Układy Cyfrowe i Systemy Wbudowane 1

Język VHDL

dr inż. Jarosław Sugier
Jaroslaw.Sugier@pwr.wroc.pl
IIAR, pok. 227 C-3

Literatura

- Język VHDL: M. Zwoliński(...) / K. Skahill(...) / IEEE Standard 1076 (PWr!)
- Architektury układów PLD, CPLD: www...; **www.xilinx.com**
- J. Kalisz „Podstawy elektroniki cyfrowej", WKiŁ
- C. Zieliński „Podstawy projektowania układów cyfrowych", PWN
- J. Baranowski, B. Kalinowski, Z. Nosal „Układy elektroniczne. Cz. 3: Układy i systemy cyfrowe", WNT 1994
- 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;
        -- Carry
        if A = '1' and B = '1' then
            C <= '1';
        else
            C <= '0';
        end if;
    end process;
end architecture Behavioral;
```


Porty i sygnały

```
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;
```


Klauzula generic

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


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;
```

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;
```


Wektory i napisy bitowe

```
type BIT_VECTOR is array (NATURAL range <>) of BIT;

signal DataBus  : BIT_VECTOR( 7 downto 0);
(...)

DataBus <= "10000000";
DataBus <= B"1000_0000";
DataBus <= X"80";
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 );
```


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

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 );

(...)
-- Shift right (this must be synchronous!):
ASCII <= '0' & ASCII( 7 downto 1 );

-- Arithmetic shift right:
ASCII <= ASCII( 7 ) & ASCII( 7 downto 1 );

-- Shift left:
ASCII <= ASCII( 6 downto 0 ) & '0';
```


Pakiet STD_LOGIC_1164

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

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

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

U !

```
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;
```

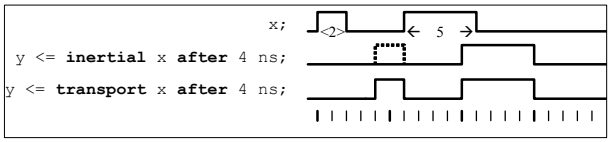


```
constant and_table : stdlogic_table := (
--      ---------------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -        |   |
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),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

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' )   -- | - |
);
```


Przypisanie sygnału

```
y <= x;
y <= ( x1 and x2 ) or ( ( x3 nand x4 ) xor x5 );
```




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

-- Wielokrotna zmiana w reakcji na pojedyńczą
-- zmianę sygnału:
y <= x after 4 ns, not x after 8 ns;

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


Przypisanie warunkowe when...else i selektywne with ... select

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


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;
```


Instrukcja procesu

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

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:

```
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:

```
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:

```
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;
```


Rejestr przesuwny:

```
entity SReg8b is
    port ( Din : in  STD_LOGIC;
           Clk : in  STD_LOGIC;
           Q   : out STD_LOGIC_VECTOR( 7 downto 0 ) );
end SReg2b;

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;
```


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;
```

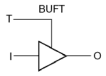

Zatrzask z asynchronicznym kasowaniem:

```
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;
```

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;
```



Licznik z asynchronicznym kasowaniem:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.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: std_logic_vector(3 downto 0);
begin
    process (C, CLR)
    begin
        if (CLR='1') then
            tmp <= "0000";
        elsif (C'event and C='1') then
            tmp <= tmp + 1;
        end if;
    end process;

    Q <= tmp;

end archi;
```

!

Licznik modulo (i z sygnałem zezwalającym):

```
process ( Clk, Clr )
begin
  if Clr = '1 then
    Cnt4b <= "0000";
  elsif rising_edge( Clk ) and CE = '1' then
    if Cnt4b = "1011" then
      Cnt4b <= "0000";
    else
      Cnt4b <= Cnt4b + 1;
    end if;
  end if;
end process;
```


Licznik ładowalny:

```
entity counters_3 is
    port(C, ALOAD : in std_logic;
         D : in std_logic_vector(3 downto 0);
         Q : out std_logic_vector(3 downto 0));
end counters_3;

architecture archi of counters_3 is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (C, ALOAD, D)
    begin
        if (ALOAD='1') then
            tmp <= D;
        elsif (C'event and C='1') then
            tmp <= tmp + 1;
        end if;
    end process;

    Q <= tmp;

end archi;
```


Licznik rewersyjny:

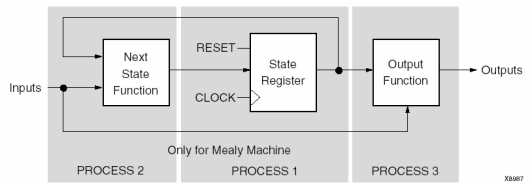
```
entity counters_6 is
    port(C, CLR, UP_DOWN : in std_logic;
         Q : out std_logic_vector(3 downto 0));
end counters_6;

architecture archi of counters_6 is
    signal tmp: std_logic_vector(3 downto 0);
begin
    process (C, CLR)
    begin
        if (CLR='1') then
            tmp <= "0000";
        elsif (C'event and C='1') then
            if (UP_DOWN='1') then
                tmp <= tmp + 1;
            else
                tmp <= tmp - 1;
            end if;
        end if;
    end process;

    Q <= tmp;

end archi;
```


Maszyny stanów



```
library IEEE;
use IEEE.std_logic_1164.all;
entity fsm_3 is
    port ( clk, reset, x1 : IN std_logic;
           outp           : OUT std_logic);
end entity;

architecture beh1 of fsm_3 is
    type state_type is (s1,s2,s3,s4);
    signal state, next_state: state_type ;
begin

    process1: process (clk,reset)
    begin
        if (reset ='1') then
            state <=s1;
        elsif (clk='1' and clk'Event) then
            state <= next_state;
        end if;
    end process process1;

    process2 : process (state, x1)
    begin
        next_state <= state;
        case state is
            when s1 =>  if x1='1' then
                            next_state <= s2;
                        else
                            next_state <= s3;
                        end if;
            when s2 => next_state <= s4;
            when s3 => next_state <= s4;
            when s4 => next_state <= s1;
        end case;
    end process process2;

    process3 : process (state)
    begin
        case state is
            when s1 => outp <= '1';
            when s2 => outp <= '1';
            when s3 => outp <= '0';
            when s4 => outp <= '0';
        end case;
    end process process3;

end beh1;
```


Instrukcja generacji

a) for ... generate

```
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;
```


Procesy

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

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
```



```
[ 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;
```


Cykle symulacji


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

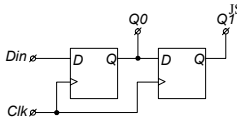
```
architecture DFlow of Ex1 is
  signal S : STD_LOGIC;
begin
  S <= A or B;
  Y <= B xor S;
end architecture;
```

```
Cykl:    A   B   S   Y    Opis:
(...)    '0' '0' '0' '0'
10ns     '1' '0' '0' '0'  (a) A←1 (b) A'event, wyk. S<=... :
                                  '1'/10ns na POW_S
10ns + Δ '1' '0' '1' '0'  (a) S←1 (b) S'event, wyk. Y<=... :
                                  '1'/10ns na POW_Y
10ns + 2Δ '1' '0' '1' '1' (a) Y←1 (b) Y'event
                                  (koniec)
```

```
process (A, B, S)
begin
  S <= A or B;
  Y <= B xor S;
end process;
```

?


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



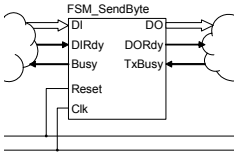

```
Cykl    Clk Din  Q0  Q1  Opis:
(...)   '0' '1' '0' '0'
10ns    '1' '1' '0' '0'  (a) Clk ← 1
                         (b) Clk'event, wykonanie procesu:
                             '1'/10ns na POW_Q0,
                             '0'/10ns na POW_Q1

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

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


Przykład:

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);
```



```
(...)
begin

  -- 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 );


  -- State register (with asynchronous reset) = process1
  process ( Clk, Reset )
  begin
    if Reset = '1' then
      State <= sReset;
    elsif rising_edge( Clk ) then
      State <= NextState;
    end if;
  end process;

(...)
```



```
(...)

-- Next state decoding = process1
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;

(...)
```

```
(...)
  -- Outputs = process3
  with HalfByte select          -- 0-15 => ASCII '0'-'F'
    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;

  DORdy <= '1' when State = sSendH or State = sSendL
           else '0';

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

end RTL;
```


Testbench

```
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');
```



```
  --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 300 ns, '0' after 300 ns + Tclk;
```



```
  -- Byte source
  process
    type typeArray is array ( 0 to 3 )
                                  of STD_LOGIC_VECTOR( 7 downto 0 );
    variable arrBytes : typeArray := ( X"10", X"20", X"3A", X"4F" );
  begin
    wait for 500 ns;
    for i in arrBytes'RANGE loop
      if Busy = '1' then
        wait until Busy = '0';
      end if;
      wait for 7.1 * Tclk;
      DI <= arrBytes( i );
      DIRdy <= '1';
      wait for Tclk;
      DIRdy <= '0';
    end loop;
    wait; -- will wait forever
  end process;
```



```
  -- ASCII sink
  process
  begin
    loop
      wait until rising_edge( Clk ) and DORdy = '1';
      TxBusy <= '1';
      wait for 11 * Tclk;  -- e.g. 11 * Tclk
      TxBusy <= '0';
    end loop;
  end process;
end architecture;
```