-wise OR by $F$=11, where $C_0$ is the carry bit to the least significant bit (LSB). The results are word *Y*, zero flag *Z* and carry flag $C_3$.

In VHDL the LSM object declaration looks like the following.

```
use  Cnetwork.all;
entity LSM is
  port(F : in BIT_VECTOR(1 downto 0);-- function
       A : in BIT_VECTOR(3 downto 0);-- first operand
       B : in BIT_VECTOR(3 downto 0);-- second operand
       C0: in BIT;                   -- carry input
       Y : out BIT_VECTOR(3 downto 0);-- result
       C3: out BIT;                          -- carry output
       Z : out BIT    -- zero flag
       );
end LSM;
```

**Behavioral model of LSM.**

The entity in its behavioral representation is represented by the algorithm of its behave not to consider the concrete element basis of integral technology. But on the contrary to the algorithm represented by the usual programmer language, here the strict operation sequence is given which is executed in time. By the execution process the operations are executed in sequential-parallel order: sequential operators are executed sequentially and parallel ones do in parallel.

The behavioral model of LSM can be described as the following architecture.

```
architecture BEH of LSM is
  signal ai,bi,ci,yi:integer;
  signal ybi:BIT_VECTOR(4 downto 0);
begin
  -- input data are represented by integers
  ai<= BIT_TO_INT(A);
  bi<= BIT_TO_INT(B);
  ci<= BIT_TO_INT(C0);
-- Adder-subtractor
ADDER:yi<= ai+bi+ci when F(0)='0' else
             ai-bi-ci;
-- Result multiplexor
MUX:with F select
  ybi<= INT_TO_BIT(yi,5) when "00"|"01",--Adder-subtractor
          '0'&(A and B) when  "10",      -- AND
          '0'&(A or  B) when others;    --OR
  C7<= ybi(4);                   --        carry otput
  Z<='1' when ybi(3 downto 0)="0000" else '0';--Zero flag
  Y<= ybi(3 downto 0);         --          Result
end BEH;
```

In the architecture declaration the signals are declared, which are used as intermediate signals in the calculations. After the declarative part of the architecture, the descriptional part of it stays between the key words `begin` and `end`. Firstly, the type conversion of the signals is used for conversion of bit vectors `A, B, C0` to integer signals `ai,bi,ci` using type conversion functions from the package `Cnetwork`. In the operator which is marked by the mark `ADDER`, the adder-subtractor is described, Depending on the condition `F(0)`, it performs addition or subtraction of integers. It worth to mention that in VHDL all parallel operators can be marked by the name which helps to understand the program and simplifies its deвug.

The operator marked by MUX describes a multiplexor which due to the code F selects either adder output, or logic function AND or OR. The 4-bit addition-subtraction result is transferred to 5-bit vector taking into account the overflow bit. The logic operation result is expanded by a 0 bit to get the 5-dit vector as the arithmetic result gets. This expansion is implemented using the bit concatenation operation: '0'&(A or B). Such expansion is needed because the signal assignment operation affords that the left signal type has to be of the same type as the right signal is. Here the ybi bit width which is left to '<=' has to be equal to the formula result bit width which is right to.

The result Y is formed as 4 low bits of the signal ybi, and the result C is formed as the MSB of it. The zero flag Z is formed as a result of comparing to zero of 4 low bits of the signal ybi.

It worth to mention that this example is described by the style for the synthesis. But because the custom type conversion functions are used except the standard functions like ones from the package IEEE.Numeric_bit, then the synthesis result can have too large hardware volume. Therefore, below is described the LSM example which is described using the standard package IEEE.Numeric_bit. Such an example is compiled (synthesized) with small hardware volume (only всего 15 LUT).

```
architecture NUM of LSM is
  signal ai,bi,yi,bp1:signed(4 downto 0);
  signal ybi:BIT_VECTOR(4 downto 0);
begin
  ai<= RESIZE(signed(A),5);-- 1 sign bit is added
  bi<= RESIZE(signed(B),5);
  bp1<=bi     when C0='0' else -carry bit is added
       bi+1;
  yi<= ai+bp1  when F(0)='0' else
       ai-bp1;
  MUX:with F select
  ybi<= bit_vector(yi) when "00"|"01",
        '0'&(A and B) when  "10",
        '0'&(A or  B) when others;
  C3<= ybi(4);
  Z<='1' when ybi(3 downto 0)="0000" else '0';
  Y<= ybi(3 downto 0);
end NUM;
```

**Structured LSM model based on LUTs**.

In this example the LSM is designed which is based on the LUTs, asn id described by the structure style. Such project is directly mapped into FPGA. I.e. each described LUT component is implemented in the respective LUT of the FPGA.

LSM has *n* stages according to its digits. Each stage can be represented as a composition of 3 LUT. First of them LNI implements the one bit addition or logic operation with the input data *Ai*,*Bi.* The second – LNO – adds carry bit *Ci* from the previous stage to the LNI result *Xi.* The third - LNC – adds carry *Ci* to sum or difference of *Ai* and *Bi* , and calculates the carry $C_{i+1}$ to the next stage. The structure of a single stage is shown in Fig. 1.

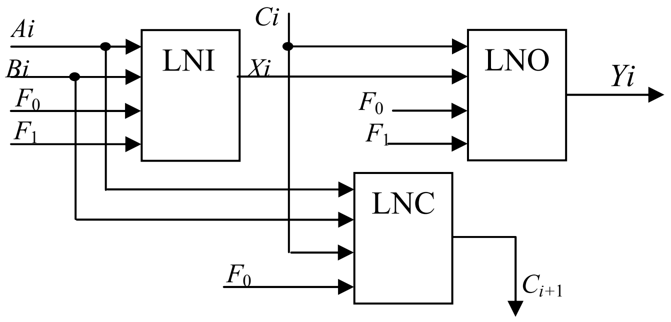

Fig.1. LSM stage structure

These LUTs are coded according the following truth tables 1-3.

Table 1. Truth table for LNI

| Address | Function | F1,F0 | Bi | Ai | Xi |
|---|---|---|---|---|---|
| 0 | A+B | 00 | 0 | 0 | 0 |
| 1 | | | 0 | 1 | 1 |
| 2 | | | 1 | 0 | 1 |
| 3 | | | 1 | 1 | 0 |
| 4 | A-B | 01 | 0 | 0 | 1 |
| 5 | | | 0 | 1 | 0 |
| 6 | | | 1 | 0 | 0 |
| 7 | | | 1 | 1 | 1 |
| 8 | A&B | 10 | 0 | 0 | 0 |
| 9 | | | 0 | 1 | 0 |
| 10 | | | 1 | 0 | 0 |
| 11 | | | 1 | 1 | 1 |
| 12 | $A \vee B$ | 11 | 0 | 0 | 0 |
| 13 | | | 0 | 1 | 1 |
| 14 | | | 1 | 0 | 1 |
| 15 | | | 1 | 1 | 1 |

Table 2. Truth table for LNO

| Address | Function | F1,F0 | Xi | Ci | Yi |
|---|---|---|---|---|---|
| 0 | A+B | 00 | 0 | 0 | 0 |
| 1 | | | 0 | 1 | 1 |
| 2 | | | 1 | 0 | 1 |
| 3 | | | 1 | 1 | 0 |
| 4 | A-B | 01 | 0 | 0 | 1 |
| 5 | | | 0 | 1 | 0 |
| 6 | | | 1 | 0 | 0 |
| 7 | | | 1 | 1 | 1 |
| 8 | A&B | 10 | 0 | 0 | 0 |
| 9 | | | 0 | 1 | 0 |
| 10 | | | 1 | 0 | 1 |
| 11 | | | 1 | 1 | 1 |
| 12 | $A \vee B$ | 11 | 0 | 0 | 0 |
| 13 | | | 0 | 1 | 0 |
| 14 | | | 1 | 0 | 1 |
| 15 | | | 1 | 1 | 1 |

Table 3. Truth table for LNC

| Address | Function | F0 | Ci | Bi | Ai | $C_{i+1}$ |
|---|---|---|---|---|---|---|
| 0 | A+B | 0 | 0 | 0 | 0 | 0 |
| 1 | | | 0 | 0 | 1 | 0 |
| 2 | | | 0 | 1 | 0 | 0 |
| 3 | | | 0 | 1 | 1 | 1 |
| 4 | | | 1 | 0 | 0 | 0 |
| 5 | | | 1 | 0 | 1 | 1 |
| 6 | | | 1 | 1 | 0 | 1 |
| 7 | | | 1 | 1 | 1 | 1 |
| 8 | A-B | 1 | 0 | 0 | 0 | 0 |
| 9 | | | 0 | 0 | 1 | 1 |
| 10 | | | 0 | 1 | 0 | 0 |
| 11 | | | 0 | 1 | 1 | 0 |
| 12 | | | 1 | 0 | 0 | 1 |
| 13 | | | 1 | 0 | 1 | 1 |
| 14 | | | 1 | 1 | 0 | 0 |
| 15 | | | 1 | 1 | 1 | 1 |

The right column of the tables is the information for coding the respective LUTs. These codes will be equal to X"E896" , X"CC96", and X"B2E8", respectively.

Then a single LSM stage is described as the following.

```
LNI:LUT4 generic map(mask=>X"6917")
         port map(a=>A(i),b=>B(i),c=> F(0),d =>F(1),
          Y =>X(i));
LNO:LUT4 generic map(mask=>X"6933")
         port map(a=>c(i),b=>X(i),c=> F(0),d =>F(1),
          Y =>yi(i));
LNC:LUT4 generic map(mask=>X"174D")
         port map(a=>A(i),b=>B(i),c=> c(i),d =>F(0),
          Y =>c(i+1));
```

Signal `yi` is equal to the output signal - port `Y` in mode `out` , because it is used also as an input signal for the function of the zero flag deriving. Therefore, the LUT `LNO` output could not be attached to the port `Y` directly.

To describe the whole LSM, these operators can be doubled *n*=4 times by the copy-paste operations, corecting the indexes properly. But the use of the operator **generate** is more effective one.

The zero flag signal `Z` has a 1 meaning only on a single combination of 16 possible combinations of bits `yi` and therefore, it can be generated by a single LUT with the mask code X"8000".

The resulting architecture looks like the following.

```
architecture STR_LUT of LSM is
   signal c,x,yi: BIT_VECTOR(4 downto 0);
   component LUT4 is
     generic(mask:bit_vector(15 downto 0):=X"ffff";
             td:time:=1 ns);
      port(a : in BIT;
           b : in BIT;
           c : in BIT;
           d : in BIT;
           Y : out BIT);
   end  component;
begin
         c(0)<=C0;
--  ALU network
LSM_STR:for i in 0 to 3 generate
  LNI:LUT4 generic map(mask=>X"6917")
           port map(a=>A(i),b=>B(i),c=> F(0),d =>F(1),
             Y =>X(i));
  LNO:LUT4 generic map(mask=>X"6933")
           port map(a=>C(i),b=>X(i),c=> F(0),d =>F(1),
             Y =>yi(i));
  LNC:LUT4 generic map(mask=>X"174D")
           port map(a=>A(i),b=>B(i),c=> c(i),d =>F(0),
             Y =>c(i+1));
  end generate;
-- Zero result deriving
  UZ:LUT4 generic map(mask=>X"8000")
           port map(a=>yi(3),b=>yi(2),c=> yi(1),d =>yi(0),
             Y =>Z);
  Y<=yi(3 downto 0);
  C3<=c(4);   --выход переноса
end STR_LUT;
```

In the architecture declaration space the used components are declared, like the component `LUT4`.

The resulting structure which was built by the compiler-synthesizer XST of the system Xilinx Webpack is shown on Fig2. Its hardware volume occupies 13 LUT, i.e.

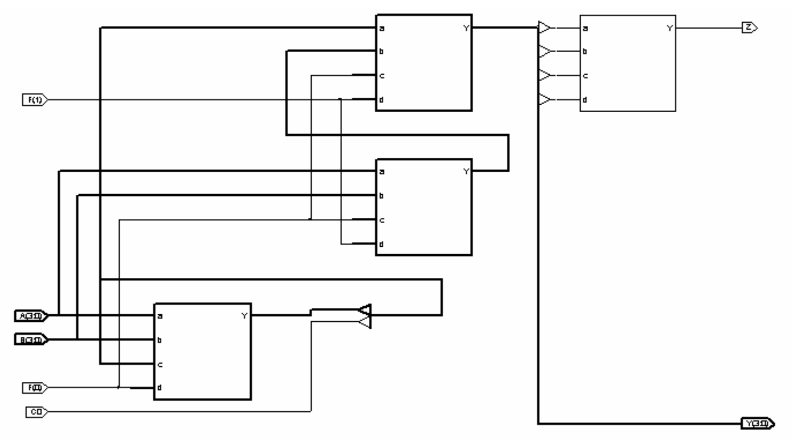

Fig2. LSM structure on the base of LUTs.

**Structured model of LSM on the base of PLD**.

The previous example of LSM based on LUT can be easily redesigned using PLD-type elements except LUTs. Then all the PLDs have the equal object declaration:

```
use  Cnetlist.all;
entity PLM_4 is
  generic(td:time:=1 ns);           -- delay for modeling
  port(A : in BIT;
       B : in BIT;
       C : in BIT;
       D : in BIT;
       Y : out BIT);
end PLM_4;
```

The LNI architecture of the function $X_i$ calculation (see tabl.1) looks like the following:

```
architecture PLMI of PLM_4 is
     begin
       Y<=(          -- arithmetic operations
         (not D and not C and not B and a)        -- A+B
         or (not D and not C and B and not A)
         or (not D and  C and not B and not A)    -- A-B
         or (not D and  C and  B and A)
         --         logic operations
        or  ( D and not C and B and A    )        -- AND
         or (( D and C) and not(not B and not A))    -- OR
         )              after td;   --delay
     end PLMI;
```

It is considered that at the inputs `D,C,B,A` the signals `F(1),F(0),B(i),A(i)` are inputted, respectively . Analogously according to the tables 2,3 the blocks LNO and LNS are described as the architectures `PLMO` and `PLMS`, respectively

```
architecture PLMO of PLM_4 is
     begin
       Y<=(          -- arithmetic operations
         (not D and not C and not B and A) --  A+B
         or (not D and not C and B and not A)
         or (not D and  C and not B and not A)--  A-B
         or (not D and  C and  B and A)
                    --logic operations
        or ( D and not C and  B)      --  AND
         or( D and C and  B)         --  OR
         )              after td;
     end PLMO;
```

```
architecture PLMS of PLM_4 is
     begin
       Y<=(          -- arithmetic operations
         (not D and not C and  B and A)          -- A+B
```

```
        or (not D and C and not(not B and not A))
        or ( D and not C and not B and A)        --A-B
        or ( D and C and not B)
        or ( D and C and  B and A)
        )               after td;   --
    end PLMS;
```

The unit OR-NOT which derives the zero flag is described by the following architecture

```
architecture PLM_NOR of PLM_4 is
     begin
      Y<=not (D or C or  B or A)  -- функция ИЛИ-НЕ
          after td;     --задержка элемента
     end PLM_NOR;
```

Now when architectures of all the components are prepared the description of whole LSM can be done. It is below.

```
     architecture STR_PLM of LSM is
      signal c,x,yi: BIT_VECTOR(4 downto 0); --inner signals
      component PLM_4 is       --used conponent
     --with different architectures
      generic(td:time:=1 ns);
            port(a : in BIT;
                  b : in BIT;
                  c : in BIT;
                  d : in BIT;
                  Y : out BIT);
      end   component;

     begin
      c(0)<=C0;--input carry
     -- 4 bits of LSM
      LSM_STR:for i in 0 to 3 generate
            LNI:entity PLM_4(PLMI) port map  --  LNI
               (a=>A(i),b=>B(i),c=> F(0),d =>F(1),
                  Y =>X(i));
            LNO: entity PLM_4(PLMO) port map  --  LNO
               (a=>C(i),b=>X(i),c=> F(0),d =>F(1),
                  Y =>yi(i));
            LNC:entity PLM_4(PLMS) port map   --  LNC
               (a=>A(i),b=>B(i),c=> c(i),d =>F(0),
                  Y =>c(i+1));
      end generate;

      UZ:entity PLM_4(PLM_NOR) port map       --NOR
               (yi(3),yi(2),yi(1),yi(0),Z);

      Y<=yi(3 downto 0);                          -- Result
      C3<=c(4);

     end STR_PLM;
```

One can to find that by the component instantiation with the architectures `PLMI, PLMO, PLMS` the named binding of ports and signals is used. Such binding is preferable? Because it shows the port-signal relation explicitely, and it stops the errors occuring. When the entity `PLM_4(PLM_NOR)` is instantiated the positioning binding is used, i.e. the attached signal stays in the respective position. Such a binding is agreeable here because the function NOR does not depend on the input permutations, and it is difficult to bind with an error.

**Testbench for LSM**

The digital networks are usually tested on all the design stages. To test the VHDL – models automatically the testbench is usually used. One of the testing methods consists in comparing the testing model with the perfect model. Consider the testbench for the architecture `LSM(STR_LUT)`, for which the perfect model is the architecture `LSM(BEH)`.

Such a testbench – the object `lsm_tb` – is below.

```
entity lsm_tb is
end lsm_tb;
architecture TB_ARCHITECTURE of lsm_tb is
  component lsm --tested objects
        port(F : in BIT_VECTOR(1 downto 0);
              A : in BIT_VECTOR(3 downto 0);
              B : in BIT_VECTOR(3 downto 0);
              C0: in BIT;
              Y : out BIT_VECTOR(3 downto 0);
              C3: out BIT;
              Z : out BIT );
  end component;
  component    RANDOM_GEN is      --random number generator
        generic(n:positive:=4; --output data width
              tp:time:=100 ns   ;      -- CLK period
              SEED:positive:=12345);    -- initial state
        port(CLK:out BIT;
              Y : out BIT_vector(n-1 downto 0));
  end component;
--testing signals
  signal F : BIT_VECTOR(1 downto 0):="00";
  signal C0 : BIT:='0';
  signal A,B : BIT_VECTOR(3 downto 0);
--proved signals
  signal Y1,Y2,Y : BIT_VECTOR(3 downto 0);
  signal C31,C32,C, Z1,Z2,Z: BIT;
begin
  G1: RANDOM_GEN                          --operand A generator
      generic map(n=>4,SEED=>1234)
        port map(CLK=>open,Y =>A);
  G2: RANDOM_GEN                          -- operand B generator
      generic map(n=>4,SEED=>8765)
      port map(CLK=>open,Y =>B);

  UUT1 :entity lsm(STR_LUT)          --Ttested component
      port map (    F => F,     A => A,     B => B,     C0 => C0,
        Y => Y1,  C3 => C31, Z => Z1);
  UUT2 :entity lsm(BEH)              --perfect component
        port map (  F => F,A => A,    B => B,     C0 => C0,
        Y => Y2, C3 => C32,     Z => Z2     );
--comparers to signal comparing
 COMP_Y:     Y<=Y1 xor Y2;
 COMP_C:     C<=C31 xor C32;
 COMP_Z:     Z<=Z1 xor Z2;
end TB_ARCHITECTURE;
```

Entity `lsm_tb` has not the input-output ports and therefore it has the simple declaration. The perfect and tested components LSM have the equal interfaces but the different architectures. Therefore they are declared as a single entity. But when they are instantiated, they are marked as `entity lsm(BEH)` and `entity lsm(STR_LUT)`. When the random number generators are instantiated, the generic constant *n*=4 gives their bit width, and the generic constants `SEED` give the different initial states to provide different values of the signals A and B.

The structure of the derived testbench which is gebnerated by the tool Code2Graphics of the Active HDL is shown in Fig. 3.The control signals Y and C0 are exchanged by hand in the modeling process.

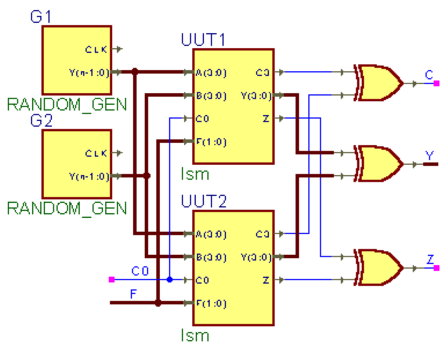

Fig.3. Testbench structure

The modeling results are compared by the XOR function. If the modeling results are equal then these functions are zeroed.

5. **Laboratory exersize implementation**

According to the variant number the task is selected.The task parameters include:

- Resulting network bit width n;
- A set of LSM operations;
- A set of output signals.

The task variant is selected on the base of 2 last figures *ab* of the student credit record.

Then the bit width is n=2(a+4).

The table 4 shows the LSM function set to be implemented

Table 4. LSM function set

| b | Function set | Output signals |
|---|---|---|
| 0 | AND, OR,  A+B, A+B+Ci, 1 | Y, Cn |
| 1 | AND, XOR, NOT A AND B, A+B, 1 | Y, N |
| 2 | OR, XOR, NOT A AND B, A+B, 0 | Y, Cn |
| 3 | AND, A+B, A-B, A+B+Ci, A-B-Ci | Y, Z |
| 4 | AND, OR,  A-B, A-B-Ci, 1 | Y, Cn |
| 5 | OR, XOR, NOT A AND B, A-B, 1 | Y, N |
| 6 | OR, NOT A AND B, A+B, A-B, 0 | Y, Cn |
| 7 | AND, OR, A+B, A+B+Ci, 1 | Y, Z |
| 8 | AND, OR, XOR, A+B, A+B+Ci, 0 | Y, Z,N |
| 9 | AND, OR, XOR, A+B,A-B | Y, Cn, Parity |

The laboratory exersize implementation has 3 stages: behavioral model development, structural model development and testbench development wit the model testing.

**Behavioral model development.**

The behavioral model of LSM is described by the dataflow style using the ooperations with bit vectors and integers, and functions of the package Cnetlist or the package IEEE.Numeric_bit. Here the VHDL editor and compiler of Active HDL are used.

After LSM describing as the architecture LSM(BEH), it is tested by the inputting some sets of testing signals during its simulation.

**Structural model development**

Such model is described by the structural style. The proper components from the file Cnetwork_Lib.VHD are used.

**Thestbench development and model testing**

The testbench shown above serves as an example of such a testbench. It is redesigned according to the needs of the given variant of the tested entity.

When the models are tested due to the signal waveforms the model correct operation is proven and the signal delays between inputs and outputs are measured. The derived waveforms are replaced to the report using options ctrl-c and ctrl-v. Dur th the testing results the conclusions to the laboratory exersize is done.

**6. Laboratory exersize report**

The laboratory exersize report must contain:

- Goal of the work,
- LSM variant description,
- PLD model synthesis process,
- VHDL texts of the behavioral and structural models of LSM,
- Waveforms of the testbench,
- Measured delays,
- Conclusions.