



Kierunek Elektronika i Telekomunikacja, zaoczne, IV rok
**Projektowanie Systemów Cyfrowych
w Językach Opisu Sprzętu**

Galopem przez VHDL

Program wykładu

Jednostki projektowe

`entity, architecture, configuration, package`

Pojęcia leksykalne

literały, identyfikatory, obiekty, wyrażenia

Opis strukturalny

`map, generate`

Instrukcje sekwencyjne

`process, wait, if, case, loop, next, exit, assert, function, procedure`

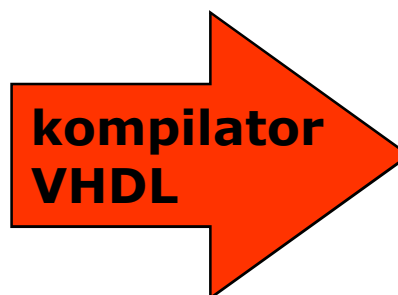
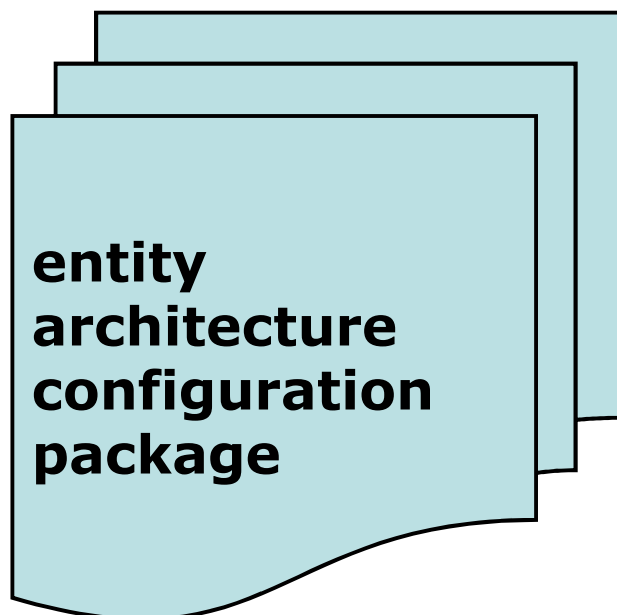
Instrukcje współbieżne

przypisania: bezwarunkowe, warunkowe i decyzyjne, podprogramy, `block`

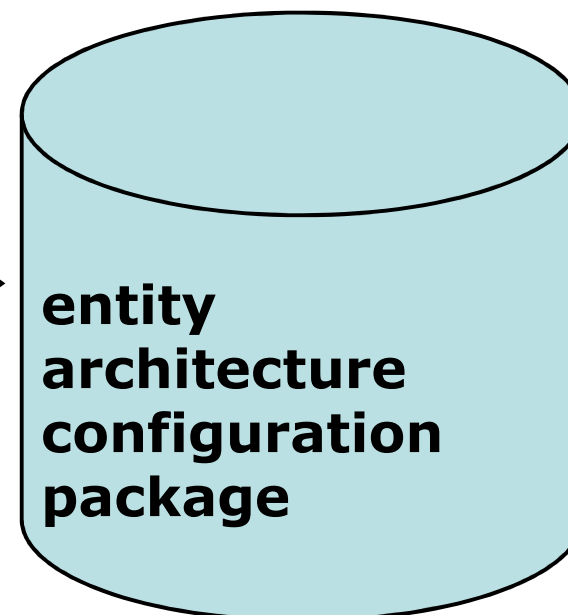
Typy złożone

tablice: jednowymiarowe i wielowymiarowe

Jednostki projektowe



Biblioteki





```
-- deklaracja entity definiuje komponent  
-- i jego połączenie ze światem zewnętrznym
```

```
entity ENTITY_NAME is
```

```
    port (
```

```
        < PORT_NAME: <mode> <type>; >
```

```
    );
```

```
end ENTITY_NAME;
```

```
-- deklaracja architecture definiuje sposób działania
```

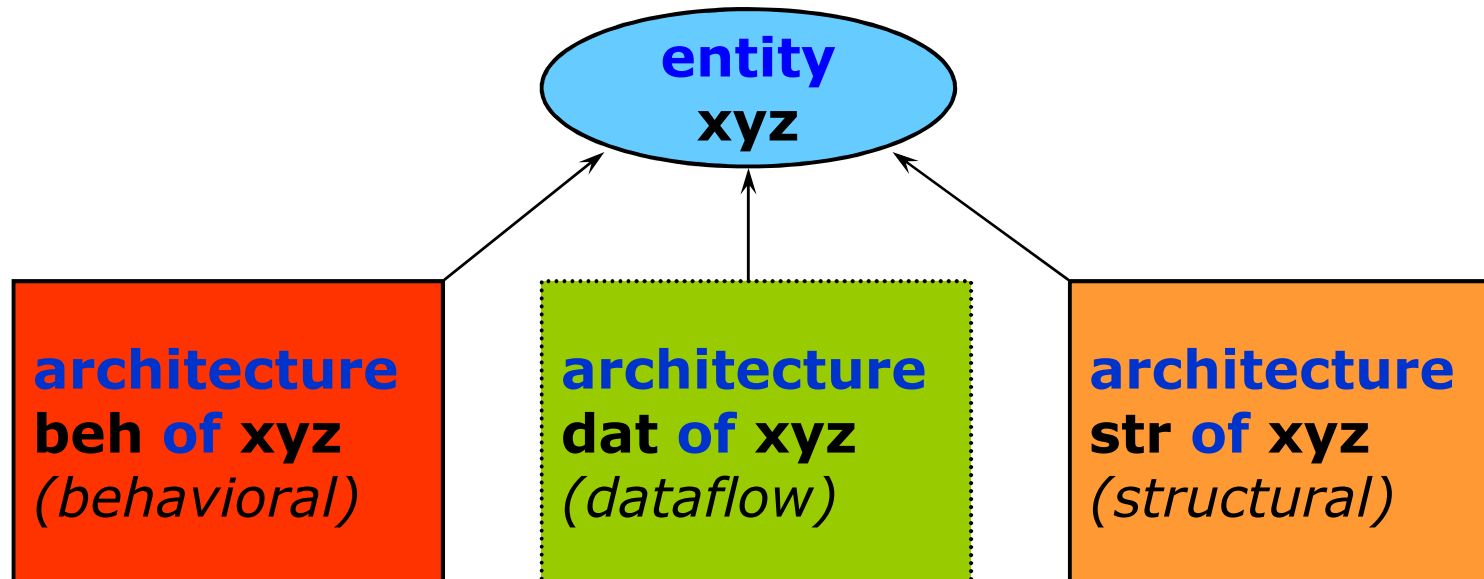
```
-- komponentu, do którego się odwołuje
```

```
architecture ARCH_NAME of ENTITY_NAME is
```

```
begin
```

```
    <statements>
```

```
end ARCH_NAME;
```



-- Przykład komparator

```
entity COMPARE is
  port (A,B: in bit;
        C: out bit);
end COMPARE;
```

```
architecture arch_behavioral of COMPARE is
begin
  process (A,B) -- A,B - zmienne "aktywne" w procesie
  begin
    -- sekwencyjne operatory przypisań
    if (A=B) then
      C <= '1' after 1 ns;
    else
      C <= '0' after 1 ns;
    end if;
  end process;
end arch_behavioral;
```

Zespół operacji sekwencyjnych, opisujących zachowanie modelu.



Jednostki projektowe

Deklaracja `architecture` – *dataflow style*

```
architecture arch_dataflow of COMPARE is
begin -- współbieżne operatory przypisań
    C <= not (A xor B) after 1 ns;
end arch_dataflow
```

Zespół współbieżnych przypisań, opisujących przepływ danych.

```
architecture arch_structural of COMPARE is
signal I: bit;
  component XOR2 port (x,y: in bit; z: out bit);
  end component;
  component INV port (x: in bit; z: out bit);
  end component;
begin
  U0: XOR2 port map (A,B,I);
  U1: INV port map (I,C);
end arch_structural;
```

Zespół połączonych elementów, opisujących strukturę modelu.

- pozwala na wybranie jednej z architektur dla danej `entity`
- stanowi wygodny sposób dokumentowania wersji projektu
- likwiduje konieczność rekompilacji całego projektu
gdy wymagana jest zmiana tylko kilku komponentów

```

configuration TESTBENCH_FOR_top of top_tb is
  for TB_ARCHITECTURE
    for UUT : top
      use entity work.top(structure);
    end for;
  end for;
end TESTBENCH_FOR_top;

```

Diagram illustrating the components of the `configuration` declaration:

- `TESTBENCH_FOR_top`: nazwa konfiguracji
- `top_tb`: konfigurowana entity
- `TB_ARCHITECTURE`: konfigurowana architektura
- `UUT : top`: konfigurowany komponent
- `work`: nazwa biblioteki
- `top`: wybrana architektura

Grupuje używane wspólnie deklaracje stałych, podprogramów, komponentów czy typów zmiennych.

```
package my_constans is  
    constant unit_delay: time := 1 ns;  
end my_constans;
```

```
Y <= '0' after work.my_constans.unit_delay;
```

```
package STANDARD
```

W każdej implementacji VHDL pakiet ten definiuje m.in. typy danych, jak `bit`, `boolean`, `bit_vector`, `character`, `string`, `text` itp.

- **literały**
napisy reprezentujące dane; ze sposobu ich zapisu wynikają ich wszystkie właściwości, w tym ich wartości
- **identyfikatory (nazwy)**
ciągi liter i cyfr, identyfikujące obiekty
- **obiekty**
sygnały, zmienne, stałe, parametry
- **wyrażenia**
wzory ujmujące operatory i argumenty, określające sposób obliczenia lub określenia wartości

Literały pojedyncze (skalary)

`character` - pojedynczy znak objęty apostrofami, np: `'A'` lub `'a'`
`bit` - reprezentuje wartość binarną `'1'` lub `'0'`
`std_logic` - reprezentuje wartość sygnałów wg. IEEE 1164:

`'U'` niezainicjalizowany (*uninitialized*)
`'X'` nieznany (*forcing an unknown*)
`'0'` silne zero (*forcing 0*)
`'1'` silne jeden (*forcing 1*)
`'Z'` wysoka impedancja (*Hi-Z*)
`'W'` słaby nieznany (*weak unknown*)
`'L'` słabe zero (*weak 0*)
`'H'` słabe jeden (*weak 1*)
`'-'` nieistotny (*don't care*)

boolean - reprezentuje dwie dyskretne wartości:

true TRUE True
false FALSE False

real - reprezentuje wartość zmiennoprzecinkową, np: **1.3** lub
-344.0E+23, typowo od **-1.0E+38** do **1.0E+38**

z precyzją co najmniej sześciu cyfr po przecinku

integer - reprezentuje wartość całkowitą, n.p.: **+1, 862** lub **-257**,
+123_456, 16#00FF#, typowo od **-2,147,483,647** do
+ 2,147,483,647

time - reprezentują jedyną predefiniowaną wielkość fizyczną,
to jest czas: **62 fs**, (**ps, ns, us, ms, sec, min, hr**)

Literały wielokrotne (tablice, wektory)

string - ciąg znaków objęty cudzysłowami, n.p.: **"x"**, **"hold time"**

bit_vector - **"0001_1100"**, **x"00FF"**

std_logic_vector - **"101Z"**, **"UUUUUU"**

Literały dla typów „numerycznych”:

Literały dziesiętne:

14
7755
156E7
188.993
88_670_551.453_909
44.99E-22

Literały o innych podstawach:

16#FE# -- 254
2#1111_1110# -- 254
8#376# -- 254
16#D#E1 -- 208
16#F.01#E+2 -- 3841.00
2#10.1111_0001#E9 -- 1506.00

Literały dla typów „fizycznych”:

b"11111110" - reprezentacja binarna
B"1111_1110" - równoważna reprezentacja binarna
x"FE" - równoważna reprezentacja szesnastkowa
O"376" - równoważna reprezentacja ósemkowa

Literały wielkości fizycznych:

60 sec 100 m 5 kohm 177 A

Identyfikatory podstawowe

Muszą zaczynać się od litery. Potem mogą następować litery, cyfry lub podkreślnik (*underscore*). Podkreślnik nie może być ostatni, ani nie może występować w sąsiedztwie innego podkreślnika.

VHDL nie rozróżnia wielkości liter (*case insensitive*): `XyZ` \Leftrightarrow `xyz`.

Identyfikatory nie mogą być takie jak słowa kluczowe (około 100).

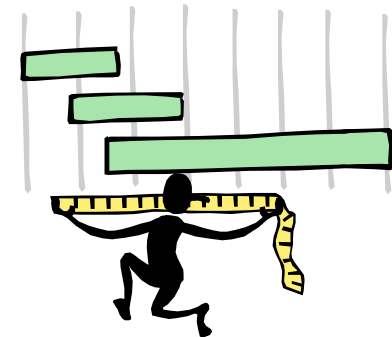
Przykłady: `XYZ`, `x3`, `s(3)`, `s(1 to 4)`, `my_defs`.

Zakres zmienności typu można ograniczać:

`range` {*low_val* **to** *high_val* | *high_val* **downto** *low_val*}

np: `integer range 1 to 10;`

`real range 1.0 to 10.0;`



Większość obiektów musi być deklarowana w sposób jawny.
Niektóre obiekty (np.: identyfikatory iteracji w pętli, sygnały powstałe z innych sygnałów w wyniku użycia atrybutów) deklarowane są w sposób domyślny.

Do deklaracji obiektów (ich nazwy i typu) należą deklaracje: stałych, zmiennych, sygnałów lub plików.

Deklaracje sygnałów

- **skalarnych:** `signal name(s): type[range][: = expression];`
- **tablicowych:** `signal name(s): array_type [(index)][: = expression];`
- **entity:** `port (name(s): direction type [range][: = expression]);`

Deklaracje zmiennych (w zakresie procesu)

- **skalarnych:**

`variable name(s): type[(range)][:= expression];`

- **tablicowych:**

`variable name(s): array_type [(range)][:= expression];`

np:

```
variable Index: integer range 1 to 50;
```

```
variable Cycle: time range 10ns to 50ns := 10ns;
```

```
variable MEMORY: bit_vector (0 to 7);
```

```
variable x,y: integer;
```

W VHDL'92 wprowadzono zmienne globalne do komunikacji pomiędzy procesami.



Argumenty wyrażeń muszą sobie odpowiadać
pod względem typów

Konwersje typów:

<code>integer (3.0)</code>	<code>integer</code>
<code>real (3)</code>	<code>real</code>
<code>integer * time</code>	<code>time</code>
<code>nanos + picos</code>	<code>time</code>
<code>nanos / picos</code>	<code>integer</code>

```
variable My_Data, My_Sample: integer;
```

```
...
```

```
My_Data := integer(74.94 * real(My_Sample));
```

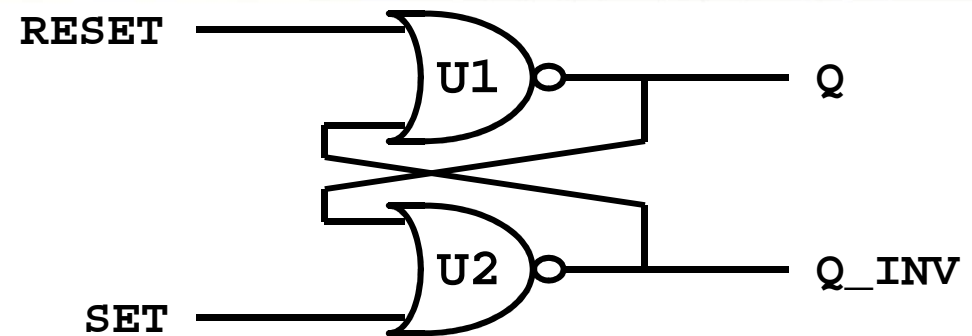
```
Vector <= CONV_STD_LOGIC_VECTOR(Integer_Variable);
```

Operatory wyrażeń:

logiczne	<code>and</code>	<code>or</code>	<code>nand</code>	<code>nor</code>	<code>xor</code>	<code>not</code>
relacji	<code>=</code>	<code>/=</code>	<code><</code>	<code><=</code>	<code>></code>	<code>>=</code>
połączenia	<code>&</code>					
arytmetyczne	<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>**</code>	
	<code>mod</code>	<code>rem</code>	<code>abs</code>			
VHDL'92	<code>sll</code>	<code>srl</code>	<code>sla</code>	<code>sra</code>	<code>rol</code>	<code>ror</code> <code>xnor</code>

Typy argumentów:

takie same	: <code>and or nand nor xor not</code> <code>= /= < <= > >= + - * /</code>
integer	: <code>mod rem</code>
integer exp	: <code>**</code>
numeryczny	: <code>abs</code>

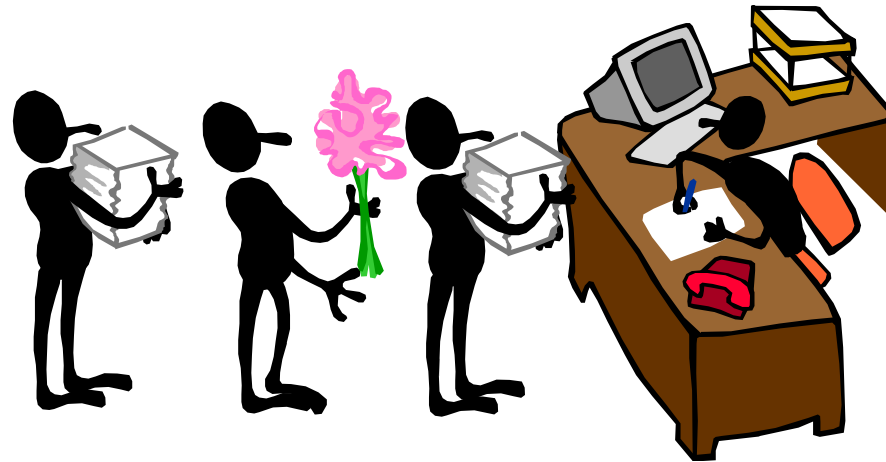


```

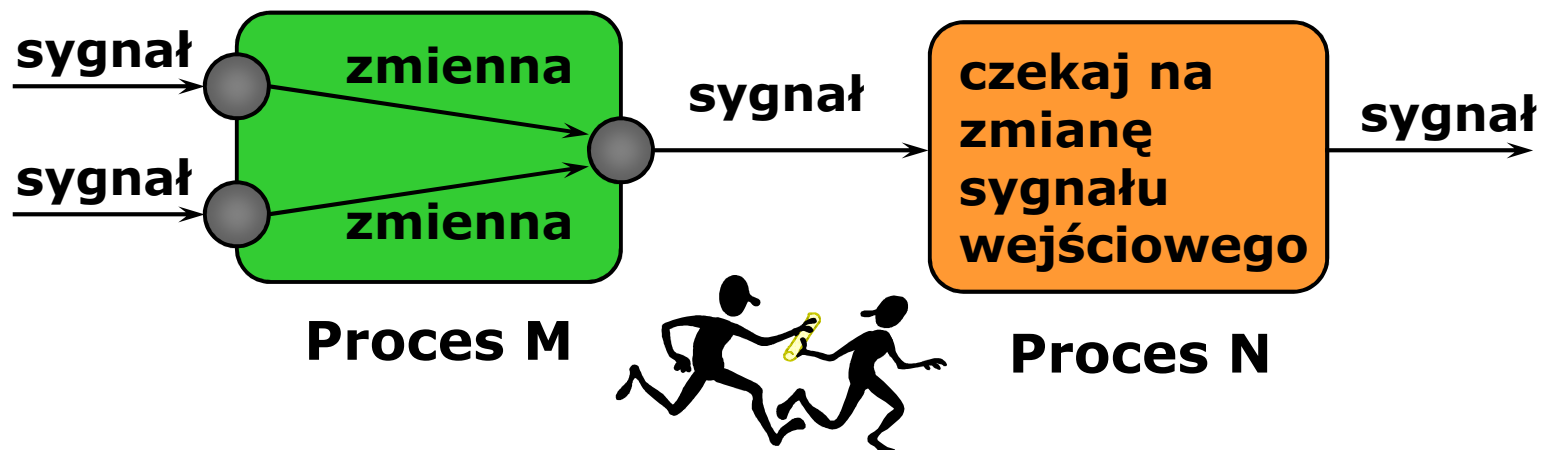
architecture STRUCT of RS_FLOP is
--deklaracja komponentu (component declaration)
  component NOR2 port (A,B: in bit; X: out bit);
  end component;
begin
--podstawienie komponentu (component instantiation)
  U1: NOR2 port map (RESET,Q_INV,Q);
  U2: NOR2 port map (Q,SET,Q_INV); --notacja pozycyjna
end STRUCT;

-- lub notacja specyfikacyjna, np:
-- U1: NOR2 port map (A => RESET, X => Q, B => Q_INV);
  
```

- **process** (*współbieżna!*)
- instrukcje przypisania
- **wait**
- **if**
- **case**
- `null`
- `loop`
- `next`
- `exit`
- `assert`
- `podprogramy`



- jest instrukcją współbieżną!
- definiuje część architektury, gdzie instrukcje interpretowane są sekwencyjnie,
- zawiera wyłącznie instrukcje sekwencyjne,
- musi zawierać albo listę sygnałów aktywujących, albo instrukcję **wait**,
- zapewnia możliwość “programowego” definiowania działania,
- ma możliwość zmian sygnałów zdefiniowanych w **architecture** i / lub **entity**





Instrukcje sekwencyjne

Instrukcja **process**

Instrukcja przypisania

Składnia:

[*etykieta* :]
process [(*lista sygnałów aktywujących*)]
[*podprogram*]
[*typ*]
[*stała*]
[*zmienna*]
[*inne deklaracje*]
begin
 instrukcje sekwencyjne
end process [*etykieta*];

Instrukcja przypisania wartości do zmiennych

zmienna := wyrażenie ;

Instrukcja przypisania wartości do sygnałów

*sygnał <= wyrażenie [**after** delay];*

Składnia:

```
wait [on sygnaty]  
      [until warunek]  
      [for wyrażenie_czas]
```

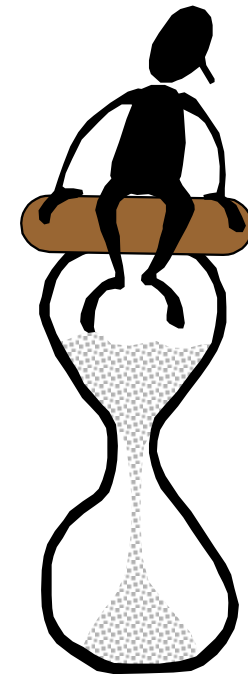
Przykłady:

```
wait on a,b; -- aktywacja procesu  
wait until x > 10;  
wait for 10 ns;  
wait; -- waits forever
```

Poniższe zapisy są równoważne:



```
process  
  ...  
  wait on a,b;    <=>    process (a,b);  
  ...  
end process;  
  ...  
end process;
```



Instrukcje sekwencyjne

Instrukcja **if**

Składnia:

```
if warunek then instrukcje_sekwencyjne ;  
    [elsif warunek then instrukcje_sekwencyjne] ;  
    [else instrukcje_sekwencyjne] ;  
end if ;
```

Przykład:

```
process (R, CLK)  
begin  
    if R = '0' then  
        operand(7 downto 0) <= "00000000";  
    elsif CLK = '1' and CLK'event then  
        operand(7 downto 0) <= DATAB;  
    end if ;  
end process ;
```

Szczególnie wygodna np. do dekodowania kodów, stanu automatu lub stanu magistrali.

Składnia:

case wyrażenie **is**

when *val*

 [**when** *val1* | *val2*

 [**when** *val3* **to** *val4*

 [**when** **others**

 => *instrukcje_sekwencyjne* ;

 => *instrukcje_sekwencyjne* ;

 => *instrukcje_sekwencyjne* ;

 => *instrukcje_sekwencyjne* ;

end case ;

Przykład:

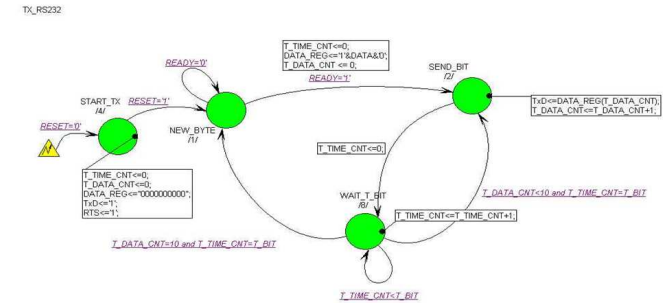
case BCD_int **is**

when 0 => LED <= "11111110" ;

when 1 => LED <= "0110000" ;

 ...

end case ;



Instrukcje współbieżne

- przypisania wartości sygnałom
 - bezwarunkowe (*unconditional*)
 - warunkowe (*conditonal*)
 - decyzyjne (*selected*)
- podprogramy
- block



W obydwu poniższych przykładach skutek przypisania jest ten sam:

```
architecture sequential of MULTIPLEXER is
begin
    process (A, INDEX)
    begin
        OUTPUT <= A (INDEX); -- sekwencyjne
    end process;
end sequential;
```

```
architecture concurrent of MULTIPLEXER is
begin
    OUTPUT <= A (INDEX); -- współbieżne
end concurrent;
```

Składnia:

sygnał <= {wyrażenie **when** warunek **else**} wyrażenie ;

Przykład:

```
DATA <= ROM when ADR < x"2000" else  
      RAM when ADR < x"6000" else  
      "ZZZZZZZZ" ;
```

Analogiczne do sekwencyjnej instrukcji **if**, ale:

- wykonują się bez uwzględnienia kolejności,
- używane w opisach typu *dataflow* i *structural*,
- syntezują się do logiki kombinacyjnej.

Uwaga! Nie można przypisana jak wyżej użyć wewnątrz **process**.

Składnia:

```
with wyrażenie select  
sygnał <= {wyrażenie when wartości,};
```

Przykład:

```
with digit select  
out <= '1' when 0 | 9,  
       '0' when 1 to 8,  
       'Z' when others;
```



Analogiczne do sekwencyjnej instrukcji **case**.

Uwaga! Nie można przypisana jak wyżej użyć wewnątrz **process**.

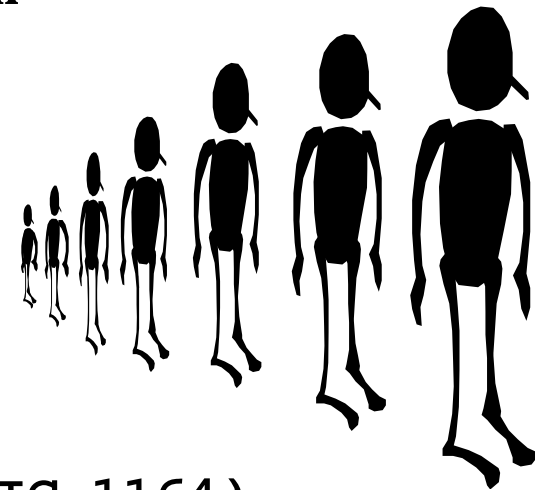
Zaawansowane typy danych

- Predefiniowane typy danych
- Typy rozszerzone (*Extended Types*)
 - Typy wyliczeniowe (*Enumerated Types*)
 - Podtypy (*Subtypes*)
- **Typy złożone (*Composite Types*)**
 - Tablice (*Arrays*)
 - *jednowymiarowe (wektory)*
 - *wielowymiarowe*
 - Rekordy (*Records*)
- Inne typy predefiniowane (*Other Predefined Types*)
 - Pliki (*Files*)
 - Linie (*Lines*)

- Tablice składają się z elementów tego samego typu.
- Są one używane do opisu magistral, rejestrów i innych zbiorów elementów sprzętowych.
- Elementy tablic mogą być skalarami lub elementami złożonymi. (Nie można tworzyć np. tablic plików !!)
- Dostęp do poszczególnych elementów dzięki użyciu wskaźników.

Jedynymi predefiniowanymi typami tablicowymi są:

- `bit_vector` (`package STANDARD`)
- `string` (`package STANDARD`)
- `std_logic_vector` (`package STD_LOGIC_1164`)



Użytkownik musi sam deklarować nowe typy tablic dla elementów `real` i `integer`.

Sposób dostępu zależy od sposobu deklaracji.

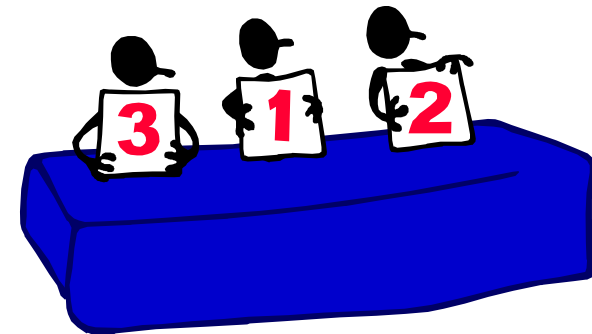
Przykłady:

```
variable c: bit_vector (0 to 3);
variable d: bit_vector (3 downto 0);
c := "1010";
d := c;
```

c(0) c(1) c(2) c(3)

1	0	1	0
---	---	---	---

d(3) d(2) d(1) d(0)



b(4 to 7)

dowolny zakres

c(4)

indeks poza zakresem

c(1.0)

błędny typ indeksu

Przykład: dla typu `bit_vector`:

<code>c := "1010";</code>	stała typu <code>bit_vector</code>
<code>c := x"A";</code>	j.w., uwaga na długość!
<code>c := S & T & M & W;</code>	4 połączone sygnały 1-bitowe
<code>c := ('1', '0', '1', '0');</code>	4-bitowy agregat
<code>c := 10;</code>	niedozwolony

Fragment tablicy (*slice*)

Przypisania mogą zachodzić pomiędzy fragmentami wektorów.

Przykłady:

```
variable a: bit_vector (3 downto 0);  
variable c: bit_vector (8 downto 1);  
c(6 downto 3) := a;
```

`c(6 downto 3)` a nie `c(3 to 6)`

- kierunek indeksów musi być
taki sam jak w deklaracji !

Literał tablicowy może zawierać listę elementów w notacji pozycyjnej i / lub specyfikacyjnej, tworząc tzw. agregat.

Składnia:

```
[type_name' ] ([choice =>] expression1 { , [others =>] expression2 }
```

Przykłady:

```
variable a,b: bit := '1';  
variable x,y: bit_vector (1 to 4);  
-- notacja pozycyjna  
x := bit_vector('1', a nand b, '1', a or b);  
-- notacja specyfikacyjna  
y := (1 => '1', 4 => a or b, 2 => a nand b, 3 => '1');
```

=> czytaj: “otrzymuje”

Przy użyciu notacji specyfikacyjnej, [*choice* =>] wskazuje na jeden lub kilka elementów.

[*choice* =>] może zawierać wyrażenie (n.p.: (*i mod 2*) =>), wskazujące na jeden element lub zawierać zakres (n.p.: *3 to 5* => lub *7 downto 0* =>), wskazujący na sekwencję elementów.

Notacja pozycyjna musi być użyta przed notacją specyfikacyjną.

Przykład:

```
variable b: bit;  
variable c: bit_vector (8 downto 1);  
c := bit_vector('1',b,5 downto 2 => '1',others => '0');
```

Bardzo wygodne:

```
Counter <= (others => '0');  
Data_Bus <= (others => 'Z');
```

Ciąg dalszy
nastąpi...

