Kurs języka VHDL

***V**ery **H**igh (Speed Integrated Circuits) **D**escription **L**anguage*

Józef Kalisz, Wojskowa Akademia Techniczna, 2008

Początek: lata 80-te XX w.

Kontrakt VHSIC (*Department of Defense, USA*)

Podstawa: język ADA

Normalizacja: IEEE

(*Institute of Electrical and Electronic Engineers*)

Od 1987 kolejne wersje normy IEEE Standard 1076

Norma IEEE Std 1164: pakiet std_logic_1164

VHDL

Projektowanie sprzętu cyfrowego

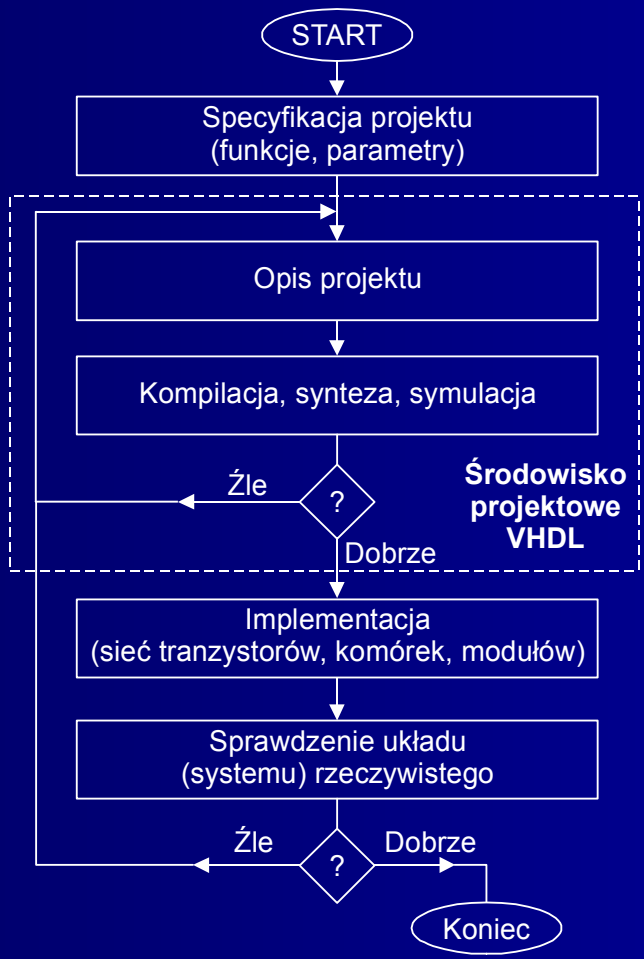
1. Opis tekstowy: plik w języku VHDL
2. Kompilacja pliku
3. Sprawdzenie, symulacja funkcjonalna

Dalszy ciąg procesu projektowania

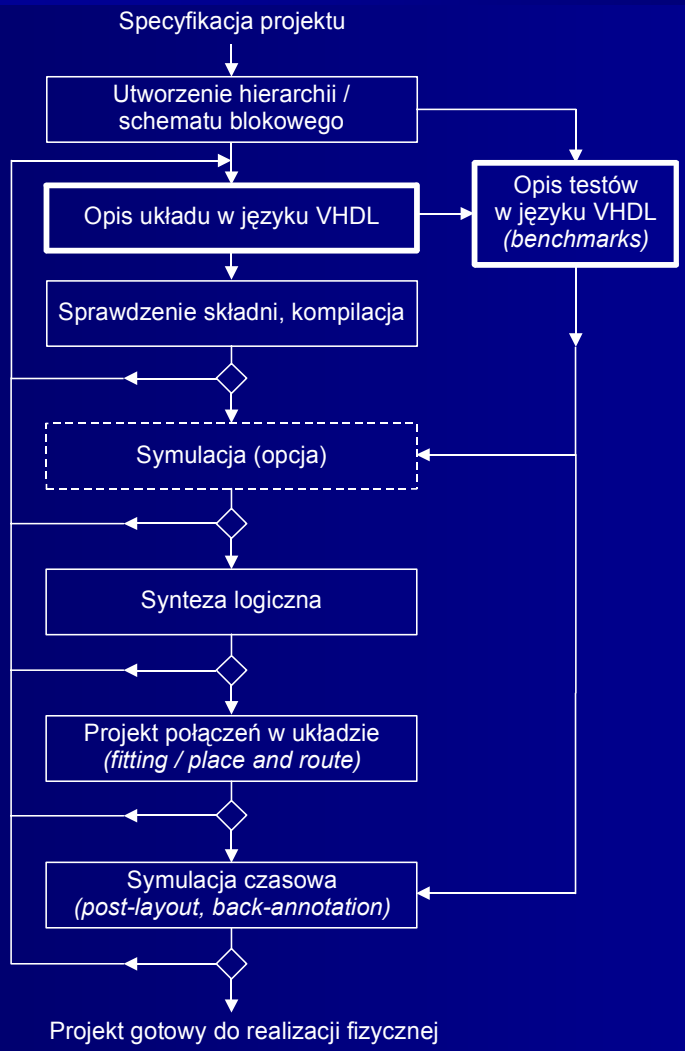
4. Synteza logiczna
5. Projekt układu scalonego
6. Symulacja czasowa
7. Weryfikacja praktyczna

VHDL

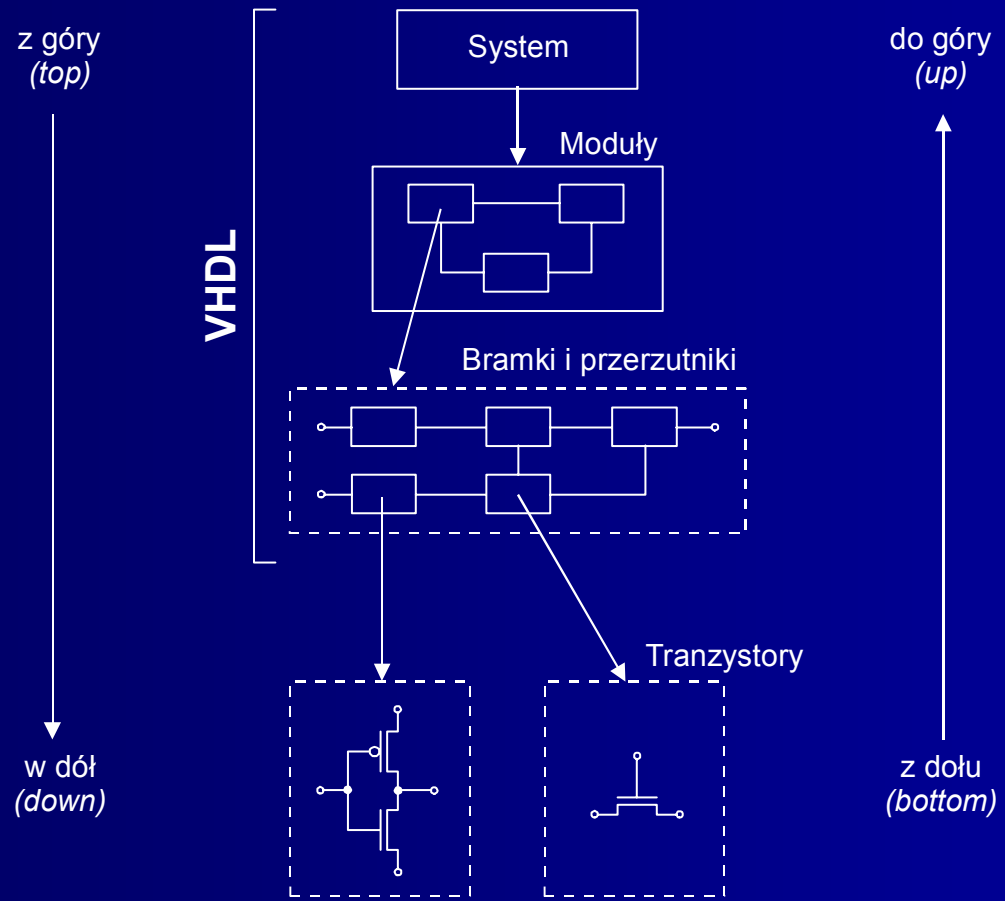
Proces projektowania



Projektowanie w środowisku VHDL



Hierarchia w procesie projektowym



OPIS UKŁADÓW CYFROWYCH W JĘZYKU VHDL

- Opis: plik tekstowy - zbiór modułów o portach we/wy
- Stosowana czcionka: `Courier, Lucida Console`
- Nazwy: wyłącznie litery alfabetu, cyfry i znak _
- Nazwa musi się rozpoczynać od litery i nie może kończyć znakiem _
- Bez „polskich liter” (ą, ę, ...)!
- W nazwach i słowach kluczowych nie odróżnia się liter małych od dużych
- Komentarze w każdym wierszu po dwóch kreskach (--)
- Słowa kluczowe: wytłuszczenie (**entity**), wersaliki (ENTITY), kolor (**entity**)

Przykład opisu:
3-wejściowa bramka NAND
jako jednostka projektowa (moduł)


```
-- NAND3  (umowne oznaczenie modułu)

--  deklaracja jednostki (projektowej):
entity nand3 is
   port(a,b,c : in bit;            --  wejscia
            y  : out bit);         --  wyjscie
   end nand3;

-- deklaracja architektury:
architecture a1 of nand3 is
begin
   y <= not(a and b and c);        --  cialo architektury
end a1;
```

<= symbol przypisania wartości wyrażenia do sygnału

VHDL

Składnia jednostki projektowej

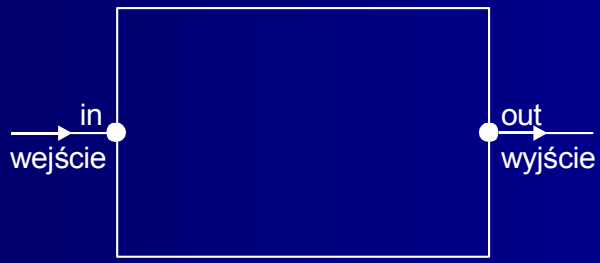
[…] – pojedyncze elementy opcjonalne
{…} – powtarzalne elementy opcjonalne
| – alternatywa („albo-albo”)

```
[deklaracje bibliotek i polecenia uzycia pakietow]
entity nazwa_jednostki is -- deklaracja jednostki
   [generic(lista_deklarowanych_stalych);]
   [port(lista_sygnalow_wej. in | inout nazwa_typu;
    lista_sygnalow_wyj. out | buffer | inout nazwa_typu);]
end nazwa_jednostki;
architecture nazwa_architektury of nazwa_jednostki is
   [czesc_deklaracyjna: typy, sygnaly_wewnetrzne, stale]
begin
   {instrukcja_wspolbiezna;}  -- cialo architektury
end nazwa_architektury;
```

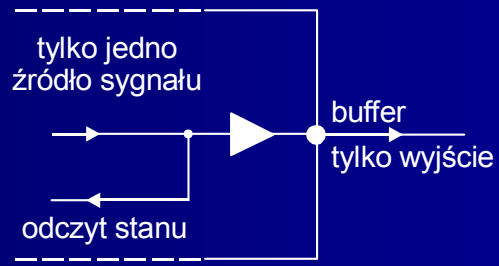
- Klauzule **port** i **generic** tworzą nagłówek jednostki
- Ciało architektury zawiera ciąg instrukcji współbieżnych
- Może być więcej architektur (o różnych nazwach)

Porty jednokierunkowe: `in`, `out`
Port `buffer`: tylko wyjście z odczytem stanu
Port dwukierunkowy: `inout`

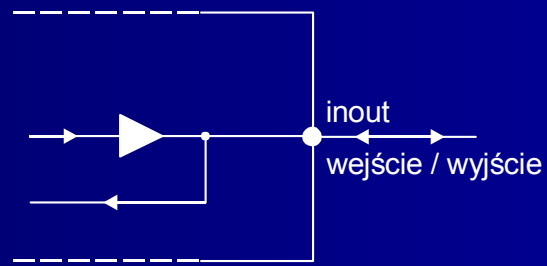
a)



b)



c)



VHDL

Sekwencyjność operacji ► instrukcja procesu

```
-- Przerzutnik T wyzwalany zboczem narastajacym, z wejsciem zerujacym
entity fft is
   port (T,C,R    : in bit;  -- C – zegar, R - zerowanie
             Q    : out bit);
end fft;
architecture a1 of fft is
begin
   process (C,R)              -- (C,R): lista wrazliwosci
      variable tq  :  bit;    -- czesc deklaracyjna procesu,
                              -- wprowadzenie zmiennej lokalnej
   begin                      -- poczatek opisu ciala procesu
      if  R = '1' then tq := '0';          -- zerowanie
         elsif C'event and C = '1' and T = '1' then
         -- jesli jest narastajace zbocze zegara i T = '1', to
         tq := not tq;        -- zmiana stanu (przypisanie do zmiennej)
      end if;
      Q <= tq;                -- przypisanie do sygnalu Q
   end process;               -- koniec procesu
end a1;
```

`:=` symbol przypisania do zmiennej

- zmienna lokalna `tq` umożliwia pamiętanie „obecnego” stanu przerzutnika
- `'event` – atrybut do wykrycia zdarzenia (zmiany stanu sygnału C)

  `C'event = true | false`

VHDL

Zastosowanie sygnału wewnętrznego
(w części deklaracyjnej architektury)

```
-- Inny opis architektury przerzutnika T
architecture a2 of fft is
   signal  tp  :  std_logic;  -- sygnal wewnetrzny
begin
   process (C,R)
   begin
      if  R = '1' then tp <= '0';
        elsif C'event and C = '1' then
        tp <= tp xor T;  -- przypisanie sekwencyjne
      end if;
   end process;
   Q <= tp;    -- przypisanie wspolbiezne
end a2;
```

- Zaleca się stosowanie zmiennej lokalnej: mniej zasobów, symulacja natychmiastowa
- Sygnały wewnętrzne nie mogą być deklarowane wewnątrz procesu
- Sygnały wewnętrzne mają rodzaj **inout**
  (możliwy zapis i odczyt do modelowania połączeń wewnętrznych)

Wykorzystanie portu buffer lub inout, który umożliwia odczyt stanu

```
-- Przerzutnik T z wyjsciem buffer
entity fft1 is
   port (T,C,R  : in bit;
         Q : buffer bit);
end fft1;
architecture a1 of fft1 is
begin
   process (C,R)
   begin
      if  R = '1' then Q <= '0';
         elsif C'event and C = '1' and T = '1' then
         Q <= not Q;  --  odczyt i negacja stanu Q
      end if;
   end process;
end a1;
```

VHDL

Inny przykład stosowania procesu

```
-- Przerzutnik D
entity ffd is
   port(D,C,R : in bit;
              Q : out bit);
end ffd;
architecture a1 of ffd is
begin
   process(C,R)
      begin
         if  R = '1' then Q <= '0' ;
            elsif (C'event and C = '1') then Q <= D;
         end if;
   end process;
end a1;
```

- Nie użyto zmiennej lokalnej, ani sygnału wewnętrznego, ani wyjścia **buffer** lub **inout**, gdyż przerzutnik D nie wymaga odczytu stanu obecnego.

VHDL

Style opisu architektury

Styl behawioralny – *behavioral* (opis działania)

- algorytmiczny

  opis sekwencji stanów, z użyciem procesu i instrukcji sekwencyjnych

- przepływowy - *dataflow*

  opis przepływu danych podczas przetwarzania, instrukcje współbieżne, równania boolowskie;

  także opis układów sekwencyjnych z rejestrami – styl RTL – *Register Transfer Logic*

Styl strukturalny (opis budowy czyli zapis połączeń komponentów schematowych)

OBIEKTY do zapisu danych

- sygnały, zmienne, stałe i pliki
- TYPY służą do wyrażania wartości obiektów

- Sygnały są funkcjami czasu
- Składnia deklaracji sygnału:

```
signal nazwa_sygnalu : nazwa_typu [ograniczenie][:=
wyrazenie];
```

Np.

```
signal reset : bit := '1';  -- inicjalizacja '1'
signal y : bit_vector(0 to 3) := ('0','1','1','0');
signal alfa : integer range 0 to 255; -- ograniczenie
signal mi : bit_vector(9 downto 0):= (others => '1');
```

- (others => '1') ustala wartość początkową elementów wektora mi równą '1'
- Operator (=>) oznacza przyporządkowanie

VHDL

Zmienne (variable)
nie są funkcjami czasu

- Stosowane pomocniczo tylko w obrębie procesu lub podprogramu
- Przypisanie do zmiennej (:=)

  na przykład pp := x + 1;

Deklaracja zmiennej:

```
variable nazwa_zmiennej : nazwa_typu [ograniczenie][:=
wyrazenie];
```

Np.

```
variable uu  :  integer range 0 to 127 := 5;
variable pp  :  integer range 500 downto 5 := 100;
```

VHDL

Stałe przypisują nazwy wartościom stałym określonego typu

Deklaracja stałej:

```
constant nazwa_stalej : typ := wyrazenie;
```

Np.

```
constant UCC : bit := '1';
Constant masa : bit;  -- domyslna wartosc '0'
constant fi : integer := 12;
constant vec : bit_vector(7 downto 0) := "10010011";
```

VHDL

Typy skalarne

- numeryczne
- wyliczeniowe
- fizyczne

Liczby całkowite – integer

- Zakres predefiniowany od – ($2^{31} - 1$) do + ($2^{31} - 1$)
- Można zadeklarować nowy typ podając zakres, np.

```
type dzien_miesiaca is range 0 to 31;
```

- Operatory arytmetyczne

```
+, - , *, /, **, abs, rem, mod
```

- Operatory relacyjne

```
=, /=, <, <=, >, >=
```

Np. `a+b+c, (a*3 - 5)/b, 2**c`

Podtypy standardowe, predefiniowane

```
subtype natural is integer range 0 to 2147483647;
subtype positive is integer range 1 to 2147483647;
```

VHDL

Typy skalarne

- numeryczne
- wyliczeniowe
- fizyczne

Liczby rzeczywiste – `real` – b. rzadko stosowane

- Zakres predefiniowany od – 1.0E+38 do + 1.0E+38
- Można zadeklarować samodzielnie zakres, np.

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

Typ bitowy – `bit` – b. często stosowany

- Zakres `('0', '1')` – to są znaki, a nie liczby!
- Operatory logiczne:
  `not, and, or, nand, nor, xor, xnor`
- Operator unarny `not` ma najwyższy priorytet
- Operatory relacyjne : `=, /=, <, <=, >, >=`
- Operacje relacyjne (predykaty) mają wyższy priorytet niż logiczne

Np.

```
a and (b or c) -- funkcja logiczna, wynik typu bit
(not a or c) /= '1'  -- predykat, wynik typu boolean
```

VHDL

Typy skalarne

- numeryczne
- wyliczeniowe
- fizyczne

- Wada typu `bit`: m.in. brak stanu wysokiej impedancji i stanu nieokreślonego

Typ dziewięciowartościowy `std_ulogic` (IEEE Std 1164-1993), unresolved

```
type std_ulogic is   (
     'U',    -- stan niezainicjowany
     'X',    -- wymusza stan nieznany
     '0',    -- wymusza stan 0
     '1',    -- wymusza stan 1
     'Z',    -- stan wysokiej impedancji
     'W',    -- słaby stan nieznany (odczyt)
     'L',    -- słabe 0 (odczyt), rownowazne polaczeniu
                  -- przez rezystor z masa)
     'H',    -- słabe 1 (odczyt), rownowazne polaczeniu
                  -- przewodu przez rezystor
                  -- z napieciem zasilania)
     '-'     -- stan nieokreslony, podobnie jak 'X');
```

- Wartości `'U'`, `'W'`, `'-'` są metalogiczne, tylko do symulacji

VHDL

Typy skalarne

- **numeryczne**
- **wyliczeniowe**
- **fizyczne**

Typ `std_logic` - resolved (jednoznaczny), najczęściej stosowany

- Zdefiniowany w pakiecie `std_logic_1164`

Typ boolowski: `boolean`

- Zakres: `(false, true)`
- Operatory: głównie `=` , `/=`
- Operacje jak dla typu bitowego

  Np. predykaty `(a = b)`, `(b /= c)`

Typ znakowy: `character`

- Znaki są zapisywane z użyciem apostrofów, np. `'b','b','b'`
- Zakres: 128 znaków ze zbioru ASCII

`(..,'0','1','2',..,'A','B','C',..,'a','b','c',..)`

- Norma IEEE Std 1076-2001 rozciąga zbiór znakowy na tablicę 256 znaków ujętych w normie ISO 8859 (wersja zachodnia)

VHDL

Typy skalarne

- numeryczne
- wyliczeniowe
- fizyczne

Typy wyliczeniowe są definiowane przez listę identyfikatorów (nazw) lub listę literałów (pojedynczych liter lub cyfr ujętych w apostrofy). Np.

```
type kolory is (niebieski,zielony,czerwony);  -- nazwy
type trojkowy is ('3','6','9'); -- literały
type litery is ('A','B','C','c'); -- literały
```

- w definicji typu każdy identyfikator ma indeks, np. niebieski => 0, '6' => 1, 'c' => 3

- Poniższe dwa typy są identyczne:

```
type duze is (A,ALA,ELA,WANDA);  -- nazwy
type male is (a,ala,ela,wanda);  -- nazwy
```

Typ fizyczny czasowy: time

Zakres predefiniowany: (0 to 2147483647) jednostek:
fs, ps, ns, us, ms, sec, min, hr

Np. x <= not a and b after 12 ns;

Jak skrócić czas kompilacji i uzyskać po syntezie prostsze struktury logiczne?

- Zawężać zakresy typów standardowych i definiować własne podtypy

Np.
```
subtype bajtowy is integer range 0 to 255;
subtype mlitery is character range 'a' to 'z';
```

- Tworzyć własne typy przez podanie odpowiedniego zakresu

Np.
```
type zakres256 is range 0 to 255;
type dekada is range 0 to 9;
```

- Operacje (np. dodawanie) można realizować na danych tylko tego samego typu!
- Nie można więc dodać danych o typach `zakres256` i `dekada`
- Do tego celu można utworzyć podtyp, np.

```
subtype dekada is zakres256 range 0 to 9;
```

VHDL

Typy złożone
array i string

- Zbiory indeksowane: każdy element zbioru ma liczbę (indeks), określającą pozycję elementu w zbiorze

- Tablica (array) – gdy elementy zbioru mają jeden typ

```
-- np. definicja slowa 16-bitowego
type slowo16 is array(0 to 15) of bit;
```

- Typ łańcuchowy (string) ► łańcuchy znakowe (słowa)

  np. `constant pik : string(1 to 5) := ”Alina”;`
- Słowa wymagają użycia cudzysłowów
- Łączenie słów (konkatenacja): `”100”&” razy”`

  Operacje na znakach: np. `c <= pik(1 to 3)&”cja”;`

VHDL

Typ złożony
bit_vector

- Tablica jednowymiarowa to wektor
- Gdy elementy wektora są typu bit, wówczas stosujemy typ predefiniowany bit_vector(format)

```
type bit_vector is array (natural range <>) of bit;
```

„Skrzynka” <> - format nieokreślony (wymaga określenia), np.

```
variable dbyte : bit_vector(15 downto 0);
```

- Zapis wektorów: np. (’1’,’0’,’0’,’1’) albo ”1001”

Na przykład

```
variable a : bit_vector(5 downto 0) := ”110010”;
```

Operacje na bitach: np. `d <= a(3 downto 1)&”010”;`

- Zaleca się stosować format malejący
- Zapis wektorów w HEX, np.

```
constant mol : bit_vector(7 downto 0) := x”3F”;
```

VHDL

Typ złożony standard_logic_vector

- Aby użyć elementy wektora typu std_logic, stosujemy typ predefiniowany std_logic_vector(format)

Np.

```
port (A, B : out std_logic_vector(3 downto 0));
variable dekada : std_logic_vector(9 downto 0);
```

Można zrealizować przypisania

```
A <= "X01Z";
dekada := "10XX0ZZ1U0";
```

- Stosowanie tego typu (i typu std_logic) wymaga wpisania na początku modułu klauzuli użycia pakietu:

```
use ieee.std_logic_1164.all;
```

VHDL

Typy złożone `signed` i `unsigned`

(norma IEEE Std 1076.3-1997)

- Aby wektory o elementach typu `bit` lub `std_logic` można stosować jak równoważne liczby
- Typ `signed` dla liczb całkowitych ze znakiem w kodzie ZU2
- Typ `unsigned` dla liczb naturalnych - bez znaku
- Na skrajnej prawej pozycji w wektorze jest najmniej znacząca cyfra
- Typy te są ujęte w pakietach standardowych języka. Aby je użyć, trzeba na początku modułu wpisać klauzulę `use ieee.numeric_std.all;`

  Np. zmienna `variable m : unsigned(7 downto 0);`

  przybiera wartości w zakresie od 0 do $2^8 - 1$
- Możliwe są operacje `m + 1, m*7, if m = 30...` i podobne
- Niemożliwe jest jednak przypisanie liczby całkowitej, np. `m := 14;`
- Wtedy trzeba użyć typu łańcuchowego, np. `m := "00001110";`

  (niekiedy jest akceptowany zapis w HEX, np. `m := X"0E";`)

VHDL

Atrybuty

do identyfikacji cech typów i sygnałów

- Zapis `(typ | obiekt)'atrybut` zwraca poszukiwaną cechę
- `'event` zwraca wartość `true` jeśli miało miejsce zdarzenie
- Rozpoznawanie granic typów skalarnych i złożonych, np.

```
type zakres is range 15 downto -5;
variable P: integer := zakres'left;      -- P := 15
variable R: integer := zakres'right;     -- R := -5
variable S: integer := zakres'high;      -- S := 15
variable T: integer := zakres'low;       -- T := -5
```

- `'length` zwraca długość wektora, np. wektora `alfa`

```
variable U: integer := alfa'length;
```

- `'range` identyfikuje format (zakres indeksowania) wektora

  Np. jeśli `signal w : std_logic_vector(0 to 15);`

  to `w'range` zwraca zakres `(0 to 15)`

- `'reverse_range` zwraca odwrócony zakres

VHDL

Pakiety i biblioteki

package, library

- Pakiety (package) zawierają obiekty, podprogramy, modele, funkcje i procedury do łatwego stosowania w różnych projektach
- Pakiety standardowe (IEEE) i niestandardowe (indywidualne)
- Biblioteki (library) zawierają skompilowane pakiety i realizowane projekty
- Biblioteka work domyślnie zawiera wyniki realizowanych projektów
- Domyślnie przed każdym projektem istnieje (niewidoczny) zapis

```
library std, work;    -- deklaracja bibliotek
use std.standard.all; -- klauzula uzycia
```

- Aby użyć typy std_logic i std_logic_vector należy na początku modułu umieścić klauzule:

```
library ieee;
use ieee.std_logic_1164.all;
```

VHDL

Instrukcje współbieżne

Instrukcje współbieżne można bezpośrednio użyć w ciele architektury

- przypisanie do sygnału (<=)
- instrukcja procesu `process`
- instrukcja współbieżnego wywołania procedury
- instrukcja łączenia komponentów `port map`
- instrukcja powielania `generate`
- instrukcja blokowa `block`

Dwie pierwsze są podstawowe

VHDL

Instrukcje przypisania do sygnału: podstawowe, warunkowe, selektywne

Przypisanie podstawowe

```
sygnal <= [transport] wyrazenie [after odcinek_czasu];
```

- `wyrazenie` musi dawać wynik tego samego typu co `sygnal`
- Klauzule `transport` i `after` służą do opisu opóźnień typu `time`
- Są one pomijane w syntezie i służą tylko do symulacji

Przypisanie warunkowe `when-else`

```
sygnal <=  {wyrazenie when warunek [and | or warunek]
     else} wyrazenie_koncowe;
```

- Symulator bada warunki (o wyniku typu `boolean`) kolejno i realizuje przypisanie przy pierwszym spełnionym warunku
- Należy wymienić wszystkie możliwe warunki lub wpisać `wyrazenie_koncowe`, które domyślnie spełnia warunek `when others` („dla pozostałych warunków”)

VHDL

Przypisanie warunkowe

Przykład opisu multipleksera 2-na-1

```
architecture a4 of mux2_1 is
begin
  y <= a when s = '0' else b; -- pamietac o apostrofach!
end a4;
```

albo z użyciem typu std_logic

```
architecture a5 of mux2_1 is
begin
  y <= a when s = '0' else
       b when s = '1' else 'X'; -- albo '0' zamiast 'X'
end a5;
```

albo z użyciem klauzuli **unaffected** (nie wszystkie kompilatory ją akceptują!)

```
       b when s = '1' else unaffected;
```

- ► Do opisu większych multiplekserów trzeba jako adres użyć wektor s
- ► Wada: ważna jest kolejność warunków ► po syntezie bardziej złożone układy ► zaleca się stosować przypisanie selektywne

VHDL

Przypisanie selektywne
with-select

```
with wybor select
{sygnal <=  wyrazenie when wartosc_wyboru,}
 sygnal <=  wyrazenie_koncowe when others;
```

Przykład opisu multipleksera grupowego 4-na-1 bajtów

```
entity muxg8_4_1 is
   port (a,b,c,d : in std_logic_vector(7 downto 0);
              adr  : in std_logic_vector(1 downto 0);
               y   : out std_logic_vector(7 downto 0));
end muxg8_4_1;
architecture a1 of muxg8_4_1 is
begin
   with adr select
      y <=  a when "00",
            b when "01",
            c when "10",
            d when "11",
            "XXXXXXXX" when others;
end a1;
```

VHDL

Przypisanie selektywne

Zapis tablicy stanów układu kombinacyjnego

Np. dla funkcji 6 zmiennych

```
a  :  in std_logic_vector(5 downto 0);
```

i sygnale wyjściowym

```
y  :  out std_logic;
```

można opisać tablicę stanów:

```
with a select
y <= '1' when
     --  tu trzeba wypisac wszystkie stany a,
     --  dla ktorych y = 1, na przyklad
           "001100" | "10100X" | "1110XX",
       '0' when
     --  tu trzeba wypisac wszystkie stany a,
     --  dla ktorych y = 0, na przykład
           "X01010" | "1100X0" | "100010",
       'X' when others;
```

- Symbol (|) oznacza wybór („albo”). Nie oznacza on funkcji **or**.
- Można również wykorzystać równoważniki dziesiętne (elementy zbioru T)

VHDL

Instrukcja procesu
process

- Instrukcja współbieżna – zapisywana w ciele architektury
- Składnia:

```
[etykieta:]process [(lista_wrazliwosci)][is]
        [czesc_deklaracyjna]
begin
   {instrukcja_sekwencyjna;}
end process [etykieta];
```

- Lista wrażliwości zawiera nazwy sygnałów, które podczas symulacji powodują wykonanie instrukcji sekwencyjnych wewnątrz procesu, gdy tylko którykolwiek z tych sygnałów zmieni wartość
- Lista ta jest opcjonalna, lecz gdy jej nie ma, to musi być wprowadzona wewnątrz procesu sekwencyjna instrukcja czekania `wait`
- Opcjonalna etykieta służy tylko do polepszenia czytelności zapisu i generalnie zaleca się jej stosowanie
- W części deklaracyjnej mogą być umieszczone deklaracje typów (`type`), stałych (`constant`) i zmiennych lokalnych (`variable`)

VHDL

INSTRUKCJE SEKWENCYJNE

do opisu procesów oraz podprogramów (procedury, funkcje)

Lista zawiera dwie instrukcje wykorzystywane także jako współbieżne

- przypisanie podstawowe do sygnału (<=)
- wywołanie procedury

oraz

- przypisanie do zmiennej (:=)
- instrukcja czekania `wait`
- instrukcja warunkowa `if-then-else`
- instrukcja wyboru `case`
- instrukcja pętli `loop` i związane instrukcje `exit` i `next`
- instrukcja pusta `null`
- instrukcja testowa `assert`

VHDL

INSTRUKCJE SEKWENCYJNE
wait

Oznacza przejściowe wstrzymanie wykonania procesu lub podprogramu. Trzy opcje:

```
wait [until warunek]      -- czekaj dopoki
   [on nazwa_sygnalu {,nazwa_sygnalu}] -- czekaj na
   [for odcinek_czasu];   -- czekaj przez
```

Opis architektury przerzutnika D przy uzyciu instrukcji **wait on**, czyli „czekaj na (zmianę sygnału (sygnałów))”

```
architecture a2 of ffd is
begin
   process     -- nie ma listy wrazliwosci!
   begin
      wait on (R,C);     -- czekaj na zmiane R,C
      if  R = '1' then Q <= '0';
         elsif (C'event and C = '1') then Q <= D;
      end if;
   end process;
end a2;
```

VHDL

INSTRUKCJE SEKWENCYJNE
if-then-else

- Sekwencyjny odpowiednik współbieżnej instrukcji przypisania warunkowego `when-else`, której nie można stosować w obrębie procesu ani podprogramu
- Składnia:

```
if warunek then {instrukcja sekwencyjna;}
   {elsif warunek then {instrukcja sekwencyjna;}}
   [else {instrukcja sekwencyjna;}]
end if;
```

- Wynik warunku jest typu `boolean` (`false` lub `true`)
- Gdy warunek jest spełniony (`true`), to wykonywana jest sekwencja instrukcji następująca bezpośrednio po słowie `then`
- W przeciwnym razie (`else`) wykonywana jest inna sekwencja albo sprawdzany jest kolejny warunek (`elsif warunek then`) i tak dalej
- Sprawdzanie warunków następuje kolejno, a więc pierwsze wypisane warunki mają odpowiednio wyższy priorytet

VHDL

INSTRUKCJE SEKWENCYJNE
case

- Sekwencyjny odpowiednik współbieżnej instrukcji przypisania selektywnego `with-select-when`, której nie można stosować w obrębie procesu ani podprogramu
- Składnia:

```
case wyrazenie is
   {when wybor => {instrukcja sekwencyjna;}}
   [when others => {instrukcja sekwencyjna;}]
end case;
```

- `wybor` jest pojedynczą wartością `wyrazenia` albo grupą takich wartości
- Przy opisie należy wymienić wszystkie wzajemnie wyłączające się wartości, albo dla wartości nieużywanych wprowadzić zapis

```
when others => sekwencja_instrukcji;
```

lub

```
when others => null;  --  brak dzialania
```

- Sprawdzanie wartości `wyboru` następuje równolegle (jednocześnie), czyli żadna z nich nie ma priorytetu względem innych

VHDL

INSTRUKCJE SEKWENCYJNE
case

- Jako wybor można użyć wyrażenia alternatywnego, grupującego kilka wartości przy użyciu symbolu albo (|)

- Na przykład, jeśli c jest liczbą całkowitą i y jest sygnałem out typu std_logic

```
case c is
   when 1 | 3 | 4 | 14 => y <= '1';
   when 2 | 5 | 6 => y <= '0';
   when 7 | 9 => y <= 'X';
   when others => null;
end case;
```

VHDL

INSTRUKCJE SEKWENCYJNE
loop

- Pętla `loop` umożliwia zapis powtarzania sekwencji instrukcji
- Trzy rodzaje pętli: `for`, `while` i pętle nieskończone
- Jeśli liczba iteracji (obiegów pętli) jest z góry znana, stosujemy instrukcję `for-loop`

```
[etykieta:] for parametr in zakres loop
  {instrukcja sekwencyjna;}
end loop [etykieta];
```

- `parametr` pętli jest zazwyczaj indeksem zakresu definiowanego przez granice i kierunek indeksowania (`to | downto`).
- Na przykład, inwersję wszystkich bitów wektora `A` można opisać

```
for i in 0 to A'length - 1 loop
  B(i) <= not A(i);   -- B zawiera negacje bitow A
end loop;
```

- Indeks (w tym przykładzie `i`) nie musi być odrębnie deklarowany i jest rozpoznawany wyłącznie w obrębie tej instrukcji
- Atrybut `'length` umożliwia identyfikację długości wektora
- Opcjonalna `etykieta` zwiększa czytelność programu

VHDL

INSTRUKCJE SEKWENCYJNE
while-loop

- Jeśli liczba iteracji zależy od wyniku sprawdzania warunku przed powtórzeniem pętli, to stosuje się instrukcję **while-loop**

```
[etykieta:] while warunek loop
          {instrukcja sekwencyjna;}
          end loop [etykieta];
```

- Na przykład można opisać wirtualny zegar, który działa tylko wtedy gdy zmienna boolowska flaga ma wartość true:

```
process
begin
  while flaga loop
    zegar <= not zegar;  -- typ bit lub std_logic
    wait for okres_zegara/2;  -- typ time
  end loop;
end process;
```

VHDL

INSTRUKCJE SEKWENCYJNE
wyjście z pętli (exit, next)

Instrukcja

```
exit[etykieta_petli][when warunek];
```

służy do wyjścia z pętli, jeśli warunek jest spełniony

► Można zrezygnować z opcji [when warunek], ale oznacza to bezwarunkowe wyjście z pętli przy pierwszym napotkaniu słowa exit podczas wykonywania instrukcji

► warunek można też wprowadzić stosując instrukcję warunkową if-then:

```
if warunek then exit;
end if;
```

Instrukcja

```
next [etykieta_petli][when warunek]
```

służy do zakończenia wykonywania bieżącej iteracji pętli i przejście do następnej, jeśli warunek jest spełniony. Jednocześnie indeks pętli zwiększa się o jeden.

VHDL

INSTRUKCJE SEKWENCYJNE
null, assert

Instrukcja pusta **null** wskazuje, że nie jest wykonywane żadne działanie poza przejściem do wykonania następnej instrukcji

► Często stosowana na końcu instrukcji **case**

Instrukcja testowa **assert** służy do sprawdzania poprawności symulacji i alarmowania w razie błędów przez odpowiednie komunikaty tekstowe

```
[etykieta:] assert warunek
   [report "tekst_komunikatu"]
   [severity severity_level];  --  poziom zagrożenia
```

► Jeśli warunek nie jest spełniony, to generowany jest komunikat alarmowy

► Opcja **report** umożliwia wyświetlenie wpisanego komunikatu, a opcja **severity** (zagrożenie) wyświetla słowo informujące o jednym z czterech stopni zagrożenia:

Failure (uszkodzenie), Error (błąd), Warning (ostrzeżenie) i Note (uwaga)

Np.
```
assert (t_setup < 2 ns)
report "Za krotki czas ustalenia!"
severity Warning;
```

i jeśli warunek nie jest spełniony, to symulator zareaguje komunikatem Assertion violation.

VHDL

PODPROGRAMY: FUNKCJE I PROCEDURY

- **Funkcja** jest podprogramem, którego parametrami są wyłącznie sygnały wejściowe i który jako wynik zwraca jedną wartość określonego typu.
- **Procedura** jest podprogramem, który na liście parametrów ma zarówno wejścia i wyjścia.
- **Funkcja** jest traktowana jak wyrażenie, które można stosować tylko w obrębie procesu lub podprogramu.
- **Procedura** (ściślej – jej wywołanie) jest traktowana jak instrukcja, która może być współbieżna lub sekwencyjna, zależnie od miejsca jej użycia.
- Do tworzenia podprogramów można stosować wyłącznie instrukcje sekwencyjne.
- W obrębie podprogramów, podobnie jak wewnątrz procesów, nie mogą być deklarowane sygnały wewnętrzne, lecz tylko zmienne lokalne.
- W odróżnieniu od procesów, w podprogramach wartości zmiennych lokalnych są ustalane od nowa przy każdym wywołaniu podprogramu.
- Oznacza to, że zmienne lokalne nie mogą być wykorzystane do „pamiętania” stanów w podprogramach.

VHDL

FUNKCJE (1)

- Funkcja przed użyciem (czyli wywołaniem) musi być zadeklarowana.
- Składnia deklaracji funkcji:

```
function nazwa [(lista_parametrow)] return typ is
  [czesc deklaracyjna]
  --  zazwyczaj deklaracje zmiennych lokalnych
  --  nie mozna deklarowac sygnalow!
begin
  {instrukcja sekwencyjna;}
  -- przynajmniej jedna instrukcja musi
  -- zawierać słowo return(wyrazenie)
end [function][nazwa];
```

- Deklarację funkcji umieszcza się globalnie w odrębnym i zadeklarowanym pakiecie albo lokalnie w części deklaracyjnej architektury lub w nagłówku procesu.
- Wywołanie funkcji wykonuje się używając jej nazwy jako wyrażenia z podaniem listy parametrów przechodzących. Wartości tych parametrów nie są zmieniane przez funkcję.

VHDL

FUNKCJE (2)

- Argumenty funkcji, czyli parametry formalne z listy są rodzaju **in** i domyślnie należą do klasy stałych (**constant**), ale muszą być jawnie deklarowane w klasie **signal**, jeśli w obrębie funkcji wykorzystuje się na przykład atrybuty sygnału.
- Parametry funkcji nie mogą należeć do klasy zmiennych (**variable**).
- Przy wywołaniu funkcji parametry formalne są zastępowane przez parametry aktualne (przechodzące).
- Instrukcja **return** kończy wykonywanie funkcji i zwraca wynik funkcji, czyli wartość zadeklarowanego typu.
- W obrębie funkcji nie można stosować instrukcji **wait**.
- Przykład funkcji komparacji trzech bitów:

```
function komp3(a,b,c : std_logic) return boolean is
begin
   if ((a xor b) nor (a xor c)) = '1' then
      return true;
      else return false;
   end if;
end komp3;
```

W powyższym przykładzie funkcja zwraca wynik typu **boolean**, ale można ją łatwo zmodyfikować, aby otrzymać wynik typu **std_logic**.

VHDL

FUNKCJE (3)

- Funkcja do konwersji wektora typu `std_logic_vector` na liczbę całkowitą (`integer`).
- Podobną rolę pełni typ `unsigned`, ale tylko w odniesieniu do operacji arytmetycznych.

```
function vec_int(signal a : std_logic_vector) return integer is
   variable p : integer;
begin
   p := 0;
   for i in a'range loop
      p := p * 2;
      case a(i) is      -- można uzyc instrukcje if-then-else
         when '1' | 'H' => p := p + 1;
         when others => null;
      end case;
   end loop;
   return p;
end vec_int;
```

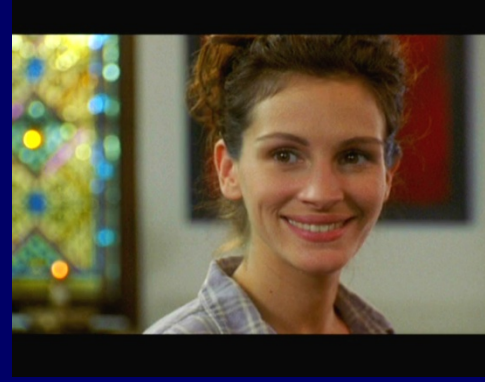
- Wzór do zamiany słowa dwójkowego $B = b_{n-1}b_{n-2}...b_1b_0$ na równoważnik dziesiętny:

  $L(B) = ((...((b_{n-1}\cdot 2 + b_{n-2})\cdot 2 + ...)\cdot 2 + b_1)\cdot 2 + b_0$

- Atrybut sygnałowy `a'range` umożliwia iteracje w pętli, rozpoczynając od lewostronnego bitu.
- Ponieważ użyto atrybutu sygnałowego, więc trzeba było zadeklarować parametr formalny `a` jako `signal`. Również parametr aktualny musi należeć do klasy `signal`.

VHDL

FUNKCJE (4)



- Wiele funkcji jest zdefiniowane w pakiecie standardowym std_logic_1164.
- Np. funkcje wykrywające narastające i opadające zbocze sygnału:

```
function rising_edge(signal s : std_ulogic) return boolean;
function falling_edge(signal s : std_ulogic) return boolean;
```

- Czyli zamiast pisać `if zegar'event and (zegar = '1') then...`, można użyć zapis `if rising_edge(zegar) then... .`
- Typem obiektu może być std_logic.
- Można utworzyć „uniwersalne" funkcje logiczne, automatycznie rozpoznające liczbę argumentów, jeśli wejścia zadeklaruje się w postaci wektora bez podania formatu. Np. funkcja NAND:

```
function fnand(signal a: std_logic_vector) return std_logic is
  variable p  :  std_logic;
begin
  p := '1';
  for i in a'range loop
    p := p and a(i);
  end loop;
  return not p;
end fnand;
```

- Np. jeśli były zadeklarowane sygnały `signal s,t : in std_logic_vector(0 to 7);` oraz `signal r : in std_logic;` to można funkcję fnand użyć w postaciach

```
if fnand(s) = '1' then...;
r <= fnand(s) or fnand(t); -- error!: signal r is in!
```

VHDL

PROCEDURY (1)

- **Procedura** może zwracać więcej wyników niż jeden. Składnia deklaracji procedury:

```
procedure nazwa [(lista-_parametrow)] is
   [czesc deklaracyjna]
   --  zazwyczaj deklaracje zmiennych lokalnych
   --  nie mozna deklarowac sygnalow!
begin
   {instrukcja sekwencyjna;}
-- moze być uzyte słowo return (bez wyrazenia) do zakonczenia procedury
end [procedure][nazwa];     -- normalne zakonczenie procedury
```

- Parametry formalne: klasa **constant**, **variable** lub **signal**, rodzaj: **in**, **out**, lub **inout**. Na liście należy definiować klasę i rodzaj parametru.
- Rodzaj **in** - do wprowadzania wartości parametrów do procedury, **out** - do zwracania wartości parametrów z procedury (czyli jej wyników), **inout** - do modyfikacji jednego parametru (wprowadzanego i zwracanego).
- Podobnie jak funkcja, procedura musi być zadeklarowana albo globalnie w pakiecie, albo lokalnie: w deklaracji architektury albo w nagłówku procesu.
- Procedura może być wywołana w ciele architektury jako instrukcja współbieżna (pod warunkiem, że żaden z parametrów na liście nie należy do klasy **variable**) lub w obrębie procesu bądź innego podprogramu jako instrukcja sekwencyjna.

VHDL

PROCEDURY (2)

Przykład procedury:

Tester parzystości wektora A o automatycznie rozpoznawalnej długości i generujący dwa sygnały wyjściowe: P (gdy liczba jedynek jest parzysta) i N (gdy jest odwrotnie).

```
procedure test_par(
  signal A   : in std_logic_vector;
  signal P,N : out std_logic) is
  variable t : std_logic;    --  zmienna lokalna
begin
  t := '1';
  for i in A'range loop
    t := t xor A(i);
  end loop;
  P <= t;
  N <= not t;
end test_par;
```

VHDL

PROCEDURY (3)

- Poniżej do pamiętania i odczytu „obecnego” stanu przerzutnika wykorzystano port o rodzaju **inout**
- Jest to jedyna możliwość do wykorzystania w tej procedurze
- Nie można zastosować do tego celu ani sygnału wewnętrznego lub wyjścia **buffer** (niedopuszczalne w procedurze) ani zmiennej lokalnej (nie przechowuje wartości po wykonaniu procedury)

Przerzutnik JK wyzwalany zboczem narastającym, z asynchronicznym zerowaniem i ustawianiem

```
procedure FFJK1
     (signal C,J,K,R,S  :  in std_logic;
      signal  Q  :  inout std_logic;
      signal  QN :  out std_logic) is
   variable   JK :  std_logic_vector(0 to 1);
   variable   re :  boolean;
begin
   JK := J & K;
   re := rising_edge(C);
   if R = '1' then Q <= '0';
      elsif S = '1' then Q <= '1';
      elsif re then
         case JK is
            when "10" => Q <= '1';
            when "01" => Q <= '0';
            when "11" => Q := not Q;  -- ???
            when others  =>  null;
         end case;
   end if;
   QN <= not Q;
end FFJK1;
```

VHDL

PROCEDURY (4)

Do opisu przerzutnika JK można wykorzystać jego logiczne równanie funkcyjne:

```
procedure FFJK2
   (signal C,J,K,R,S :  in  std_logic;
    signal Q    :  inout std_logic;
    signal QN   :  out std_logic) is
   variable re  :  boolean;
begin
   re := rising_edge(C);
   if R = '1' then Q <= '0';
      elsif S = '1' then Q <= '1';
      elsif re then
         Q <= (J and not Q) or (not K and Q); -- równanie funkcyjne
   end if;
   QN <= not Q;
end FFJK2;
```

- Niektóre kompilatory nie akceptują używania wewnątrz procedur atrybutu `'event` ani predefiniowanej funkcji `rising_edge()`
- Można wtedy użyć funkcję `rising_edge()`, ale nie wewnątrz instrukcji `if-then-else`, lecz z odrębnym przypisaniem do zmiennej lokalnej, jak pokazano wyżej

VHDL

PROCEDURY (5)
Wywoływanie procedury

- Przy wywoływaniu procedury trzeba na liście parametrów zastąpić nazwy parametrów formalnych nazwami parametrów aktualnych

- (1) Przyporządkowanie niejawne (domyślne).
  Na przykład

```
FFJK2(zegar,a,b,reset,set,Q4,Q4not);
```

  Liczba i kolejność nazw musi ściśle odpowiadać deklaracji procedury.

- (2) Przyporządkowanie jawne: po lewej stronie symbolu przyporządkowania (=>) wpisuje się parametr formalny, a po prawej stronie parametr aktualny. Na przykład

```
FFJK2(C=>zegar,J=>a,K=>b,R=>reset,S=>set,Q=>Q4,QN=>Q4not);
```

  Ponieważ wszystkie przyporządkowania są opisane kompletnie, zatem kolejność ich wypisania może być dowolna.

VHDL

PRZYKŁADY OPISU (1)

Rejestry równoległe

Można wykorzystać model przerzutnika D i zamienić typ sygnałów wejściowych i wyjściowych ze skalarnego na złożony

```
-- Rejestr n-bitowy
entity regn is
  generic (n : positive := 8);   -- deklaracja n = 8
  port  (C,R : in std_logic;
           D : in std_logic_vector(n-1 downto 0);
           Q : out std_logic_vector(n-1 downto 0));
end regn;
architecture a1 of regn is
begin
  process (R, C)
  begin
    if  R = '1' then Q <= (others => '0');
      elsif rising_edge(C) then Q <= D;
    end if;
  end process;
end a1;
```

VHDL

PRZYKŁADY OPISU (2)

Rejestry równoległe

Uniwersalna procedura dla rejestru równoległego

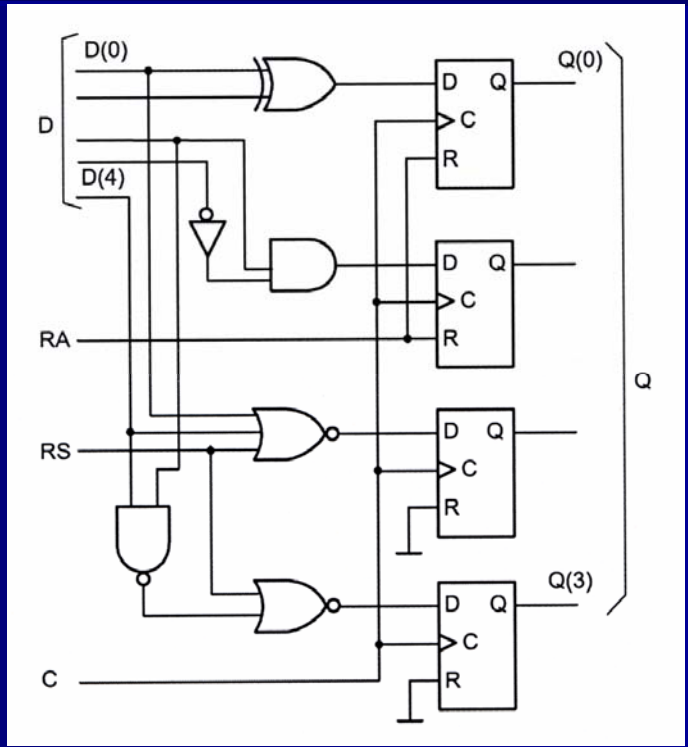
- Liczba bitów (n) rejestru nie jest jawnie deklarowana
- Do rozpoznania tej liczby stosuje się atrybut `'range`

```
procedure reg
   (signal C,R  : in std_logic;
     signal  D  : in std_logic_vector;
     signal  Q  : out std_logic_vector) is
   variable ff : boolean; --  niekonieczne
begin
   ff := rising_edge(C);     --  niekonieczne
   if R = '1' then Q <= (Q'range => '0');
   -- zerowanie wektora o dowolnej dlugosci
      elsif ff then Q <= D;
      -- zamiast ff mozna uzyc rising_edge(C)
   end if;
end reg;
```

VHDL

PRZYKŁADY OPISU (3)

Sieć bramek i rejestr



RA – zerowanie asynchroniczne

RS – zerowanie synchroniczne

```
entity siec is
    port (C,RA,RS  :  in std_logic;
                D  :  in std_logic_vector(4 downto 0);
                Q  :  out std_logic_vector(3 downto 0));
end siec;
```

VHDL

PRZYKŁADY OPISU (4)

Sieć bramek i rejestr – opis klasyczny (bez użycia procedury)

```
architecture a1 of siec is
    begin                          -- opcjonalne etykiety p0..p3 polepszaja czytelnosc opisu
      p0:  process                                    -- bez listy wrazliwosci, potrzebne wait
           begin
              if RA = '1' then Q(0) <= '0';           -- zerowanie asynchroniczne
                  else wait until rising_edge(C);     -- czekaj na zbocze zegara
                  Q(0) <= D(0) xor D(1);              -- wpisz sygnal z wyjscia bramki XOR
              end if;
           end process p0;
      p1:  process(C,RA)                              -- jest lista, nie ma wait
           begin
              if RA = '1' then Q(1) <= '0';
                  elsif rising_edge(C) then
                  Q(1) <= D(2) and not D(3);          -- wpisz sygnal z wyjscia bramki AND
              end if;
           end process p1;
      p2:  process
           begin
              wait until rising_edge(C);
              Q(2) <= not(D(0) or D(4) or RS);        -- zerowanie synchroniczne
           end process p2;
      p3:  process(C)
           begin
              if rising_edge(C) then
                  if RS = '1' then Q(3) <= '0';       -- zerowanie synchroniczne
                     else Q(3) <= D(2) and D(4);
                  end if;
              end if;
           end process p3;
    end a1;
```

VHDL

PRZYKŁADY OPISU (5)

Sieć bramek i rejestr – opis z użyciem procedury

- Proces p3 można zapisać krócej, stosując opis logiczny

```
p3: process(C)
begin
   if rising_edge(C) then
      Q(3) <= (not RS) and D(2) and D(4);
   end if;
end process p3;
```

- Wykorzystanie procedury
  - Wprowadzamy asynchroniczne wejścia zerujące w postaci wektora R
  - Zakładamy, że długości wektorów D, R i Q są takie same

```
procedure regr
  (signal C    :  in std_logic;
   signal R,D  :  in std_logic_vector;
   signal Q    :  out std_logic_vector) is
   variable rr :  boolean;
begin
   rr := rising_edge(C);
   for i in R'range loop
      if R(i)= '1' then Q(i) <= '0';
         elsif rr then Q(i) <= D(i);
      end if;
   end loop;
end regr;
```

VHDL

PRZYKŁADY OPISU (6)

Sieć bramek i rejestr

- Załóżmy że procedura ta jest zawarta w roboczym pakiecie pak, umieszczonym w bibliotece work.
- Aby ją wywołać, trzeba określić parametry aktualne (przechodzące) procedury.
- Stosując styl przepływowy (RTL) można utworzyć prostszy opis architektury pokazanego przykładu:

```
library ieee;                          -- klauzula standardowa
use ieee.std_logic_1164.all;           -- klauzula standardowa
use work.pak.all;    -- dodatkowo, jesli korzystamy z pakietu pak
entity siec is
   port (C,RA,RS  :  in std_logic;
               D  :  in std_logic_vector(4 downto 0);
               Q  :  out  std_logic_vector(3 downto 0));
end siec;
architecture a2 of siec is
   --  tutaj trzeba wpisac procedure regr
   --  jesli nie korzystamy z pakietu pak
   signal   w,z   :  std_logic_vector(3 downto 0);
begin
   w(0) <= D(0) xor D(1);
   w(1) <= D(2) and not D(3);
   w(2) <= not(D(0) or D(4) or RS);
   w(3) <= (not RS) and D(2) and D(4);
   z <= ('0','0',RA,RA);
   regr(C,z,w,Q);       -- wywolanie procedury z parametrami aktualnymi
end a2;
```

VHDL

PRZYKŁADY OPISU (7)

Styl strukturalny

- Opis strukturalny stanowi tekstowy odpowiednik schematu logicznego lub blokowego.
- Architektura opisuje sieć połączeń układów określanych jako komponenty.
- Komponenty są opisywane standardowymi modułami i deklarowane jako **component**.
- Aby opisać sieć „połączeń" sygnałów wyjściowych (typu **out**) komponentów z portami wejściowymi (typu **in**) komponentów trzeba wprowadzić sygnały pomocnicze.
- Do wejść komponentów można „przyłączyć" nie tylko sygnały, ale również wyrażenia lub wartość stałą, albo można określone wejścia pozostawić otwarte, przypisując im słowo **open**.
- Deklarację połączeń komponentów tworzy się wykorzystując współbieżną instrukcję łączenia („osadzania") komponentów **port map**

```
[etykieta:] nazwa_komponentu port map (lista);
```

- Na liście opisuje się połączenia przez odpowiednie przyporządkowania lub podstawienia.
- Stosując zapis jawny przyporządkowań stosuje się symbole przyporządkowania (=>):

```
port_wewnetrzny => sygnal_zewnetrzny | wyrazenie | open
```

  - wtedy kolejność wypisania przyporządkowań nie jest istotna,
  - można pominąć opis tych portów, których stany mają stałe wartości, zadeklarowane wcześniej
  - nazwy sygnałów pomocniczych mogą być takie same jak nazwy portów komponentów:
    - wtedy zapis wydaje się nieco prostszy,
    - lecz trzeba opisywać przyporządkowania takie jak C=>C i Q=>Q, co jest trudniejsze w interpretacji.

- Stosując zapis niejawny (podstawienia) pomija się nazwy portów wewnętrznych i symbol przyporządkowania
  - wtedy trzeba zachować kolejność i pozycje portów podane w deklaracji komponentów i opisać wszystkie porty.

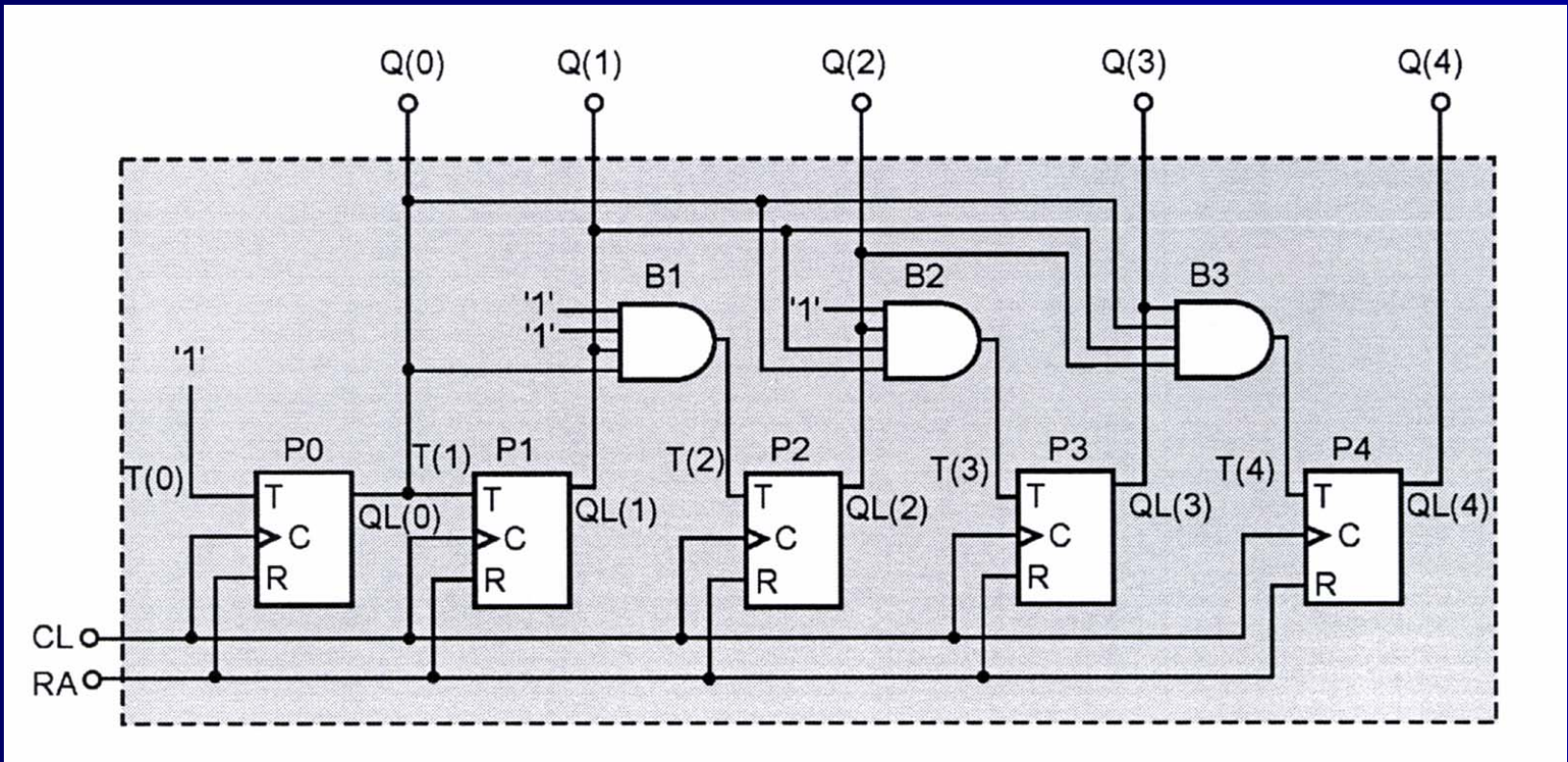
- Moduły komponentów mogą być wpisywane bezpośrednio przed ich deklaracją albo mogą być umieszczane w odpowiednich pakietach. Bardzo pomocne są pakiety prymitywów.

VHDL

PRZYKŁADY OPISU (8)

Styl strukturalny – opis licznika

Pięciobitowy licznik synchroniczny z przeniesieniami równoległymi



VHDL

PRZYKŁADY OPISU (9)

Styl strukturalny – opis licznika

Pięciobitowy licznik synchroniczny z przeniesieniami równoległymi

- Będą użyte dwa komponenty: czterowejściowa bramka AND i przerzutnik T.
- Komponenty są opisane na początku, czyli ich skompilowane opisy będą w bibliotece roboczej work.

```
-- czterowejsciowa bramka AND - komponent
library ieee;
use ieee.std_logic_1164.all;
entity and4 is
   port (A,B,C,D   :  in std_logic;
                Y   :  out  std_logic);
end and4;
architecture a1 of and4 is
begin
   y <= a and b and c and d;    -- moga byc male litery
end a1;
```

- Komponent przerzutnika T można opisać analogicznie jak na slajdzie 10

VHDL

PRZYKŁADY OPISU (10)

Styl strukturalny – opis licznika

Pięciobitowy licznik synchroniczny z przeniesieniami równoległymi

```
-- przerzutnik T - komponent
library ieee;
use ieee.std_logic_1164.all;
entity fft is
   port (T,C,R :  in std_logic;
               Q  :  out std_logic);
end fft;
architecture a1 of fft is
begin
   process(C,R)
      variable tt   :   std_logic;
   begin
      if  R = '1' then tt := '0';
        elsif rising_edge(C) then
        tt <= tt xor T;
      end if;
      Q <= tt;
   end process;
end a1;
```

VHDL

PRZYKŁADY OPISU (11)

Styl strukturalny – opis licznika

```
--  model licznika
library ieee;
use ieee.std_logic_1164.all;
   -- use work.pak.all; gdy komponenty sa w pakiecie pak
entity licznik5 is
   port(CL,RA  : in std_logic;
             Q : out std_logic_vector(4 downto 0));
end licznik5;
architecture struktura of licznik5 is
   --  deklaracje komponentow
   component and4
      port (A,B,C,D : in std_logic := '1'; -- inicjalizacja:
   --  domyslnie na wszystkich wejsciach jest stan '1'
                 Y : out  std_logic);
   end component;
   component fft
      port (T,C,R : in std_logic;
                Q : out std_logic);
   end component;
   --  deklaracja sygnalow pomocniczych
   signal T,QL :  std_logic_vector(4 downto 0);

-- cialo architektury licznika na nastepnym slajdzie
```

VHDL

PRZYKŁADY OPISU (12)

Styl strukturalny – opis licznika

```
begin
     --  opis polaczen przez przyporzadkowania jawne:
     T(0) <= '1';
     P0: fft port map (T=>T(0),C=>CL,R=>RA,Q=>QL(0));
     T(1) <= QL(0);
     P1: fft port map (T=>T(1),C=>CL,R=>RA,Q=>QL(1));
     --  na wejsciach C i D jest domyslny stan '1':
     B1: and4   (A=>QL(0),B=>QL(1),Y=>T(2));
     P2: fft port map (T=>T(2),C=>CL,R=>RA,Q=>QL(2));
     --  przyporzadkowania niejawne:
     B2: and4 port map (QL(0),QL(1),QL(2),'1',T(3));
     --  teraz trzeba bylo opisac wszystkie porty
     P3: fft port map (T(3),CL,RA,QL(3));
     B3: and4 port map (QL(0),QL(1),QL(2),QL(3),T(4));
     P4: fft port map (T(4),CL,RA,QL(4));
     Q <= QL;
end struktura;
```

VHDL

PRZYKŁADY OPISU (13)

Multiplekser

Przykład opisu w stylu przepływowym, funkcyjnym (por. slajd 34).

- Zaleca się stosować instrukcję współbieżną **with-select-when**, która implikuje realizację równoległej struktury logicznej.

```
-- Multiplekser 4-na-1
entity mux4 is
   port (a,b,c,d : in std_logic);
              adr  : in std_logic_vector(1 downto 0);
                y : out std_logic);
end mux4;
architecture a1 of mux4 is
begin
   with adr select
   y <= a when "00",
        b when "01",
        c when "10",
        d when others; -- albo: d when "11",
                       --       'X' when others;
end a1;
```

VHDL

PRZYKŁADY OPISU (14)

Multiplekser

Przykład opisu w stylu przepływowym, logicznym

```
architecture a2 of mux4 is
begin
  y <= a and (not adr(0)) and (not adr(1)) or
       b and adr(0) and (not adr(1)) or
       c and (not adr(0)) and adr(1) or
       d and adr(0) and adr(1);
end a2;
```

VHDL

PRZYKŁADY OPISU (15)

Multiplekser

- W opisie procedury (styl behawioralny) należy użyć instrukcji sekwencyjnej `case-is-when`:

```
procedure mux4(
  signal a,b,c,d : in  std_logic;
  signal     adr : in  std_logic_vector(1 downto 0);
  signal     y   : out std_logic) is
begin
  case adr is
    when "00" => y <= a;
    when "01" => y <= b;
    when "10" => y <= c;
    when others => y <= d; -- albo when "11" => y <= d;
  end case;                --      when others => y <= 'X';
end mux4;
```

VHDL

PRZYKŁADY OPISU (16)

Konwerter kodu

- Proste konwertery można opisać odpowiednią tablicą stanów logicznych
- Przykład procedury opisujacej konwerter kodu 1-z-10 na kod BCD8421 (koder)

```
procedure conv_to_BCD(
  signal ten : in std_logic_vector(9 downto 0);
  signal bcd : out std_logic_vector(3 downto 0)) is
begin
  case ten is
    when "0000000010" => bcd <= "0001";
    when "0000000100" => bcd <= "0010";
    when "0000001000" => bcd <= "0011";
    when "0000010000" => bcd <= "0100";
    when "0000100000" => bcd <= "0101";
    when "0001000000" => bcd <= "0110";
    when "0010000000" => bcd <= "0111";
    when "0100000000" => bcd <= "1000";
    when "1000000000" => bcd <= "1001";
    when others => bcd <= "0000";
  end case;
end conv_to_BCD;
```

VHDL

PRZYKŁADY OPISU (17)

Komparator

- Jednobitowy komparator można opisać funkcją XNOR:

```
entity komp1 is
   port(p,q  :  in std_logic;
            y  :  out std_logic);
end komp1;
architecture a1 of komp1 is
begin
   y <= p xnor q;
end a1;
```

- Do porównywania wektorów najprościej bezpośrednio sprawdzać relację równości `P = Q`:

```
procedure komp
   (signal  P :  in std_logic_vector;
    signal  Q :  in std_logic_vector;
            y :  out std_logic) is
begin
   if P = Q then y <= '1' else y <= '0';
   end if;
end komp;
```

VHDL

Opis strukturalny układów o powtarzalnych komponentach

Instrukcja `for-in-generate`

- Uproszczenie opisu strukturalnego złożonych układów
- Podstawowa instrukcja powielania

```
etykieta: for indeks in zakres generate
      [czesc deklaracyjna]
      [begin] {instrukcja wspolbiezna;}
      end generate [etykieta];
```

- Warunkowa instrukcja powielania

```
etykieta: if warunek generate
      [czesc deklaracyjna]
      [begin] {instrukcja wspolbiezna;}
      end generate [etykieta];
```

VHDL

PRZYKŁADY OPISU (18)

Licznik synchroniczny z przeniesieniami szeregowymi

- Liczba bitów licznika jest deklarowana jednorazowo w wierszu **generic** (w przykładzie N = 8).
- Komponentem jest przerzutnik T, do którego wejścia T jest dołączona trójwejściowa bramka AND.
- Sygnał T ma port **inout**, które jest również wejściem dla bramki w kolejnym komponencie.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity ffg is
   port (e1,e2,e3,C,R : in std_logic;
                    Q : out std_logic;
                    T : inout std_logic);
end ffg;
architecture a1 of ffg is          -- opis komponentu
   signal tt  :  std_logic;
begin
   T <= e1 and e2 and e3;          -- 3-wejsciowa bramka AND
   process (C,R)                   -- przerzutnik T
   begin
      if  R = '1' then tt <= '0';
        elsif rising_edge(C)then tt <= tt xor T;
      end if;
   end process;
      Q <= tt;
   end a1;
```

VHDL

PRZYKŁADY OPISU (19)

Licznik synchroniczny z przeniesieniami szeregowymi

```
library IEEE;
use IEEE.std_logic_1164.all;
entity licz is
   generic (N : positive := 8); -- liczba bitow licznika
   port(CL,RA,EN : in std_logic;
                Q : out std_logic_vector(N-1 downto 0));
end licz;
architecture gener of licz is
   component ffg port (e1,e2,e3,C,R  : in std_logic;
                                  Q  : out std_logic;
                                  T  : inout std_logic);
   end component;
   signal TL,QL : std_logic_vector(N-1 downto 0);
begin
   G0: ffg port map (e1=>EN,e2=>'1',e3=>'1',C=>CL, R=>RA,T=>TL(0),
          Q=>QL(0)); -- pierwszy przerzutnik
   G1: for i in 1 to N-1
        generate  --  powielanie
   G2: ffg port map (e1=>EN,e2=>QL(i-1),e3=>TL(i-1),C=>CL,R=>RA,T=>TL(i),
          Q => QL(i));
        end generate;
     Q <= QL;
   end gener;
```

VHDL

PRZYKŁADY OPISU (20)

Dzielnik częstotliwości przez 10 z symetrycznym przebiegiem wyjściowym

```
entity dzielnik_10 is
   port(ck  : in  std_logic;
        y   : out std_logic);
end dzielnik_10;
architecture a1 of dzielnik_10 is
   signal stan  : std_logic_vector (3 downto 0);
begin
   process (ck)
   begin
      if rising_edge(ck) then   -- kodowanie tablicy przejść
         case stan is           -- wykorzystanie stanow nieokreslonych
              when "1011" => stan <= "011X";   -- 2 stany
              when "011X" => stan <= "010X";   -- + 2 stany
              when "010X" => stan <= "000X";   -- + 2 stany
              when "000X" => stan <= "0010";   -- + 1 stan
              when "0010" => stan <= "0011";   -- + 1 stan = 8 stanow
              when "0011" => stan <= "111X";
              when "111X" => stan <= "110X";   -- na pozycji MSB jest 5 zer i 5 jedynek
              when "110X" => stan <= "100X";   -- co oznacza symetryczny przebieg
              when "100X" => stan <= "1010";   -- na wyjściu stan(3)
              when "1010" => stan <= "1011";
              when others => stan <= "1011";   -- podobnie 8 stanow
         end case;  -- razem 16 mozliwych stanow do optymalizacji projektu
      end if;
  end process;
  y <= stan(3);           -- WebPack 9.2i, Spartan-3A XC3S50A:
end a1;                -- 551 MHz, 2 slices, 4 FF, 3 LUT
```

VHDL

PRZYKŁADY OPISU (21)

Dzielnik częstotliwości przez 10 z symetrycznym przebiegiem wyjściowym

```
-- Algorytmiczny opis architektury
architecture a2 of dzielnik_10 is
  signal stan: std_logic_vector(3 downto 0):= "0000";
begin
  process (clk)
  begin
    if rising_edge(clk) then
        -- najpierw piec stanow od "X000" do "X100"
      if stan(2 downto 0) = "100" then
        -- nastepne piec stanow
        stan <= not stan(3) & "000";
        else stan <= stan + 1;
      end if;
    end if;
  end process;
  q <= stan(3);
end a2;

Inny zapis algorytmu:

     if stan(2 downto 0) < "100" then
        -- nastepne piec stanow
        stan <= stan + 1;
        else stan <= not stan(3) & "000";
      end if;
```

VHDL

PRZYKŁADY OPISU (22)

Automat o czterech stanach – detektor sekwencji stanów 01-01-10

```
entity auto4 is
  port (X : in  std_logic_vector(1 downto 0);
        ck : in  std_logic;
         y : out std_logic);
end auto4;
architecture b2 of auto4 is
  type stan is (S1, S2, S3, S4);  -- typ wyliczeniowy do zapisu czterech stanow
  signal S : stan;
begin
  process (ck, S)
  begin
    if rising_edge(ck) then
      case S is                -- opis tekstowy grafu (tablicy) przejsc
        when S1 => y <= '0';
          if X = "01" then S <= S2; else null; end if;   -- wykrycie stanu 01
        when S2 => y <= '0';
          if X = "01" then S <= S3; else S <= S1; end if;  -- kolejny stan 01 ?
        when S3 => y <= '0';
          if X = "10" then S <= S4;       -- i jeśli wystapi teraz stan 10 to bingo!
            elsif X = "01" then S <= S3; else S <= S1;
          end if;
        when S4 => y <= '1';      -- sekwencja wykryta!
          if X = "01" then S <= S2; else S <= S1; end if;
      end case;
    end if;
  end process;
end b2;
```