Digital Network Design
Short Practical Tutorial

Laboratory exercise 1
Arithmetic and logic unit

1 Goal:
The goal is to achieve knowledge and practical experience in ALU design for modern computers, to get programming and debugging experience 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 instruction opcode set.

3. ALU design example
Consider the example of 4-bit ALU, 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.

```
library IEEE;
use IEEE.Numeric_bit.all;
entity ALU 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 ALU;
```

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 ALU can be described as the following architecture.

```
architecture NUM of ALU is
  signal ai,bi,yi,bp1:signed(4 downto 0);
  signal ybi:BIT_VECTOR(4 downto 0);
begin
  ai<= '0'& signed(A);-- 1 bit is added to select the carry out bit
  bi<= '0'& signed(B);
  bp1<=bi     when C0='0' else -carry bit is added
       bi+1;
```

```
ADDER: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;
```

In the architecture declaration the signals are declared, which are used as intermediate signals in the calculations. The type `signed` is the subtype of the type `BIT_VECTOR`, which represents an integer value with MSB as a sign. After the declarative part of the architecture, the description 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 signed signals `ai,bi.` They are made in one bit longer than input signals to derive the carry out bit. Note that explicit type conversion is allowed between closely related types only, which are here `signed` and `BIT_VECTOR` of equal length.

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 gives. 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`.

Such an example is compiled (synthesized) with small hardware volume (only 15 LUT).

4. Testbench for ALU

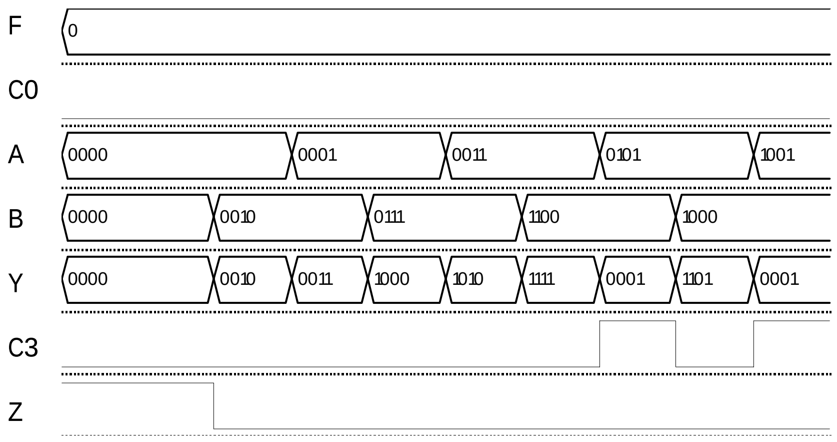
The digital networks are usually tested on all the design stages. To test the VHDL – models automatically the testbench is usually used. Consider the testbench for the architecture ALU(NUM). Such a testbench – the object `ALU_tb` – is below.

```
entity ALU_tb is
end ALU_tb;
architecture TB_ARCHITECTURE of lsm_tb is
  component ALU --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;
--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 Y : BIT_VECTOR(3 downto 0);
  signal C3,Z: BIT;
begin
  F<="00";
  C0<='0';
  A<="0000", "0001" after 30 ns,"0011" after 50 ns,"0101" after 70 ns,"1001" after 90 ns;
```

```
B<="0000", "0010" after 20 ns,"0111" after 40 ns,"1100" after 60 ns,"1000" after 80 ns;
UUT: ALU          --tested component
    port map (F => F, A => A,B => B,C0 => C0,
    `             Y => Y,  C3 => C3, Z => Z);
 end TB_ARCHITECTURE;
```

Here the input data signals A and B are generated as waveforms, i.e. their values are exchanged after some delay which is given by the AFTER clause.

The resulting modeled waveforms are shown in the following figure. The result correctness is proven "by hand", i.e. one proves that by addition (see fig.) 0101+1100=0001 and C3=1 because 5+(-4) = 1. The modeling is implemented with different values both F and C0 proving ALU operation in different modes.



5. **Laboratory exercise implementation**

Each exercise variant has a set of parameters, which are numbered by natural numbers. A set of them is derived from the record-book number of the student. Consider 3 last figures a_2, a_1, a_0, of the record-book number. Then the variant number is

$N = 100a_2 + 10a_1 + a_0 = 2^9b_9+2^8b_8+2^7b_7+2^6b_6+2^5b_5+2^4b_4+2^3b_3+2^2b_2+2^1b_9+b_0$,

where b_i are the bits of the number N in the binary representation.

The data bit width N_{DB} is selected from the Table 1. The data are the twos complement integers.

Table 1

b_5, b_4	00	01	10	11
N_{DB}	8	16	24	32

The ALU operations to be implemented are represented in the Table 2.

Table 2

b_2, b_1, b_0	ALU operations	Output signals
000	AND, OR, NAND, ADD, ADDI, ADDC, SRA	Y, Cn
001	AND, XOR, ADD, ADDI, SUB, SRA	Y, N, Parity
010	AND, OR, XOR, ADD, ADDI, ADDC, SUB	Y, Cn
011	AND, XOR, ADD, ADDI, SUB, SRA	Y, Z, Parity
100	AND,NAND, ADD, ADDC, SUB, SRA	Y, Cn
101	AND, XOR, ADD, ADDC, SUB, SUBB,SRL,SRA	Y, N
110	AND, XOR, ADD, ADDC, SUB, SRA,	Y, Cn
111	AND, OR, XOR, ADD, ADDC, SUB, SRA	Y, Z

Where *x* NAND *y* is NOT(*x* & *y*); XOR – exclusive OR; ADDI – addition with the immediate operand, which sign bit is expanded to the whole data width, i.e. ALU must have the additional input for that operand, which bit width is 8; ADDC – addition with the carry bit; SUBB – subtraction with a borrow (carry bit); SRA – arithmetical right shift to a single bit, SRL – logical shift.

The laboratory exercise implementation has 2 stages: behavioral model development, and testbench development wit the model testing.

Behavioral model development.

The behavioral model of ALU is described by the dataflow style using the operations with bit vectors and integers, and functions of the package Cnetwork or the package IEEE.Numeric_bit. Here the VHDL editor and compiler of Active HDL 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. Finally, the conclusions to the laboratory exercise based on the testing result are done.

6. Laboratory exercise report

The laboratory exercise report must contain:

- Goal of the work,
- ALU description,
- VHDL texts of the behavioral model and testbench,
- Waveforms of the testbench,
- Conclusions.

Laboratory exercise 2

Program Counter

1. Goal:

to achieve theoretical and practical knowledge in program counter design. The laboratory exercise serves as a lesson to program in VHDL the behavior both combinational networks and triggers as well.

2. Theoretical informatiom

The program counter (PC) named also as instruction counter (ICTR) is used for the calculation the next instruction address in the CPU. Depending on the instruction type and condition flag, PC outputs different addresses of the next instruction.

If the running instruction is not a branch instruction, or a branch instruction but the branch condition is false then the next instruction address is formed from the recent address by the addition of the instruction length code of *k* bytes. If this instruction is the branch instruction and the branch condition is true then the next instruction is the branch address. This address is derived from the address field of the instruction as the absolute address (call-type instruction), or the relative displacement, which is added to the PC state. By the initialization the PC state must be reset.

3. PC design example

Consider the PC which forms 13-bit width address, adds the 8-bit displacement by the signal ED and installs the absolute address AA by the signal WR. Let the address increment is always a 1, i.e. the instruction length is stable. Such PCs are built in the RISC-like processors. Then the following VHDL model represents such a PC.

```
library IEEE;
use IEEE.Numeric_Bit.all;
entity PC is
      port(CLK: in BIT;  -- clock input
            R :  in BIT;  -- reset
            EPC: in BIT;  -- PC operation enable
            WR: in BIT;  -- write absolute address
            ED: in BIT;  -- enable displacement addition
            AA: in BIT_VECTOR(12 downto 0);   -- absolute address
            D : in BIT_VECTOR(7 downto 0); -- branch address displacement
            A : out BIT_VECTOR(12 downto 0));  -- output address
end PC;

architecture PC of PC is
      signal ai:unsigned(12 downto 0); --inner signal - register-counter
begin
      PC_P: process(CLK, R)    --process describes the program counter
      begin
            if R='1' then
                  ai<=(others=>'0');         -- PC reset
            elsif RISING_EDGE(CLK) then
                  if EPC='1' then
                        if WR<='1' then
                              ai<=unsigned(AA);-- absolute address loading
                        elsif ED='1' then
                              ai<=ai + unsigned(D); --displacement addition
                        else
                              ai<=ai+1; -- increment
                        end if;
                  end if;
            end if;
      end process;

      A<=BIT_VECTOR(ai);
end PC;
```

Here the signal `ai` is the inner signal, which is inferred by the synthesizer as the register-counter. This signal is assigned to the output port A. The port-signal A could not be used instead of `ai` because he has the out-mode, which prohibites it to be an operand in a statement. The behavior of PC is described by the process PC_P. The process is such parallel operator which describes the sequential behavior. The operators in the process body are implemented sequentially from the first operator to the last one. The process is started by the event of exchange of selected signals which form the sensitivity list, and which is put in the brackets after the reserved word Process.

The conditions of the if-then-elsif-else operator are proven sequentially. And if some condition is true then the respective operators are implemented, and this logical operator is finished. Here the condition `RISING_EDGE(CLK)` is the function call which returns true if at considered moment the rising edge of the clock signal occurs. This means that the signal `ai` is written by the new value, i.e. it behaves as the register.

The signal condition `EPC='1'` allows writing to `ai`, that means that PC may be waiting on `EPC`, for example, when in CPU some wait states are introduced.

4. Testbench for PC

One of the possible testbenches for PC is the following

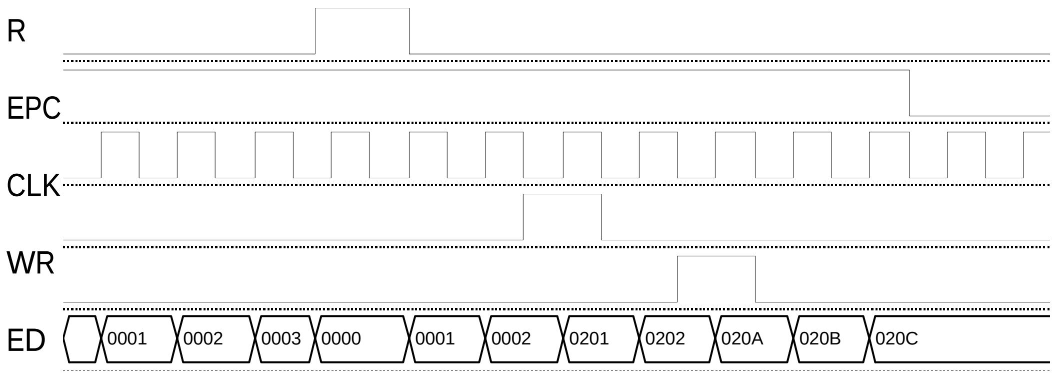
```
library IEEE;
use IEEE.Numeric_Bit.all;
entity PC_tb is
end pc_tb;
architecture TB_ARCHITECTURE of pc_tb is
      component pc is port(
            CLK : in BIT;
            R : in BIT;
            EPC : in BIT;
            WR : in BIT;
            ED : in BIT;
            AA : in BIT_VECTOR(12 downto 0);
            D : in BIT_VECTOR(7 downto 0);
            A : out BIT_VECTOR(12 downto 0) );
      end component;

      signal CLK, R, EPC, WR, ED: BIT;
      signal AA, A: BIT_VECTOR(12 downto 0);
      signal D : BIT_VECTOR(7 downto 0);
begin
      R<='0', '1' after 33 ns, '0' after 45 ns;
      CLK<=not clk after 5 ns;                    --clock generator
      EPC<='1', '0' after 110 ns;
      WR<='0', '1' after 60 ns, '0' after 70 ns;
      ED<= '0', '1' after 80 ns, '0' after 90 ns;
      D<="00001000";
      AA<="0001000000001";

      UUT : pc   port map ( CLK => CLK, R => R,EPC => EPC, WR => WR,
                 ED => ED, AA => AA,D => D, A => A);
end TB_ARCHITECTURE;
```

Here the waveform assignment operator `CLK<=not clk after 5 ns;` after each 5 nanosecond inverts the value of the clock signal, i.e. it generates the meander waveform with the period of 10 nanoseconds. Note, that all the bit signals including this one are initialized before modeling as zeros. Therefore, the rising edges of the clock signal occur in 5-th, 15-th, etc. nanosecond.

The resulting waveforms which show the correct PC operation are in the figure below.



5. **Laboratory exercise implementation**

Each exercise variant has a set of parameters, which are numbered by natural numbers. A set of them is derived from the record-book number of the student. Consider 3 last figures a_2, a_1, a_0, of the record-book number. Then the variant number is

$N = 100a_2 + 10a_1 + a_0 = 2^9b_9+2^8b_8+2^7b_7+2^6b_6+2^5b_5+2^4b_4+2^3b_3+2^2b_2+2^1b_1+b_0$,

where b_i are the bits of the number N in the binary representation.

The instruction address bit width N_{IA} is selected from the Table 1.

Table 1

b_9, b_8	00	01	10	11
N_{IA}	14	16	18	20

The program counter PC has the data width N_{IA} as well.

The displacement bit width is selected from the Table 2.

Table 2

b_1, b_0	00	01	10	11
N_D	8	10	12	14

Laboratory exercise 3
Program Counter

1. Goal:
to achieve theoretical and practical knowledge in memory unit designing like the register array. The laboratory exercise serves as a lesson to program in VHDL the behavior both combinational networks and triggers as well.

2.Theoretical information

The register array (RA) is intended for the fast speed access and transfer of n-bit wide data through a set of ports. In the laboratory exercise the data bit width N_R from 4 to 16, and the array volume from 8 to 32 words are considered. Each of RA ports has its own address bus. The port number is equal to 2 or 3. The ports are marked as B, D, or Q. One of ports is intended for writing, and another – for reading are. The ports can be bidirectional, i.e. they are used both for reading and for writing. In the laboratory exercise the bidirectional busses are recommended to be implemented on the tristate bus.

The writing operation is done always on the rising edge of the clock signal or write signal. The reading operation is done immediately just after the address code is set.

3. **Example of the RA description**

Consider an example of the 3-port RA design with the volume of 8 sixteen bit words. The RA entity declaration looks like the following.

```
    use work.CNetlist.all;
    entity RA is
      port(CLK:in BIT; -- синхровход
           WR:in BIT;  -- сигнал записи
           AB:in BIT_VECTOR(2 downto 0);-- адрес канала B
           AD:in BIT_VECTOR(2 downto 0);-- адрес канала D
           AQ:in BIT_VECTOR(2 downto 0);-- адрес канала Q
           Q: in BIT_VECTOR (15 downto 0);-- данное канала Q
           B: out BIT_VECTOR(15 downto 0);-- данное канала B
           D: out BIT_VECTOR(15 downto 0));-- данное канала D
    end RA;
```

Behavioral model of RA

The main feature of the RA model consists in that that two parallel reading operations and one writing operation are implemented through 3 different addresses. This behavior is described by the following architecture.

```
architecture BEH of RA is
    type MEM8X16 is array(0 to 7) of BIT_VECTOR(15 downto 0);
    signal addr,do: BIT_VECTOR(15 downto 0);
begin
RA8:process(CLK,AD,AB) ---- блок регистровой памяти --------------
          variable RAM: MEM8x16;
          variable addrq,addrd,addrb:natural;
        begin
          addrq:= BIT_TO_INT(AQ);
          addrd:= BIT_TO_INT(AD);
          addrb:= BIT_TO_INT(AB);
          if CLK='1' and CLK'event then
                if WR = '1' then
                   RAM(addrq):= Q;  -- запись
                end if;
          end if;
          B<= RAM(addrb);          -- чтение канала B
          D<= RAM(addrd);          -- чтение канала D
    end process;
end BEH;
```

This description is written by the style for synthesis. The synthesizer implements in on the set of triggers. Fig. below illustrates the structure of this RA.

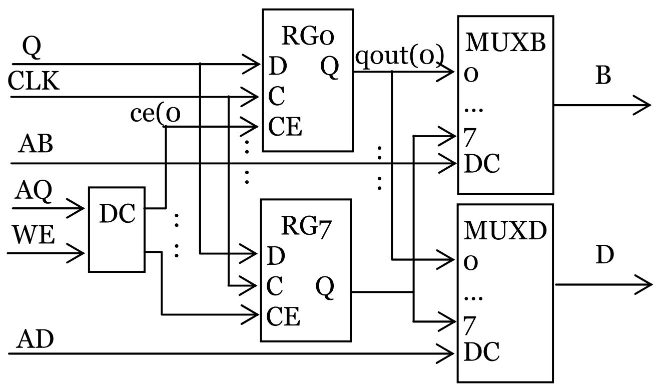


Fig.3.1. RA structure

4. **Laboratory exercise implementation**

Each exercise variant has a set of parameters, which are numbered by natural numbers. A set of them is derived from the record-book number of the student. Consider 3 last figures a_2,a_1,a_0, of the record-book number. Then the variant number is

N = $100a_2 + 10a_1 + a_0 = 2^9b_9+2^8b_8+2^7b_7+2^6b_6+2^5b_5+2^4b_4+2^3b_3+2^2b_2+2^1b_9+b_0$,

where b_i are the bits of the number N in the binary representation.

The data bit width N_{DB} is selected from the Table 1. The data are the twos complement integers.

Table 1

b_5, b_4	00	01	10	11
N_{DB}	8	16	24	32

The register file volume N_R is selected from the Table 2.

Table 2

b_3, b_2	00	01	10	11
N_R	8	16	32	8*4 (4 banks of 8 registers)

When 4 banks are considered, then the register file consists of 4 equal separate parts, one odf them is selected by the 2-bit bank select bus.

The register address number N_{RA} is selected from the Table 3.

Table 3

b_1, b_0	00	01	10	11
N_{RA}	1	2	3	3

When N_{RA} =2 one of the addresses is used both for reading and for writing. When N_{RA} =3 one of the addresses is used only for writing.

Laboratory exercise 4

Finite State Machine

1. Goal:

to achieve theoretical and practical knowledge in the finite state machine (FSM) designing. The laboratory exercise serves as a lesson to program in VHDL the authomata behavior, which is implemented as FSM.

2.Theoretical information

FSM is a sequential logic network, which implements the given control algorithm as the predefined sequence of its states. Its next state S_{i+1} depends on the previous one S_i and on the input signals X_j. Its outputs Y_p depend logically on its state S_i. Such FSM is named as a Moore FSM. When its outputs Y_p depend both on its state S_i and on input signals X_j, then it is named as a Mealy FSM. The FSM algorithm is fully described by its state graph (state diagram) or by FSM chart. Modern CADs synthesize the FSM from its state graph automatically.

The nodes of the state graph represent the FSM states, and its directed edges represent the branches from one state to another. In the Moore state graph the node S_k is labelled by variables Y_P, separated by a slash "/", which output a 1, when FSM stays in this state. The edges are labelled by the input labels which are the boolean functions of the input variables, and which derive the branch conditions. They can be labelled by the output labels, that are output variables, when it its the Mealy state graph. In Fig.4.1. is an example of the state graph, which has both Moore and Mealy outputs.

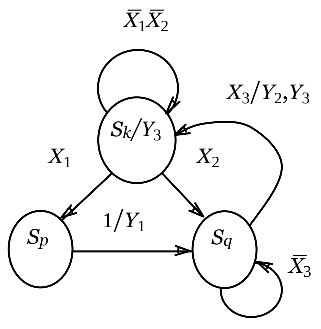


Fig.4.1

If we label an edge X_iX_j/Y_pY_q, this means if inputs X_i and X_j are 1 (we don't care what the other input values are), the outputs Y_p and Y_q are 1 and (other outputs are 0), and we will traverse this arc to go to the next state. For example, in the graph in Fig.1 the state S_k remains the same when $\overline{X}_1\overline{X}_2 = 1$ and it is exchanged to the state S_k when $X_1 = 1$, providing the output signal $Y_1 = 1$. In order to have a completely specified proper state graph, in which the next state is always uniquely defined for every input combination, we must place the following constraints on the input labels for every state S_k:

If F_i and F_j are any pair of input labels (boolean functions) on edges exiting state S_k, then $F_i \cdot F_j = 0$, if $i \neq j$.

If n edges exit state S_k, and they have input labels $F_1, F_2, \ldots, F_n$, respectively, then $F_1 \vee F_2 \vee \ldots \vee F_n = 1$.

The first condition assures us that at most one input label can be 1 at any given time, and the second condition assures us that at least one input label will be 1 at any given time. Therefore, exactly one label will be 1, and the next state will be uniquely defined for every input combination. For example in Fig.4.1 conditions are satisfied for all the nodes.

As an alternative to state graphs, a state machine flowchart, or FSM chart is. Just as flowcharts are useful in software design, they are useful in the hardware design of digital systems. The state in it is represented by a state box. It contains a state name, followed by a slash "/" and an optional output list. A state code may be placed outside the box. A decision box is represented by a diamond-shaped symbol with true and false branches. The condition placed in the box is a Boolean expression that is evaluated to determine which branch to take. The conditional output box, which has curved ends, contains a conditional output list.

An FSM chart is constructed from FSM blocks. Each of them contains exactly one state box, together with the decision boxes and conditional output boxes, associated with that state. Block has one entrance path and one or more exit paths. Each block describes the operation during the time that FSM is in one state. A path through a block from entrance to exit is referred to as a link path.

Certain rules must be followed when constructing an FSM block. First, for every valid combination of input variables, there must be exactly one exit path defined. This is necessary since each allowable input combination must lead to a single next state. Second, no internal feedback within a block is allowed.

Table 4.1

Present state Z_2Z_1	Next state, $X_3X_2X_1$=								Present output, $X_3X_2X_1$=							
	000	001	010	011	100	101	110	111	000	001	010	011	100	101	110	111
S_k 00	00	01	10	–	00	01	10	–	Y_3							
S_p 01	10								Y_1							
S_q 10	10		00		10		00		0	Y_2, Y_3						

It is easy to convert a state graph to an equivalent FSM chart. The chart, which is equivalent to one in Fig.4.1, has three blocks – one for each state. The Moore output Y_3 is placed in the state box S_k, since it does not depend on the input. Some condition nodes X_1, X_2 have a single output. This is explained by the fact that the Mealy outputs Y_1, Y_2 appear in conditional output boxes, since they depend on both the state and input. The resulting FSM chart is shown in Fig.4.2.

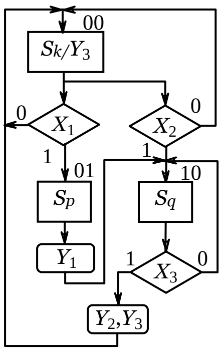


Fig.4.2

The FSM network contains a set of triggers and a logic network. Triggers store the FSM state, orher words, they form a state register. The logic network consists of two parts. One of them generates signals D_i, which are the trigger stimulating functions. The another one outputs the resulting signals Y_j.

First of all, the states S_k are given the concrete values. There is a set of methods of coding the states. The method selection depends on the number of states, if the FSM is optimized for speed or hardware volume, or error immunity. The one-hot coding means that for n state FSM the n-bit wide state word is selected, in which a 1 stays in the k-th position, when coding the state S_k. For example, FSM with the state graph in Fig.4.1 would have the state coding 001, 010, 100. This coding is usually used when the state number is small. It provides usually the highest speed, because the trigger stimulating functions occur to be rather small. When the state graph contains the long chains of nodes, the state register can be implemented as a shift register.

The natural number coding is used in most of cases, especially, when the state graph contains the long chains of nodes. In the example on Fig.4.2, the codes are 00, 01, and 10. In this situation, the state register behaves as a counter. When the states are coded by the Gray codes, then in most of state branches, only a single bit of the code is exchanged. This serves both to LN minimization and to error immunity. In the combined coding, the code word is divided into 2 or more fields, each of them are coded by some coding method. Here the advantages of different methods can be used. For example, when the code word has two n-bit fields, which have one-hot coding, then such code word can code up to n^2 states.

Then, the state table is built. This table has the columns of present state, next state and present output. The next state column contains the subcolumns, which are coded by the bits of the input signals. These subcolumns show what next state is for the given value of the input variable. The present output column has the similar form. Table 4.1. is the state table for the FSM chart in Fig.4.2.

From the next state columns of the state table the trigger stimulating functions are derived, as signals that force the D triggers of the state register to be set:

$$D_1 = \bar{Z}_2\bar{Z}_1(\bar{X}_3\bar{X}_2X_1 \vee \bar{X}_3X_2X_1 \vee X_3\bar{X}_2X_1 \vee X_3X_2X_1) = \bar{Z}_2\bar{Z}_1X_1;$$
$$D_2 = \bar{Z}_2\bar{Z}_1(\bar{X}_3X_2\bar{X}_1 \vee \bar{X}_3X_2X_1 \vee X_3X_2\bar{X}_1 \vee X_3X_2X_1) \vee \bar{Z}_2Z_1 \vee Z_2Z_1 \vee Z_2\bar{Z}_1\bar{X}_2 =$$
$$= \bar{Z}_2\bar{Z}_1X_2 \vee Z_1 \vee Z_2\bar{Z}_1\bar{X}_2.$$

Note that the don't-care states in the combination X_2X_1=11 and in the state Z_2Z_1 = 11 (for D_2) are assigned as a 1. The output functions are derived from the present output columns as well:

$$Y_1 = Z_1;\ \ Y_2 = Z_2X_3;\ Y_3 = \bar{Z}_2\bar{Z}_1 \vee Z_2X_3;$$

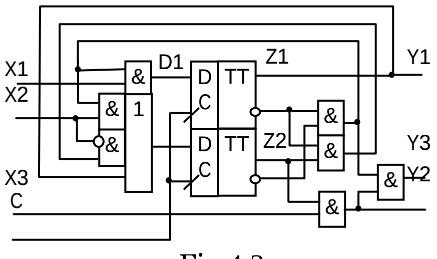


Fig.4.3

The resulting FSM network is shown in Fig.4.3.

Once the state graph or the FSM chart are build, then the FSM can be easily described in VHDL and then its network can be synthesized by a compiler. Here a case statement can be used to specify what happens in each state. Each condition box corresponds directly to an if statement. The following program describes FSM with the chart in Fig.4.3.

```
entity FSM1 is port(C,X1,X2,X3:in bit;
                        Y1,Y2,Y3:out bit);
end FSM1;
architecture beh of FSM1 is
    signal S,D:bit_vector(0 to 1);        -- state codes
begin
  LN:process(X1,X2,X3,S) begin            --LN model
        Y1<='0';Y2<='0';Y3<='0'; --usual output states
        case S is
        when "00" => Y3<='1';    -- current state Sk
              if X1='1' then
                          D<="01"; --next state
                   elsif X2='1' then
                   D<="10";
              else   D<="00"
              end if;
        when "01"=> D<="10"; Y1<='1'; -- state Sp
        when others=> if X3='1' then   -- state Sq
                   D<="00";Y2<='1';Y3<='1';
              else      D<="10";
              end if;
        end case;
    end process;
 RG:process(C) begin               -- state register
        if C='1' and C'event then
              S <= D; -- update state on rising edge of C
        end if;
    end process;
end beh;
```

The first process represents the logic network of FSM, and the second process updates the state register on the clock. The signals *Y1, Y2, Y3* are turned on in the appropriate states, and they must be turned off when the state *S* changes. A convenient way to do this is to set them all to 0 at the start of the process.

4. **Laboratory exercise variants**

Each exercise variant has a set of parameters, which are numbered by natural numbers. A set of them is derived from the record-book number of the student. Consider 3 last figures a_2,a_1,a_0, of the record-book number. Then the variant number is

$N = 100a_2 + 10a_1 + a_0 = 2^9b_9+2^8b_8+2^7b_7+2^6b_6+2^5b_5+2^4b_4+2^3b_3+2^2b_2+2^1b_1+b_0$,

where b_i are the bits of the number N in the binary representation.

The FSM must implement the slot machine control. Consider the slot machine, which solds some product, which is costs N_C kopecks. Then this product will be outputted, when the sum N of the coins, which are dropped by the customer, is equal to N_C. The nominal values of the coins can be V_1**,** V_2, and V_3. Therefore, the FSM must provide the output signal OK=1 when $N_C= a_1V_1+ a_2V_2+ a_3V_3$, where $a_1, a_2, a_3 \in \{0,1,2,3,...\}$

The values N_C , V_1**,** V_2, and V_3 are selected from the Table 4.2.

Table 4.2

b_3,b_2,b_1,b_0	N_C	V_1	V_2	V_3
0 0 0 0	20	5	10	-
0 0 0 1	30	5	25	-
0 0 1 0	50	10	25	50
0 0 1 1	55	10	25	-
0 1 0 0	60	10	25	-
0 1 0 1	60	10	25	50
0 1 1 0	70	10	50	-
0 1 1 1	75	25	50	-
1 0 0 0	100	25	50	-
1 0 0 1	125	25	50	
1 0 1 0	125	25	50	100
1 0 1 1	150	25	50	-
1 1 0 0	150	50	100	-
1 1 0 1	175	25	100	-
1 1 1 0	200	50	100	-
1 1 1 1	300	50	100	-

5. **Example of the FSM design**

Consider an example of the slot machine with the following parameters :

N_C	V_1	V_2	V_3
25	5	10	-

The FSM entity declaration looks like the following.

```
entity FSM is
 port(CLK:in BIT; -- synchrosignal
      RST:in BIT;  -- reset input
      V1:in BIT;-- first type coin is dropped
      V2:in BIT;-- second type coin is dropped
      DR:out BIT;-- the instruction: "Drop a coin"
      OK:out BIT;-- the sum is achieved
      ERR: out BIT);-- the achieved sum is false
end FSM;
```

State graph design

The state graph is designed according to the algorithm of the slot machine operation. The state graph nodes form a set of levels or stages. Ther first node belongs to the first stage, it is the state when the machine is resetted. The second stage is formed by the nodes, which fix the events, when the first coin V_1 or V_2 is dropped. The third stage is formed by the nodes, which fix the events, when the second coin of the value V_1 or V_2 is dropped, etc. And the last stage is formed by the final nodes, when the needed sum is achieved or this sum is error one. The states are named according to the sum, which is got, say, when the recent sum is 10 kopecks then the state is N10. The derived state graph is illustrated by the Fig.4.4.

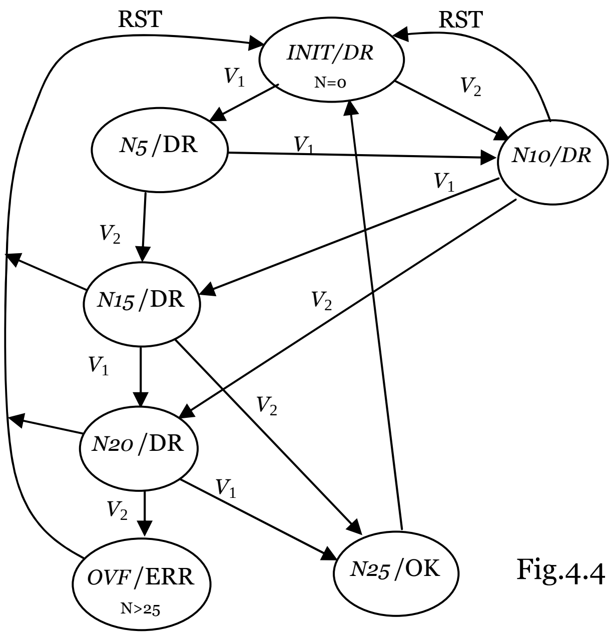


Fig.4.4

FSM description

The algorithm represented by the state graph on Fig.4.4. is described as the following architecture.

```
architecture BEH of FSM is
  Type STATE is  (INIT, N5, N10, N15, N20, N25, OVF);--states of the FSM
  signal st : STATE;
begin
  STATES:  process(CLK,RST)       -- FSM process
  begin
        if RST='1' then
              st<=INIT;
        elsif CLK='1' and CLK'event then
              case st is    --next FSM state depending on actual state and signals V1 and V2
                    when INIT => if V1='1' then st<=N5;
                                 elsif V2='1' then st<=N10;
                                 end if;
                    when N5 => if V1='1' then st<=N10;
                                 elsif V2='1' then st<=N15;
                                 end if;
                    when N10 => if V1='1' then st<=N15;
                                 elsif V2='1' then st<=N20;
                                 end if;
                    when N15 => if V1='1' then st<=N20;
                                 elsif V2='1' then st<=N25;
                                 end if;
                    when N20 => if V1='1' then st<=N25;
                                 elsif V2='1' then st<=OVF;
                                 end if;
                    when N25 => st<=INIT;
                    when others => null; --when other states – nothing to do
              end case;
        end if;
  end process;
-- output signals
  DR<='1' when st=INIT or  st=N5 or  st=N10 or st=N15 or st=N20 else '0';
  OK<='1' when st = N25 else '0';  -- the sum is achieved
  ERR<='1' when st=OVF else '0';-- the achieved sum is false
end BEH;
```

The derived FSM model was modeled using the signal stimulating. By this process, the signals RST, V1 and V2 were stimulated by the buttons (Hotkey mode of the stimulator). The resulting waveforms are in the Fig.4.5, and they show the correct FSM operation.

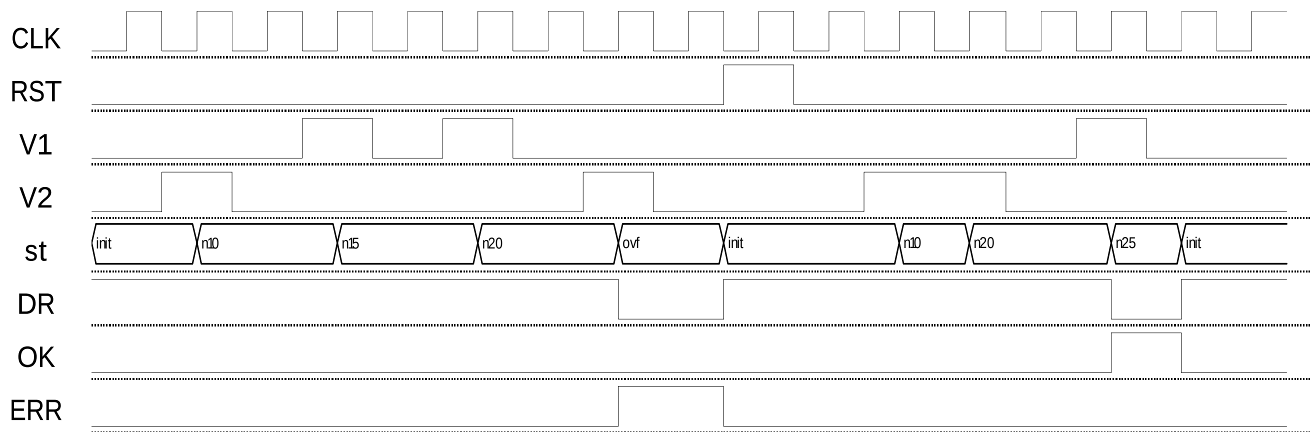


Fig.4.5.