

VHDL cz.1

Rafał Walkowiak

IIIn PP

16.12.2017

VHDL

- **VHDL** (*ang. **V**ery **H**igh **S**peed **I**ntegrated **C**ircuits **H**ardware **D**escription **L**anguage*) jest popularnym językiem opisu sprzętu używanym w komputerowym projektowaniu układów cyfrowych w technologii układów programowalnych: FPGA i ASIC.

Charakterystyka VHDL

- Zawiera użyteczne konstrukcje semantyczne umożliwiające zwięzłą specyfikację złożonych układów cyfrowych.
- Projekt w języku VHDL może posiadać wielopoziomową hierarchię.
- Możliwe jest korzystanie z bibliotek gotowych elementów.
- Możliwe jest tworzenie podukładów wykorzystywanych jako tzw. komponenty w tym samym lub innych projektach.

Struktura projektu

- Definiowanie **jednostek projektowych** na wielu poziomach.
- Określenie **jednostki projektowej najwyższego poziomu** – aktualnie realizowanej – testowanej.
- Pozostałe jednostki projektowe możliwe do wykorzystania jako **komponenty** jednostki najwyższego poziomu.
- Wykorzystanie jednostek projektowych z **dołączonych bibliotek** jako komponentów innych jednostek projektowych.

Jednostka projektowa – desing entity

Moduł deklaracji jednostki – wej-wy, parametry

```
-- deklaracja standardowej
    biblioteki i pakietu
library ieee;
-- import elementów pakietu biblioteki
use ieee.std_logic_1164.all;
entity sumator is
port(
a:      in std_logic;
b:      in std_logic;
cin:    in std_logic;
s:      out std_logic;
cout:   out std_logic);
-- sygnały
end sumator ;
```

Moduł opisu działania jednostki

```
architecture pierwsza of sumator is
signal s1:std_logic; --deklaracja sygnału

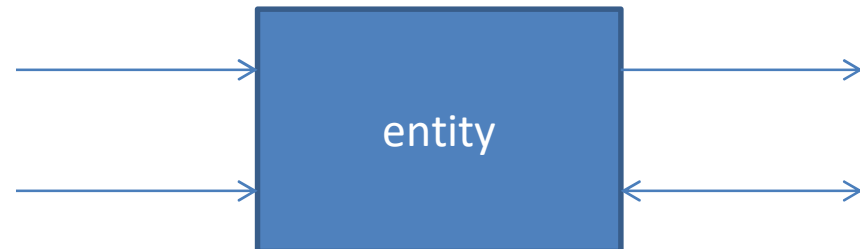
begin
-- ZBIÓR WSPÓŁBIERZNIE
-- REALIZOWANYCH PROCESÓW
- KAŻDA LINIA ODDZIELNY PROCES
s1<= (a and not b) or (b and not a);
s <= (cin and not s1) or (not cin and s1);
cout<= (a and b) or (s1 and cin);
end pierwsza;
```

Deklaracja jednostki projektowej

```
entity sumator is
port(
a:    in std_logic;
b:    in std_logic;
cin:  in std_logic;
s:    out std_logic;
cout: out std_logic);
generic; // wartości
        parametrów
end sumator ;
```

Tryby portów:

- **in** – sygnał wejściowy
- **out** – sygnał wyjściowy
- **inout** – sygnał wejściowy i wyjściowy
- **buffer** – port wyjściowy z możliwością odczytu



Opis działania jednostki – **architecture**

architecture nazwa_arch **of** nazwa_jednostki **is**

Definicje i deklaracje:
(typów, podtypów, stałych, sygnałów,
komponentów, konfiguracji)

begin

- instrukcje przypisania wartości do sygnałów
- procesy
- komponenty

end nazwa_arch;

Elementy strukturalne języka VHDL

»Słowa kluczowe

»Identyfikatory

»Obiekty danych

»Operatory

»Atrybuty

»Instrukcje

Identyfikatory obiektów danych w kodzie

- Mogą się składać z liter, cyfr i znaków podkreślenia
- Wielkość liter NIE ma znaczenia
- Nazwa musi się rozpoczynać od **litera**
- Nazwa nie powinna być dłuższa niż 16 znaków

Obiekty danych

- Obiekty danych służą do przechowywania wartości.
- Wyróżnia się trzy klasy obiektów:
 - sygnał – *signal*
 - zmienne – *variable* (możliwe tylko w procesach (obiekty lokalne) lub podprogramach)
 - stałe – *constant*

Obiekty danych

Sygnał :

- Odpowiednikiem sprzętowym **sygnału jest ścieżka** w układzie scalonym. Wartości sygnałów są przypisywane zgodnie z obowiązującymi w zakresie regułami – np. z pewnym opóźnieniem.
- deklarowane w interfejsie jednostki projektowej oraz jako sygnały wewnętrzne jednostki;
- do przenoszenia informacji między zakresami typu **proces**
- posiadają typ i wartość, która zmienia się w określonych możliwych do specyfikacji i testowania momentach czasu;

signal reset: std_logic; --sygnał jednobitowy

signal dane: std_logic_vector(0 to 7); -- wektor
sygnałowy

Obiekty danych

Zmienne:

Służą do specyfikacji sposobu przetwarzania informacji lokalnie w zakresie **procesu**. Wartości **zmiennych** są przypisywane natychmiast. Zmienna nie ma odpowiednika sprzętowego i służy do obliczeń w ramach kodu VHDL.

```
variable zmienna: bit;
```

```
variable temp_var: BIT_VECTOR(0 TO 4) := "10101";
```

Typy danych

Dotyczą sygnałów, zmiennych i stałych

- BIT, BIT_VECTOR
- **STD_LOGIC, STD_LOGIC_VECTOR,**
- STD_ULOGIC,
STD_ULOGIC_VECTOR,
- INTEGER, BOOLEAN,
- TYP WYLICZENIOWY

Typ STD_LOGIC

- STD_LOGIC, STD_LOGIC_VECTOR –
 - *standardowy typ sygnałów*
 - *posiadający 9 możliwych wartości używanych do symulacji różnych konfiguracji połączeń,*
 - *typ rozstrzygalny (możliwych wiele nośników – przypisać równoległych),*
 - *funkcja rozstrzygająca*
- Zastosowanie wymaga deklaracji:
 - **LIBRARY** IEEE;
 - **use** IEEE.Std_Logic_1164.all;
 - Pakiet zawiera rozszerzenia standardowego VHDL:
 - std_logic,
 - funkcja rozstrzygająca,
 - funkcje logiczne dla std_logic, **std_logic_vector**
 - funkcje dla detekcji zboczy sygnałów: rising_edge, falling_edge
 - funkcje konwersji typu (następna strona)

STD_LOGIC - STD_LOGIC_VECTOR

- Wartości
 - ‘U’ – wartość niezainicjowana,
 - ‘0’, ‘1’, ‘X’, - wartości silne: 0, 1, stan nieznany np. efekt jednoczesnego przypisania wartości 0 i 1
 - ‘Z’ – stan wysokiej impedancji
 - ‘W’, ‘L’, ‘H’ – wartości słabe: stan nieznany, 0, 1 – służą do implementacji poziomów sygnałów generowanych przy użyciu rezystorów podciągających (połączenia typu suma na drucie) – jednoczesne podstawienia do sygnału wartości L i 1 rozstrzygane jako 1.
 - ‘-’ – stan dowolny
(wielkie litery obowiązkowo)
- STD_LOGIC_VECTOR jest tablicą obiektów STD_LOGIC
- Dla STD_LOGIC_VECTOR operatory arytmetyczne: +, - są dostępne (przeciążenie) po zastosowaniu dodatkowo:
use IEEE.std_logic_unsigned.all;

Typ *std_ulogic* – charakterystyka jak wyżej, lecz typ nierozstrzygalny

funkcje konwersji typów

use IEEE.Std_Logic_1164.all;

funkcje	Typ argumentu	Typ wyniku
to_bit(a)	std_logic	bit
to_stdulogic(a)	bit	std_logic
to_bitvector(a)	std_logic_vector	bit_vector
to_stdlogicvector(a)	bit_vector	std_logic_vector

Typy skalarne - 1

- Bez elementów lub struktury
- Przyjmują **wartości z określonego typu zakresu** lub **wartości wyszczególnione** w deklaracji.

bit - typ dwuwartościowy 0, 1;

integer – liczby całkowite zakres predefiniowalny ($2^{31} - 1$) do $-(2^{31} - 1)$, można ograniczyć: np. - **variable** a: integer **range** – 10 **to** 10;

real – liczby rzeczywiste, zakres predefiniowany od $-1.0E+38$ do $+1.0E+38$

można ograniczyć zakres, np:

type przedzial is range -10.0 **to** +10.0;

Typy skalarne -2

Typ wyliczeniowy (definiowany przez użytkownika):

```
type wyliczeniowy is (zielony, zolty, czerwony);
```

Przykład użycia trybu wyliczeniowego:

```
signal stan: wyliczeniowy;  
case stan is  
  when zielony =>  
    stan <= zolty;  
  when zolty =>  
    stan <= czerwony;  
  .....  
end case;
```

Typ tablicowy

- Zakres typu tablicowego zmiennej lub stałej jest definiowany w ramach deklaracji typu lub deklaracji zmiennej za pomocą słów kluczowych: „to” „downto”

Przykłady:

```
TYPE Byte IS ARRAY (7 DOWNT0 0) OF STD_LOGIC;  
signal MAGISTRALA : Byte;
```

```
signal m1,m2: std_logic_vector(1 to 3);  
variable X,Y: bit_vector (0 to 7);
```

```
m1<=m2; -- podstawienie sygnałów
```

```
M1(3 to 4) <= M2( 0 to 1); -- podstawienie zakresu elementów wektora sygnału
```

```
X(3 to 4) := Y( 0 to 1); -- podstawienie zakresu elementów tablicy zmiennej
```

```
m1(1 to 2) <= "00"; --podwójny "" dla wartości wektora
```

```
m1(1) <='0'; --pojedynczy ' ' dla wartości bitu
```

```
m2(3 downto 1) <= m1(3 downto 1); -- --podstawienie zakresu elementów wektora  
sygnału
```

Konkatenacja

- Operator konkatenacji: &
- Pozwala na łączenie obiektów w nową tablicę. Łączonymi obiektami mogą być tablica/e i skalar/y.

Wektor <= (Data1(0 to 2) & data2(2 to 5) & singlebit);

-- typ obiektu do podstawienia wyniku powinien być typem tablicowym o rozmiarze będącym sumą rozmiarów elementów składowych

Stałe

- Stałe to obiekty danych, których wartości nie zmieniają się w trakcie symulacji.
- Stałe mogą być zadeklarowane w bloku deklaracji jednostki projektowej, architektury, pakiecie, procesie, funkcji i procedurze.
- Składnia: **constant nazwa : typ := wartość;**
- Przykłady użycia:
constant zero:STD_LOGIC_VECTOR(3 **DOWNTO** 0):="0000";
constant tp: time:= 5 ns; -- predefiniowany typ opisu czasu

Sygnały

- Sygnały mogą być zdefiniowane w **bloku deklaracji jednostki** lub w **części deklaracyjnej architektury** (pomiędzy architecture a begin).
- Przypisanie wartości do sygnału nie jest musi być natychmiastowe lecz jest „harmonogramowane”.
- Z sygnałem skojarzone są: typ i wartość
- Składnia deklaracji:

signal nazwa_sygnału:

typ_sygnału[:=wartość_początkowa];

Przykłady:

signal s1, s2 : bit;

signal liczba : integer := 7;

Atrybuty sygnałów

Dostarczają dodatkowych informacji o obiektach (np. sygnałach, zmiennych, typach lub komponentach).

obiekt'atrybut[(parametr)];

- Predefiniowane atrybuty:
 - 'left; 'right;'high;'low;
 - lewy, prawy, najmniejszy, największy element danego typu, 'left = 'high jeśli definiowano sygnał poprzez downto
 - 'length;'range; - liczba bitów, zakres
 - 'event; 'stable; - przyjmuje wartość true/false –
informacja, że sygnał zmienił lub nie wartość

Atrybuty sygnałów - przykłady

Użycie atrybutów dla:

```
signal x: std_logic_vector(7 downto 0)
```

Znaczenie: x'left (7), x'right (0) , x'high (7),
x'range (7 downto 0) x'length (8)

```
x(x'left) <= '1';          -- najstarszy bit =1
```

```
x <= (x'high => '1', x'low => '1', others => '0');
```

```
-- 10000001
```

Atrybuty sygnałów i typów -przykłady

type count **is** integer **range** 127 **downto** 0;

type states **is** (idle, read, write);

count'left oznacza 127

count'high oznacza 0

states'right oznacza write

states'high oznacza idle

If Clock'event **and** Clock = '1' **then** Q <= D;

Analogiczne do 'event i 'stable informacje dostarczają funkcje
określone dla sygnałów - rising_edge(nazwa_sygnalu),
falling_edge(nazwa_sygnalu)

Instrukcje

Sekwencyjne

- Porządek zapisu instrukcji sekwencyjnych określa działanie układu.
- Instrukcje sekwencyjne są stosowane w specyfikacji behawioralnej (ang. behavioral description).
Przed wszystkim w tzw. procesach

Współbieżne

- Zachowanie układu jest niezależne od kolejności instrukcji przyporządkowania sygnałów.
- Stosowane w specyfikacji typu „przepływ danych” (ang. dataflow description).

Operatory

- Służą do określenia sposobu wyznaczenia nowych wartości.
- Specyfikacja w tabeli poniżej wg malejącego priorytetu
- Dla operacji numerycznych konieczna zgodność typu argumentów

Klasa	operator	Typ danych
negacja	NOT	INTEGER, BIT, STD_LOGIC
MNOŻENIA	*,/, mod, rem	INTEGER
ZNAKU	+-	INTEGER
SUMOWANIA	+-	INTEGER, BIT, STD_LOGIC
PRZESUWANIA	<i>sll, srl, sla, sra, rol, ror</i> <i>Przesunięcie wpis 0,</i> <i>Przesunięcie wpis wartości</i> <i>krańcowej, przesuniecie cykliczne</i>	STD_LOGIC,BIT (dla tablic)
RELACYJNE	=, /, <, <=, >, >=	Argumenty – tego samego typu; wynik – BOOLEAN
LOGICZNE konieczność określenia kolejności realizacji !!!	<i>and, or, nand, nor, xor, xnor</i>	BIT, BOOLEAN, STD_LOGIC

Instrukcja przypisania

- Instrukcja przypisania (realizowana **sekwencyjnie** w procesie, **równolegle** poza procesem):

Nazwa_sygnalu <= **wyrażenie**;

- Przykład:

```
architecture strukturalna of error_test is  
  signal q_reg, q_next : std_logic_vector (7 downto 0);  
begin  
  error <= we(2) and (we(1) or we(0));  
  b <= (0=>1,others => '0'); -- przypisuje 0 do wszystkich  
    bitów oprócz zerowego wektora sygnału  
  q_reg <= q_next; -- przypisanie wektorów  
end a;
```

Instrukcja przypisania

X <= '1'; --skalar

Y <= "0000" -- wektor 4 bitowy

Wektor1 <= wektor2 & bit ;

Z <= a nand c; -- nadanie wartości bez opóźnienia

Z <= a nand c after 10 ns ;

-- nadanie wartości po czasie 10 s opóźnienia (opóźnienie inercyjne)

Do obiektu **sygnal** można przyporządkować wartość obiektu **zmienna** (i zmiennej wartość sygnału również) pod warunkiem przynależności do tego samego typu.

Realizacja instrukcji współbieżnych

- Instrukcja współbieżnego przypisania wartości do sygnału ma **ukrytą listę czułości**, która zawiera **wszystkie sygnały** znajdujące się po prawej stronie symbolu $\leq$.
- Instrukcja współbieżnego przypisania jest wykonywana zawsze, gdy nastąpi **zmiana dowolnego sygnału** występującego na liście czułości.
- Dla typów **rozstrzygalnych współbieżne przypisania** wartości do sygnałów powodują utworzenie **kolejnych nośników** (ang. driver) dla uaktualnianych sygnałów. W przypadku uaktualniania sygnału przez wiele instrukcji dla utworzonych nośników sygnału stosowana jest **funkcja rozstrzygająca określająca** stan sygnału. Wykorzystanie w przypadku magistral.
- W przypadku wielokrotnego przypisania w **instrukcjach sekwencyjnych** stosowany jest jeden nośnik sygnału.

Przypisanie selektywne: with-select-when

Instrukcje współbieżne

-- Przyporządkowanie do sygnału wybranego sygnału

with w select

x <= a when v1 | v2,
b when 1 to 3,
z when others;

-- w, x, sygnały

-- x:= a gdy w=v1 lub w=v2

Przypisanie warunkowe: when else

Instrukcje współbieżne

-- warunkowe przyporządkowanie wartości do sygnału

-- nadaje się do wejść o różnym priorytecie

```
y <=    j when a=b else  
        k when c=d else  
        m when others;
```

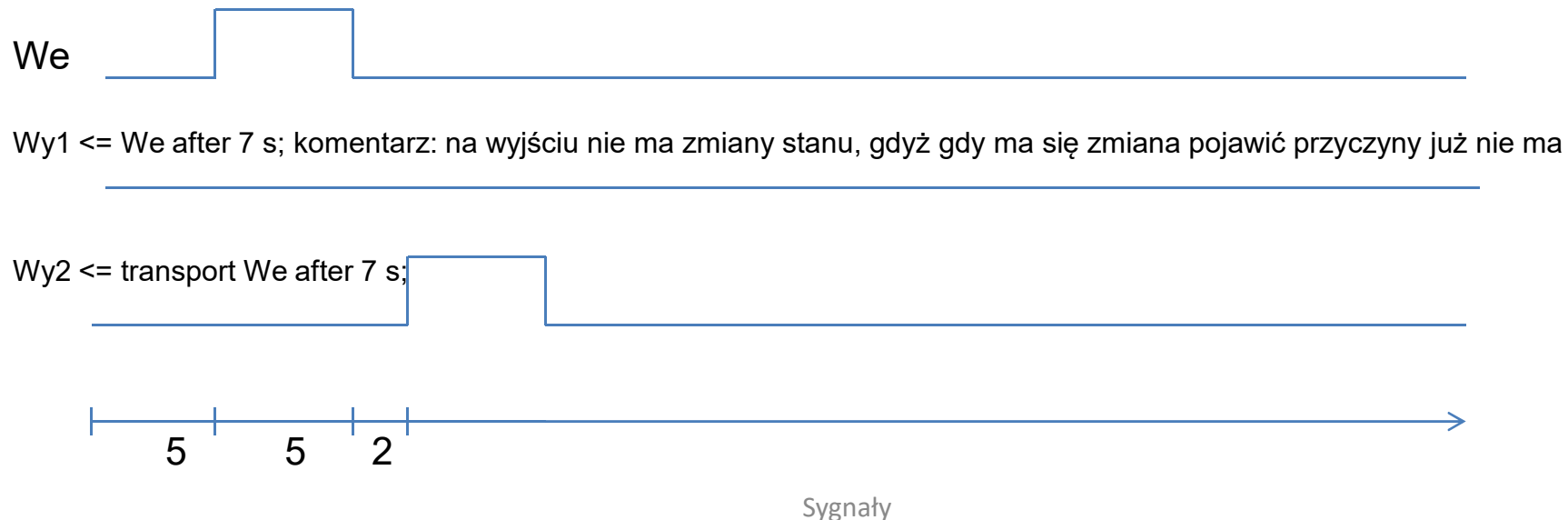
Z <= A when S= '1' else B;

-- zmiana wartości sygnału S (**sygnał** po prawej stronie wyrażenia przypisania - **S** znajduje się na „liście czułości”) powoduje wykonanie instrukcji przypisania o wyniku zależnym od wartości sygnału S (zakładamy że A i B nie są sygnałami, jeśli są sygnałami to tak samo ich zmiana wyzwała nowe przypisanie).

Typy opóźnień

Opóźnienie inercyjne: efekt szybkich zmian sygnałów wejściowych obserwowany gdy przyczyna jest aktywna po okresie czasu równym wielkości opóźnienia (większość praktycznie spotykanych opóźnień w układach cyfrowych).

Opóźnienie transportowe: efekt zmian sygnałów wejściowych obserwowany niezależnie od szybkości zmian na wejściach.



Generic

- Klauzula pozwalająca na przekazywanie informacji ze środowiska do bloku kodu.
- Umożliwia **dostarczenie z zewnątrz bloku wartości parametrów** bloku .
- Zdefiniowane wielkości przy użyciu parametru określonego klauzulą **generic** mogą określać parametry modułów takie jak szerokość magistral, liczba bitów rejestrów, komparatorów, licznik pętli, parametry czasu.
- Dzięki konstruktorowi **generic** wykorzystanie komponentu może być elastyczne, gdyż w zależności od potrzeb można określić wartość parametru układu. Użycie zamiast **generic** obiektu **constant** wymaga przygotowania wielu różnych architektur układu – dla każdego potrzebnego wariantu.

entity rom is

generic (width: integer :=16);

port (Data :out bit_vector (width-1 downto 0));

.....

Specyfikacja behawioralna układu

- Opis działania układu poprzez opis sposobu przetwarzania danych za pomocą sekwencji operacji niezbędnych do uzyskania oczekiwanego wyniku przetwarzania.
- **Proces** w VHDL jest sposobem opisu kolejnych kroków działań w przetwarzaniu danych.

Proces (1)

Składnia:

```
[etykieta:] process [ ( lista_czułości ) ] [ is ]
```

```
--definicje i deklaracje
```

```
begin
```

```
--instrukcje_sekwencyjne;
```

```
end process [ etykieta ] ;
```

- Lista czułości - jest to lista **sygnałów**, których zmiana powoduje **aktywację procesu** (ponowne wykonanie wyspecyfikowanych kroków – lista czułości równoważna jest z **wait on** na końcu procesu)
- Przypisanie wartości sygnałów bazuje na bieżącej wartości sygnałów znajdujących się po prawej stronie znaku <=. Modyfikowane sygnały zostaną uaktualnione dopiero **po wykonaniu wszystkich** instrukcji procesu; uaktualnienie powoduje „**zawieszenie**” procesu. Sygnały zostają uaktualnione wartością **ostatniego** przypisania. Zmiany dotyczące obiektów zmiennych (**variable**) są wykonywane natychmiast.

Proces (2)

1. Występuje wewnątrz architektury.
2. Instrukcje wewnętrzne wykonywane są sekwencyjnie
3. Procesy nie mogą być zagnieżdżane.
4. W części deklarycyjnej procesu można definiować: typy, podtypy, stałe, atrybuty i zmienne.
5. W części deklarycyjnej procesu nie można deklarować sygnałów.
6. Instrukcje danego procesu wykonywane w kolejności wystąpienia w kodzie, lecz równolegle z instrukcjami innych procesów.
7. Zgodnie z semantyką języka VHDL **sygnał zachowuje** wartość jeśli nie następuje do niego przypisanie.
8. Ze względu na pkt 7 warto zatem dla czytelności kodu przypisywać wartości we wszystkich gałęziach konstrukcji **if** lub przypisać im wartość domyślną na początku procesu.

Podstawienie sygnałów

współbieżne

```
A,B,C std_logic;
```

```
begin
```

```
A<=B;
```

```
A<=C;
```

```
end
```

B	C	A
0	Z	0
1	Z	1
0/1	1/0	NIEDOZWOLONE
Z	0	0
Z	1	1

szeregowe

```
A,B,C std_logic;
```

```
process (clk)
```

```
Begin
```

```
if clk'event and clk ='1' then
```

```
    A<=B;
```

```
    A<=C;
```

```
    D<=A
```

```
end
```

A ⁺	B	D ⁺	D ⁺⁺
C	NIEISTOTNE	A	C
X ⁺ TO WARTOŚĆ X PO 1 ZBOCZU NAR. CLK			
X ⁺⁺ TO WARTOŚĆ X PO 2 ZBOCZU NAR. CLK			

Instrukcje sekwencyjne

- Do wykorzystania w *procesach*

```
wait for 50 ns;           -- czekaj 50 ns
```

```
wait until enable = '0';      -- czekaj aż enable będzie ponownie = 0
```

```
wait on input_A until input_B='1';
```

```
-- czekaj aż nastąpi zmiana input_A pod
```

```
-- warunkiem że input_B stanie się ponownie = 1
```

```
wait on input_A, input_B;           -- lista czułości;
```

- Konstrukcje:

— If

– Case

— For

Instrukcja warunkowa if – then - else

- Instrukcja sekwencyjna
- Część **elsif** może wystąpić dowolną ilość razy
- Część **else** nie musi wystąpić
- Może być zagnieżdżana
- Składnia:

```
if warunek then  
    sekwencja_instrukcji1  
    {elsif warunek then  
        sekwencja_instrukcji2 }  
    [else sekwencja_instrukcji3]  
end if ;
```

Konstrukcje behawioralne i strukturalne - porównanie

Behawioralna (w procesie)

```
if a='0' then if b='0' then y2<='1';  
                else y1<='0';  
                end if;  
else if b='1' then y2<='0';  
                else  
                    y1<='1';  
                end if;  
end if;
```

-- przy braku spełnienia warunków wartość
sygnału pozostaje bez zmian

strukturalna

```
y2<=(not a and not b) or  
      (y2 and (not a or not b)) ;  
y1<=(a and not b) or  
      (y1 and (a or not b));
```

a	b	y1	y2
0	0	y1	1
0	1	0	y2
1	1	y1	0
1	0	1	y2

Instrukcja wyboru - case

- Instrukcja sekwencyjna
 - Można do porównania wskazać kilka możliwych wartości **zmiennej** lub **zakres zmiennej**
 - Wartości nie mogą się powtarzać
 - Składnia: **case** *zmienna* **is**
when *wybór1* => *sekwencja1_instrukcji* ;
when *wybór2* => *sekwencja2_instrukcji* ;
[**when** *others* => *sekwencja3_instrukcji* ;]
end case;
- w kodzie powyżej znak => oznacza „to”

Pętle for i while

Instrukcje sekwencyjne

```
etykieta1: for i in 3 downto 0 loop  
    if reset = '1' then  
        dane_wy(i) <= '0';  
    end if;  
End loop etykieta1;
```

```
i:=3;  
etykieta2: while ( i > 0 ) loop  
    if reset = '1' then  
        dane_wy(i) <= '0';  
    end if;  
    i:=i-1;  
End loop etykieta2;
```

--wyjścia z pętli

```
exit etykieta when i > 14;  
if i>14 then exit etykieta;
```

Przerwa w realizacji pętli (1)

-- gdy reset jest równy 0 zeruj elementy wektora A

L5: for licznik in 1 to 10 loop

exit L5 when reset = '1';

 A(licznik):='0';

end loop L5;

Przerwa w realizacji pętli (2)

```
-- jeśli się zmieni stan magistralii  
-- zbadaj stan magistrali – zlicz liczbę jedynek
```

```
signal Magistrala: bit_vector (3 downto 0);  
signal Jedyнки: integer;
```

```
LiczJedyнки:process(Magistrala)  
  variable liczba_jedynek:integer:=0;  
  begin  
    for licznik in 3 downto 0 loop  
      next when Magistrala(licznik)='0';  
      liczba_jedynek:= liczba_jedynek+1;  
    end loop;  
    Jedyнки<= liczba_jedynek;  
  end process LiczJedyнки
```

Sygnał a zmienna w procesie

signal A integer;

variable B integer;

process (A);

A<= A+1; -- wyrażenie niepoprawne

-- operacja powtarzana

-- nieskończenie wiele razy

B := B+1; -- wyrażenie poprawne

end process;

Proces przerzutnika D

```
-- Przerzutnik synchroniczny z
-- zerowaniem synchronicznym
-- kompletny, uruchomiony projekt
library IEEE;
use ieee.std_logic_1164.all;
entity D is      port (
    D      : in std_logic;
    CLK    : in std_logic;
    CLR:    in std_logic;
    pre    : in std_logic;
    Q      : out std_logic;
    NQ     : out std_logic );
end d;
```

architecture first of D is

begin

Synch_D: **process** (CLK)

constant low:std_logic:='0';

constant high:std_logic:='1';

begin

if rising_edge(CLK) then

if (**CLR='0'**) then

Q<=low; NQ<= high;

else if ((CLR='1') and (**PRE='0'**))
then

NQ<=low; Q<= high;

else

Q<=D; NQ<= not D;

end if;

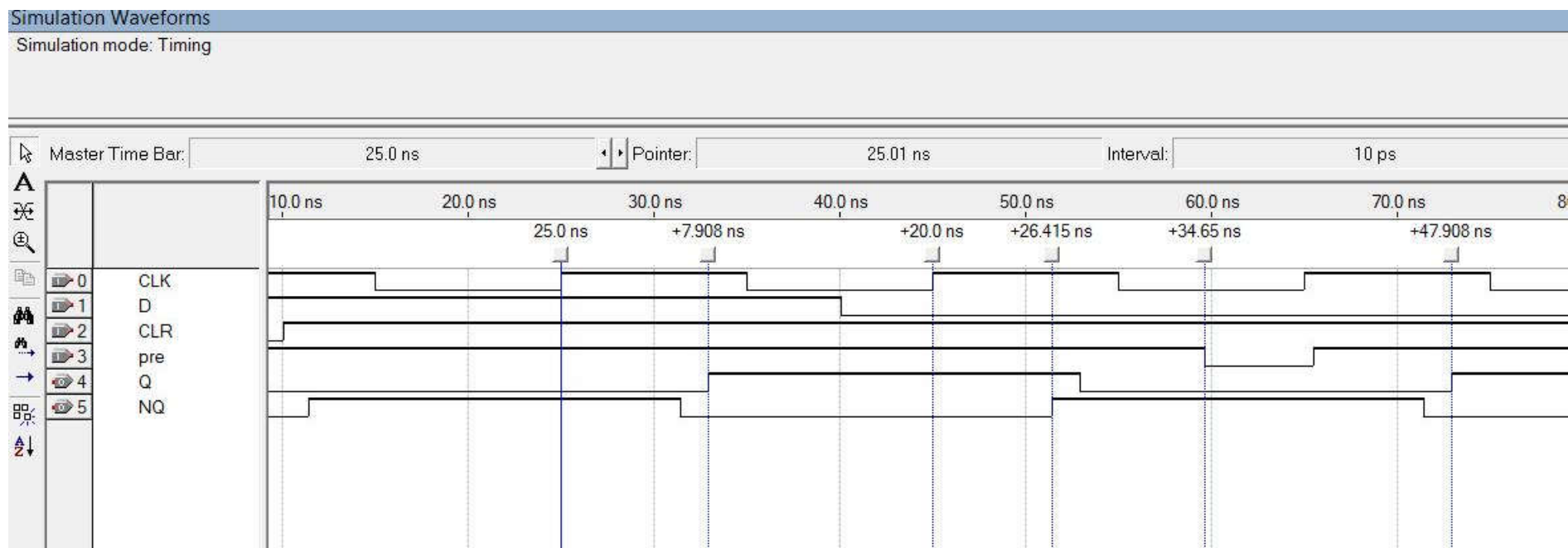
end if;

end if;

end process Synch_D;

end first;

Przerzutnik D synchroniczny z zerowaniem synchronicznym



Przerzutnik zatraskowy z zerowaniem

```
--kompletny uruchomiony projekt
library IEEE;
use ieee.std_logic_1164.all;
entity D_Latch is
    port
        ( D      : in std_logic;
          C      : in std_logic;
          CL     : in std_logic;
          pr     : in std_logic;
          Q      : out std_logic;
          NQ     : out std_logic
        );
end d_l;
```

architecture first of D_Latch is
begin

```
LATCH: process (C, D, CL, PR)
constant low:std_logic:='0';
constant high:std_logic:='1';
```

begin

```
if CL=low then
```

```
    Q<=low; NQ<= high;
```

```
else if (CL=high) and (PR=low)
then NQ<=low; Q<= high;
```

```
    else if C=high then
```

```
        Q<=D; NQ<= not D;
```

```
    end if;
```

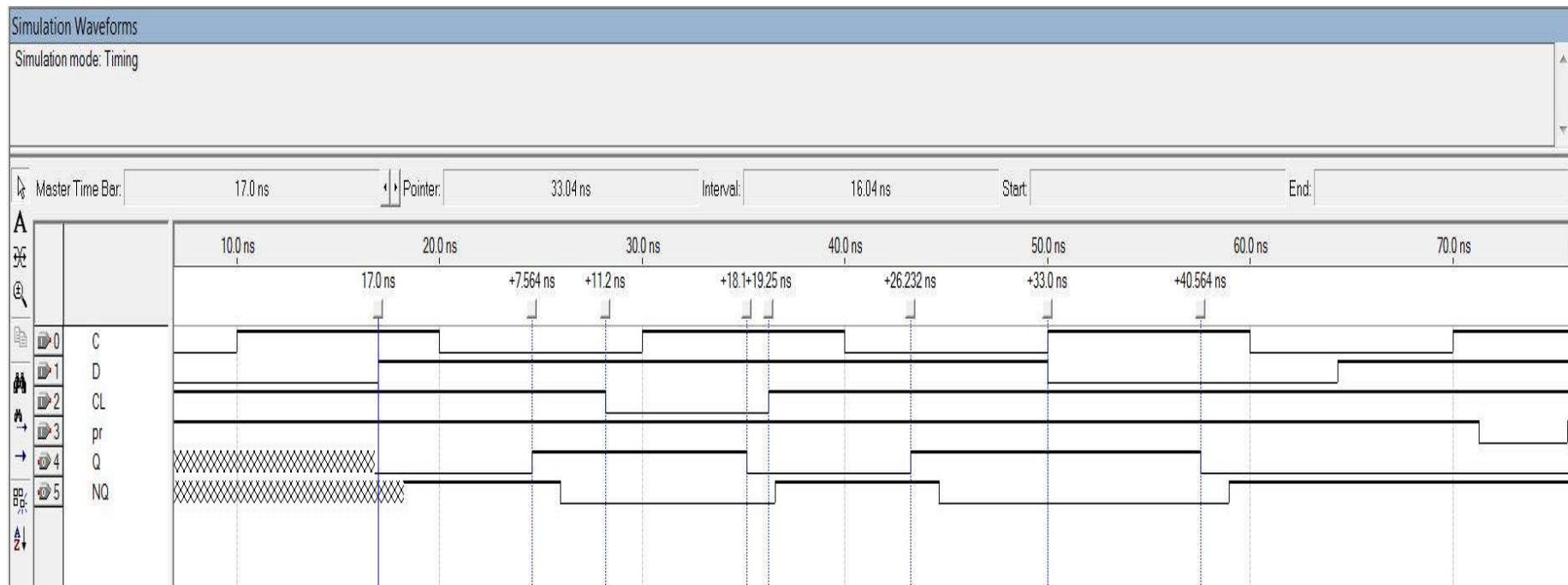
```
end if;
```

```
end if;
```

```
end process LATCH;
```

```
end first;
```

Przerzutnik zatrzaskowy z zerowaniem



Proces multipleksera

```
Mux 2to1: process (A,B,SEL)
begin
  Y<=A;
  if (SEL='1') then Y <= B; -- instrukcja sekwencyjna
end if;
end process Mux2to1
```

Komponent

- Jednostka projektowa raz zdefiniowana i przeznaczona do wielokrotnego wykorzystania w projektach.
- Komponenty można tworzyć w ramach projektu w pliku głównym, plikach dołączonych do projektu lub w bibliotece projektu.
- W celu korzystania z komponentów konieczne są:
 - **Definicja** komponentu lub wskazanie biblioteki- źródła
 - **Deklaracja** komponentu w **bloku deklaracji architektury jednostki** projektowej wykorzystującej komponent (zawiera porty - parametry formalne)
 - **Konkretyzacja** – specyfikacja wykorzystania komponentu (zawiera przyporządkowanie do parametrów formalnych parametrów aktualnych związanych z wykorzystaniem komponentu)
- Trzy rodzaje specyfikacji użycia (konkretyzacji) układu: bezpośrednia, powiązania portów pozycją w specyfikacji, powiązania portów nazwami portów.
- Projekty zawierające komponenty do wykorzystania w innych projektach mogą stanowić pakiety przechowywane w bibliotekach.

Przykład wykorzystania komponentu

konkretyzacja poprzez powiązanie portów poprzez pozycję w specyfikacji

architecture strukturalna of uklad1 is

-- deklaracja komponentu w architekturze jego wykorzystania

component mux3_5to1

port(adres,d1,d2,d3,d4,d5,d6,d7,d8: in std_logic_vector(2 downto 0);

wy:out std_logic_vector(2 downto 0)); -- parametry formalne

end component;

-- pozostałe deklaracje

begin

-- użycie jednostki – konkretyzacja komponentu w innym innym układzie

mult1:mux3_5to1 port map (adr,we0, we1,we2,we3,we4,we5,we6,we7,wy);

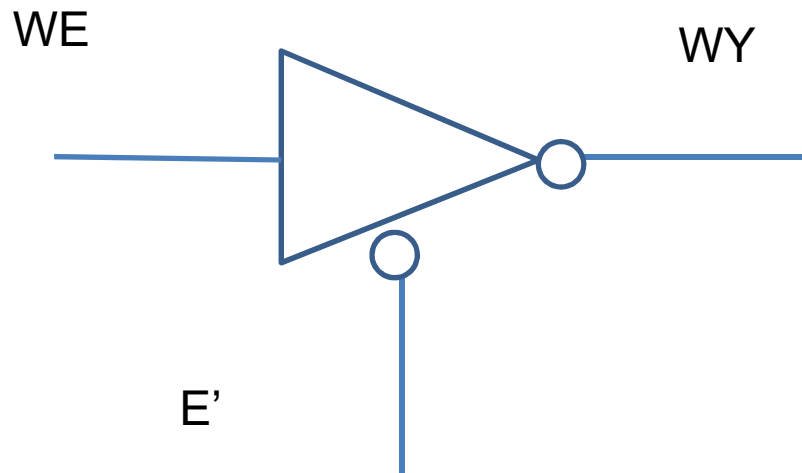
-- są to parametry aktualne związane z użyciem komponentu

--pozostała część definicji architektury układ1

end strukturalna;

Bramki trójstanowe

Trzeci stan logiczny



WE	E'	WY
0	0	1
0	1	Z
1	0	0
1	1	Z

Koncepcja:

W niektórych implementacjach bramek logicznych oprócz logicznego "0" i logicznej "1" istnieje trzeci stan logiczny

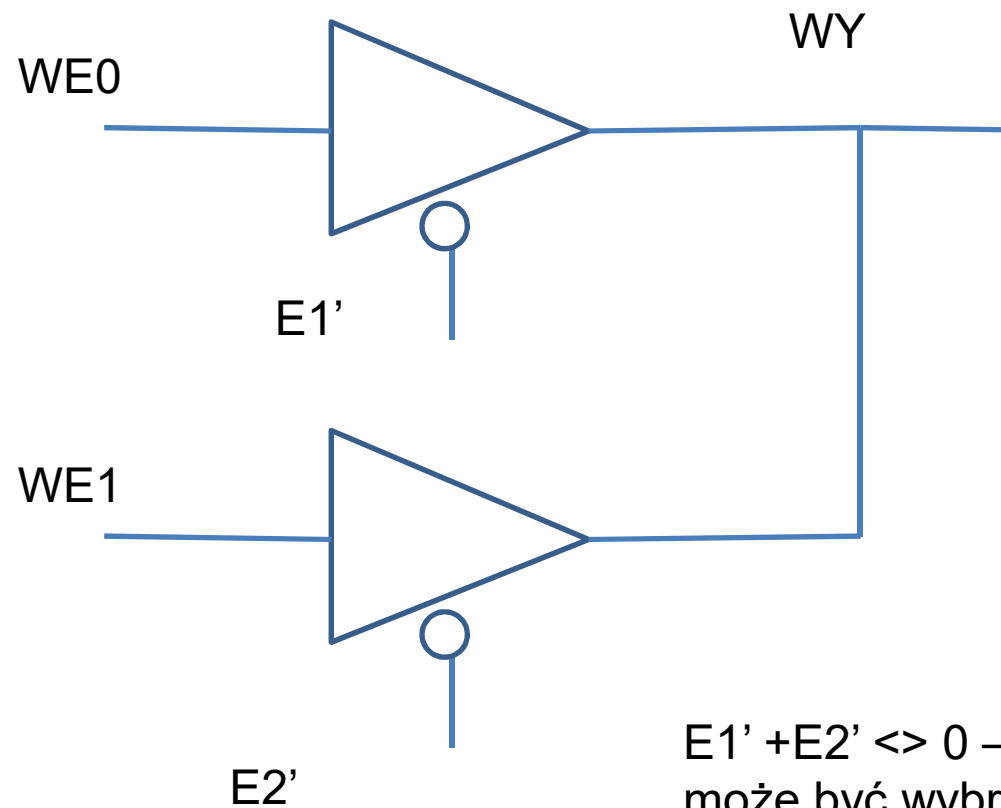
- stan wysokiej impedancji (ang. high impedance) Z.

Gdy **wyjście układu** nie jest połączone galwanicznie z układem cyfrowym znajduje się ono w stanie wysokiej impedancji – nie jestysterowane (ani do poziomu wysokiego ani niskiego).

Zastosowanie:

podłączenie układu do magistrali jako jednego z możliwych źródeł jejysterowania. Konieczne zapewnienie wykluczającego się wyboru źródła.

Bramki trójstanowe zastosowanie



WE0	WE1	E1'	E2'	WY
0	X	0	1	0
1	X	0	1	1
X	0	1	0	0
X	1	1	0	1
X	X	1	1	Z

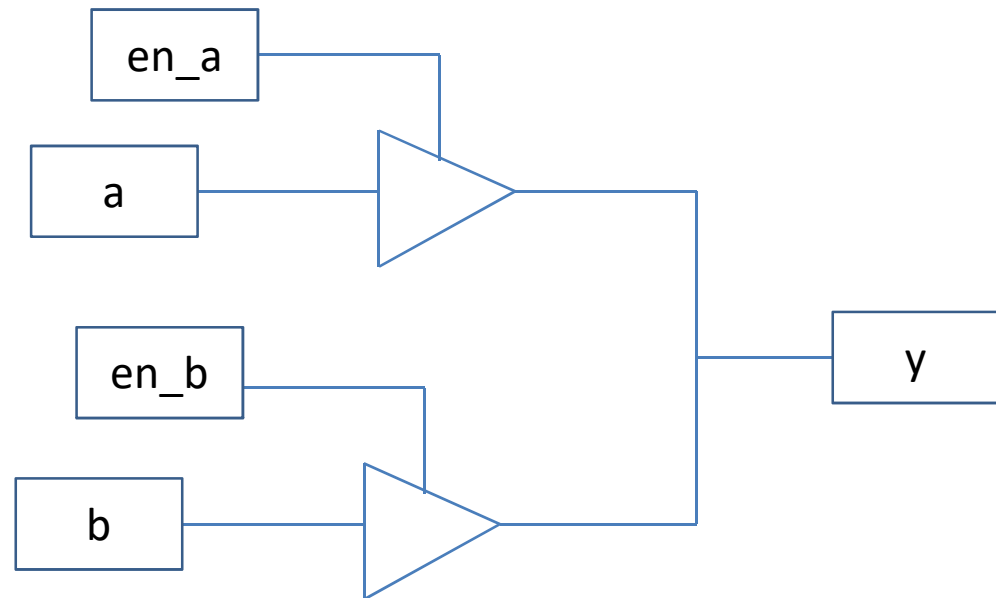
$E1' + E2' \neq 0$ – TYLKO jedno wejście ENABLE może być wybrane

```
Library IEEE;  
Use IEEE.std_logic_1164.all;  
entity 2_zrodla_sygnalu is
```

```
    port (  
        en_a: in boolean;  
        en_b: in boolean;  
        a: in std_logic;  
        b: in std_logic;  
        y: out std_logic);  
end zrodla;  
architecture przyklad of  
    2_zrodla_sygnalu is  
begin  
    y<= a when en_a else 'Z';  
    y<= b when en_b else 'Z';  
end przyklad;
```

-- wielokrotne równoległe przypisanie wartości
do portu wyjściowego, funkcja
rozstrzygająca użyta do określenia stanu
sygnału (typy rozstrzygalne sygnałów).

Sygnały wielostanowe – magistrala (1,0,Z)



-- Altera devices contain tri-state buffers in the I/O. Thus, a tri-state buffer must feed a top-level I/O in the final design. Otherwise, the Quartus II software will convert the tri-state buffer into logic.

Przykład: odwróć magistralę

```
constant MSB : natural := 7;  
subtype ZakresMagistrali is natural range MSB downto 0;  
  
process (MagistralaWejsciowa)  
  variable NowaMagistrala: bit_vector(ZakresMagistrali)  
  
  begin  
    for i in ZakresMagistrali loop  
      NowaMagistrala(MSB-i) := MagistralaWejsciowa(i);  
    end loop;  
  end process;
```