Układy Cyfrowe i Systemy Wbudowane 1

Język VHDL

dr inż. Jarosław Sugier
Jaroslaw.Sugier@pwr.edu.pl
K30W04ND03, pok. 227 C-3

Literatura

- Język VHDL: M. Zwoliński(...) / K. Skahill(...) / IEEE Standard 1076 (PWr!)
- Architektury układów PLD, CPLD: www...; **www.amd.com (Xilinx)**
- J. Kalisz „Podstawy elektroniki cyfrowej", WKiŁ
- C. Zieliński „Podstawy projektowania układów cyfrowych", PWN
- J. Pasierbiński, P. Zbysiński „Układy programowalne w praktyce", WKiŁ
- T. Łuba (red.) „Synteza układów cyfrowych", WKiŁ
- Pong P. Chu „RTL hardware design using VHDL", J. Wiley

Jednostki i architektury

```
library UNISIM;
use UNISIM.VComponents.all;

entity HalfAdder is
    port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           S : out STD_LOGIC;
           C : out STD_LOGIC);
end entity HalfAdder;

architecture Structural of HalfAdder is
begin
    XOR_gate : XOR2 port map ( A, B, S );
    AND_gate : AND2 port map ( A, B, C );
end architecture Structural;

architecture Dataflow of HalfAdder is
begin
    S <= A xor B;
    C <= A and B;
end architecture Dataflow;
```



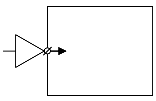
```
architecture Behavioral of HalfAdder is
begin
    process( A, B )
    begin
        -- Sum
        if A /= B then
            S <= '1';
        else
            S <= '0';
        end if;
    end process;

    process( A, B )
    begin
        -- Carry
        if A = '1' and B = '1' then
            C <= '1';
        else
            C <= '0';
        end if;
    end process;

end architecture Behavioral;
```

```
...
-- Or, mixed in one proces:
process( A, B )
begin
    -- Sum
    if A /= B then
        S <= '1';
    else
        S <= '0';
    end if;
    -- Carry
    if A = '1' and B = '1' then
        C <= '1';
    else
        C <= '0';
    end if;
end process;
...
```


Porty i sygnały: tryby pracy portów

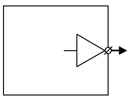
Synteza:

1) **`in`** – READ ONLY



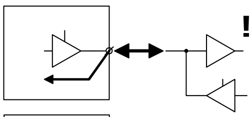
✔

2) **`out`** – WRITE ONLY



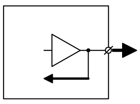
✔

3) **`inout`** – bidirectional (3-state buffers, etc.)



!!

4) **`buffer`** – „out with read capability"



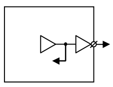
✘

Porty i sygnały: sygnały wewnętrzne

```
entity AndNand is
    port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : in  STD_LOGIC;
           WY_And  : out  STD_LOGIC;
           WY_Nand : out  STD_LOGIC);
end AndNand;

architecture DataflowBad of AndNand is
begin
    WY_And  <= A and B and C;
    WY_Nand <= not WY_And;
end DataflowBad;
```

Compilation:
`HDLParsers:1401 - Object WY_And of mode OUT can not be read.`

```
architecture DataflowOK of AndNand is
    signal Int_And : STD_LOGIC;
begin
    Int_And <= A and B and C;
    WY_And  <= Int_And;
    WY_Nand <= not Int_And;
end DataflowOK;
```



Wektory i napisy bitowe

Standardowy typ z biblioteki `STD_LOGIC_1164`:

```
  type STD_LOGIC_VECTOR is array (NATURAL range <>)
                                     of STD_LOGIC;
```

Użycie:

```
  (...)
  signal DataBus  : STD_LOGIC_VECTOR( 7 downto 0);
  (...)

  DataBus <= "10000000";
  DataBus <= B"1000_0000";
  DataBus <= X"80";                       -- O"..." = octal
  DataBus <= ( '1', '0', '0', '0', '0', '0', '0', '0' );
  DataBus <= ( '1', others => '0' );
  DataBus <= ( 7 => '1', others => '0' );

  DataBus <= ( others => '0' );

  HalfByte <= DataBus( 7 downto 4 );

  MSB <= DataBus( 7 );
```


Operator '&'

```
  signal ASCII : STD_LOGIC_VECTOR( 7 downto 0 );
  signal Digit : STD_LOGIC_VECTOR( 3 downto 0 );
  (...)
  ASCII <= "0011" & Digit;   -- X"3" & Digit;
  ASCII <= X"3" & Digit ( 3 ) & Digit ( 2 ) &
                Digit ( 1 ) & Digit ( 0 );
  (...)
  -- These must be synchronous:
  --   ...shift right
  ASCII <= '0' & ASCII( 7 downto 1 );
  --   ...arithmetic shift right:
  ASCII <= ASCII( 7 ) & ASCII( 7 downto 1 );
  --   ...rotate left:
  ASCII <= ASCII( 6 downto 0 ) & ASCII( 7 );
```


Klauzula generic

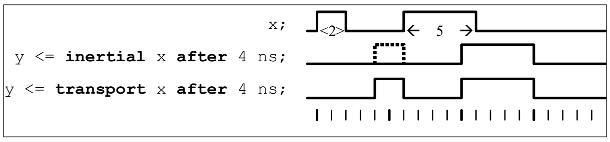
```
  entity identifier is
     generic ( parameter_declarations ); -- optional
     port ( port_declarations );         -- optional
     [ declarations ]                    -- optional
  begin                                  \__ optional
     [ statements ]                      /
  end entity identifier ;
```

Np.

```
  entity Buf is
     generic ( N : POSITIVE := 8;          -- data width
               Delay : DELAY_LENGTH := 2.5 ns );
     port ( Input  : in STD_LOGIC_VECTOR( N-1 downto 0 );
            OE     : in STD_LOGIC;
            Output : out STD_LOGIC_VECTOR( N-1 downto 0 ) );
  end entity Buf;
```


Przypisanie sygnału

```
  y <= x;
  y <= (x1 and x2) or x3;
```




```
  -- Domyślnie:
  y <= x;              <=>   y <= inertial x after 0 ns;
  y <= x after 4 ns;   <=>   y <= inertial x after 4 ns;

  -- Przypisanie wielokrotne:
  y <= '0', '1' after 100 ns, '0' after 120 ns;

  -- Np. w opisach sekwencyjnych (powtarzanych w pętlach):
  Clk <= '1', '0' after ClkPeriod / 2;
```


Przypisanie warunkowe `when...else`

```
 entity MUX_4 is
        port( A, B, C, D : in STD_LOGIC;
              Sel : in STD_LOGIC_VECTOR( 1 downto 0 );
              Y : out STD_LOGIC );
 end MUX_4;
 architecture Dataflow of MUX_4 is
 begin
     Y <= A when Sel = "00" else
          B when Sel = "01" else
          C when Sel = "10" else     ... lub ...
          D;                         D when Sel = "11" else
 end Dataflow;                       'X';
```

- **UWAGA!**

```
      Y <= '1' when A = '1' and B = '1' else
           '0' when A = '0' and B = '0';          ???
```

WARNING:Xst:737 - Found 1-bit latch for signal <Y>. Latches may be generated from incomplete case or if statements. We do not recommend the use of latches in FPGA/CPLD designs, as they may lead to timing problems.

Przypisanie selektywne `with ... select`

```
   (...)

   architecture Dataflow2 of MUX_4 is
   begin
       with Sel select
           Y <= A when "00",
                B when "01",
                C when "10",
                D when "11",
                'X' when others;
   end Dataflow2;
```

- połączenie opcji (jako OR) - znakiem '|':

```
      with Data( 2 downto 0 ) select
          PE <= '1' when "001" | "010" | "100" | "111",
                '0' when others;
```

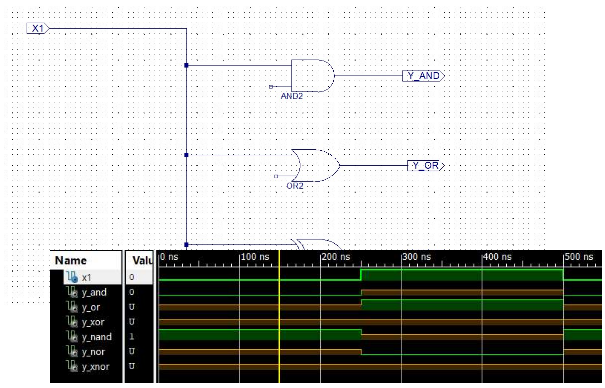
Instancje komponentów

```
entity XOR_2WE is
   generic( Tp : DELAY_LENGTH );
   port ( I1, I2 : in  STD_LOGIC;
          O : out  STD_LOGIC);
end XOR_2WE;
architecture A of XOR_2WE is
begin
  O <= I1 xor I2 after Tp;
end A;
```

```
entity AND_2WE is
   generic( Tp : DELAY_LENGTH );
   port (I1, I2 : in  STD_LOGIC;
         O : out  STD_LOGIC);
end AND_2WE;
architecture A of AND_2WE is
begin
   O <= I1 and I2 after Tp;
end A;
```

```
entity HalfAdder is
(...)
architecture Structural of HalfAdder is
    component XOR_2WE is
        generic( Tp : DELAY_LENGTH );
        port ( I1, I2 : in  STD_LOGIC; O : out  STD_LOGIC);
    end component;
    component AND_2WE is
        generic( Tp : DELAY_LENGTH );
        port ( I1, I2 : in  STD_LOGIC; O : out  STD_LOGIC);
    end component;
begin
  XOR_gate : XOR_2WE generic map( 5 ns ) port map( A, B, S );
  AND_gate : AND_2WE generic map( Tp => 3 ns )
                     port map( O=>C, I1=>A, I2=>B );
end architecture Structural;
```


Niepodłączone WE (open) a symulacja behawioralna:



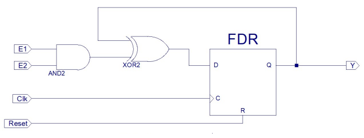
Przykład 1

ISE: plik z pobudzeniami testowymi

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;

ENTITY Top_sch_tbw IS
END Top_sch_tbw;

ARCHITECTURE behavioral OF Top_sch_tbw IS

   COMPONENT Top
   PORT( We2   :  IN STD_LOGIC;
         We1   :  IN STD_LOGIC;
         Wy   :   OUT   STD_LOGIC);
   END COMPONENT;

   SIGNAL We2  :  STD_LOGIC;
   SIGNAL We1  :  STD_LOGIC;
   SIGNAL Wy   :  STD_LOGIC;

BEGIN

   UUT: Top PORT MAP(
      We2 => We2,
      We1 => We1,
      Wy => Wy
   );

   WE1 <= '0', '1' after 100 ns, '0' after 300 ns;
   WE2 <= '0', '1' after 200 ns, '0' after 400 ns;


END;
```


Przykład 2

ISE: plik vhf na podstawie sch




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

entity SimpleSch is
   port ( Clk   : in    STD_LOGIC;
          E1    : in    STD_LOGIC;
          E2    : in    STD_LOGIC;
          Reset : in    STD_LOGIC;
          Y     : out   STD_LOGIC);
end SimpleSch;

architecture BEHAVIORAL of SimpleSch is

   signal XLXN_1  : STD_LOGIC;
   signal XLXN_2  : STD_LOGIC;
   signal Y_DUMMY : STD_LOGIC;

   component FDR
      port ( C : in    STD_LOGIC;
             D : in    STD_LOGIC;
             R : in    STD_LOGIC;
             Q : out   STD_LOGIC);
   end component;
   component XOR2
      port ( I0 : in    STD_LOGIC;
             I1 : in    STD_LOGIC;
             O  : out   STD_LOGIC);
   end component;
   component AND2
      port ( I0 : in    STD_LOGIC;
             I1 : in    STD_LOGIC;
             O  : out   STD_LOGIC);
   end component;

begin

   Y <= Y_DUMMY;

   XLXI_1 : FDR
      port map (C=>Clk,
                D=>XLXN_1,
                R=>Reset,
                Q=>Y_DUMMY);

   XLXI_2 : XOR2
      port map (I0=>XLXN_2,
                I1=>Y_DUMMY,
                O=>XLXN_1);

   XLXI_3 : AND2
      port map (I0=>E2,
                I1=>E1,
                O=>XLXN_2);

end BEHAVIORAL;
```


Instrukcja generacji

a) for … generate

Np.

```
entity FullAdder is
    port ( A, B : in  STD_LOGIC_VECTOR( 7 downto 0 );
           CI   : in  STD_LOGIC;
           S    : out  STD_LOGIC_VECTOR( 7 downto 0 );
           CO   : out  STD_LOGIC);
end FullAdder;
architecture Dataflow of FullAdder is
    signal Cint : STD_LOGIC_VECTOR( 8 downto 0 );
begin
lb: for i in 0 to 7 generate
        S(i) <= A(i) xor B(i) xor Cint(i);
        Cint(i + 1) <= ( A(i) and B(i) ) or
          ( A(i) and Cint(i) ) or ( B(i) and Cint(i) );
    end generate;
    Cint( 0 ) <= CI;
    CO <= Cint( 8 );
end Dataflow;
```


b) if … generate

```
label: if condition generate       -- label required
          block_declarative_items   \__ optional
       begin                        /
          concurrent_statements
       end generate label;
```

Instrukcja procesu

```
[label:] process [ ( sensitivity_list ) ] [ is ]
             [ declarative_items ]
         begin
             sequential_statements
         end process [ label ];
```

Np. było:

```
process( A, B ) is
begin
    if A /= B then
        S <= '1';
    else
        S <= '0';
    end if;
    if A = '1' and B = '1' then
        C <= '1';
    else
        C <= '0';
    end if;
end process;
```


Instrukcje sekwencyjne

(...)

Instrukcja `wait`:

```
wait for 10 ns;                 -- timeout
wait until clk = '1';           -- warunek logiczny
wait until A > B and ( S1 or S2 );
wait on sig1, sig2;             -- lista wrażliwości
```

`wait until ...` – czeka na zdarzenie, tj. zmianę sygnału (zawsze wstrzyma wykonanie procesu!); jeśli nie o to chodzi to trzeba np. dodać warunek:

```
if Busy /= '0' then
    wait until Busy = '0';
end if;
```



```
[ label: ] if  condition1  then
               statements
           elsif  condition2  then \_ optional
               statements          /
           ...
           else                    \_ optional
               statements          /
           end if [ label ] ;
```

```
architecture DF of MUX_4 is
begin
  Y <= A when Sel = "00" else
       B when Sel = "01" else
       C when Sel = "10" else
       D when Sel = "11" else
       'X';
end DF;
```

```
architecture DF_Eq of MUX_4 is
begin
  process ( Sel, A, B, C, D )
  begin
    if Sel = "00" then
        Y <= A;
    elsif Sel = "01" then
        Y <= B;
    elsif Sel = "10" then
        Y <= C;
    elsif Sel = "11" then
        Y <= D;
    else
        Y <= 'X';
    end if;
  end process;
end DF_Eq;
```



```
[ label: ] case  expression  is
              when choice1 =>
                 statements
              when choice2 =>    \_ opt.
                 statements      /
              ...
              when others =>     \_ opt. if all choices
                 statements      /  covered
           end case [ label ] ;
```

```
architecture DF2 of MUX is
begin
  with Sel select
    Y <= A when "00",
         B when "01",
         C when "10",
         D when "11",
        'X'when others;
end DF2;
```

```
architecture DF2_Eq of MUX_4 is
begin
  process ( Sel, A, B, C, D )
  begin
    case Sel is
      when "00" =>   Y <= A;
      when "01" =>   Y <= B;
      when "10" =>   Y <= C;
      when "11" =>   Y <= D;
      when others => Y <= 'X';
    end case;
  end process;
end DF2_Eq;
```



```
[ label: ] loop
               statements   -- use exit to abort
           end loop [ label ] ;

[ label: ] for variable in range loop
               statements
           end loop [ label ] ;

[ label: ] while  condition  loop
               statements
           end loop [ label ] ;
```

```
next;
next outer_loop;  -- label of loop instr.
next when A > B;
next this_loop when C = D or A > B;
```

```
exit;
exit outer_loop;
exit when A > B;
exit this_loop when C = D or A > B;
```


Inne instrukcje

Instrukcja **`null`**

Np.:

```
case (...) is
  (...)
  when others => null;
end case;
```

Instrukcja **`report`**

```
report "Message" [ severity SevLevel ];
type SEVERITY_LEVEL is (NOTE, WARNING, ERROR, FAILURE);
```

Np.

```
report "Parity bit error" severity WARNING;
```

Instrukcja **`assert`**

```
assert boolean_cond [report "Msg"] [severity SevLevel];
```

⇒ *Domyślnie:* `SevLevel = ERROR`

Instrukcja **`return`**

Konieczna w funkcjach, opcjonalna w procedurach.

Wywołanie procedury

Sygnały synchroniczne

```
entity DFF is
  port ( D   : in  STD_LOGIC;
         Clk : in  STD_LOGIC;
         Q   : out STD_LOGIC );
end DFF;

architecture RTL of DFF is
begin
  process ( Clk )
  begin
    if Clk'Event and Clk = '1' then
      Q <= D;
    end if;
  end process;
end architecture;
```

```
entity TFF is
  port ( T   : in  STD_LOGIC;
         Clk : in  STD_LOGIC;
         Q   : out STD_LOGIC );
end TFF;

architecture RTL of TFF is
  signal Q_int : STD_LOGIC := '0';
begin
  Q <= Q_int;
  process ( Clk )
  begin
    if Clk'Event and Clk = '1' then
      if T = '1' then
        Q_int <= not Q_int;
      end if;
    end if;
  end process;
end architecture;
```

- Pakiet STD_LOGIC_1164:

```
function rising_edge  (signal s : STD_ULOGIC) return BOOLEAN;
function falling_edge (signal s : STD_ULOGIC) return BOOLEAN;

if Clk'Event and Clk = '1' then...  =>  if rising_edge(Clk) then...
```


Clock Enable (~FDE):

```
entity DFF_E is
  port( D   : in  STD_LOGIC;
        CE  : in  STD_LOGIC;
        Clk : in  STD_LOGIC;
        Q   : out STD_LOGIC );
end DFF_E;

architecture RTL of DFF_E is
begin
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      if CE = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;
```


Asynchronous Clear + Enable: (FDCE)

```
entity DFF_CE is
  port( D   : in  STD_LOGIC;
        Clr : in  STD_LOGIC;
        CE  : in  STD_LOGIC;
        Clk : in  STD_LOGIC;
        Q   : out STD_LOGIC );
end DFF_CE;

architecture RTL of DFF_CE is
begin
  process ( Clk, Clr )
  begin
    if Clr = '1' then
      Q <= '0';
    elsif rising_edge(Clk) then
      if CE = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;
```

Synchronous Reset + Enable: (FDRE)

```
entity DFF_RE is
  port( D   : in  STD_LOGIC;
        Rst : in  STD_LOGIC;
        CE  : in  STD_LOGIC;
        Clk : in  STD_LOGIC;
        Q   : out STD_LOGIC );
end DFF_RE;

architecture RTL of DFF_RE is
begin
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      if Rst = '1' then
        Q <= '0';
      elsif CE = '1' then
        Q <= D;
      end if;
    end if;
  end process;
end architecture;
```


ERROR:Xst:827: Signal Q cannot be synthesized, bad synchronous description (…)

```
process ( Clk, Clr )
begin
  if Clr = '1' then
    Q <= '0';
  elsif rising_edge(Clk) then
    if CE = '1' then
      Q <= D;
    end if;
  end if;
end process;
```

⇓ FDCE OK

```
process ( Clk, Clr )
begin
  if Clr = '1' then
    Q <= '0';
  end if;
  if rising_edge(Clk) then
    if CE = '1' then
      Q <= D;
    end if;
  end if;
end process;
```

⇓ ERROR: bad synchronous descr.! (FDEC?)

```
process ( Clk, Clr )
begin
  if rising_edge(Clk) then
    if CE = '1' then
      Q <= D;
    end if;
  elsif Clr = '1' then
    Q <= '0';
  end if;
end process;
```

⇓ ERROR: bad synchronous descr.! (FDEC?)

⇒ Symulacja behawioralna nie widzi problemu

```
process ( Clk )
begin
  if rising_edge( Clk ) then
    if Rst = '1' then
      Q <= '0';
    elsif CE = '1' then
      Q <= D;
    end if;
  end if;
end process;
```

⇓ FDRE OK

```
process ( Clk )
begin
  if rising_edge( Clk ) then
    if Rst = '1' then
      Q <= '0';
    end if;
    if CE = '1' then
      Q <= D;
    end if;
  end if;
end process;
```

⇓ Synthesizable to ~FDER, *not* recommended

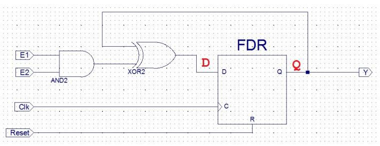
```
process ( Clk )
begin
  if rising_edge( Clk ) then
    if CE = '1' then
      Q <= D;
    end if;
    if Rst = '1' then
      Q <= '0';
    end if;
  end if;
end process;
```

⇓ Synthesizable to FDRE, *bad* style!

Rejestr przesuwny:

```
entity SReg8b is
    port ( Din : in  STD_LOGIC;
           Clk : in  STD_LOGIC;
           Q   : out STD_LOGIC_VECTOR( 7 downto 0 ) );
end SReg8b;

architecture RTL of SReg8b is
  signal iQ : STD_LOGIC_VECTOR( 7 downto 0 );
begin
  Q <= iQ;
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      iQ( 7 downto 0 ) <= iQ( 6 downto 0 ) & Din;
    end if;
  end process;
end architecture;
```


Np.




```
architecture RTL of SimpleSch is

  signal D, Q : STD_LOGIC;

begin

  D <= ( E1 and E2 ) xor Q;

  process( Clk )
  begin
    if rising_edge( Clk ) then
      if Reset = '1' then
        Q <= '0';
      else
        Q <= D;
      end if;
    end if;
  end process;

  Y <= Q;

end RTL;
```

Licznik binarny z asynchronicznym kasowaniem:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity counters_1 is
   port(C, CLR : in STD_LOGIC;
        Q : out STD_LOGIC_VECTOR(3 downto 0));
end counters_1;
architecture archi of counters_1 is
   signal tmp: UNSIGNED(3 downto 0);
begin
   process ( C, CLR )
   begin
      if CLR = '1' then
         tmp <= "0000";
      elsif rising_edge( C ) then
         tmp <= tmp + 1;
      end if;
   end process;
   Q <= STD_LOGIC_VECTOR( tmp );
end archi;
```

!

Licznik modulo (z zerowaniem i sygnałem zezwalającym):

```
(...)
process ( Clk )
begin
  if rising_edge( Clk ) then
    if Rst = '1' then
      tmp <= "0000";
    elsif CE = '1' then
      if tmp = "1001" then
        tmp <= "0000";
      else
        tmp <= tmp + 1;
      end if;
    end if;
  end if;
end process;
(...)
```


Licznik ładowalny (asynchronicznie):

```
process ( Clk, ALOAD, D )
begin
  if ALOAD = '1' then
    tmp <= D;
  elsif rising_edge( Clk ) then
    tmp <= tmp + 1;
  end if;
end process;
```

Licznik rewersyjny:

```
process ( Clk, CLR )
begin
  if CLR = '1' then
    tmp <= "0000";
  elsif rising_edge( Clk ) then
    if UP_DOWN = '1' then
      tmp <= tmp + 1;
    else
      tmp <= tmp - 1;
    end if;
  end if;
end process;
```


Zatrzask:

```
entity latches_1 is
    port(G, D : in std_logic;
         Q : out std_logic);
end latches_1;

architecture archi of latches_1 is
begin
    process (G, D)
    begin
        if (G='1') then
            Q <= D;
        end if;
    end process;
end archi;
```

```
Q <= D when G = '1';
```

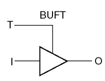

Zatrzask z asynchronicznym kasowaniem:

```
architecture archi of latches_2 is
begin
    process (CLR, D, G)
    begin
        if (CLR='1') then
            Q <= '0';
        elsif (G='1') then
            Q <= D;
        end if;
    end process;
end archi;
```

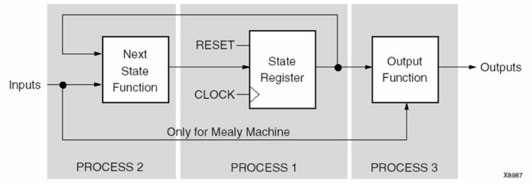
```
Q <= '0' when CLR = '1' else
     D   when   G = '1';
```

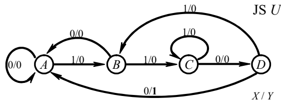

Bufor trójstanowy:

```
entity three_st_2 is
    port(T : in  std_logic;
         I : in  std_logic;
         O : out std_logic);
end three_st_2;

architecture archi of three_st_2 is
begin
    O <= I when (T='0') else 'Z';
end archi;
```



Maszyny stanów






```
entity SequenceDet is
   port ( Clk : in  STD_LOGIC;
          Reset : in  STD_LOGIC;
          x : in  STD_LOGIC;
          y : out  STD_LOGIC);
end SequenceDet;

architecture Behavioral of SequenceDet is

   type state_type is ( A, B, C, D );
   signal state, next_state : state_type;

begin

   process1 : process( Clk )
   begin
      if rising_edge( Clk ) then
         if Reset = '1' then
            state <= A;
         else
            state <= next_state;
         end if;
      end if;
   end process process1;

   process2 : process( state, x )
   begin
      next_state <= state;    -- by default

      case state is

         when A =>
            if x = '1' then
               next_state <= B;
            end if;

         when B =>
            if x = '1' then
               next_state <= C;
            else
               next_state <= A;
            end if;

         when C =>
            if x = '0' then
               next_state <= D;
            end if;

         when D =>
            if x = '0' then
               next_state <= A;
            else
               next_state <= B;
            end if;

      end case;
   end process process2;

   -- In place of process3:
   y <= '1' when state = D and x = '0'
        else '0';

end Behavioral;
```


FSM in XST (Xilinx Synthesis Tool)

Process Properties - HDL Options

Category | Switch Name | Property Name | Value

Synthesis O
HDL Optio
Xilinx Spec

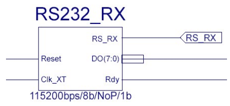
```
=========================================================================
*                          HDL Synthesis                                *
=========================================================================
(...)
Synthesizing Unit <SequenceDet>.
    Related source file is "C:/XilinxPrj/FSM_VHDL/SequenceDet.vhd".
    Found finite state machine <FSM_0> for signal <state>.
    -----------------------------------------------------------------------
    | States              | 4                                             |
    | Transitions         | 8                                             |
```

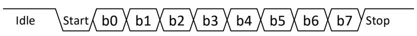
```
=========================================================================
*                       Advanced HDL Synthesis                          *
=========================================================================
(...)
Analyzing FSM <FSM_0> for best encoding.
Optimizing FSM <state/FSM> on signal <state[1:2]> with gray encoding.
-------------------
 State | Encoding
-------------------
 a     | 00
 b     | 01
 c     | 11
 d     | 10
-------------------
(...)
```

Unit
(...)

Opisy dla symulacji

Przykład 1: Nadawanie bajtu do odbiornika RS-232 (UUT: `port in RS_RX`)



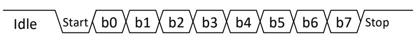



```
entity Tbw_RS232_RX is
end Tbw_RS232_RX;

architecture Simulation of Tbw_RS232_RX is

   component RS_RCEIVER
      port( ...
            RS_RX : in STD_LOGIC;
            ... );
   end component;

   signal RS_RX : STD_LOGIC := '1';
   constant Clk_Period : DELAY_LENGTH := 1 us / 50; -- e.g. 50MHz
   ( ... )
begin

  -- Instantiate the Unit Under Test (UUT)
   uut: RS_RCEIVER port map (
          ...
          RS_RX => RS_RX,
          ... );

   Reset <= '0';

   Clk_XT <= not Clk_XT after Clk_Period / 2;
   (...)
```

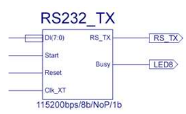

Opisy dla symulacji

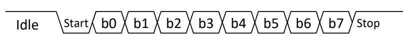
Przykład 1: Nadawanie bajtu do odbiornika RS-232 (UUT: `port in RS_RX`)

```
(...)
-- Stimula for the RS_RX input:
process
  procedure RS_Transm( Byte : STD_LOGIC_VECTOR( 7 downto 0 ); Bod : POSITIVE ) is
    variable Frame : STD_LOGIC_VECTOR( 9 downto 0 ) := '1' & Byte & '0';
  begin
    for i in 0 to 9 loop
      RS_RX <= Frame( i );
      wait for 1 sec / Bod;
    end loop;
  end procedure;
begin
  RS_RX <= '1';
  wait for 1.5 us;
  RS_Transm( X"F0", 115200 );
  wait for 2 * 1 sec / 115200;    -- the line is idle for e.g. two bits
  RS_Transm( X"71", 115200 );
  wait; -- will wait forever
end process;
(...)
```



Przykład 2: Odbiór bajtu z nadajnika RS-232 (UUT: `port out RS_TX`)

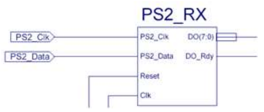


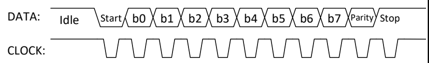



```
(...)
process
   constant Bod : POSITIVE := 115200;
   constant BitTime : DELAY_LENGTH := 1 sec / Bod;
   variable Byte : STD_LOGIC_VECTOR( 7 downto 0 );

begin
   loop
      wait until falling_edge( RS_TX );
      wait for BitTime / 2;
      next when RS_TX /= '0';         -- false start bit
      report "RS232 TX: start BIT detected";
      for i in 0 to 7 loop
         wait for BitTime;
         Byte( i ) := RS_TX;
      end loop;
      report "RS232 TX: received " &
             INTEGER'Image( to_integer( UNSIGNED( Byte ) ) ); -- in decimal :(
      wait for BitTime;
      assert RS_TX = '1' report "RS232 TX WARNING: invalid stop bit"
                                                            severity WARNING;
   end loop;
end process;
(...)
```

Przykład 3: Nadawanie bajtów do portu PS/2 (UUT: `PS2_Data, PS2_Clk`)

```
(...)
process
   procedure TransmPS2( Byte : STD_LOGIC_VECTOR( 7 downto 0 ) ) is
      variable Frame : STD_LOGIC_VECTOR( 10 downto 0 ) := "11" & Byte & '0';
   begin
      -- Parity calculation
      for i in 0 to 7 loop
         Frame( 9 ) := Frame( 9 ) xor Byte( i );
      end loop;

      -- Transmission of the frame; Freq.Clk = 10kHz (Tclk = 100 us)
      for i in 0 to 10 loop
         PS2_Data <= Frame( i );
         wait for 5 us;
         PS2_Clk <= '0', '1' after 50 us;
         wait for 95 us; -- 100us per loop
      end loop;
   end procedure;
begin
   PS2_Data <= '1';
   PS2_Clk  <= '1';
   wait for 15 us;

   TransmPS2( X"F0" );
   wait for 200 us;

   TransmPS2( X"81" );
   wait; -- will wait forever
end process;
(...)
```





Pakiet STANDARD

```
type INTEGER is range --usually typical INTEGER-- ;
subtype NATURAL is INTEGER range 0 to INTEGER'HIGH;
subtype POSITIVE is INTEGER range 1 to INTEGER'HIGH;
type REAL is range --usually double precision f.p.-- ;
type BOOLEAN is (FALSE, TRUE);
type CHARACTER is ( --256 characters-- );
type STRING is array (POSITIVE range <>) of CHARACTER;
type BIT is ('0', '1');
type TIME is range --implementation defined-- ;
  units
     fs;          -- femtosecond
     ps  = 1000 fs; -- picosecond
     ns  = 1000 ps; -- nanosecond
     us  = 1000 ns; -- microsecond
     ms  = 1000 us; -- millisecond
     sec = 1000 ms; -- second
     min = 60  sec; -- minute
     hr  = 60  min; -- hour
  end units;
subtype DELAY_LENGTH is TIME range 0 fs to TIME'HIGH;
```


Operatory

```
**     exponentiation,      numeric ** integer,    result numeric
abs    absolute value,      abs numeric,           result numeric
not    complement,          not logic or boolean,  result same

*      multiplication,      numeric * numeric,     result numeric
/      division,            numeric / numeric,     result numeric
mod    modulo,              integer mod integer,   result integer
rem    remainder,           integer rem integer,   result integer

+      unary plus,          + numeric,             result numeric
-      unary minus,         - numeric,             result numeric

+      addition,            numeric + numeric,     result numeric
-      subtraction,         numeric - numeric,     result numeric
&      concatenation,       array or element,      result array
```

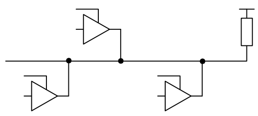


```
sll    shift left logical,     log. array sll integer,      result same
srl    shift right log.,       log. array srl integer,      result same
sla    shift left arith.,      log. array sla integer,      result same
sra    shift right arith.,     log. array sra integer,      result same
rol    rotate left,            log. array rol integer,      result same
ror    rotate right,           log. array ror integer,      result same

=      equality,                                    result boolean
/=     inequality,                                  result boolean
<      less than,                                   result boolean
<=     less than or equal,                          result boolean
>      greater than,                                result boolean
>=     greater than or equal,                       result boolean

and    logical and,            log. array or boolean,       result same
or     logical or,             log. array or boolean,       result same
nand   logical nand,           log. array or boolean,       result same
nor    logical nor,            log. array or boolean,       result same
xor    logical xor,            log. array or boolean,       result same
xnor   logical xnor,           log. array or boolean,       result same
```


Pakiet STD_LOGIC_1164

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

type STD_ULOGIC is ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     'O',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

U !



```
type STD_ULOGIC_VECTOR is array ( NATURAL range <> ) of
STD_ULOGIC;
```



```
function resolved ( s : STD_ULOGIC_VECTOR ) return STD_ULOGIC;

constant resolution_table : stdlogic_table := (
--      ---------------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -         |   |
--      ---------------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
        ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
        ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
        ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
        ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
        ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
    );
(...)

-------------------------------------------------------------------
-- *** industry standard logic type ***
-------------------------------------------------------------------
subtype STD_LOGIC is resolved STD_ULOGIC;
type STD_LOGIC_VECTOR is array ( NATURAL range <>) of STD_LOGIC;
```

```
-- truth table for "and" function
constant and_table : stdlogic_table := (
--      ----------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -        |   |
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
);
-- truth table for "or" function
constant or_table : stdlogic_table := (
--      ----------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -        |   |
        ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' ),  -- | U |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | X |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 0 |
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  -- | 1 |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | Z |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | W |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | L |
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  -- | H |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' )   -- | - |
);
```



```
-- truth table for "xor" function
constant xor_table : stdlogic_table := (
--      ----------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -        |   |
--      ----------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ),  -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | X |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 0 |
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ),  -- | 1 |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | W |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | L |
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ),  -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )   -- | - |
    );

-- truth table for "not" function
constant not_table: stdlogic_1d :=
--  -------------------------------------------------
--  |   U    X    0    1    Z    W    L    H    -   |
--  -------------------------------------------------
     ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' );
```


Pakiet NUMERIC_STD (IEEE Std. 1076.3)

- Pre 1076.3: Synopsys libraries `std_logic_unsigned`/ `std_logic_signed`
  – now obsolete, but used to be *defacto* industry standard

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
. . .
// The two NUMERIC_STD types:
signal A_u : UNSIGNED(3 downto 0); -- "s.l.v." interpreted as unsigned
signal B_s : SIGNED(3 downto 0);   -- "s.l.v." interpreted as signed
. . .
A_u <= "1111";   -- 15 decimal
B_s <= "1111";   -- -1 decimal
. . .
A_u <= A_u + 1;  -- overloaded '+' operator; also '-', '<'...
```



```
-- Obligatory explicit type conversions:
std_l_vec  <= STD_LOGIC_VECTOR( unsigned | signed );
unsigned   <= UNSIGNED( std_l_vec );
signed     <=   SIGNED( std_l_vec );

-- Conversion from/to integer types:
unsigned   <= TO_UNSIGNED( 128, 8 );  -- value, vector_size
signed     <= TO_SIGNED( -7, 8 );
std_l_vec  <= STD_LOGIC_VECTOR( TO_UNSIGNED( 1076, 32 ) );
int        <= TO_INTEGER( unsigned | signed );
int        <= TO_INTEGER( (UN)SIGNED( std_l_vec ) );

-- Carry out in additions
Result_8b <= Arg1_8b + Arg2_8b;      -- carry out is lost
Result_9b <= ('0' & Arg1_8b) + ('0' & Arg2_8b);
Carry_Out <= Result_9b( 8 );         -- OK
-- Note: Result_9b <= Arg1_8b + Arg2_8b  will not work!
```


Atrybuty

`NamedEntity'AttrName[(ParameterList)]`

Atrybuty wektorów / typów wektorowych

`A'LEFT` is the leftmost idx of array A or constrained array type.

`A'RIGHT` is the rightmost idx of array A or constrained array type.

```
type bit_array is array (-15 to 15) of bit;
variable L: INTEGER := bit_array'Left; -- L has a value of -15
. . .
for i in some_vector'left to some_vector'left + 4 loop
   . . . browse the first 5 elements . . .
```

`A'HIGH` is the highest idx of array A or constrained array type.

`A'LOW` is the lowest idx of array A or constrained array type.

```
variable H: INTEGER := bit_array'High; -- H has a value of 15
```

`A'LENGTH` is the integer value of the number of elements in array A.

```
variable LEN: INTEGER := bit_array'Length -- LEN has a value of 31
```

`A'RANGE` is the range `A'LEFT to A'RIGHT` or `A'LEFT downto A'RIGHT`.

```
for i in some_vector'Range loop
   . . . browse all elements . . .
```


`A'REVERSE_RANGE` is the range of A with `to` and `downto` reversed.

`A'ASCENDING` is boolean `true` if range of A defined with `to`.

Atrybuty typów

`T'LEFT` is the leftmost value of type T (largest if `downto`)

`T'RIGHT` is the rightmost value of type T (smallest if `downto`)

```
variable V: INTEGER := INTEGER'Left; -- value of -2,147,483,648
```

`T'HIGH` is the highest value of type T.

`T'LOW` is the lowest value of type T.

```
variable V: REAL := REAL'High; --  1.701411e+308
```

`T'POS(X)` is (zero-based) integer position of X in the discrete type T.

```
type state_type is (Init, Hold, Strobe, Read, Idle);
variable P: integer := state_type'Pos( Hold ); -- value of 1
. . .
-- State number to the out port (synthesizable by XST!)
DbgOutput <= to_unsigned( state_type'Pos( state ), 3 );
```

`T'VAL(X)` is the value of discrete type T at integer position X.

```
variable V: state_type := state_type'Val( 2 ); -- value of Strobe
```

```
T'PRED(X)        is the value of discrete type T that is the predecessor of X.
T'SUCC(X)        is the value of discrete type T that is the successor of X.
  variable V: state_type := state_type'Succ(Init); -- value of Hold

T'LEFTOF(X)      is the value of discrete type T that is left of X.
T'RIGHTOF(X)     is the value of discrete type T that is right of X.
           -- Different from 'Pred / 'Succ only in a subtype
           -- which changed order of the base type

T'ASCENDING      is boolean true if range of T defined with to.

T'IMAGE(X)       is a string representation of X that is of type T.
  report INTEGER'Image( to_integer( unsigned( slv ) ) );
T'VALUE(X)       is a value of type T converted from the string X.
  constant Pi: REAL := REAL'Value( "3.141" );

T'BASE           is the base type of type T
  POSITIVE'Base'Left  -- INTEGER'Left or -2,147,483,648
```


Np. 1) Konwersja STRING na literał bitowy


```
constant slvHello : STD_LOGIC_VECTOR(39 downto 0)
                              := X"48_65_6C_6C_6F";  -- "Hello"
```

Albo:

```
function string_to_slv( S : STRING ) return STD_LOGIC_VECTOR is
   constant strLength : NATURAL := S'Length;
   constant Snorm : STRING( 1 to strLength ) := S;
   variable Result: STD_LOGIC_VECTOR(strLength * 8 - 1 downto 0);
begin
   for i in 0 to strLength - 1 loop
      Result(i * 8 + 7 downto i * 8) :=
         STD_LOGIC_VECTOR( to_unsigned(
                       CHARACTER'Pos( Snorm(strLength - i) ), 8) );
   end loop;
   return Result;
end function;
(...)
constant slvHello : STD_LOGIC_VECTOR :=
                                string_to_slv( "Hello" );
```


Np. 2) Konwersja slv Hex na slv ASCII

X"0123A" → X"30_31_32_33_41"

```
function slvHex_to_slvASCII( slvHex : STD_LOGIC_VECTOR )
                                        return STD_LOGIC_VECTOR is
   constant NoDigits : INTEGER := slvHex'Length / 4;
   variable Hex      : STD_LOGIC_VECTOR( 3 downto 0);
   variable ASCII    : STD_LOGIC_VECTOR( 7 downto 0);
   variable slvASCII : STD_LOGIC_VECTOR(NoDigits * 8 - 1 downto 0);
begin
   for i in 0 to NoDigits - 1 loop
      Hex := slvHex(i * 4 + 3 downto i * 4);
      case Hex is
         when X"A"   =>   ASCII := X"41";
               -- STD_LOGIC_VECTOR( to_unsigned( Character'POS('A') ) );
         when X"B"   =>   ASCII := X"42";
         when X"C"   =>   ASCII := X"43";
         when X"D"   =>   ASCII := X"44";
         when X"E"   =>   ASCII := X"45";
         when X"F"   =>   ASCII := X"46";
         when others =>   ASCII := X"3" & Hex;  -- '0'...'9'
      end case;
      slvASCII(i * 8 + 7 downto i * 8) := ASCII;
   end loop;
   return slvASCII;
end function;
```


Atrybuty obiektów

`E'SIMPLE_NAME`	is a string containing the name of entity E.
`E'INSTANCE_NAME`	is a string containing the design hierarchy including E.
`E'PATH_NAME`	is a string containing the design hierarchy of E to design root.

Atrybuty sygnałów (na potem)

`S'EVENT`	true if signal S has had an event this simulation cycle.
`S'STABLE`	signal: true if no event is occurring on signal S.
`S'STABLE(t)`	signal: true if no even has occurred on signal S for t units of time.
`S'ACTIVE`	true if signal S is active during current simulation cycle.
`S'QUIET`	signal: true if S is quiet. (no event this simulation cycle)
`S'QUIET(t)`	signal: true if S has been quiet for t units of time.
`S'TRANSACTION`	bit signal, the inverse of previous value each cycle S is active.
`S'LAST_EVENT`	the time since the last event on signal S.
`S'LAST_ACTIVE`	the time since signal S was last active.
`S'LAST_VALUE`	the previous value of signal S.
`S'DELAYED(t)`	signal: the value of S at time now - t .
`S'DRIVING`	true if the process is driving S.
`S'DRIVING_VALUE`	is the current driving value of signal S in the process.

Atrybuty definiowane przez użytkownika

```
-- Declaration:
attribute Name : AttributeType;
-- Application:
attribute Name of ObjectName : ObjectClass is Value;
```

ObjectClass = **signal** | **type** | **function** | **architecture** | ...

Np.

Rozpoznawany przez XST atrybut ustalający kodowanie dowolnego typu wyliczeniowego:

```
type statetype is (ST0, ST1, ST2, ST3);
attribute enum_encoding : STRING;
attribute enum_encoding of statetype : type is "001 010 100 111";
```


Cykle symulacji


```
entity Ex1 is
  port(
    A, B, C : in  STD_LOGIC;
    Y     : out STD_LOGIC );
end entity;
```

```
architecture DFlow of Ex1 is
  signal S : STD_LOGIC;
begin
  S <= A or B;
  Y <= C xor S;
end architecture;
```

Testbench:

```
entity tbw_Ex1 is
end tbw_Ex1;

architecture behavior of tbw_Ex1 is

   component Ex1
   port(
        A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        C : in  STD_LOGIC;
        Y : out  STD_LOGIC
      );
   end component;

  signal A : STD_LOGIC := '0';
  signal B : STD_LOGIC := '0';
  signal C : STD_LOGIC := '0';

    signal Y : STD_LOGIC;

    begin

       uut: Ex1 port map (
              A => A,
              B => B,
              C => C,
              Y => Y
            );

       A <= '1' after 10 ns;

    end;
```

Cykle symulacji


```
 (...)
    signal A : STD_LOGIC := '0';
    signal B : STD_LOGIC := '0';
    signal C : STD_LOGIC := '0';
    signal Y : STD_LOGIC;
 (...)
```

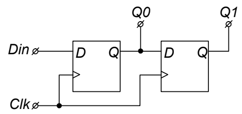
TBW:
```
A <= '1' after 10 ns;
```

UUT:
```
S <= A or B;
Y <= C xor S;
```

```
               A B C S Y
Init:          0 0 0 U U     Wykonanie przypisań:
                             A <= (...): transakcja '1'/10 ns → POW_A
                             S <= (...): transakcja '0'/ 0 ns → POW_S
                             Y <= (...): transakcja 'U'/ 0 ns → POW_Y
Cykle:
 0ns                         (a) S ← 0 (Event), Y ← U (Active)
               0 0 0 0 U     (b) Y <= (...): transakcja '0'/ 0 ns → POW_Y

 0ns + Δ                     (a) Y ← 0 (Event)
               0 0 0 0 0     (b) null
10ns                         (a) A ← 1 (Event)
               1 0 0 0 0     (b) S <= (...): transakcja '1'/10 ns → POW_S

10ns + Δ                     (a) S ← 1 (Event)
               1 0 0 1 0     (b) Y <= (...): transakcja '1'/10 ns → POW_Y

10ns + 2Δ                    (a) Y ← 1 (Event)
               1 0 0 1 1     (b) null
                   (KONIEC)
```



```
process ( Clk )
begin
  if rising_edge( Clk ) then
    Q0 <= Din;
    Q1 <= Q0;
  end if;
end process;
```




```
  Cykl    Clk Din  Q0  Q1  Opis:
  (...)   '0' '1' '0' '0'
  10ns                     (a) Clk ← 1
          '1' '1' '0' '0'  (b) Clk'Event, wykonanie procesu:
                               trans. '1'/10ns → POW_Q0,
                               trans. '0'/10ns → POW_Q1

10ns + Δ                   (a) Q0 ← 1, Q1 ← 0
          '1' '1' '1' '0'  (b) Q0'event, Q1 tylko active
                                   (koniec)

 process( Clk, Din, Q0, Q1 )...?
 (...)
10ns + Δ                   (a) Q0 ← 1, Q1 ← 0
          '1' '1' '1' '0'  (b) Q0'Event, wykonanie procesu:
                               warunek if niespełniony
                                  (koniec)
```


Atrybuty sygnałów

`S'EVENT`	true if signal S has had an event this simulation cycle.
`S'STABLE`	signal: true if no event is occurring on signal S.
`S'STABLE(t)`	signal: true if no even has occurred on signal S for t units of time.
`S'ACTIVE`	true if signal S is active during current simulation cycle.
`S'QUIET`	signal: true if S is quiet. (no transaction this simulation cycle)
`S'QUIET(t)`	signal: true if S has been quiet for t units of time.
`S'TRANSACTION`	`BIT` signal, the inverse of previous value each cycle S is active.
`S'LAST_EVENT`	the time since the last event on signal S.
`S'LAST_ACTIVE`	the time since signal S was last active.
`S'LAST_VALUE`	the previous value of signal S.
`S'DELAYED(t)`	signal: the value of S at time now - t .

`t = 0` ⇒ time is delta

Within a process driving `S`:

`S'DRIVING`	true if the process is driving S.
`S'DRIVING_VALUE`	current driving value of signal S in the process.

Przykłady

1) Funkcja `rising_edge()`:

```
function rising_edge  (signal s : STD_ULOGIC) return BOOLEAN is
begin
   return (s'EVENT and (To_X01(s) = '1') and
                             (To_X01(s'LAST_VALUE) = '0'));
end;
```

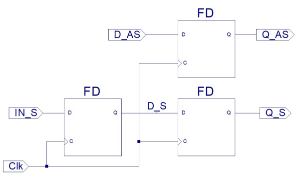
gdzie

```
  CONSTANT cvt_to_x01 : logic_x01_table :=
      -- 'U'   'X'   '0'   '1'   'Z'   'W'   'L'   'H'   '-'
     (  'X',  'X',  '0',  '1',  'X',  'X',  '0',  '1',  'X'   );
```

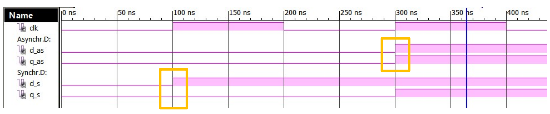
2) Wykrycie trwającego co najmniej 100 µs stanu zerowego na linii `PS2_Clk`:

```
wait until PS2_Clk'DELAYED'LAST_EVENT > 100 us  and
                                     PS2_Clk'LAST_VALUE = '0';
```


Problem: zmiana sygnału WE w momencie jego próbkowania



TBW:
```
(...)
   Clk  <= not Clk after 100 ns;
   IN_S <= '1';
   D_AS <= '1' after 300 ns;
(...)
```

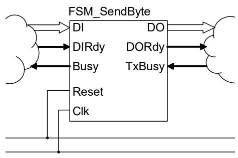



```
 Cykl   Clk   D_AS   Q_AS
(...)   '0'   '0'    '0'
300ns                       (a) Clk ← 1, D_AS ← 1
        '1'   '1'    '0'    (b) Clk'Event, wykonanie procesów:
                                trans. 1/300ns → POW_Q_AS,
                                trans. 1/300ns → POW_Q_S

                           ( . . . )
```

⇒ Nie ma możliwości zdefiniowania przypisania `D_AS` w momencie „300 + Δ"

Projekt

Moduł transkodujący otrzymany bajt na dwa znaki ASCII




```
entity FSM_SendByte is
    port ( Clk : in  STD_LOGIC;
           Reset : in  STD_LOGIC;
           DI : in  STD_LOGIC_VECTOR (7 downto 0);
           DIRdy : in  STD_LOGIC;
           TxBusy : in  STD_LOGIC;
           DO : out  STD_LOGIC_VECTOR (7 downto 0);
           DORdy : out  STD_LOGIC;
           Busy : out  STD_LOGIC  );
end FSM_SendByte;

architecture RTL of FSM_SendByte is
  type state_type is ( sReset, sReady, sWaitH, sSendH,
                                         sWaitL, sSendL );
  signal State, NextState : state_type;
  signal regDI : STD_LOGIC_VECTOR (7 downto 0);
  signal HalfByte : STD_LOGIC_VECTOR (3 downto 0);
```

JS UCiSW 1

```
(...)
begin

  -- FSM: State register
  process ( Clk )
  begin
    if rising_edge( Clk ) then
      if Reset = '1' then
        State <= sReset;
      else
        State <= NextState;
      end if;
    end if;
  end process;

  -- FSM: Next state decoding
  process ( State, DIRdy, TxBusy )
  begin

    NextState <= State; -- default

    case State is
      when sReset =>
        NextState <= sReady;

      when sReady =>
        if DIRdy = '1' then
          NextState <= sWaitH;
        end if;

      when sWaitH =>
        if TxBusy = '0' then
          NextState <= sSendH;
        end if;

      when sSendH =>
        NextState <= sWaitL;

      when sWaitL =>
        if TxBusy = '0' then
          NextState <= sSendL;
        end if;

      when sSendL =>
        NextState <= sReady;

    end case;
  end process;
(...)
```

67

JS UCiSW 1

```
(...)
  -- Outputs
  DORdy <= '1' when State = sSendH or State = sSendL
          else '0';

  Busy  <= '1' when State /= sReady
          else '0';

  -- Other, i.e.: input register (with CE)...
  regDI <= DI when rising_edge( Clk ) and State = sReady;
                                          -- SKRÓT PROCESU!

  -- ...halfByte selection...
  HalfByte <= regDI( 7 downto 4 ) when State = sSendH or
                                       State = sWaitL
              else regDI( 3 downto 0 );

(...)
```

68

JS UCiSW 1

```
(...)
 -- ...transcoding X"0" - X"F" to ASCII '0'-'F'
  with HalfByte select
    DO <= X"30" when "0000",
          X"31" when "0001",
          X"32" when "0010",
          X"33" when "0011",
          X"34" when "0100",
          X"35" when "0101",
          X"36" when "0110",
          X"37" when "0111",
          X"38" when "1000",
          X"39" when "1001",
          X"41" when "1010",
          X"42" when "1011",
          X"43" when "1100",
          X"44" when "1101",
          X"45" when "1110",
          X"46" when others;
end RTL;
```

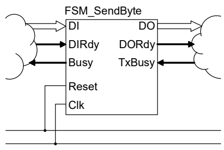
- Nie próbować opisać wszystkiego w jednym procesie
- Każdy sygnał przypisywany w osobnej instrukcji współbieżnej, np. 1 proces / 1 sygnał
- Unikać długich opisów sekwencyjnych, w tym wielokrotnego przypisywania sygnału podczas jednego wykonania procesu
- Jeśli sygnał ma pamiętać swój stan ⇒ `rising_edge( Clk )`, bo inaczej będą zatrzaski

69

Testbench

JS UCiSW 1

```
entity Test_vhd is
end Test_vhd;

architecture behavior of Test_vhd is
  -- component Declaration for the Unit Under Test (UUT)
  component FSM_SendByte
  port(
    Clk : in STD_LOGIC;
    Reset : in STD_LOGIC;
    DI : in STD_LOGIC_VECTOR(7 downto 0);
    DIRdy : in STD_LOGIC;
    TxBusy : in STD_LOGIC;
    DO : out STD_LOGIC_VECTOR(7 downto 0);
    DORdy : out STD_LOGIC;
    Busy : out STD_LOGIC
    );
  end component;
```




```
  --Inputs
  signal Clk :  STD_LOGIC := '0';
  signal Reset :  STD_LOGIC := '0';
  signal DIRdy :  STD_LOGIC := '0';
  signal TxBusy :  STD_LOGIC := '0';
  signal DI :  STD_LOGIC_VECTOR(7 downto 0) := (others=>'0');
```

70

JS UCiSW 1

```
  --Outputs
  signal DO :  STD_LOGIC_VECTOR(7 downto 0);
  signal DORdy :  STD_LOGIC;
  signal Busy :  STD_LOGIC;
  -- AUX
  constant Tclk : TIME := 1 us / 50;  -- MHz
begin
  -- Instantiate the Unit Under Test (UUT)
  uut: FSM_SendByte port map(
    Clk => Clk,
    Reset => Reset,
    DI => DI,
    DIRdy => DIRdy,
    TxBusy => TxBusy,
    DO => DO,
    DORdy => DORdy,
    Busy => Busy
    );

  -- Global clock 50MHz
  Clk <= not Clk after Tclk / 2;

  -- Reset
  Reset <= '1' after 100 ns, '0' after 100 ns + Tclk;
```

71

JS UCiSW 1

```
  -- Byte source
  process
    type typeByteArray is array ( NATURAL range <> )
                             of STD_LOGIC_VECTOR( 7 downto 0 );
    constant arrBytes : typeByteArray
                              := ( X"10", X"20", X"3A", X"4F" );
  begin
    wait for 200 ns;
    for i in arrBytes'RANGE loop
      if Busy = '1' then
        wait until Busy = '0';
      end if;
      wait for 7.1 * Tclk;    -- .1 to avoid rising_edge Clk
      DI <= arrBytes( i );
      DIRdy <= '1';
      wait for Tclk;
      DIRdy <= '0';
    end loop;
    wait; -- forever
  end process;
```

72

```
  -- ASCII sink
  process
  begin
    loop
      wait until rising_edge( Clk ) and DORdy = '1';
      TxBusy <= '1';
      wait for 11.1 * Tclk;  -- e.g. 11.1 * Tclk
      TxBusy <= '0';
    end loop;
  end process;
end architecture;
```