



Galopem przez V...erilog

Jednostka projektowa

module, interface & body (style)

Pojęcia leksykalne

literały, identyfikatory, typy danych, wektory, wyrażenia (operandy i operatory):

operandy: stałe, parametry, sieci, zmienne, bity i pola bitowe, tablice i pamięci

operatory: arytmetyczne, logiczne, relacji, równości, bitowe, redukcyjne,
przesunięcia, warunkowe, połączenia, replikacji

Opis strukturalny

podstawienie, *generate*

Opis *data-flow*

przypisania ciągle: bezwarunkowe i warunkowe, podprogramy

Opis behawioralny

initial, always, if, case, casez, casex, begin, end,
repeat, while, ...



- **sygnały:**
obiekty, jak np. sieci czy zmienne
- **przypisania ciągłe:**
przypisania **assign** (współbieżne)
- **konstrukcje proceduralne:**
np. bloki proceduralne **initial** i **always**
- **bloki sekwencyjne:**
np. blok **begin** / **end**
- **instrukcje proceduralne:**
instrukcje w obrębie konstrukcji / bloków proceduralnych / sekwencyjnych

```
-- deklaracja module definiuje komponent:  
  
-- jego połączenie ze światem zewnętrznym  
module module_name (port_list)
```

Declarations:

reg, wire, parameter, input, output, inout, function, task ...

```
-- oraz jego funkcjonalność lub strukturę
```

Statements:

*initial statement, always statement, module instantiation,
gate instantiation, UDP instantiation, continuous assignments ...*

```
endmodule [module_name]
```

module module_name

port list

port declarations

data type declarations

INTERFACE

parameters

declarations of **wires**, **regs** and other variables

instantiation of lower-level modules or primitives

data flow statements (**assign**)

always and **initial** blocks

(all behavioral statements go into these blocks)

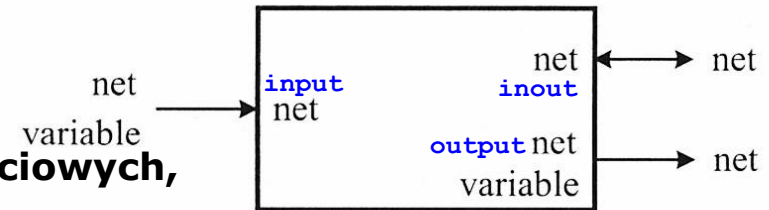
tasks and functions

endmodule [module_name]

BODY

Rodzaje portów i zasady ich łączenia:

- **input:**
 - deklaruje grupę sygnałów jako portów wejściowych,
 - muszą być deklarowane jako **wire**,
 - na zewnątrz mogą się łączyć z **wire** lub **reg**.
- **output:**
 - deklaruje grupę sygnałów jako portów wyjściowych,
 - mogą być deklarowane jako:
 - **wire** – dla modelowania strukturalnego / dataflow,
 - **reg** – dla modelowania behawioralnego,
 - na zewnątrz mogą się łączyć tylko z **wire**,
 - jeśli używane w **initial** lub **always**, muszą być zadeklarowane jako **reg**.
- **inout:**
 - deklaruje grupę sygnałów jako portów dwukierunkowych (mogą być używane jako wejścia albo wyjścia, ale nie jednocześnie).
 - muszą być deklarowane jako **wire**,
 - na zewnątrz mogą się łączyć tylko z **wire**,
 - jeśli używane w **initial** lub **always**, muszą być zadeklarowane jako **reg**.



Na potrzeby tego slajdu:

net ⇔ **wire**

variable ⇔ **reg**

Deklaracja interfejsu:

- lista portów,
- deklaracja portów (kierunku i ew. długości),
- deklaracja typów danych.

```
input  [net_type] [signed] [range] port names {, port names};
output [net_type] [signed] [range] port names {, port names};
output      [reg] [signed] [range] port names {, port names};
inout  [net_type] [signed] [range] port names {, port names};

output [port_var_type] port names; //port_var_type: integer or time
```

NOTICE

//port list style
/Verilog-1995

```
module adder(a,b,s);
input  [2:0] a,b;
output [3:0] s;
reg [3:0] s;
```

//port list style
//Verilog-2001

```
module adder(a,b,s);
input  [2:0] a,b;
output reg [3:0] s;
```

//port list
//declaration style
//Verilog-2001

```
module adder(
    input [2:0] a,b,
    output reg [3:0] s
);
// (ANSI C style)
```

▪ **Opis strukturalny:**

Zespół połączonych komponentów: modułów, UDP, podstawowych bramek lub podstawowych kluczy, opisujący strukturę modelu. Zapewnia mechanizm dla konstruowania hierarchicznych projektów dużych systemów cyfrowych.

- *Switch Level*: zespół połączonych kluczy tranzystorowych
- *Gate Level*: zespół połączonych bramek logicznych

▪ **Opis *data flow*:**

Zespół współbieżnych przypisań, opisujących przepływ danych między wejściami a wyjściami (dobrze nadaje się do opisu logiki kombinacyjnej).

▪ **Opis behawioralny:**

Zespół operacji sekwencyjnych, opisujących zachowanie modelu, nie wnikając w szczegóły implementacji sprzętowej.

NOTICE

Opis RTL modeluje synchroniczny obwód cyfrowy jako przepływy sygnałów między rejestrami oraz operacje, wykonywane na tych sygnałach.

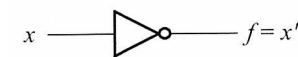
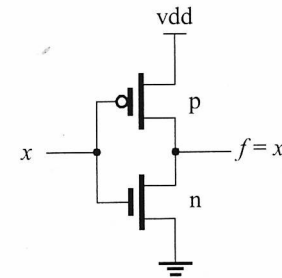
Poziomy abstrakcji opisu

Opis strukturalny (*circuit level – switch level*)

```

module mynot(output f, input x);
  //body declaration
  supply1 vdd;
  supply0 gnd;
  //not gate body
  pmos p1 (f, vdd, x);
  nmos n1 (f, gnd, x);
endmodule

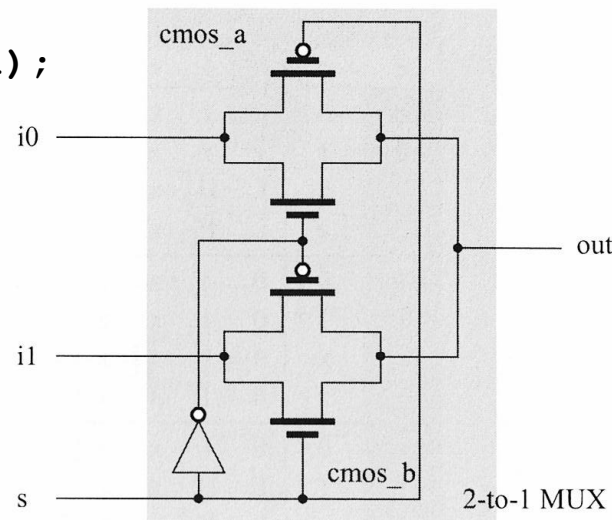
```



```

module mymux_switch(out, s, i0, i1);
  output out;
  input s, i0, i1;
  //body declaration
  wire sbar;
  mynot (sbar, s);
  //instantiate switches
  cmos (out, i0, sbar, s);
  cmos (out, i1, s, sbar);
endmodule

```



Zespół połączonych kluczy, opisujących strukturę modelu.

```
module mymux_gate(out, s, i0, i1);  
output out;  
input s, i0, i1;  
  
//body declaration  
wire sbar;  
wire a0, a1;  
  
//instantiate gates  
    not (sbar, s);  
    and (a0, i0, sbar);  
    and (a1, i1, s);  
    or (out, a0, a1);  
endmodule
```

Zespół połączonych elementów cyfrowych, opisujących strukturę modelu.

```
module mymux_flow(out, s, i0, i1);  
output out;  
input s, i0, i1;  
  
//continuous assignments  
    assign out = (s == 1'b0) ? i0 : i1;  
  
endmodule
```

Zespół współbieżnych przypisań, opisujących przepływ danych.

```
module mymux_behave (out, s, i0, i1);  
output out;  
input s, i0, i1;  
  
reg out;  
  
always @(s or i0 or i1)  
//procedural statements  
    if (s == 1'b0)  
        out = i0;  
    else  
        out = i1;  
endmodule
```

Zespół operacji sekwencyjnych, opisujących zachowanie modelu.

- **literały**
napisy reprezentujące dane; ze sposobu ich zapisu
wynikają ich wszystkie właściwości, w tym ich wartości
- **identyfikatory (nazwy)**
ciągi liter, cyfr i znaków, identyfikujące obiekty
- **typy danych**
sieci (*nets*), zmienne (*variables*)
- **wyrażenia**
wzory ujmujące operatory i operandy (argumenty),
określające sposób obliczenia lub określenia wartości

Wartości logiczne – logika 4-wartościowa

- 0** zero (*logic 0, false condition*)
- 1** jeden (*logic 1, true condition*)
- z** lub **Z** wysoka impedancja (*Hi-Z*)
don't care (nieistotna – w instrukcjach **casex** / **casez**)
- x** lub **X** nieistotna (*don't care – wejścia*)
nieznana (*unknown – wyjścia – konflikt*)

Trzy rodzaje literałów: *integer, real* i *string*.

Wartości *integer* mogą być określone w formatach:

- dziesiętnym (domyślnym)
albo:
- szesnastkowym,
- ósemkowym lub
- dwójkowym (binarnym),
wymagających określenia podstawy.

- **integer** - reprezentuje wartość całkowitą, max. 32-bitową n.p.:
 - dziesiętnie: **1**, **-862** lub **+257**,
 - binarnie: **4'b1101** lub **8'b 1101_0011**
 - szesnastkowo: **8'h7f**
- **real** - reprezentuje wartość zmiennoprzecinkową, np:
 - **1.3** - po obu stronach kropki dziesiętnej musi być cyfra
 - **1.44E-9** – można używać symbolu **E** lub **e**
- **string** - ciąg znaków, n.p.: **"x"**, **"hold time"**,
String – literał *unsigned integer*, reprezentowany przez sekwencję 8-bitowych kodów ASCII. Zmienne **string** są typu **reg**, o szerokości równej $8 \times$ liczba znaków w stringu.

Literały *integer*: notacja z podstawą systemu liczbowego

Literały *integer* w notacji z podstawą systemu liczbowego mogą się składać z pięciu części:

Składnia:

`[size] ' [s/S] [base_format] base_value`

- Rozmiar (opcjonalnie): niezerowa liczba dziesiętna bez znaku, określająca szerokość danej, wyrażoną w bitach .
- Apostrof.
- Informacja o znaku (opcjonalnie): litera – mała (**s**) lub duża (**S**).
- Znak podstawy:
 - dziesiętnej – **d** lub **D**,
 - szesnastkowej – **h** lub **H**,
 - ósemkowej – **o** lub **O**,
 - binarnej – **b** lub **B**.
- Wartość: cyfry **0..9** i **a..f**, właściwe dla wybranej podstawy. Cyfry szesnastkowe **a(A)** do **f(F)** są *case-insensitive*. Wartość może być poprzedzona spacją. **?** jest innym sposobem wyrażenia wartości **Z**.

Przykłady:

```
16'habcd // 16-bit hexadecimal number
      abc // unsigned 32-bit hexadecimal number
4'b1001 // 4-bit binary number
4'sb1001 // 4-bit signed binary number, = -7
-4'sb1001 // 4-bit signed binary number, = -(-7) = +7
```

Cała wartość może być poprzedzona spacją, a cyfry wartości mogą być rozdzielone podkreślnikami (_):

```
8'b 1001_0001 // 8-bit binary number
```

Wartości **x** oraz **z** rozszerzane są na większą liczbę bitów stosownie do podstawy systemu liczbowego:

```
8'h xd // = 8'b xxxx_1101
6'o z5 // = 6'b zzz_101
```

Jeśli deklarowany rozmiar stałej jest większy niż rozmiar wartości, to wartość jest rozszerzana w lewo zerami, bitem znaku lub znakami **x** albo **z**.

```
8'b 1001 // = 8'b 0000_1001
```

```
8'sb 1001 // = 8'b 1111_1001
```

```
8'b x001 // = 8'b xxxx_x101
```

```
8'b zz001 // = 8'b zzzz_z101
```

Jeśli deklarowany rozmiar stałej jest mniejszy niż rozmiar wartości, to wartość jest obcinana od lewej, by ją dopasować do deklarowanego rozmiaru.

```
4'b 1101_1001 // = 4'b 1001
```

Dla stałych bez zadeklarowanego rozmiaru rozmiar wartości wynosi domyślnie min. 32 bity.

```
'b 1001 // = 32'b 00000000000000000000_0000000000001001
```

```
'sb 1001 // = 32'b 11111111111111111111_1111111111111001
```

Identyfikatory

Mogą zawierać znaki alfanumeryczne (litery i cyfry), podkreślnik (*_ underscore*) i znak \$, a zaczynać się muszą od litery, podkreślnika lub \$.

Identyfikatory zaczynające się od \$ są zarezerwowane dla funkcji systemowych.

Verilog rozróżnia wielkości liter (*case sensitive*): **XyZ** $\neq$ **xyz**.

Identyfikatory nie mogą być takie same jak słowa kluczowe.

Przykłady: Counter, four_bit_adder, _4b_adder.

Komentarze

Komentarz w jednej linii:

// **One-line comment**

Komentarz blokowy:

/*

Block comment

*/

Coding style – use:

- lowercase for all names except:
- uppercase for constants and user-defined types
- meaningful names
- the same name for module and file
- the same name for signals throughout all the hierarchy levels

Dwie kategorie typów danych (nie są słowami kluczowymi):

- *nets* (sieci) – oznaczają węzły obwodu,
- *variables* (zmienne) – reprezentują elementy przechowujące wartość (od jednego przypisania do następnego).

Sieci (<i>nets</i>)		Zmienne (<i>variables</i>)
<code>wire</code>		<code>reg</code>
<code>tri</code>	<code>supply0</code>	<code>integer</code>
<code>wand</code>	<code>supply1</code>	<code>real</code>
<code>wor</code>	<code>tri0</code>	<code>time</code>
<code>triand</code>	<code>tri11</code>	<code>realtime</code>
<code>trior</code>	<code>triereg</code>	

Verilog-2005 LRM:

NOTICE — In previous versions of this standard, the term register was used to encompass the **reg**, **integer**, **time**, **real**, and **realtime** types, but that term is no longer used as a Verilog data category.

NOTICE

Sieci (*nets*):

- Reprezentują fizyczne połączenia między jednostkami strukturalnymi, takimi jak bramki czy moduły.
- Zasadniczo nie przechowują wartości (z wyjątkiem *triereg*).
- Są dostępne w każdym miejscu *module* .
- Muszą być sterowane przez:
 - *primitive* (bramki, klucze i UDP),
 - przypisanie ciągle *assign*,
 - wyrażenie *force* / *release*,
 - lub port w *module*.
- Jeżeli nie są wysterowane, posiadają wartość *z* (wyjątkiem jest *triereg*, która przechowuje ostatnio wysterowany stan).
- Najczęściej używany typ: *wire*

Deklaracja

wire [*signed*] [*[msb:lsb]*] *net_name* [, ...] ;

Zmienne (variables):

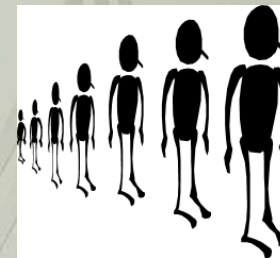
- przechowują wartość (od jednego przypisania do następnego),
- używane do modelowania elementów sprzętowych posiadających pamięć:
 - aktywnych zboczem (przerzutników),
 - aktywnych stanem (zatrząsków),
- **reg**, **integer** i **time** inicjalizowane są wartością **x**,
- najczęściej używany typ: **reg**

Deklaracja

reg [**signed**] [**msb:lsb**] *reg_name* [=reg_init] [, ...] ;

Przykłady deklaracji sieci i zmiennych:

```
wire w, x, y_in;                reg r, s, t_out;
wire [7:0] downto_bus;          reg [0:7] upto_bus;
wire signed [7:0] wir_sig;      reg signed [0:7] reg_sig;
```



- Wszystkie obiekty kategorii sieci oraz obiekty typu **reg** mogą być skalarami albo wektorami.
- Jeśli w deklaracji sieci lub zmiennej brak jest specyfikacji zakresu, to sieć zostaje skalarem.
- Jeśli określono szerokość (>1 bit), sieć zostaje wektorem.
- **[msb : lsb]** – określenie zakresu bitowego (szerokości) wektora.
- **msb** i **lsb** są wyrażeniami stałymi, nieujemnymi, typu **integer**.
- **msb** jest zawsze skrajnym lewym bitem, a **lsb** – skrajnym prawym, niezależnie od konwencji, która została użyta w deklaracji: **[high : low]** czy **[low : high]**,
- Możliwe są operacje na poszczególnych bitach lub fragmentach wektorów (*bit-select* i *part-select*).
- Obiekty typów **integer** oraz **time** nie mogą być deklarowane jako wektory, mimo tego mogą być dostępne w trybie *bit-select* i *part-select*.

Sposób dostępu zależy od sposobu deklaracji.

Przykłady:

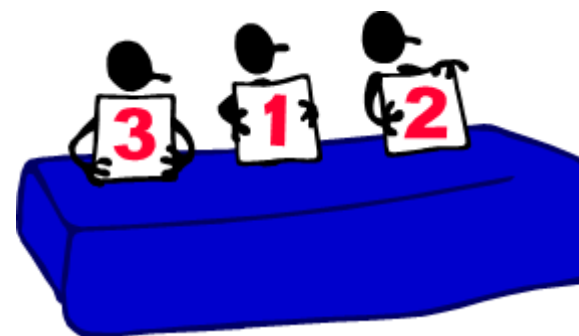
```
input  [0:3] t; //to
output [3:0] d; //downto
```

```
assign d = t;
```

t(0) t(1) t(2) t(3)

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

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



b(4 to 7)

indeksy nieujemne
dowolny zakres

t(4)

indeks poza zakresem

t(1.0)

błędny typ indeksu

- Wyrażenia składają się z operandów i operatorów.
- Wyrażenia mogą zawierać jeden, dwa lub więcej operandów połączonych jednym lub kilkoma operatorami.
- Przykłady:


```
x1 & x2 & x3;
(a & cin) | (b & cin) | (c & cin);
```

Operand	Komentarz	Operand	Komentarz
constant	<i>signed or unsigned</i>	bit-select	<i>one bit from a vector</i>
parameter	<i>signed or unsigned</i>	part-select	<i>cont bits from a vector</i>
net	<i>scalar or vector</i>	memory element	<i>one word of a memory</i>
variable	<i>scalar or vector</i>	function call	<i>system or user function</i>

Pojęcia leksykalne

Wyrażenia: operandy – stałe

- Stała (*constant*) może być ze znakiem lub bez znaku.
- Dziesiętna liczba całkowita (*integer*) jest traktowana jako liczba ze znakiem.
- Liczba całkowita określona za pomocą podstawy jest interpretowana bez znaku.

Constant	Comments
127	Signed decimal: Value = 8-bit binary vector: 0111_1111
-1	Signed decimal: Value = 8-bit binary vector: 1111_1111
-128	Signed decimal: Value = 8-bit binary vector: 1000_0000
4'b1110	Binary base: Value = unsigned decimal 14
8'b0011_1010	Binary base: Value = unsigned decimal 58
16'h1A3C	Hexadecimal base: Value = unsigned decimal 6716
16'hBCDE	Hexadecimal base: Value = unsigned decimal 48,350
9'o536	Octal base: Value = unsigned decimal 350
-22	Signed decimal: Value = 8-bit binary vector: 1110_1010
-9'o352	Octal base: Value = 8-bit binary vector 1110_1010 = unsigned decimal 234

- Parametr (**parameter**) nadaje wartość stałej.
- Wartość parametru nie może być zmieniana podczas symulacji.
- Służy do skalowania (parametryzacji) obiektów w projekcie.

```
module adder_8 (a, b, cin, sum);
```

```
parameter WIDTH = 8;
```

```
input [WIDTH-1:0] a, b;    // a & b are 8 bits (7:0)
```

```
input cin;                // cin is a scalar
```

```
output [WIDTH:0] sum;     // sum is 9 bits (8:0)
```



**Wartość przyporządkowana do elementu typu sieć (*net*)
jest interpretowana bez znaku:**

```
wire [7:0] bus_in, bus_out;  
...  
assign bus_in = -59;           // 1100_0101 = 197 unsigned  
assign bus_out = 16'hE0;       // 1110_0000 = 224 unsigned  
                                // (-32 signed?)
```

- Wartość zmiennej zadeklarowanej jako typ **reg** jest interpretowana jako liczba bez znaku.
- Wartość zmiennej zadeklarowanej jako typ **integer** jest interpretowana jako liczba ze znakiem, w kodzie U2.

```
reg [15:0] bus_out;
```

```
integer accu;
```

```
...
```

```
initial
```

```
    bus_out = -20; // = 1111_1111_1110_1100 signed  
                //   which is 65,516 unsigned
```

```
initial
```

```
    accu = -57;  
    // = 1111_1111_1111_1111_1111_1111_1100_0111 signed
```

- Bit (*bit-select*) – umożliwia wybranie pojedynczego elementu wektora.
Składnia: *vector_name* [*bit_select_expr*]
- Pole bitowe (*part-select*) – umożliwia wybranie ciągłego fragmentu wektora.
Składnia: *vector_name* [*msb_part_select_expr* : *lsb_part_select_expr*]

Przykłady:

```
reg    [15:0] data_bus;  
wire   [7:0]  op_code;  
integer count;  
data_bus [15]  //wybiera pojedynczy bit  
data_bus [7:0] //wybiera fragment  
op_code  [x+3] //wybiera pojedynczy bit  
           // o wyliczonym indeksie  
count    [5:0] //wybiera fragment obiektu integer
```

Przypisania mogą zachodzić pomiędzy fragmentami wektorów.

Przykłady:

```
reg [8:1] data_8dn;  
reg [4:0] slice_5dn;  
reg [1:3] slice_3up;
```

```
assign data_8dn[8:4] = slice_5dn;  
assign slice_3up = data_8dn[4:2];
```

Kierunek indeksów musi być taki sam jak w deklaracji !:

```
data_8dn[4:2] a nie data_8dn[2:4]
```

- Tablice służą do grupowania elementów tego samego typu w obiekty wielowymiarowe.
- W tablice mogą być łączone wszystkie typy obiektów (nie tylko **wire** i zmienne, jak w przypadku wektorów).
- Pamięć to jednowymiarowa tablica elementów typu **reg**.

Przykłady deklaracji:

```
reg mem_8x1 [7:0]; //pamięć 8 elementów 1-bitowych
                  //(skalarów)
reg [7:0] mem_1x8; //pamięć 1 słowa 8-bitowego
                  //(wektora)
reg [7:0] mem_4x8 [0:3]; //pamięć 4 słów 8-bitowych
                  //(wektorów)
reg array_2d [0:3][0:3]; //2-wymiarowa tablica elementów
                  //1-bitowych (skalarów)
```



Pojęcia leksykalne

Wyrażenia: operatory

Operator type	Operator Symbol	Operation	Number of Operands
Arithmetic	+	Add	Two or one
	-	Subtract	Two or one
	*	Multiply	Two
	/	Divide	Two
	%	Modulus	Two
Logical	&&	Logical AND	Two
		Logical OR	Two
	!	Logical negation	One
Relational	>	Greater than	Two
	<	Less than	Two
	>=	Greater than or equal	Two
	<=	Less than or equal	Two
Equality	==	Logical equality	Two
	!=	Logical inequality	Two
	===	Case equality	Two
	!==	Case inequality	Two

- **Logiczne (*logical*):**
 - Przyjmują wartości **1**, **0** lub **x**.
 - Jeśli operand przyjmuje wartość niezerową, np. **4'b1001**, to jego wartość logiczna jest **1** (*true*).
 - Jeśli operand przyjmuje wartość zerową lub jeśli bit w operandzie przyjmuje wartość **x** albo **z**, np. **4'b10x1**, to wartość logiczna operandu jest **0** (*false*).
- **Relacji (*relational*):**
 - Operandy **wire** i **reg** są traktowane jak liczby bez znaku.
 - Operandy **integer** i **real** są traktowane jak liczby ze znakiem.
 - Wartość **x** albo **z** w którymkolwiek operandzie zwraca **x**.
 - Jeśli operandy mają różne rozmiary, to mniejszy jest rozszerzany w lewo zerami.

- **Równości (*equality*):**
 - Operandy są porównywane bit po bicie.
 - Jeśli operandy mają różne rozmiary, to mniejszy jest rozszerzany w lewo zerami.
 - **Logical equality:**
 - zwraca **x**, jeśli którykolwiek operand ma bity o wartości **x** albo **z**.
 - **Case equality:**
 - zwraca **0**, jeśli obydwa operandy nie odpowiadają sobie dokładnie co do bitu i wartości,
 - zwraca **1**, jeśli obydwa operandy odpowiadają sobie dokładnie co do bitu i wartości.

`4'b10xz == 4'b10xz // => 0 false`

`4'b10xz === 4'b10xz // => 1 true`

Pojęcia leksykalne

Wyrażenia: operatory c.d.

Bitwise	&	AND	Two
		OR	Two
	~	Negation	One
	^	Exclusive-OR	Two
	^ ~ or ~ ^	Exclusive-NOR	Two
Reduction	&	AND	One
	~ &	NAND	One
		OR	One
	~	NOR	One
	^	Exclusive-OR	One
	~ ^ or ^ ~	Exclusive-NOR	One
Shift	<<	Left shift	One
	>>	Right shift	One
Conditional	? :	Conditional	Three
Concatenation	{ }	Concatenation	Two or more
Replication	{{ }}	Replication	Two or more

- Operacje logiczne bit do bitu na dwóch (z wyjątkiem negacji) operandach wektorowych – generują wektor wynikowy.
- Wartość **z** jest traktowana jak **x**.
- Ew. krótszy operand jest rozszerzany w lewo zerami.
- **& (and):** jeśli jeden z 2 bitów = **0**, to wynik = **0**;
 jeśli oba bity = **1**, to wynik = **1**;
 poza tym wynik = **x**.
- **| (or):** jeśli jeden z 2 bitów = **1**, to wynik = **1**;
 jeśli oba bity = **0**, to wynik = **0**;
 poza tym wynik = **x**.
- **^ (xor):** jeśli jeden bit = **1**, a drugi = **0**, to wynik = **1**;
 jeśli oba bity = **0** lub = **1**, to wynik = **0**;
 poza tym wynik = **x**.
- **~^ lub ^~:** jeśli jeden bit = **1**, a drugi = **0**, to wynik = **0**;
 (xnor) jeśli oba bity = **0** lub = **1**, to wynik = **1**;
 poza tym wynik = **x**.
- **~ (not):** jeśli bit = **1**, to wynik = **0**;
 jeśli bit = **0**, to wynik = **1**;
 poza tym wynik = **x**.



- **Redukcji (*reduction*):**
 - Operatory jedno argumentowe;
operand – wektor, wynik – skalar.
 - Bit po bicie, od prawej do lewej.
 - & (and), ~& (nand), | (or), ~| (nor), ^ (exor), ~^ lub ^~ (exnor)
 - Przykład: `assign reduce_result = & reduce_value;`
- **Przesunięcia (*shift*):**
 - *Logical shift left & right:* << >>
 - *Arithmetic shift left & right:* <<< >>>
 - Przesuwa lewy operand o liczbę bitów w prawym operandzie.
 - Wolne pozycje bitów uzupełnia zerami przy przesunięciu logicznym oraz przy przesunięciu arytmetycznym w lewo.
 - Przy przesunięciu arytmetycznym w prawo wolne pozycje bitów uzupełnia wartością MSB (bitem znaku).
 - Przykład: `assign shift_result = shift_value << 2;`



▪ Warunkowe (*conditional*):

- Składnia: *condition ? true_statement : false_statement*
- Współbieżny analog sekwencyjnych konstrukcji: **if** / **else** oraz **case**. Może być zagnieżdżony.
- Jeśli operandy mają różne rozmiary, to mniejszy jest rozszerzany w lewo zerami.

```
module mux4to1 (i, s, out_if, out_case);  
  input [3:0] i;  
  input [1:0] s;  
  output out_if, out_case;
```

```
// if-else form nesting
```

```
assign out_if = s[1] ? (s[0]?i[3]:i[2]) : s[0]?i[1]:i[0];
```

```
// case form nesting
```

```
assign out_case = (s == 0) ? i[0] :  
                  (s == 1) ? i[1] :  
                  (s == 2) ? i[2] :  
                  (s == 3) ? i[3] : 1'bz;
```

```
module mux4to1 (i, sel, out);  
    input [3:0] i;  
    input [1:0] sel;    // = 2#0x  
    output [1:0] out;  
  
    // returns x - logical equality  
    assign out[0] = (sel == 0) ? i[0] :  
                    (sel == 1) ? i[1] :  
                    (sel == 2) ? i[2] :  
                    (sel == 3) ? i[3] : 1'bz; // default  
                                              // not reached  
  
    // returns z - case equality  
    assign out[1] = (sel === 0) ? i[0] :  
                    (sel === 1) ? i[1] :  
                    (sel === 2) ? i[2] :  
                    (sel === 3) ? i[3] : 1'bz; // default  
                                              // reached  
  
endmodule
```

NOTICE

- **Połączenia (*concatenation*):**
 - Tworzy pojedynczy operand przez połączenie dwóch lub wielu operandów.
 - Operandy muszą mieć określony rozmiar (*sized*).
 - Operandami mogą być: skalarne sieci lub zmienne *reg*, wektorowe sieci lub zmienne *reg*, bity, pola bitowe lub stałe o określonym rozmiarze.
 - Składnia: { , }
 - Przykład: $y = \{a, b[2], c[3:0], 4'b1\};$
- **Replikacji (*replication*):**
 - Realizuje powtarzalne połączenie (*concatenation*).
 - Specyfikuje liczbę powtórzeń (stałe wyrażenie nie-0, nie-x i nie-z) oraz powtarzane wyrażenie (wymogi jak dla połączenia).
 - Składnia: {#_of_repetitions, {expression_1, ..., expression_N}}
 - Przykład: $y = \{a, \{4\{b[0]\}\}, c[1], 4'b1\};$



Pojęcia leksykalne

Wyrażenia: operatory – podsumowanie

Arithmetic	Bitwise	Reduction	Relational
+: add	~: NOT	&: AND	>: greater than
–: subtract	&: AND	: OR	<: less than
*: multiply	: OR	~ &: NAND	>=: greater than or equal
/: divide	^: XOR	~ : NOR	<=: less than or equal
?: modulus	~^, ~: XNOR	^: XOR	
** : exponent		~^, ^ ~ : XNOR	
	Case equality		Miscellaneous
Shift	===: equality	Logical	{ , }: concatenation
<<: left shift	!==: inequality	&&: AND	{c{ }}: replication
>>: right shift	Equality	: OR	? : conditional
<<<: arithmetic left shift	==: equality	!: NOT	
>>>: arithmetic right shift	!=: inequality		



Pojęcia leksykalne

Wyrażenia: operatory – pierwszeństwo

Operators	Symbols	Precedence
Unary	+ ! ~	Highest ↑
Exponent	**	
Multiply, divide, modulus	* / %	
Add, subtract	+ -	
Shift	<< >> <<< >>>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	& ~ & ^ ^ ~ ~	
Logical	&& 	
Conditional	?:	Lowest



module_name instance_name ([ports]) ;

Porty:

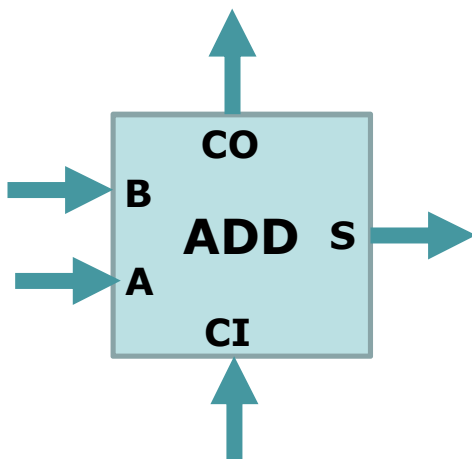
- **pozycyjne dopasowanie portów:**
 - *port_expr_1, ... , port_expr_n*
 - *porty zostają połączone wg ich kolejności na liście, na której zostały zadeklarowane przy tworzeniu modułu,*
 - *kolejność wymienienia portów ma istotne znaczenie,*
 - *porty niepołączone są omijane, np.: port_1, ,port_3*
- **nazwowe dopasowanie portów:**
 - *.port_id_1 (port_expr_1), ... , .port_id_n (port_expr_n)*
 - *porty zostają połączone poprzez wymienienie ich nazw, identyfikatory portów i odpowiadające im wyrażenia są połączone jawnie,*
 - *kolejność wymienienia portów nie ma znaczenia,*
 - *porty niepołączone są omijane albo wskazywane przez puste miejsce zamiast wyrażenia, np.: .port_id()*

Skojarzenie przez pozycję

```
module ADD_TB;
  reg a,b,ci;
  wire s,co;

  ADD SUM_INST (a,b,ci,s,co);

endmodule
```

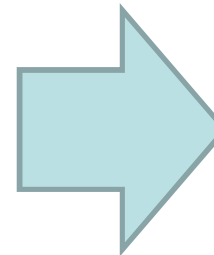


Skojarzenie przez nazwę

```
module ADD_TB;
  reg a,b,ci;
  wire s,co;

  ADD SUM_INST (
    .S(s),
    .CO(co),
    .A(a),
    .B(b),
    .CI(ci)
  );

endmodule
```



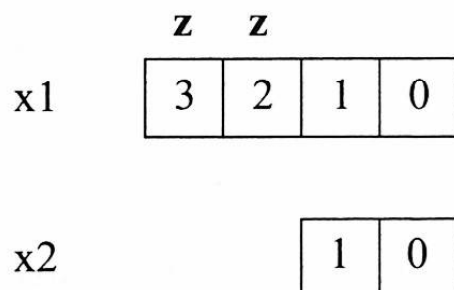
Porty:

- Mogą pozostać niepodłączone:
 - niepołączone porty wejściowe otrzymują wartość **z**,
 - niepołączone porty wyjściowe są nieużywane.
- Można łączyć porty o różnej szerokości bitowej:
 - poszczególne bity są wtedy dopasowane przez wyrównanie do prawej albo obcięcie.
- Niezależnie od metody dopasowania każde wyrażenie *port_expr* może być:
 - identyfikatorem (nazwą) sieci lub zmiennej,
 - pojedynczym bitem wektora,
 - fragmentem wektora,
 - złożeniem powyższych,
 - wyrażeniem (dla portów wejściowych).
- Obie metody dopasowania nie mogą być użyte jednocześnie dla tego samego modułu.
- Wszystkie prymitywy Verilogu (*Built-In* oraz *User-Defined*) mogą być podstawiane tylko skojarzeniem przez pozycję.

```

module name (x1, z1);
input [3:0] x1;
output [1:0] z1;
.
.
.
endmodule

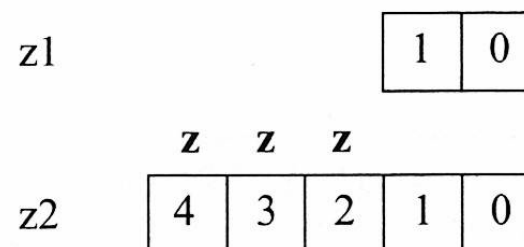
```



```

module top;
wire [1:0] x2;
wire [4:0] z2;
.
.
.
name inst1 (
.x1(x2),
.z1(z2)
);
endmodule

```



Instrukcje współbieżne Przypisanie ciągłe (*Continuous assignment*)

- **Składnia:**

assign *net_value* = *expression* ;

assign *net_1* = *expr_1* , ... , *net_n* = *expr_n* ;

- *net_value* – sieć skalarna lub wektorowa, lub ich połączenie,
- *expression* – sieci, zmienne lub wywołania funkcji,
- wartość *net_value* zmieni się, kiedy tylko zmieni się wartość *expression*,
- może sterować tylko siecią,
- wraz z innymi przypisaniami ciągłymi oraz z blokami proceduralnymi jest wykonywane współbieżnie – jego względna kolejność w tekście HDL jest nieistotna,
- może występować tylko poza blokami proceduralnymi,
- idealnie nadaje się do opisu działania funkcji kombinacyjnych.

- **Przykład:**

assign {*c_out*, *sum*[3:0]} = *a*[3:0] + *b*[3:0] + *c_in*;

Instrukcje współbieżne Przypisanie ciągłe (*Continuous assignment*)

- **Regular continuous assignment:**
`wire out;`
`assign out = in1 & in2;`
- **Net declaration assignment:**
`wire out = in1 & in2;`
 - Tylko jedno takie przypisanie na sieć, bo sieć może być zadeklarowana tylko raz.
- **Implicit net declaration:**
`wire in1, in2;`
`assign out = in1 & in2;`

NOTICE

- Domyślna deklaracja out jako `wire`.
- Dobra praktyka – deklarować jawnie.

Podobnie jak języki programowania, Verilog dostarcza trzy podstawowe grupy instrukcji:

- **instrukcje przypisania:**
 - przypisania ciągłe (*continuous assignments*): `assign`
 - przypisania proceduralne (*procedural assignments*):
 - przypisania blokujące (*blocking assignments*): `=`
 - przypisania nieblokujące (*nonblocking assignments*): `<=`
 - przypisania proceduralne ciągłe (*procedural continuous assignments*):
 - `assign / deassign, force / release`
- **instrukcje wyboru (warunkowe):**
 - `if / else, case / casex / casez`
- **instrukcje iteracyjne (pętle):**
 - `repeat, for, while, forever`

- Język HDL musi mieć możliwość uchwycenia i opisania podstawowych cech sprzętu: ciągłości i współbieżności.
- W Verilogu możliwość ta jest zapewniona dzięki konstrukcjom proceduralnym – blokom **initial** oraz **always**.
- Przy braku mechanizmów sterowania czasem symulacja nie będzie postępować.
- Sterowanie czasem to sposób na ustalanie momentów w czasie symulacji, w których wykonywane są instrukcje proceduralne.
- Verilog dostarcza dwie metody sterowania czasem:
 - sterowanie czasem przez opóźnienia (*delay timing control*),
 - sterowanie czasem przez zdarzenia (*event timing control*).

- Bloki **initial** są wykorzystywane do inicjalizacji, ustawiając wartości początkowe dla sieci i zmiennych.
- Bloki **always** pozwalają na modelowanie powtarzalnych operacji, wymaganych w modułach sprzętowych.
- Wszelkie konstrukcje / instrukcje sekwencyjne (= proceduralne, behawioralne) mogą występować wyłącznie w obrębie tych bloków.
- Bloki te wraz z przypisaniami ciągłymi są wykonywane współbieżnie – ich względna kolejność w opisie HDL jest nieistotna.
- Aktywność każdego z tych bloków zaczyna się w chwili 0.
- W jednym module może być opisana dowolna liczba bloków proceduralnych.

Blok *initial* jest w symulacji wykonywany jednokrotnie.

Składnia: *initial* [*timing_control*] *procedural_statements*

- *timing_control* – *delay control* lub *event control*
- *procedural_statements*:
 - *procedural_assignment*
 - *selection_statement*
 - *case_statement*
 - *loop_statement*
 - *wait_statement*
 - *event_trigger_statement*
 - *task_enable*
 - *procedural_continuous_assignments*
 - *disable_statement*
 - *sequential_block*
 - *parallel_block*

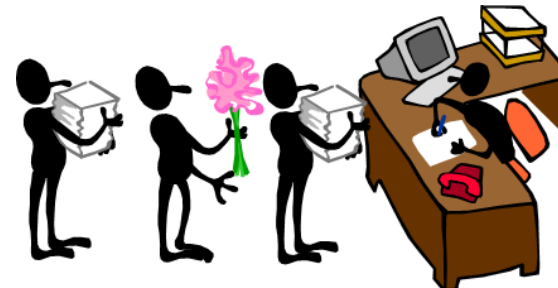
Przykład:

- Zastosowanie sterowania czasem przez opóźnienia.
- Jeśli specyfikacja opóźnienia *#delay* poprzedza przypisanie proceduralne, to zostanie ono wykonane po czasie *delay* od bieżącego momentu symulacji.
- Wykonywanie wszystkich bloków zaczyna się w chwili 0 (współbieżnie) i kończy się dla każdego bloku niezależnie.
- Dla przypisań złożonych użyto blok sekwencyjny *begin* / *end*
- Jeśli >1 blok proceduralny steruje tym samym obiektem, to wykonywane jest ostatnie przypisanie w czasie.
- Jeśli >1 blok proceduralny steruje tym samym obiektem w tym samym czasie, to wykonywane jest ostatnie przypisanie w module.

```
reg x,y;  
initial  
    x = 1'b0;           //single statement  
                        //finish @ time 0  
  
initial  
    #10 x = 1'b1;       //single statement  
                        //finish @ time 10  
  
initial  
    begin               //complex statement  
        #10 y = 1'b0;  
        #20 y = 1'b1;  
    end                 //finish @ time 30
```

- Blok sekwencyjny grupuje działania, które mają być wykonane sekwencyjnie.
- Dane działanie w obrębie bloku nie jest wykonywane dopóty, dopóki nie zostanie wykonane działanie poprzednie (wyjątkiem są przypisania nieblokujące).
- Sterowanie jest przekazywane z bloku sekwencyjnego dalej dopiero po zakończeniu jego wykonywania.
- Blokowi można nadać identyfikator (przydatny przy symulacji).
- Jeśli sekwencja składa się tylko z jednego działania, to blok sekwencyjny **begin** / **end** można pominąć.

- Składnia:
begin [: *block_idenfier*]
 sequential_procedural_statements
end



- *Regular declaration and initialization:*

```
reg carry_out;  
initial carry_out = 0;
```

- *Single statement combined declaration and initialization :*

```
reg carry_out = 0;
```

- Tylko na poziomie modułu.

- *Port list style:*

```
module adder (carry_out, ...)  
output reg carry_out = 0;
```

- Ten sam schemat, ale dla portów.

- *Port list declaration style (ANSI C style):*

```
module adder (output reg carry_out = 0, ...);
```

- Podobnie, dla portów.

NOTICE

Blok **always** jest w symulacji wykonywany wielokrotnie.

Składnia: **always** [*timing_control*] *procedural_statements*
(tak samo jak dla bloku **initial**)

Przykład: (por. Continuous assignment)

```
module mymux_beh(out, s, i0, i1);  
  output out;  
  input s, i0, i1;  
  
  always @(s or i0 or i1) //<= event timing control  
    //procedural statement(s):  
      out = (~s & i0) |  
            ( s & i1);  
  
endmodule
```

Instrukcje sekwencyjne

Przypisanie proceduralne (*Procedural assignment*)

Składnia:

[timing_control] variable [= or <=] expression ;

variable [= or <=] [timing_control] expression ;

var_value – zmienna skalarna lub wektorowa, lub ich połączenie,

- *expression* – sieci, zmienne lub wywołania funkcji,
- wartość *var_value* zmieni się, kiedy blok proceduralny zostanie aktywowany (*delay / event timing control*),
- może zapisywać tylko zmienną, tablicę (pamięć), operandy *bit-select* oraz *part-select*, lub połączenie powyższych,
- może występować tylko wewnątrz bloków proceduralnych,
- może być blokujące lub nieblokujące,
- idealnie nadaje się do opisu działania funkcji sekwencyjnych.

Przykład: (por. [Continuous assignment](#))

```
always @(a or b or c_in)
```

```
{c_out, sum[3:0]} = a[3:0] + b[3:0] + c_in;
```

Przypisania ciągłe		Przypisania proceduralne
sieć	obiekt	zmienna
assign	zapis	= lub <=
poza blokiem proceduralnym	miejsce	w bloku proceduralnym

Przypisania blokujące

- wykonywane zgodnie z porządkiem zapisu, blokują następne
- sposób na modelowanie logiki kombinacyjnej w bloku sekwencyjnym
- operator =

always begin

```
x = #5 1'b0; // x will be assigned 0 @ time 5  
y = #3 1'b1; // y will be assigned 1 @ time 8  
z = #6 1'b0; // z will be assigned 0 @ time 14
```

end

Przypisania nieblokujące

- wykonywane bez blokowania następnych w bloku sekwencyjnym,
- sposób na modelowanie wielu współbieżnych transferów danych, wywołanych wspólnym zdarzeniem
- operator <=

always begin

```
x <= #5 1'b0; // x will be assigned 0 @ time 5  
y <= #3 1'b1; // y will be assigned 1 @ time 3  
z <= #6 1'b0; // z will be assigned 0 @ time 6
```

end

Instrukcje sekwencyjne Przypisania blokujące i nieblokujące

```
initial Q = 0;
```

```
always @(posedge CLK)
begin
    Q = Q + 1; //blocking
    if (Q == 9)
        Q = 0; //blocking
end
```

```
initial Q = 0;
```

```
always @(posedge CLK)
begin
    Q <= Q + 1; //non-blocking
    if (Q == 9)
        Q <= 0; //non-blocking
end
```

Pytanie:

Jak będą liczyły powyższe liczniki ?

- **Składnia:**

- **if** (*condition*) *true_statement*
- **if** (*condition*) *true_statement*
else (*condition*) *false_statement*
- **if** (*condition_1*) *true_statement_1*
else if (*condition_2*) *true_statement_2*
[**else** *false_statement*]
- **Warunek** (*condition*) – zawsze w nawiasach ().
- **Działanie** (*statement*) może być pojedyncze lub być blokiem sekwencyjnym **begin** / **end**.
- Instrukcja zasadniczo używana do wyboru dwutorowego, ale może być zagnieżdżana, więc może być używana także do wyboru wielotorowego.
- W modelowaniu ciągłym funkcjonalność możliwa do zrealizowania za pomocą operatora warunkowego.

Składnia:

```
case (case expression)  
  (case_item1a) { ,case_item1b } : procedural_statement1  
  ...  
  (case_itemNa) { ,case_itemNb } : procedural_statementN  
  [default : procedural_statement]  
endcase
```

- Dobra praktyka – dopisywanie instrukcji **default**, nawet jeśli wszystkie możliwe przypadki są uwzględnione.
- Porównuje wyrażenia *case expression* oraz *case item* bit do bitu, uwzględniając wszystkie możliwe wartości (**0**, **1**, **x**, **z**).
- Instrukcja **casez** traktuje wartości **z** jako nieistotne, zaś instrukcja **casex** jako nieistotne traktuje wartości **x** oraz **z**. Pozwala to na ew. skrócenie listy wyrażen *case item* i efektywną konstrukcję koderów / dekoderów priorytetowych.
- W modelowaniu ciągłym funkcjonalność możliwa do zrealizowania za pomocą operatora warunkowego.



Instrukcje sekwencyjne Sterowanie czasem przez opóźnienia

- Sterowanie czasem (symulacji) przez opóźnienia polega na określaniu przedziału czasu pomiędzy (symulowanym) momentem napotkania instrukcji przez symulator, a (symulowanym) momentem jej wykonania.
- Składnia:
 # *delay_value*
- Sterowanie czasem przez opóźnienia – dwa typy:
 - Sterowanie czasem pomiędzy instrukcjami / Regularne (*Inter-assignment / Regular*)
 - Sterowanie czasem wewnątrz instrukcji (*Intra-assignment*)



- Odracza obliczenie wartości i wykonanie instrukcji przypisania na określoną liczbę jednostek czasu.
- Składnia: *#delay procedural_statement*
delay : stała lub wyrażenie
- Jeśli instrukcja nie jest podana, *#delay* powoduje tylko zwłokę o określony czas przed wykonaniem kolejnej instrukcji.
- Zapis:
 #25;
 x = a + 6;
jest równoważny:
 #25 x = a + 6;

- W poniższym przykładzie operator `<=` może być zastąpiony przez operator `=` bez wpływu na wyniki, ponieważ sterowanie czasem pomiędzy instrukcjami opóźnia wykonanie całej instrukcji.

```
reg a,b,c;  
integer d,e;  
integer count;
```

```
#25  b <= ~c;           //execute @ time 25  
#15  count <= count + 1; //execute @ time 40  
#((d+e)/2) a = b;       //execute @ time 40 + (d+e)/2  
#c   c = c + 1;         //execute @ time 40 + (d+e)/2 + c
```

- Odracza wykonanie instrukcji przypisania wartości do zmiennej na określoną liczbę jednostek czasu, ale wartość do przypisania obliczana jest w bieżącym momencie symulacji.
- Składnia: $variable = \#delay\ expression$
 $variable \leq \#delay\ expression$
 $delay$: stała lub wyrażenie
- Wszystkie wartości do przypisania są obliczane w chwili 0:

```
b = #25 ~c;           //execute @ time 25  
count = #15 count + 1; //execute @ time 40
```

```
b <= #25 ~c;           //execute @ time 25  
count <= #15 count + 1; //execute @ time 15
```

- Instrukcje proceduralne mogą być wykonywane synchronicznie ze zdarzeniami. Zdarzenie (*event*) to zmiana wartości sieci lub zmiennej (albo pojawienie się zdarzenia zadeklarowanego – *named event*).
- Sterowanie przez zdarzenia odracza wykonanie instrukcji do momentu pojawienia się danego zdarzenia.
- Dwa rodzaje sterowania zdarzeniami:
 - Sterowanie aktywowane zboczem (*edge-triggered*)
 - Sterowanie aktywowane stanem (*level-sensitive*)



- Instrukcje proceduralne mogą być wykonywane synchronicznie ze zdarzeniami. Zdarzenie (*event*) to zmiana wartości sygnału (sieci lub zmiennej) albo pojawienie się zdarzenia zadeklarowanego – *named event*).
- Sterowanie aktywowane zboczem odracza wykonanie instrukcji do momentu pojawienia się określonej zmiany danego sygnału (sieci lub zmiennej).
- Oznaczenie sterowania aktywowanego zboczem wykorzystuje symbol @ i ma następującą postać:

@ *event procedural_statement*

gdzie działanie proceduralne jest wykonywane za każdym razem kiedy pojawi się zdarzenie.

Specyfikatory zboczy:

- **edge** – zbocza: 01 0x x0
(z = x) 10 1x x1
- **posedge** – zbocza: => tabela
- **negedge** – zbocza: => tabela

np.: @ (**posedge** clk)

From	To			
	0	1	x	z
0		posedge	posedge	posedge
1	negedge		negedge	negedge
x	negedge	posedge		
z	negedge	posedge		

Wiele zdarzeń i sygnałów może powodować wykonanie instrukcji proceduralnej. Wszystkie konieczne powinny znaleźć się na liście wrażliwości (= liście zdarzeń).

- *keyword or*

```
always @(a or b or c_in)
    {c_out, sum} = a + b + c_in;
```

- *comma ,*

```
always @(a, b, c_in)
    {c_out, sum} = a + b + c_in;
```

- *implicit*

```
always @(*) // lub @*
    {c_out, sum} = a + b + c_in;
```

NOTICE

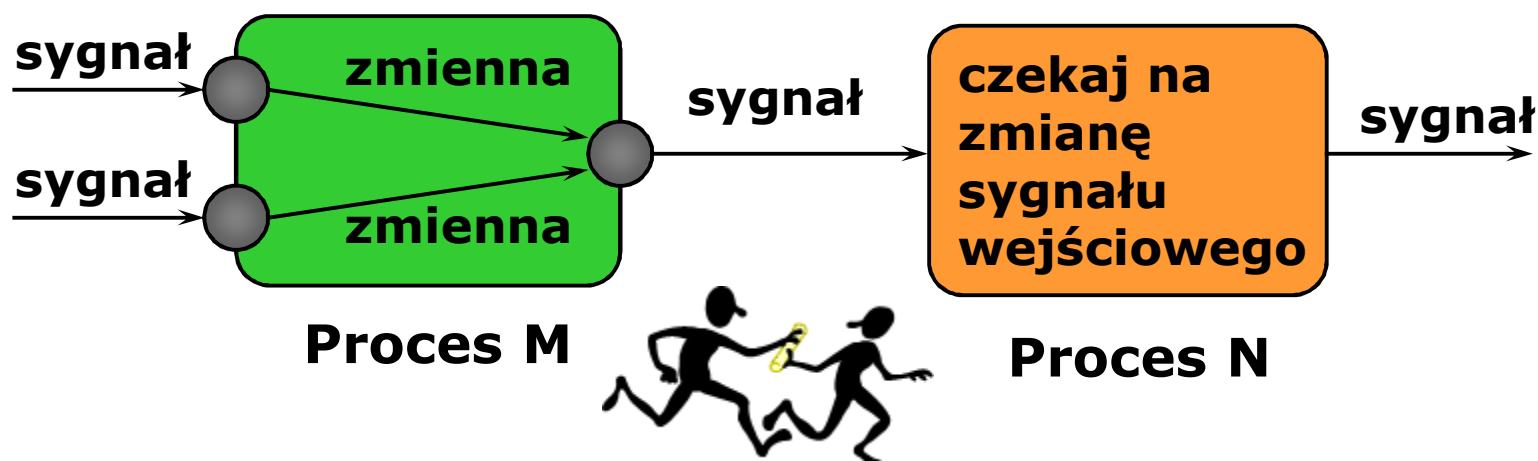
- Zdarzenie zadeklarowane
(Named Event Control, Lin 119 - 120)
- ...
 @ **event** *procedural_statement*
-
 -
 -

Level Sensitive Event Control (Lin 123)

....



- 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**



Składnia:

```
wait  [on sygnaly]
       [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:



<pre>process ... wait on a,b; end process;</pre>	$\Leftrightarrow$	<pre>process (a,b) ; ... end process;</pre>
---	-------------------	---



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 n.p. do dekodowania kodów, stanu automatu lub stanu magistral.

Składnia:

```
case wyrażenie is  
    when val                      => instrukcje_sekwencyjne ;  
    [when val1 | val2              => instrukcje_sekwencyjne ;]  
    [when val3 to val4             => instrukcje_sekwencyjne ;]  
    [when others                  => instrukcje_sekwencyjne ;]  
end case ;
```

Przykład:

```
case BCD_int is  
    when 0 => LED <= "1111110" ;  
    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:

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

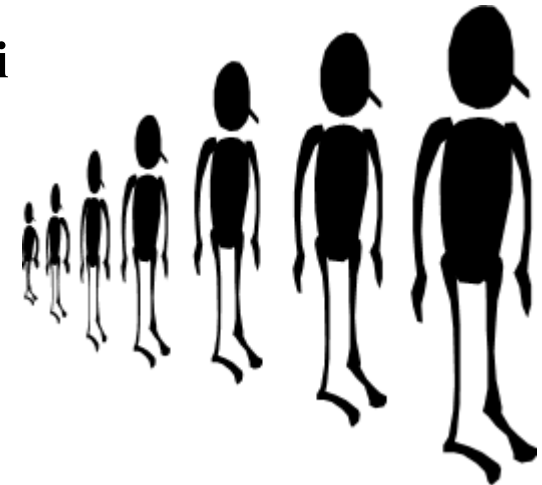
Przykład:

```
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**.

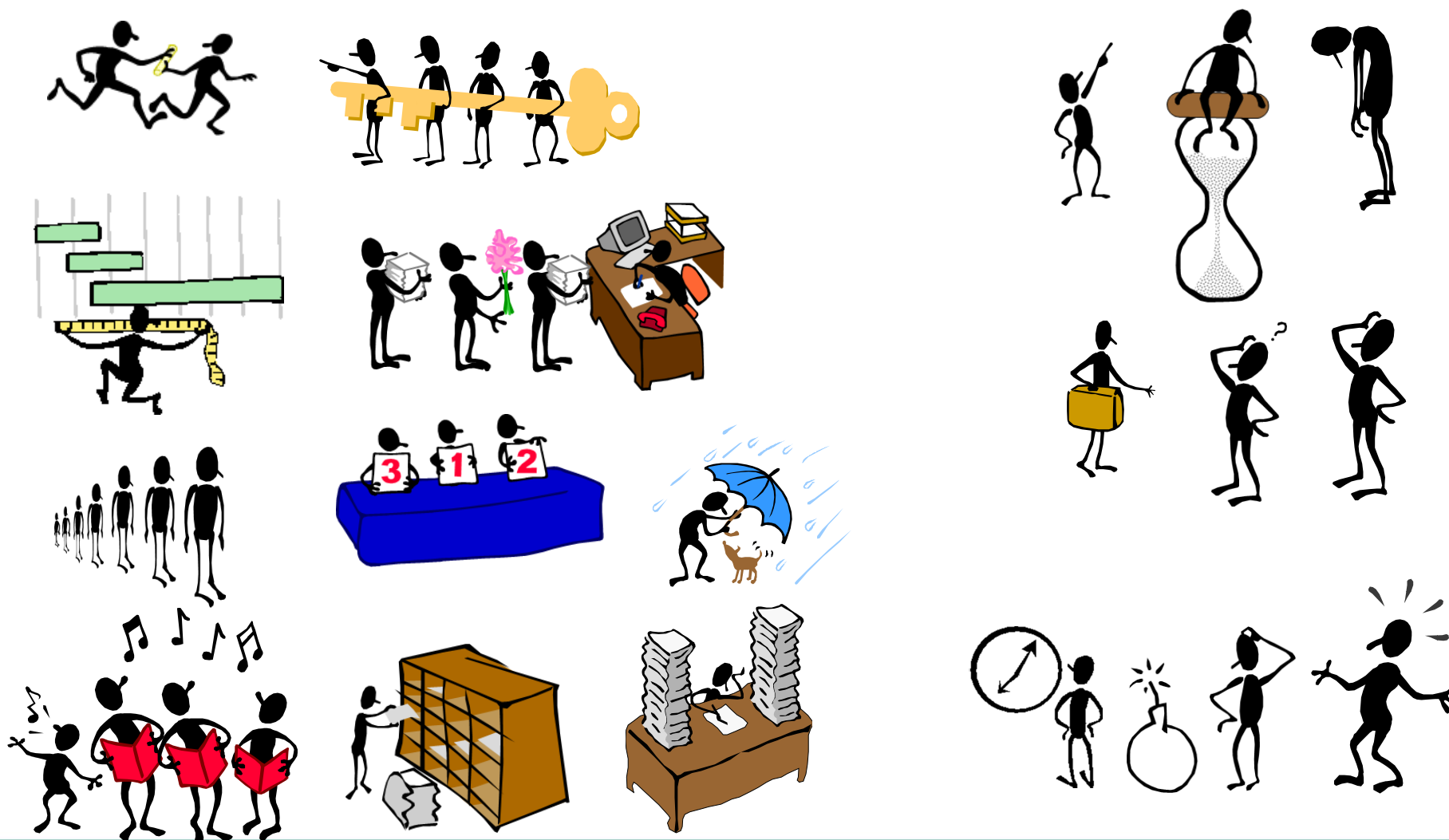
- Tablice składają się z elementów tego samego typu.
- Są one używane do opisu magistral, rejestrów i innych zbiorów elementów sprzętowych.
- Elementy tablic mogą być skalarami lub elementami złożonymi. (Nie można tworzyć np. tablic plików !!)
- Dostęp do poszczególnych elementów dzięki użyciu wskaźników.



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`.



Ciąg dalszy
nastąpi...

