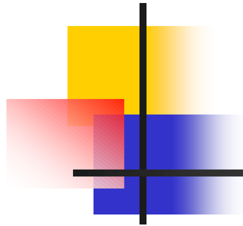


# Języki projektowania HDL

Wykład

*dr hab. inż. Marek Wójcikowski*

Ver: 2019-02-22

# Zasady zaliczenia

- Wykład (kierunkowy efekt kształcenia [K_W19])
  - 2 kolokwia: 25+25 punktów=50 punktów
  - Obecność na wykładzie +5 punktów
- Laboratorium
  - 50 punktów (kierunkowy efekt kształcenia [K_U16])
- Warunek zaliczenia:
  - Przynajmniej 25 pkt. za kolokwia

  oraz
  - przynajmniej 25 pkt. z laboratorium.

# Zasady zaliczenia (cd.)

- Wykład (kierunkowy efekt kształcenia [K_W19])
- Laboratorium (kierunkowy efekt kształcenia [K_U16])
- Progi ocen:
  - ≥90 pkt. ==> ocena 5
  - ≥80 pkt. ==> ocena 4,5
  - ≥70 pkt. ==> ocena 4
  - ≥60 pkt. ==> ocena 3,5
  - ≥50 pkt. ==> ocena 3
  - <50 pkt. ==> ocena 2

http://www.ue.eti.pg.gda.pl/fpgalab/hdl/index.html

# Daty

- 22/02 – wykład 2 godz.
- 27/02 – wykład 2 godz.
- 1/03 – wykład 2 godz.
- 6/03 – wykład 2 godz.
- 8/03 – kolokwium I (1 godz.) + wykład 1 godz.
- 20/03 – wykład 2 godz.
- 22/03 – wykład 2 godz.
- 27/03 – kolokwium II (1 godz.)

# Plan wykładu

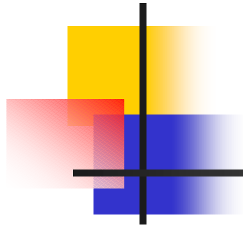
- Verilog (podstawy)
  - Koncepcja modelowania hierarchicznego
  - Podstawowe pojęcia
  - Moduły i porty
  - Modelowanie na poziomie bramek logicznych
  - Modelowanie na poziomie rejestrów (*dataflow*)
  - Modelowanie na poziomie behawioralnym
  - Zadania i funkcje

# Plan wykładu (cd.)

- VHDL (dla zaawansowanych)
  - Składnia i typy danych
  - Biblioteki
  - Jednostki projektowe, sygnały
  - Poziom strukturalny
  - Poziom RTL
  - Poziom behawioralny
  - Maszyny stanów
  - Biblioteki standardowe
  - Synteza

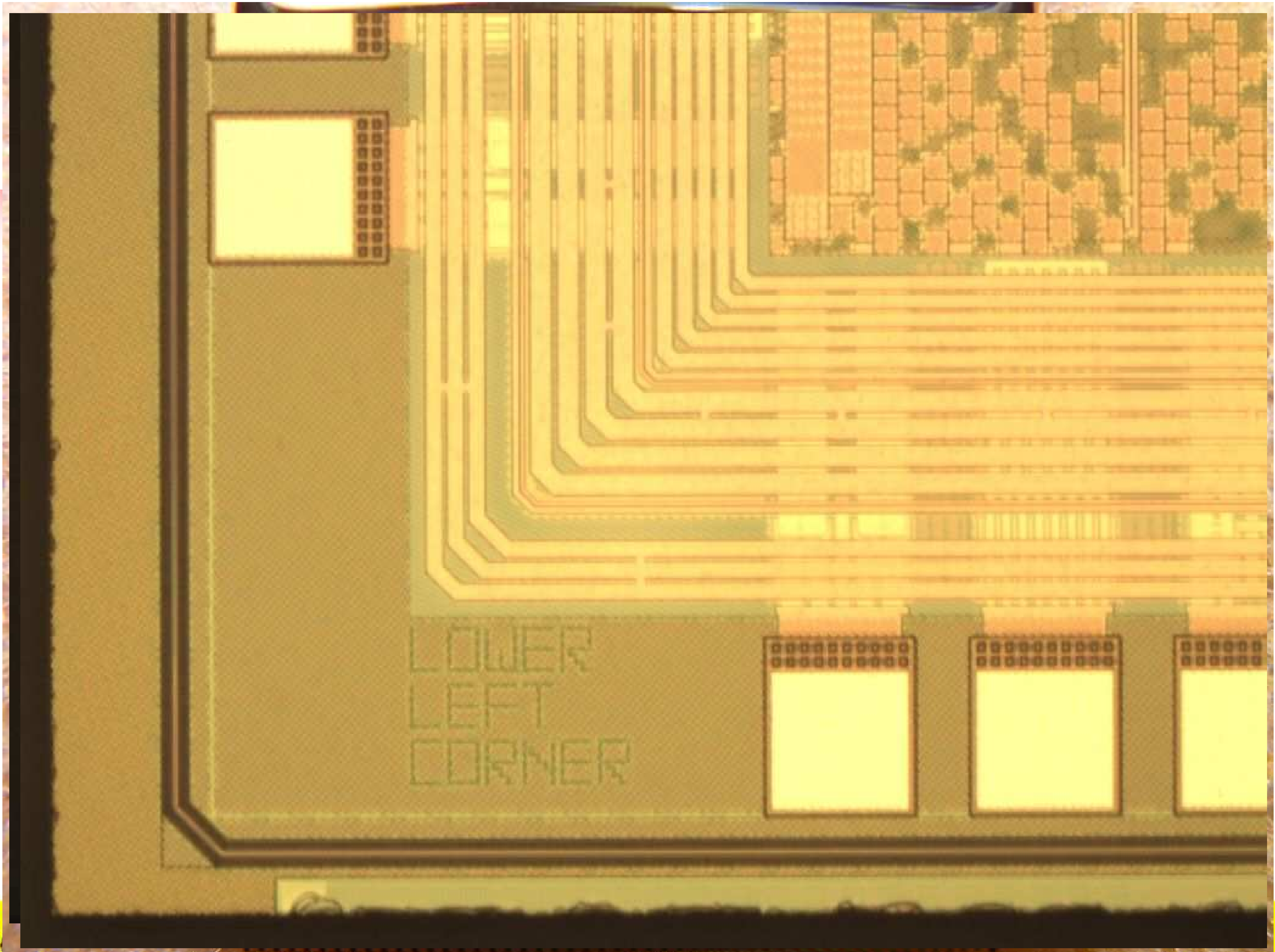
# Literatura

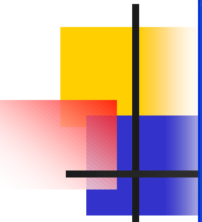

- S. Palnitkar, VERILOG HDL a Guide to Digital Design and Synthesis, SunSoft Press, 1996.
- K. Coffman, Real Word FPGA Design with Verilog, Prentice-Hall, 2000.
- M. G. Arnold, Verilog Digital Computer Design, Algorithms into Hardware, Prentice-Hall, 1999.
- A. Yalamanchili, Introductory VHDL: From Simulation to Synthesis, Prentice-Hall, 2001.
- K. Skahill, Język VHDL, Projektowanie programowalnych układów logicznych, WNT, 2001.
- K. Skahill, VHDL for Programmable Logic, Addison-Wesley Publishing Inc. 1996.
- D. Pellerin, D. Taylor, VHDL Made Easy!, Prentice Hall, 1997.
- D. E. Ott, T. J. Wilderotter, A Designer's Guide to VHDL Synthesis, Kluwer Academic Publishers, 1994.
- S. Sanfilippo, Tcl the Misunderstood, http://antirez.com/articoli/tclmisunderstood.html
- B. B. Welch, Practical Programming in Tcl and Tk, Prentice Hall1997.



# Część 1. (Verilog)

Wstęp


LOWER
LEFT
CORNER



erilog

technologia:
UMC
130nm

Cadence Encounter(r) RTL Compiler v06.20-s027_1 - /home/zobert/ic_design/synthesis/synth_3v2/sn_v8_cg_dft_jtag

File Report Tools Preferences Window Help cadence

Logical | HDL | Schematic | Physical

```
io_shell
 ├CG__RC_CG_HIER_INST(
 ├CG__RC_CG_HIER_INST(
 ⊞CORE [system_top_Te(
 ⊞JTAG_MODULE [JTAG_M/
 ⊞MK_DELAY_BL [mk_dela
 ├TBB_BSC_arf29_busy_:
 ├TBB_BSC_arf29_c0_o_1
 ├TBB_BSC_arf29_c1_o_1
 ├TBB_BSC_arf29_c2_o_1
 ├TBB_BSC_arf29_clock_
 ├TBB_BSC_arf29_data_:
 ├TBB_BSC_arf29_p0_o_1
 ├TBB_BSC_arf29_p1_o_1
 ├TBB_BSC_arf29_power_
 ├TBB_BSC_arf29_tx_rx_
 ├TBB_BSC_boot_devsel_
 ├TBB_BSC_camera_clk_:
 ├TBB_BSC_camera_data_
 ├TBB_BSC_camera_data_
 ├TBB_BSC_camera_data_
 ├TBB_BSC_camera_data_
 ├TBB_BSC_camera_data_
 ├TBB_BSC_camera_data_
 ├TBB_BSC_camera_data_
 ├TBB_BSC_camera_data_
 ├TBB_BSC_clk_pad_25MH
 ├TBB_BSC_clk_pad_50MH
 ├TBB_BSC_clk_pad_50MH
 ├TBB_BSC_dbg_tck_pad_
 ├TBB_BSC_dbg_tdi_pad_
 ├TBB_BSC_dbg_tdo_pad_
 ├TBB_BSC_dbg_tms_pad_
 ├TBB_BSC_dbg_trst_pad
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
 ├TBB_BSC_gpio_pad_io_
```

QDFZRSBELD

SMT_reg3760

RB SB CK D TD SEL Q

QDFZRSBELD

Net: PU_3259

memory_dat_pad_bicell_18__bicell_ZMA4GSLB

E E4 E8 I PD PU SMT SR IO IO O

ZMA4GSLB

io_shell 1

Design is mapped

Układ FPGA Spartan 3S200

# Znaczenie języków HDL

- Projekty mogą być opisane na bardzo wysokim poziomie abstrakcji.
- Sprawdzenie działania układu może nastąpić na bardzo wczesnym etapie projektowania.
- Opis tekstowy wraz z komentarzem dużo łatwiej zrozumieć i śledzić.

# Zastosowania języków HDL

- Wprowadzanie projektu do systemów automatycznej syntezy i projektowania.
- Opisu układu (*netlist*).
- Modelowanie i symulacja układów cyfrowych.
- Weryfikacja projektu (*test-bench*).
- Sporządzanie dokumentacji projektu.

# Popularność języka Verilog

- Łatwy do nauczenia i używania.
- W składni podobny do języka C.
- Umożliwia opis układu na różnych poziomach abstrakcji jednocześnie
- Większość popularnych systemów syntezy logicznej posiada implementację Verilogu.
- Większość producentów układów ASIC dostarcza biblioteki do symulacji po syntezie logicznej.
- *Verilog Procedural Interface* - umożliwia dopisywanie w języku C procedur współdziałających z wewnętrznymi strukturami Verilog –wywoływanie funkcji C z Verlioga i odwrotnie (kiedyś: PLI *Programming Language Interface*).

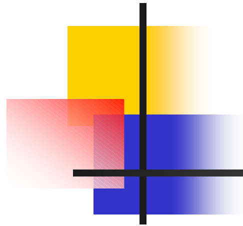
# Poziomy opisu

**Verilog**

- *Poziom kluczy*
- Poziom bramek logicznych
- Poziom rejestrów (*Dataflow*)
- Poziom behawioralny

**VHDL**:

- Poziom strukturalny
- Poziom przesłań międzyrejestrowych RTL
- Abstrakcyjny poziom behawioralny



# Część 2. (Verilog)

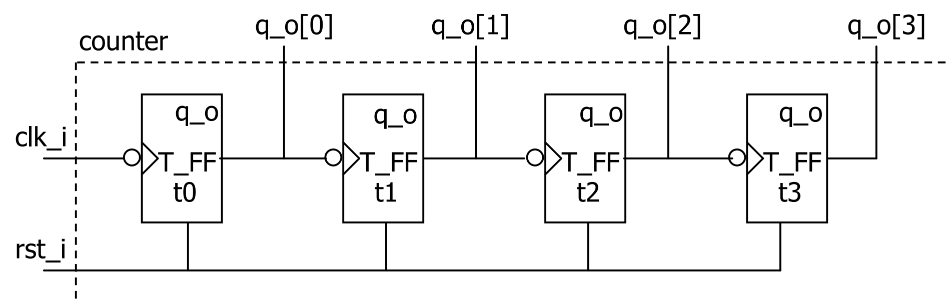
Koncepcja modelowania hierarchicznego

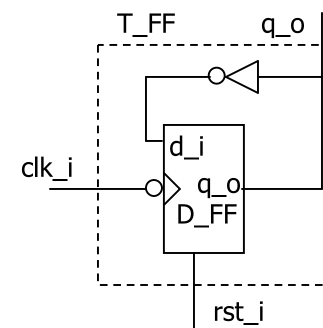
# Metodologia projektowania

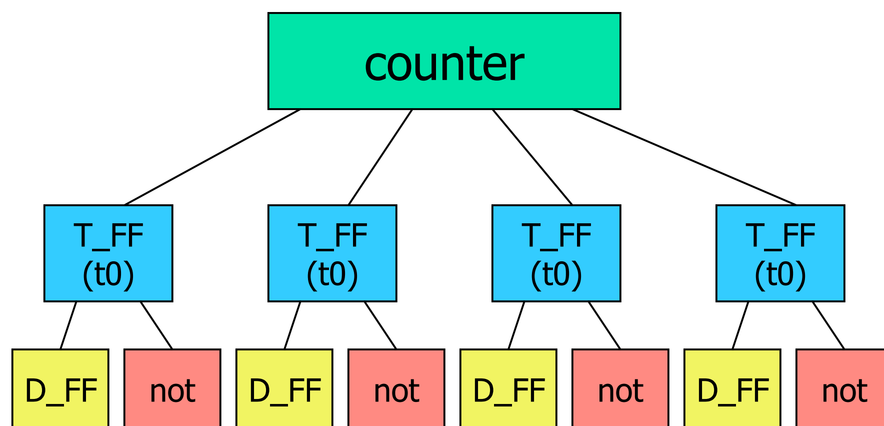
- *Top-down*
- *Bottom-up*

W praktyce stosuje się kombinację tych metodologii.

# Licznik 4-bitowy z przeniesieniem



# Moduły

```
module <nazwa_modułu> (<lista_końcówek_modułu>);
...
<wnętrze modułu>
...
...
endmodule
```

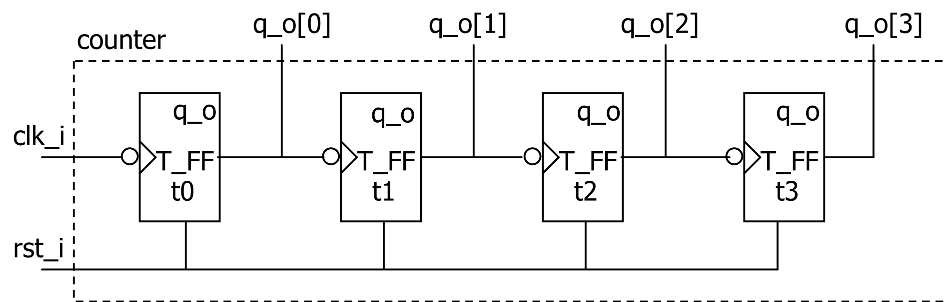
Brak średnika!

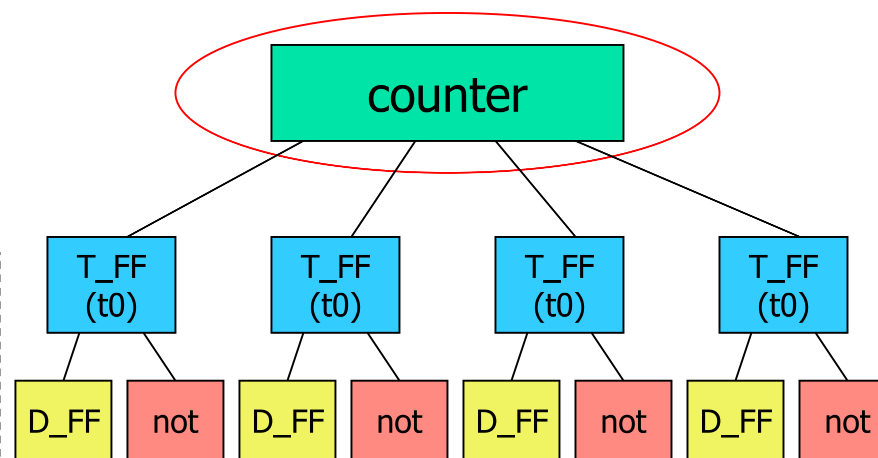
Nie wolno zagnieżdżać definicji modułów.

Moduł tylko opisuje działanie bloku układu oraz definiuje jego końcówki. Aby wykorzystać taki blok, należy go powołać do istnienia.

# Przykład – główny blok układu

```
module counter(q_o, clk_i, rst_i);
   output [3:0] q_o;
   input clk_i, rst_i;
   T_FF t0(q_o[0], clk_i, rst_i);
   T_FF t1(q_o[1], q_o[0], rst_i);
   T_FF t2(q_o[2], q_o[1], rst_i);
   T_FF t3(q_o[3], q_o[2], rst_i);
endmodule
```



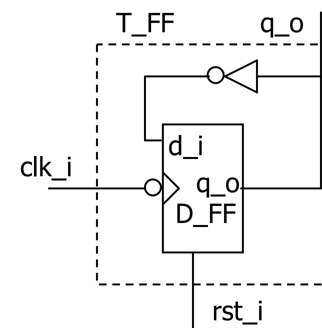



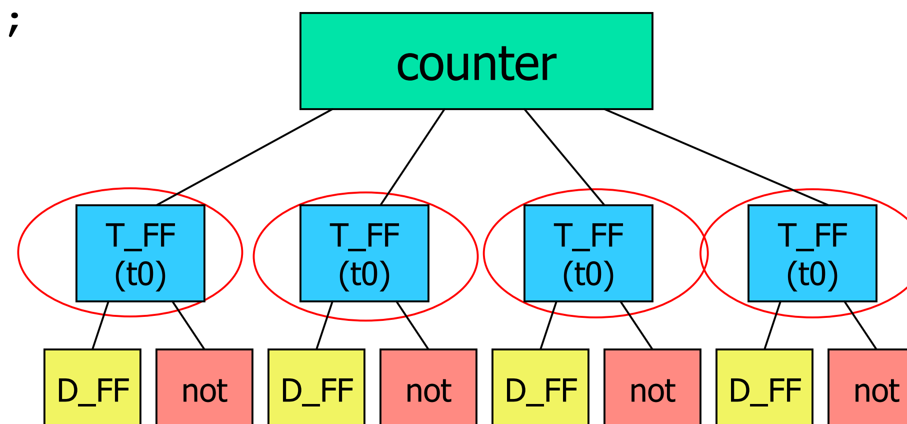
# Przykład – moduł T_FF

```
module T_FF(q_o, clk_i, rst_i);

   output q_o;
   input clk_i, rst_i;
   wire d;

   D_FF dff0(q_o, d, clk_i, rst_i);
   not n1(d, q_o);
endmodule
```



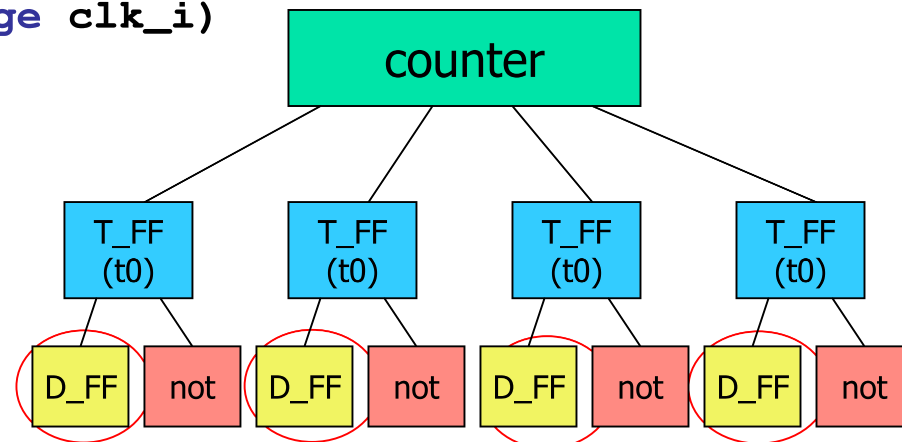



# Przykład – moduł D_FF

```
module D_FF(q_o, d_i, clk_i, rst_i);

   output q_o;
   input d_i, clk_i, rst_i;
   reg q_o;

   always @(posedge rst_i or negedge clk_i)
        if (rst_i)
                q_o = 1'b0;
        else
                q_o = d_i;
endmodule
```

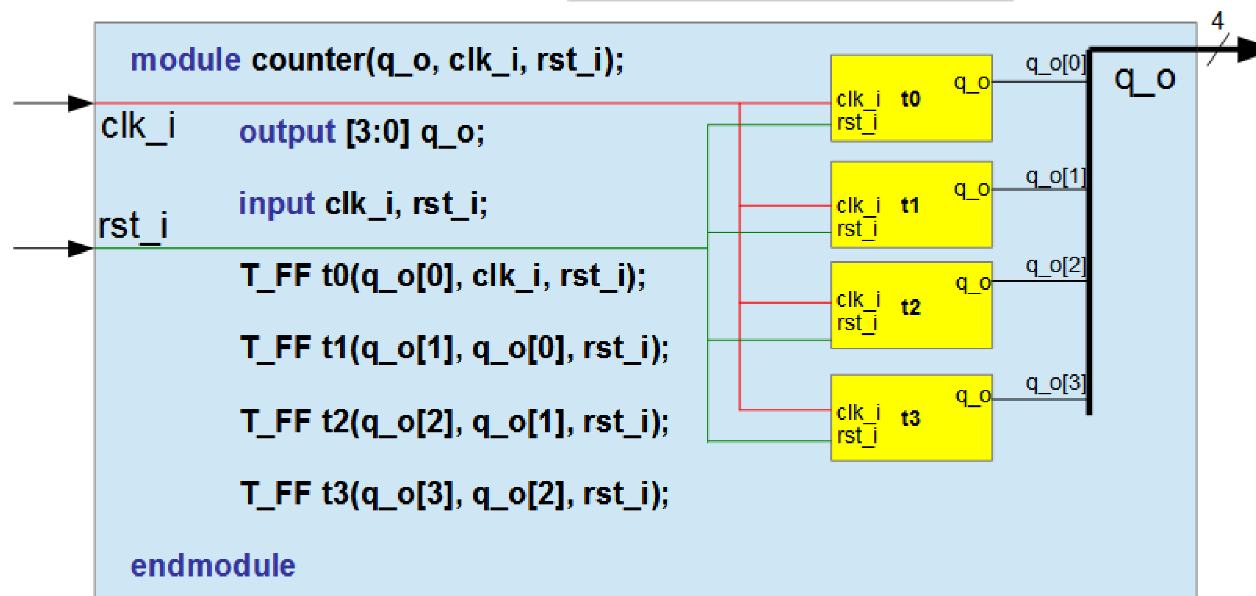


# Podsumowanie

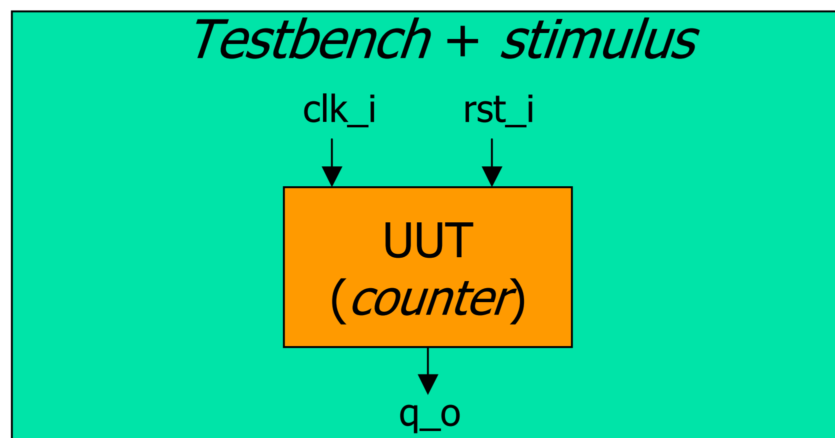
```
module T_FF(q_o, clk_i, rst_i);

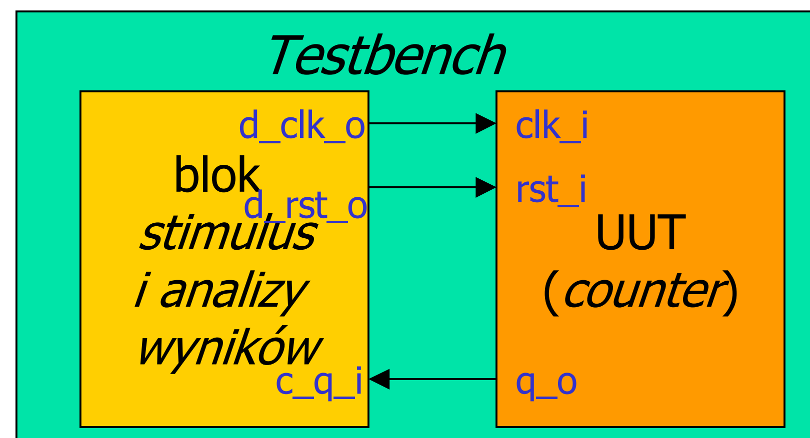
    output q_o;
    input clk_i, rst_i;
    wire d;

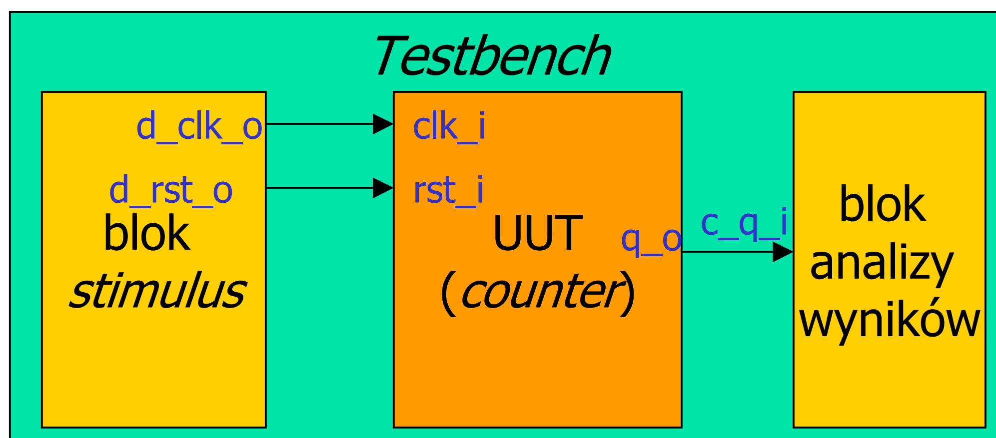
    D_FF dff0(q_o, d, clk_i, rst_i);
    not n1(d, q_o);
endmodule
```



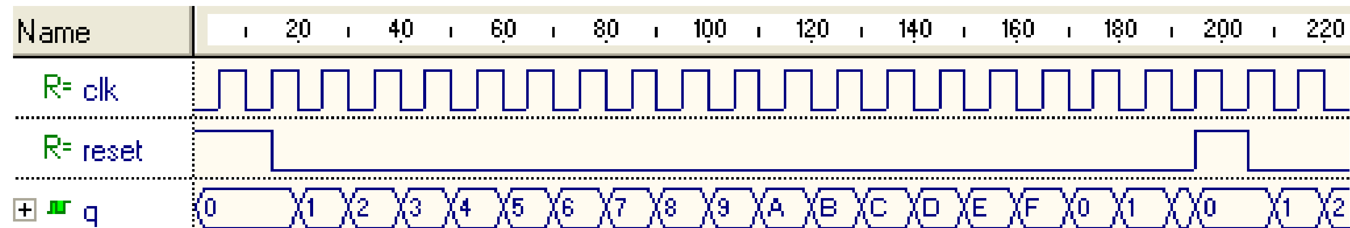
# Symulacja



# Przykład – *testbench*




```
module testbench;
   reg clk;
   reg rst;
   wire [3:0] q;

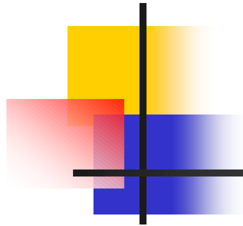
   counter cnt1(q, clk, rst);

   initial
        clk = 1'b0;
   always
        #5 clk = ~clk;

   initial
        begin
                rst = 1'b1;
                #15 rst = 1'b0;
                #180 rst = 1'b1;
                #10 rst = 1'b0;
                #20 $finish;
        end

   initial
        $monitor($time, " Output q=%d", q);

endmodule
```



# Część 3. (Verilog)

Podstawowe pojęcia

# Składnia

- Verilog rozróżnia **małe** i **DUŻE** litery!
- Słowa kluczowe piszemy **małymi** literami
- Biała spacja (spacja, tab., koniec linii) jest ignorowana (wyjątek: łańcuchy)
- Komentarze:
  - `// komentarz do końca linii`
  - ```
    /* komentarz
    przez kilka linii */
    ```
    ```

# Składnia (cd.)

- Operatory:

```
a = ~b;          // 1 argument
a = b && c;      // 2 argumenty
a = b ? c : d;   // 3 argumenty
```

- Liczby:

```
<rozmiar>'<podstawa><liczba>
```

```
<rozmiar> = liczbą dziesiętną, ilość bitów w reprezentacji
   liczby.
<podstawa> = dziesiętna 'd lub 'D, szesnastkowa 'h lub 'H,
   binarna 'b lub 'B, ósemkowa 'o lub 'O
```

```
4'b1111 // to jest 4-bitowa liczba dwójkowa
12'habc // to jest 12-bitowa liczba szesnastkowa
16'd255 // to jest 16-bitowa liczba dziesiętna
```

# Składnia (cd.)

- Liczby (cd.):
  - Liczby o nie określonej podstawie = dziesiętne
  - Liczby bez określonego rozmiaru = 32-bitowe (zależne też od komputera)

```
23456 // liczba dziesiętna
'hc3  // 32-bitowa liczba szesnastkowa
'o21  // 32-bitowa liczba ósemkowa
```

# Składnia (cd.)

- Liczby (cd.):
  - **x** oznacza stan nieokreślony
  - **z** oznacza stan wysokiej impedancji

```
12'h13x // 12-bitowa liczba szesnastkowa,
        // 4 najmniej znaczące bity są nieokreślone
6'hx    // 6-bitowa liczba szesnastkowa
32'bz   // 32-bitowa liczba o wysokiej impedancji
```

  - **x** lub **z** oznacza 4 bity w reprezentacji szesnastkowej, trzy bity w reprezentacji ósemkowej i jeden bit w dwójkowej.
  - Jeśli najbardziej znaczący bit w liczbie to 0, **z** lub **x**, to najbardziej znaczące bity są automatycznie wypełniane tą wartością. Jeśli najbardziej znaczącym bitem jest 1, to pozostałe najbardziej znaczące bity są uzupełniane zerami.

# Składnia (cd.)

- Liczby ujemne:
  - Liczby ujemne określa się poprzez dodanie znaku minus przed określeniem rozmiaru liczby.

```
-6'd3 // 6-bitowa liczba ujemna przechowywana
      // jako uzupełnienie dwójkowe.
```

- Inne uwagi:
  - W liczbach, aby zwiększyć ich czytelność, można stosować znak podkreślenia (byle nie na początku liczby).
  - Znak zapytania w liczbie jest odpowiednikiem stanu wysokiej impedancji **z**.

# Składnia (cd.)

- Łańcuchy:
  - Łańcuch musi być zdefiniowany w jednej linii - nie dopuszcza się znaków CR w łańcuchu.

# Składnia (cd.)

- Identyfikatory:
  - Można stosować znaki alfanumeryczne, znak podkreślenia, oraz **$**.
  - Nie mogą się zaczynać od **$** ani od cyfry
  - Identyfikatory zaczynające się od \:

```
\a+b-c
\**my_name**
```

# Typy danych

- Zestaw wartości

| Wartość | Znaczenie w układach cyfrowych |
|---|---|
| 0 | logiczne zero, fałsz |
| 1 | logiczna jedynka, prawda |
| x | wartość nieokreślona |
| z | stan wysokiej impedancji |

| Siła sygnału | Typ | |
|---|---|---|
| supply | Driving | najsilniejszy |
| strong | Driving | |
| pull | Driving | |
| large | Storage | |
| weak | Driving | |
| medium | Storage | |
| small | Storage | |
| highz | High Impedance | najsłabszy |

**Jeśli dwa sygnały mają równe siły, to na przewodzie jest sygnał nieokreślony x**

# Typy danych (cd.)

- Sieci:
  - połączenia pomiędzy elementami układu.
  - standardowo przyjmują szerokość 1-bitową.
  - standardową domyślną wartością jest **z**.

```
wire a;
wire b, c;
wire d=1'b0; // sieć d będzie miała zawsze wartość 0
```

# Typy danych (cd.)

- Rejestry:
  - reprezentują elementy pamiętające = zmienne, które pamiętają ostatnio zapisaną wartość.

```
reg reset; // <- deklaracja zmiennej reset
initial    // <- będzie wyjaśnione później
begin
   reset=1'b1; // ustaw reset=1
   #100 reset=1'b0; // po 100 jednostkach czasu
                    // ustaw reset=0
end
```

# Typy danych (cd.)

- Wektory:
  - Sieci (**wire**) i rejestry (**reg**) mogą być wektorami:

```
wire a; // skalar
wire [7:0] bus; // 8-bitowa magistrala
wire [31:0] busA, busB, busC;
reg [0:40] addr;
```

  - [**max:min**] lub [**min:max**] = ale zawsze lewa liczba oznacza najbardziej znaczący bit.
  - Użycie części wektorów:

```
busA[7] // 7-my bit wektora busA
bus[2:0] // trzy najmniej znaczące bity wektora
         // (nie wolno: bus[0:2]!)
```

# Typy danych (cd.)

- Liczby całkowite:

```
integer counter;
initial
  counter = -1;
```

- Liczby rzeczywiste:

```
real data;
initial
begin
  delta=4e10;
end
```

Kiedy wartość rzeczywista jest przypisywana do całkowitej, jest ona zaokrąglana do najbliższej wartości całkowitej

# Typy danych (cd.)

- Tablice:
  - **reg**, **integer**, **time**. Nie wolno dla **real**!
  - Dostęp do elementu macierzy:

    `<nazwa_tablicy>[<indeks>]`
  - Nie wolno definiować tablic wielowymiarowych*.

```
integer count[0:7]; // tablica 8 liczników
reg bool[31:0]; // tabl. 32, jednobitowych rejestrów
time chk_point[1:1000];
reg [4:0] port_id[0:7]; // tablica 8 zmiennych,
                        // każda ma 5 bitów
.............................................................
count[5] // 5-ty element tablicy count
port_id[3] // 3-ci element zmiennej (5-bitowej)
```

  - Nie można odwoływać się do pojedynczych bitów tablicy!

# Typy danych (cd.)

- Tablice (cd.):
  - Nie należy mylić tablic z wektorami!
    - Wektor to pojedynczy element o szerokości *n*-bitów.
    - Tablica to zestaw wielu elementów o szerokości 1 lub *n* bitów.
- Pamięci:
  - Pamięci = tablice rejestrów.
  - Każdy element tablicy to słowo.
  - Każde słowo może mieć długość 1 lub więcej bitów.

# Typy danych (cd.)

- Łańcuchy:
  - Przechowywane są w zmiennych typu **reg**.
  - Szerokość rejestru musi być wystarczająca do przechowania łańcucha.
  - Każdy znak w łańcuchu zajmuje 8 bitów.
  - Jeśli łańcuch jest krótszy, Verilog uzupełnia bity z lewej strony zerami.
  - Jeśli łańcuch jest za długi, to Verilog obcina lewą część łańcucha.

# Typy danych (cd.)

- Łańcuchy (cd.):

```
reg [8*18:1] string_value; // zmienna o szer. 18 bajtów
initial
   string_value="Hello Verilog Word";
```

- W łańcuchu można umieszczać znaki specjalne poprzedzone znakiem \:

```
\n     =     nowa linia
\t     =     tabulator
%%     =     %
\\     =     \
\"     =     "
\ooo   =     znak zapisany 1-3 cyfr ósemkowych
```

# Zadania systemowe – wyświetlanie na ekranie

$<słowo_kluczowe>

```
$display("Hello Verilog Word");
-- Hello Verilog Word

$display($time);
-- 230

$display("At time %d address is %h", $time, virtual_addr);
-- At time 200 address is 1fe0000001c

$display("ID of the port is %b", port_id);
-- ID of the port is 0010

$display("This string is displayed from %m level of hierarchy";
-- This string is displayed from top.pl level of hierarchy

$display("This is a \n multiline string with a %% sign");
-- This is a
-- multiline string with a % sign
```

# Zadania systemowe - wyświetlanie na ekranie (cd.)

- **$display** na koniec wstawia znak nowej linii.
- Składnia jest podobna do **printf** w C:

| | |
|---|---|
| %d lub %D | wyświetl zmienną w postaci dziesiętnej |
| %b lub %B | wyświetl zmienną w postaci binarnej |
| %s lub %S | wyświetl zmienną jako łańcuch |
| %h lub %H | wyświetl zmienną w postaci szesnastkowej |
| %c lub %C | wyświetl zmienną w postaci znaku ASCII |
| %m lub %M | wyświetl nazwę hierarchiczną (nie potrzebny argument) |
| %v lub %V | wyświetl siłę sygnału |
| %o lub %O | wyświetl zmienną w postaci ósemkowej |
| %t lub %T | wyświetl zmienną w postaci czasu |
| %e lub %E | wyświetl zmienną w postaci naukowej (np. 3.45e6) |
| %f lub %F | wyświetl zmienną w postaci zmiennoprzecinkowej |
| %g lub %G | wyświetl zmienną w postaci naukowej lub zmiennoprzecinkowej, w zależności od tego, która jest prostsza. |

# Zadania systemowe - monitorowanie wartości

- Monitorowanie zmiennych - po każdej zmianie sygnału będzie drukowana jego wartość.

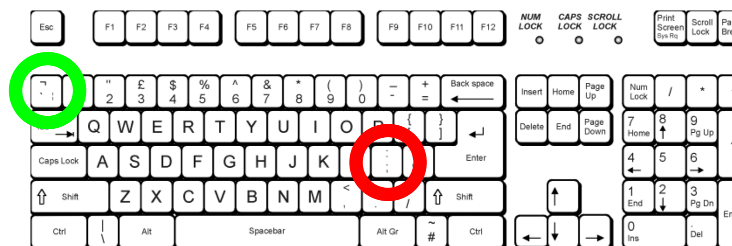
```
$monitor(p1, p2, p3, ..., pn);
```

- Parametry = jak w poleceniu **$display**. Różnicą w stosunku do **$display** jest, że **$monitor** wystarczy podać tylko raz.
- Jeśli występuje więcej niż jedno polecenie **$monitor**, to aktywne jest tylko ostatnie.

# Zadania systemowe - monitorowanie wartości (cd.)

- **$monitoron;** - załącza monitorowanie,
- **$monitoroff;** - wyłącza monitorowanie.

- Pauzowanie i koniec symulacji:
  - **$stop;** - zatrzymuje symulację i przełącza symulator w tryb interaktywny.
  - **$finish;** - kończy symulację.



Verilog

# Dyrektywy kompilatora

- **`define** definiuje makro tekstowe. Verilog zastępuje każde wystąpienie `<nazwa_makra> określonym tekstem.

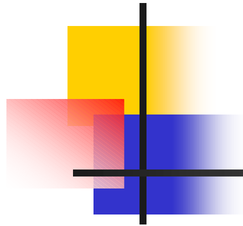
```
`define FALSE 1'b0
`define WORD_SIZE 32
`define S $stop;
`define WORD_REG reg [31:0]
```

  - Wykorzystanie makra:

```
assign out = `FALSE;
```

- **`include** - pozwala dołączyć dowolny plik:

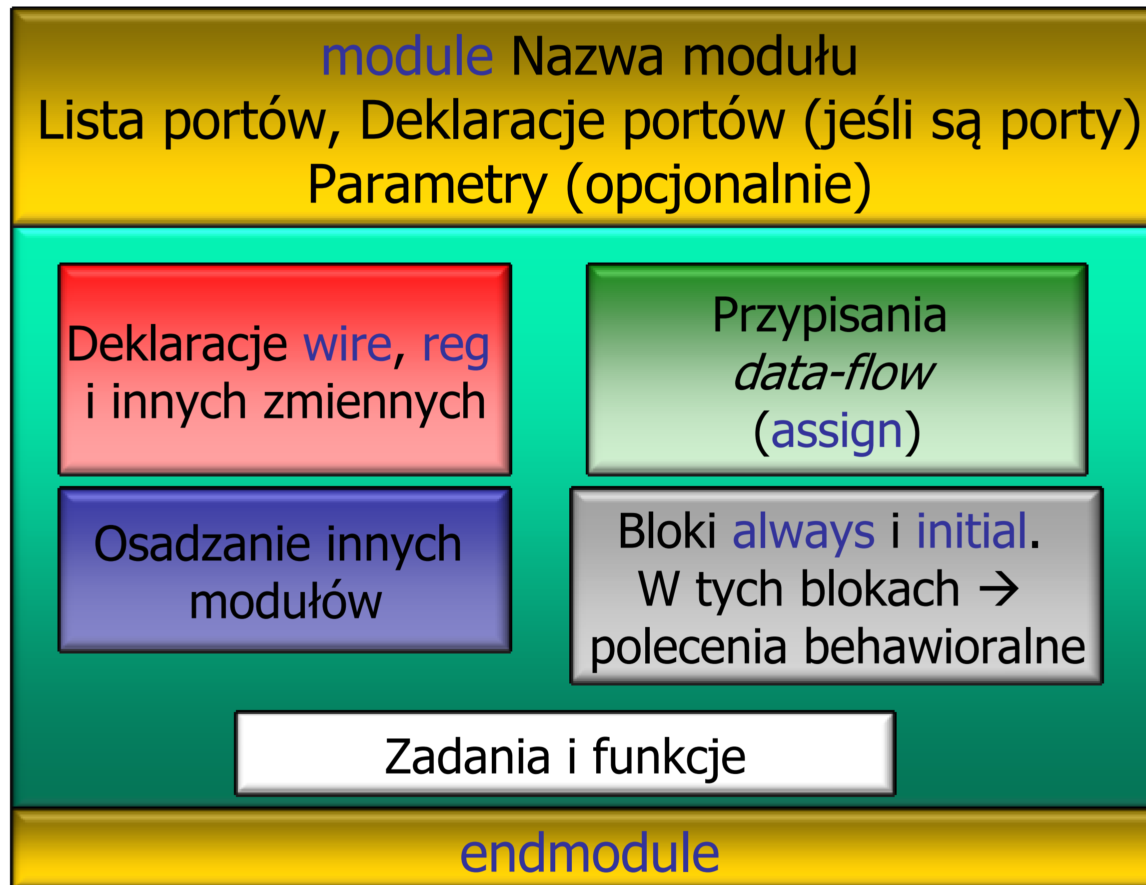
```
`include header.v
```



# Część 4. (Verilog)

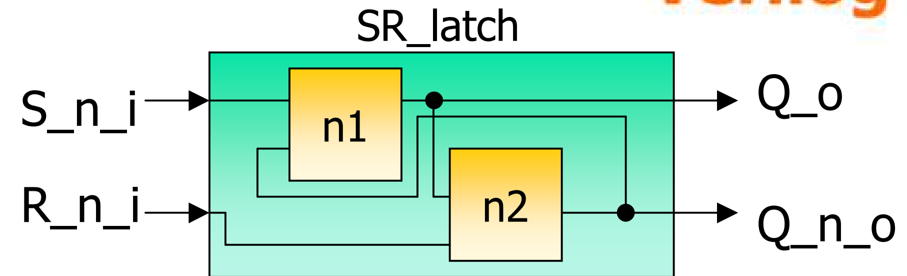
Moduły i porty

# Moduły



Verilog

# Moduły (cd.)




```
// nazwa modułu i lista portów
module SR_latch(Q_o, Q_n_o, S_n_i, R_n_i);

  // deklaracje portów
  output Q_o, Q_n_o;
  input S_n_i, R_n_i;

  // powołanie do życia innych obiektów:
  nand n1(Q_o, S_n_i, Q_n_o);
  nand n2(Q_n_o, R_n_i, Q_o);

  // słowo kluczowe endmodule
endmodule
```

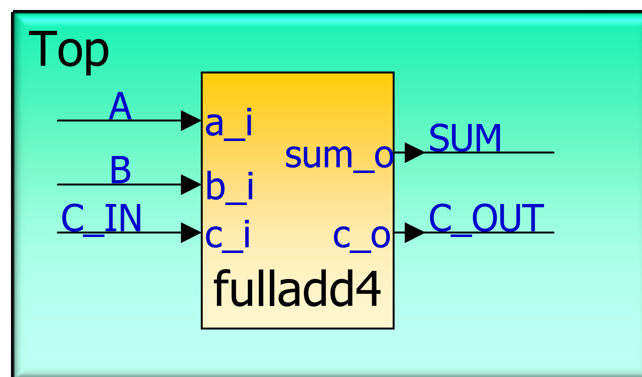
*średnik* (module SR_latch(...);)

*brak średnika* (endmodule)

# Porty

- Lista portów
  - Jeśli moduł nie wymienia sygnałów z otoczeniem - lista portów jest niepotrzebna

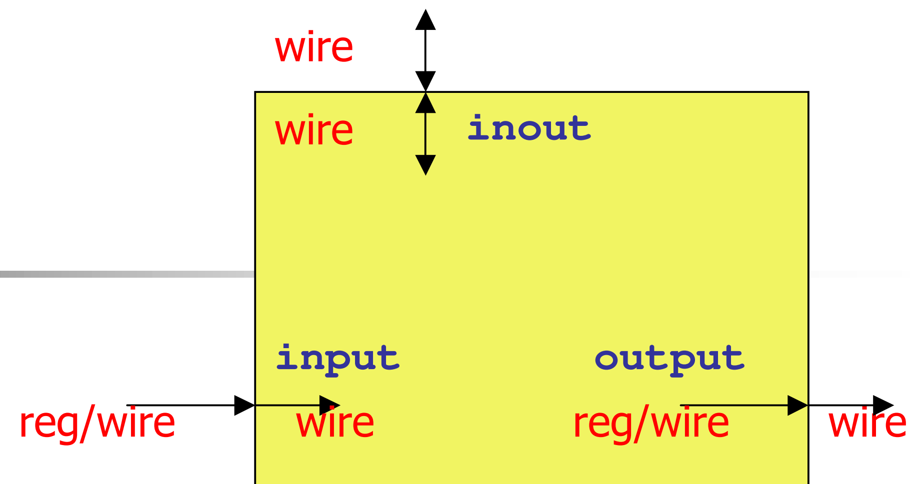
```
module fulladd4(sum_o, c_o, a_i, b_i, c_i);
module Top;
```



# Porty (cd.)

- Wszystkie porty muszą zostać zadeklarowane jako **input**, **output** lub **inout**.
- Deklaracje te niejawnie inicjują port jako **wire**. Zazwyczaj jest to wystarczające dla portów typu **input** i **inout**.
- Porty typu **output** można dodatkowo zdeklarować jako **reg**, aby mogły pamiętać wartość.

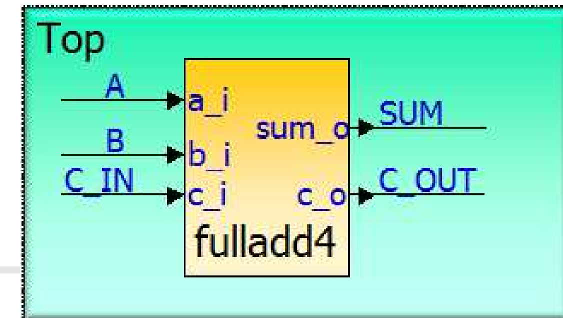
# Porty (cd.)




- Reguły łączenia portów:
  - **input** – wewnętrznie **wire**, zewnętrznie można łączyć z **reg** i **wire**.
  - **output** - wewnętrznie **reg** lub **wire**, zewnętrznie można podłączyć do **wire**. Nie można do **reg**!
  - **inout** - wewnętrznie **wire**, zewnętrznie muszą być podłączone do sieci typu **wire**.
- Dopasowanie szerokości
  - Można łączyć magistrale o różnej szerokości bitowej. Kompilator wygeneruje tylko ostrzeżenie.

```
module fulladd4(sum_o, c_o, a_i, b_i, c_i);
```

Verilog




# Porty (cd.)

- Podłączanie portów do sygnałów zewnętrznych (2 metody - nie mogą być stosowane jednocześnie):
  - Lista uporządkowana:
    - porty w tej samej kolejności jak w definicji:

```
fulladd4 fa1 (SUM, C_OUT, A, B, C_IN);
```

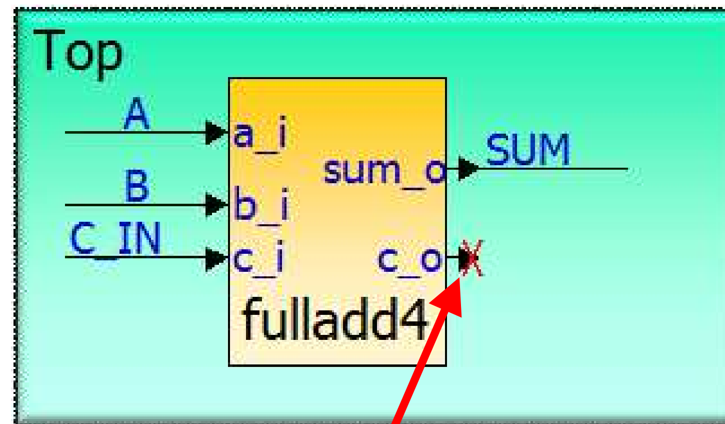
  - Łączenie poprzez podanie nazwy:

```
fulladd4 fa2(.c_o(C_OUT), .sum_o(SUM), .b_i(B),
   .c_i(C_IN), .a_i(A));
```

*nazwa portu* *do czego podłączamy*

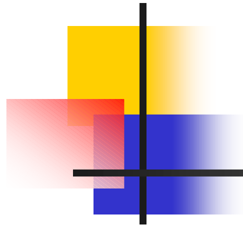
# Porty (cd.)

```
module fulladd4(sum_o, c_o, a_i, b_i, c_i);
```




- Nie podłączone porty
  - Można zostawiać porty nie podłączone.

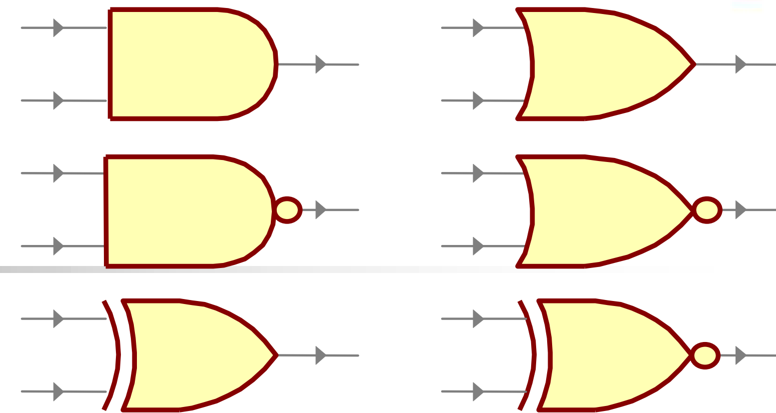
```
fulladd4 fa0(SUM, , A, B, C_IN);
// port c_o jest nie podłączony
```



# Część 5. (Verilog)

Projektowanie na poziomie
bramek logicznych

# Typy bramek



- Dostępne bramki:
  - **and**, **nand**, **or**, **nor**, **xor**, **xnor**
- Bramki o większej ilości wejść = podanie większej ilości wejść w liście portów.

```
wire out, in1, in2, in3;
and a1(out, in1, in2);
nand na3(out, in1, in2, in3);
and (out, in1, in2); //można nie podawać nazwy bramki
```

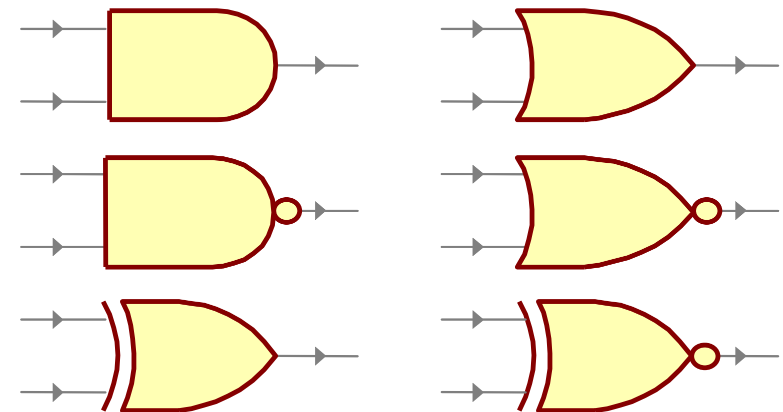
# Typy bramek (cd.)

- Bramki **and**, **nand**, **or**, **nor**, **xor**, **xnor** (cd.)

| and | 0 | 1 | x | z |
|---|---|---|---|---|
| **0** | 0 | 0 | 0 | 0 |
| **1** | 0 | 1 | x | x |
| **x** | 0 | x | x | x |
| **z** | 0 | x | x | x |

| or | 0 | 1 | x | z |
|---|---|---|---|---|
| **0** | 0 | 1 | x | x |
| **1** | 1 | 1 | 1 | 1 |
| **x** | x | 1 | x | x |
| **z** | x | 1 | x | x |

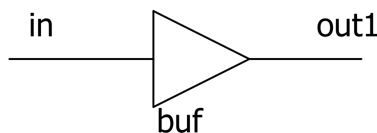
| xor | 0 | 1 | x | z |
|---|---|---|---|---|
| **0** | 0 | 1 | x | x |
| **1** | 1 | 0 | x | x |
| **x** | x | x | x | x |
| **z** | x | x | x | x |

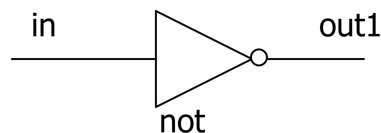


# Typy bramek (cd.)

- Bramki typu **buf** / **not**
  - Mogą one mieć jedno **wejście** i wiele **wyjść**

```
buf b1(out1, in);
not n1(out1,in);
buf b2(out1, out2, in); // wiele wyjść
not (out1, in); // nie trzeba podawać nazwy
```




buf:

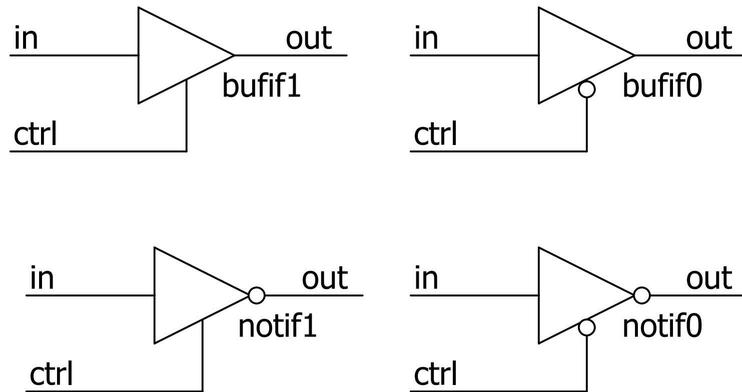
| in | out |
|---|---|
| 0 | 0 |
| 1 | 1 |
| x | x |
| z | x |

not

| in | out |
|---|---|
| 0 | 1 |
| 1 | 0 |
| x | x |
| z | x |

# Typy bramek (cd.)

- Bramki typu **bufif** i **notif**:




| Bufif1 | | ctrl | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| In | 0 | z | 0 | L | L |
| | 1 | z | 1 | H | H |
| | x | z | x | x | x |
| | z | z | x | x | x |

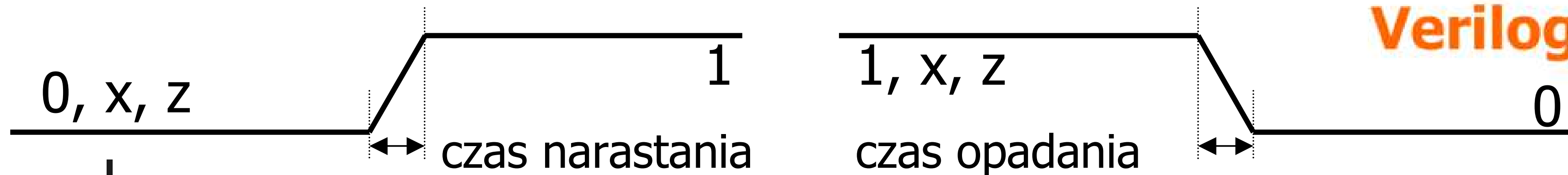
| Notif1 | | ctrl | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| in | 0 | z | 1 | H | H |
| | 1 | z | 0 | L | L |
| | x | z | x | x | x |
| | z | z | x | x | x |

| Bufif0 | | ctrl | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| In | 0 | 0 | z | L | L |
| | 1 | 1 | z | H | H |
| | x | x | z | x | x |
| | z | x | z | x | x |

| notif0 | | ctrl | | | |
|---|---|---|---|---|---|
| | | 0 | 1 | x | z |
| in | 0 | 1 | z | H | H |
| | 1 | 0 | z | L | L |
| | x | x | z | x | x |
| | z | x | z | x | x |

```
bufif1 b1 (out, in, ctrl);
bufif0 b0 (out, in, ctrl);
notif1 n1 (out, in, ctrl);
notif0 n0 (out, in, ctrl);
```

L=0, **z** lub **x**
H=1, **z** lub **x**
(w zależności od implementacji symulatora)






# Opóźnienia w bramkach

- **Czas narastania** = czas w jakim wyjście osiągnie stan **1** z dowolnego innego stanu.
- **Czas opadania** = czas w jakim wyjście osiągnie stan **0** z dowolnego innego stanu.
- **Czas wyłączenia** = czas w jakim wyjście osiągnie stan wysokiej impedancji **z** z dowolnego innego stanu.
- Jeśli wyjście zmienia się do wartości **x**, to dzieje się to po czasie najkrótszym z trzech czasów opisanych powyżej.

# Opóźnienia w bramkach (cd.)

- Możliwości określenia czasów opóźnień:
  - 1 wartość = czas wszystkich trzech przejść;
  - 2 wartości = czasy narastania i opadania, czas wyłączenia = mniejszej z tych liczb.
  - 3 wartości = czas narastania, opadania i wyłączenia.
- Jeśli nie podano żadnej wartości, wszystkie trzy czasy są równe zeru.

```
// taki sam czas dla trzech rodzajów przejść:
and #(4) a1(out, i1,i2);
// określenie czasu narastania i opadania:
and #(3, 2) a2(out, i1,i2);
// określenie trzech czasów:
bufif0 #(3, 2, 5) b1(out, in, control);
```

narastania opadania wyłączania

# Opóźnienia w bramkach (cd.)

- Wartości minimalne, typowe i maksymalne:

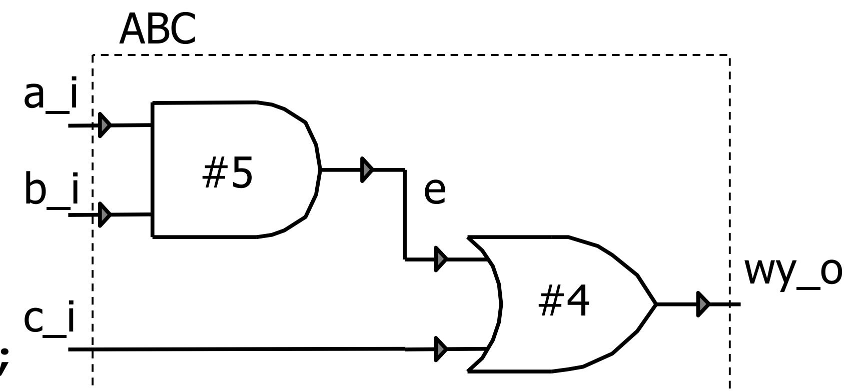
```
// jedno opóźnienie:
and #(4:5:6) a1(out, i1,i2);
       Min. Typ. Max.
// dwa opóźnienia:
and #(3:4:5, 5:6:7) a2(out, i1, i2);
      Min. Typ. Max.  Min. Typ. Max.
// trzy opóźnienia:
and #(2:3:4, 3:4:5, 4:5:6) a3(out, i1, i2);
      Min. Typ. Max. Min. Typ. Max.  Min. Typ. Max.
```

# Opóźnienia - przykład

- Moduł **ABC** ma realizować funkcję:

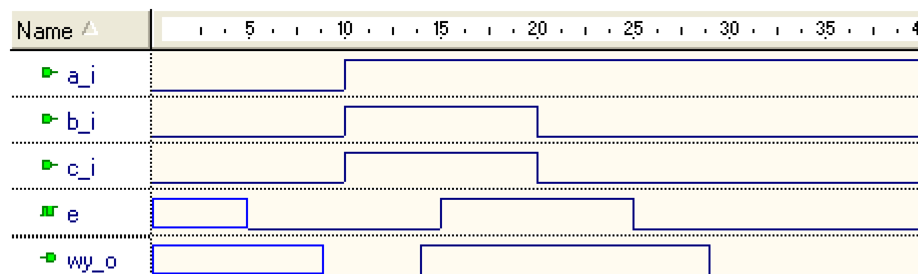
  *wy=(a*b)+c*

```
// definicja modułu A:
module ABC(wy_o, a_i, b_i, c_i);
  // deklaracje portów wej/wyj
  output wy_o;
  input a_i, b_i, c_i;
  // sieć wewnętrzna:
  wire e;
  // bramki:
  and #(5) a1(e, a_i, b_i);
  or  #(4) o1(wy_o, e, c_i);
endmodule
```

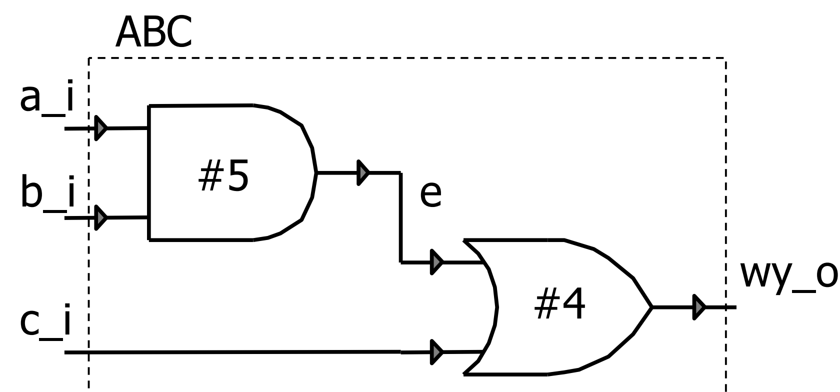


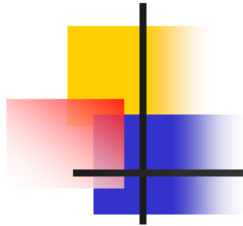
# Opóźnienia – przykład (cd.)

```
module stimulus;
  // deklaracja zmiennych:
  reg A, B, C;
  wire OUT;
  // powołanie modułu ABC:
  ABC abc1(OUT, A, B, C);
  // pobudzanie wejść, koniec symul. po 40 jedn. czasu
  initial
  begin
    A= 1'b0; B= 1'b0; C= 1'b0;
    #10 A= 1'b1; B= 1'b1; C= 1'b1;
    #10 A= 1'b1; B= 1'b0; C= 1'b0;
    #20 $finish;
  end
endmodule
```




*Waveforms are from Aldec's Active-HDL simulator*





# Część 6. (Verilog)

Modelowanie na poziomie rejestrów
(*Dataflow*)

# Przypisanie ciągłe

- Sterowanie sieci wartością logiczną.
- Zastępuje bramki.
- Składnia:
  - assign *[<siła sygnału>][<opóźnienie>] <lista przypisań>*;
- Definiowanie siły sygnału = opcjonalne (domyślnie **strong1** i **strong0**).
- Definiowanie opóźnienia = opcjonalnie (podaje się je jak w bramkach).

# Przypisanie ciągłe (cd.)

- Cechy przypisania ciągłego:
  - Lewa strona = sieć (skalar lub wektor), albo konkatenacja sieci skalarnych lub wektorowych.
  - Przypisanie ciągłe jest zawsze aktywne.
  - Wyrażenie obliczane jest zawsze, gdy dowolna ze zmiennych z prawej strony zmieni swą wartość.
  - Wynik jest natychmiast przesyłany do sieci określonej z lewej strony.
  - Wyrażenia z prawej strony mogą być rejestrami, sieciami lub wywołaniami funkcji (skalary lub wektory).

# Przypisanie ciągłe - przykład

```
assign c = a & b;
assign addr[15:0] = addr1_bits[15:0] ^
  addr2_bits[15:0];
// nawiasy klamrowe oznaczają
// łączenie (konkatenacja)
assign {c_out, sum[3:0]} = a[3:0] +
  b[3:0] + c_in;
```

# Niejawne przypisanie ciągłe

- Zamiast najpierw deklarować sieć, a potem wykonywać do niej operację przypisania:

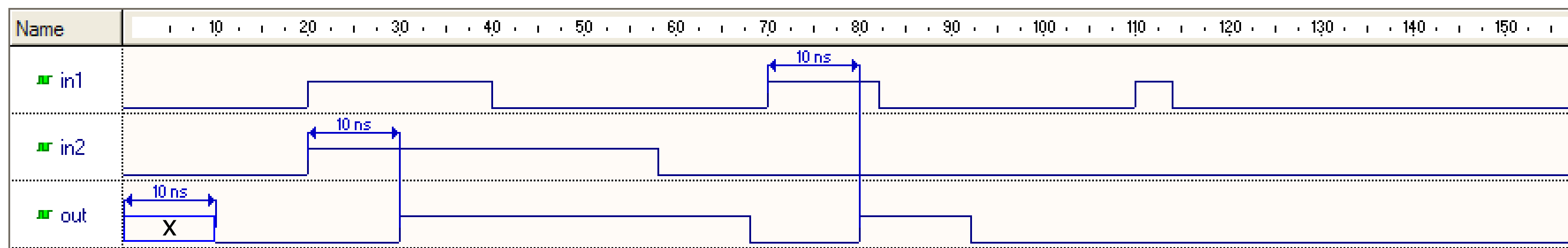
```
wire c;
assign c = a & b;
```

- można to zrobić krócej:

```
wire c = a & b;
```

# Opóźnienia

```
assign #10 c = a | b;
```




- Uwaga: Impulsy krótsze niż 10 jednostek czasowych nie przechodzą na wyjście.

# Opóźnienia (cd.)

- Niejawne określenie przypisania

```
wire c;
assign #10 c = a & b;
```

  - równoważne określenie:

```
wire #10 c = b & a;
```

- Określenie opóźnienia sieci
  - Można zdefiniować opóźnienie dla całej sieci - każda zmiana sygnału na sieci pojawi się z tym opóźnieniem.
  - Opis:

```
wire # 10 c;
assign c = a & b;
```

  - jest równoważny następującemu:

# Wyrażenia i operatory

| Typ | symbol | operacja | ilość operandów |
|---|---|---|---|
| **arytm.** | * | mnożenie | 2 |
| | / | dzielenie | 2 |
| | + | dodawanie | 2 |
| | - | odejmowanie | 2 |
| | % | modulo | 2 |
| **log.** | ! | negacja logiczna | 1 |
| | && | logiczne and | 2 |
| | \|\| | logiczne or | 2 |
| **relacje** | > | większe niż | 2 |
| | < | mniejsze niż | 2 |
| | >= | większe lub równe | 2 |
| | <= | mniejsze lub równe | 2 |
| **równość** | == | równość | 2 |
| | != | różność | 2 |
| | === | równość warunkowa | 2 |
| | !== | różność warunkowa | 2 |

# Wyrażenia i operatory (cd.)

<table>
<tr><th>Typ</th><th>symbol</th><th>operacja</th><th>ilość operandów</th></tr>
<tr><td>bitowe</td><td>~</td><td>negacja bitowa</td><td>1</td></tr>
<tr><td></td><td>&</td><td>and bitowe</td><td>2</td></tr>
<tr><td></td><td>|</td><td>or bitowe</td><td>2</td></tr>
<tr><td></td><td>^</td><td>xor bitowe</td><td>2</td></tr>
<tr><td></td><td>^~ lub ~^</td><td>xnor bitowe</td><td>2</td></tr>
<tr><td>redukujące</td><td>&</td><td>redukujące and</td><td>1</td></tr>
<tr><td></td><td>~&</td><td>redukujące nand</td><td>1</td></tr>
<tr><td></td><td>|</td><td>redukujące or</td><td>1</td></tr>
<tr><td></td><td>~|</td><td>redukujące nor</td><td>1</td></tr>
<tr><td></td><td>^</td><td>redukujące xor</td><td>1</td></tr>
<tr><td></td><td>^~ lub ~^</td><td>redukujące xnor</td><td>1</td></tr>
<tr><td>przesunięcie</td><td>>></td><td>przesunięcie w prawo</td><td>2</td></tr>
<tr><td></td><td><<</td><td>przesunięcie w lewo</td><td>2</td></tr>
<tr><td>łączenie</td><td>{}</td><td>łączenie</td><td>dowolna</td></tr>
<tr><td>replikacja</td><td>{{}}</td><td>replikacja</td><td>dowolna</td></tr>
<tr><td>warunek</td><td>?:</td><td>warunek</td><td>3</td></tr>
</table>

# Wyrażenia i operatory

- Operatory dwuargumentowe:

```
A= 4'b0011; B= 4'b0100;
D=6; E= 4;

A*B    // mnożenie, wynik: 4'b1100
D/E    // dzielenie, wynik: 1 (część ułamkowa
       //                         jest ucięta)
A+B    // dodawanie, wynik: 4'b0111
B-A    // odejmowanie, wynik: 4'b0001
```

  - Jeśli jakiś argument ma wartość **x**, to wynik też będzie miał wartość **x**.

# Wyrażenia i operatory

- Operatory dwuargumentowe (cd.):
  - Operator *modulo* zwraca resztę z dzielenia dwóch liczb (działa podobnie jak w C).

```
13%3  // wynik: 1
16%4  // wynik: 0
-7%2  // wynik: -1 (znak z pierwszego argumentu)
7%-2  // wynik: 1 (znak z pierwszego argumentu)
```

# Wyrażenia i operatory (cd.)

- Operatory jednoargumentowe

```
-4  // MINUS 4
+5  // PLUS 5
```

  - Zaleca się **nie** używać ujemnych liczb w specyfikacji **<sss>'<podstawa> <nnn>**, gdyż może to prowadzić do złych wyników (liczby w tej postaci są reprezentowane jako dopełnienie do dwóch).

```
-'d10 / 5 // co oznacza: (2^32-10)/5
```

# Wyrażenia i operatory (cd.)

- Operatory logiczne:
  - && - logiczne **and**
  - || - logiczne **or**
  - ! - logiczne **not**
  - Operatory logiczne zwracają zawsze 1-bitową wartość: **0** (fałsz), **1** (prawda) lub **x** (niejednoznaczność).
  - Jeśli argument **nie jest** równy zero, jest traktowany jako **1** logiczna (prawda).
  - Jeśli argument **jest** równy zero, to jest traktowany jako **0** logiczne (fałsz).
  - Jeśli jakiś bit argumentu jest równy **z** lub **x**, to cały argument traktowany jest jako **x** (niejednoznaczność).

# Wyrażenia i operatory (cd.)

- Operatory logiczne (cd.)

```
A=3; B=0;
A&&B // wynik: 0
A||B // wynik: 1
!A   // wynik: 0
!B   // wynik: 1

A=2'b0z; B=2'b10
A&&B // wynik: x
(a==2)&&(b==3) // wynik: 1 jeśli a=2
               // oraz b=3, w przeciwnym
               // wypadku wynik: 0
```

# Wyrażenia i operatory (cd.)

- Operatory porównania
  - Jeśli w wyrażeniu użyto operatorów porównania, wyrażenie zwraca wartość **1**, **0** lub **x**.

```
// A=4, B=3
// X=4'b1010, Y=4'b1101, Z=4'b1xxx
A<=B // wynik: 0
A>0  // wynik: 1
Y>=X // wynik: 1
Y<Z  // wynik: x
```

# Wyrażenia i operatory (cd.)

- Operatory równości:

| wyrażenie | opis | możliwe wartości |
|---|---|---|
| **a == b** | a jest równe b. Wynik nieznany, gdy a lub b mają wartość **x** lub **z**. | **0,1,x** |
| **a != b** | a jest różne od b. Wynik nieznany, gdy a lub b mają wartość **x** lub **z**. | **0,1,x** |
| **a === b** | a jest równe b, włączając **x** i **z** | **0,1** |
| **a !== b** | a jest różne od b, włączając **x** i **z** | **0,1** |

Operatory **===** i **!==** porównują bit po bicie i biorą pod uwagę również zgodność wartości **x** oraz **z**.

# Wyrażenia i operatory (cd.)

- Operatory bitowe

| | |
|---|---|
| ~ | **negacja** bit po bicie |
| & | **and** bit po bicie |
| \| | **or** bit po bicie |
| ^ | **xor** bit po bicie |
| ^~ lub ~^ | **xnor** bit po bicie |

**and bitowe**

| | 0 | 1 | x |
|---|---|---|---|
| **0** | 0 | 0 | 0 |
| **1** | 0 | 1 | x |
| **x** | 0 | x | x |

**xnor bitowe**

| | 0 | 1 | x |
|---|---|---|---|
| **0** | 1 | 0 | x |
| **1** | 0 | 1 | x |
| **x** | x | x | x |

**or bitowe**

| | 0 | 1 | x |
|---|---|---|---|
| **0** | 0 | 1 | x |
| **1** | 1 | 1 | 1 |
| **x** | x | 1 | x |

**negacja**

| | |
|---|---|
| **0** | 1 |
| **1** | 0 |
| **x** | x |

**xor bitowe**

| | 0 | 1 | x |
|---|---|---|---|
| **0** | 0 | 1 | x |
| **1** | 1 | 0 | x |
| **x** | x | x | x |

# Wyrażenia i operatory (cd.)

- Operatory redukujące
  - Operatory redukujące: **& (= and), ~& (= nand), | (= or), ~| (= nor), ^ (= xor), ^~ , ~^ (= xnor)** wymagają tylko jednego argumentu.
  - Operują na bitach wykonując odpowiednie operacje według tabel jak dla operatorów bitowych, działają od prawej do lewej.

```
// X=4'b1010
&X // 1&0&1&0 = 1'b0
|X // 1|0|1|0 = 1'b1
^X // 1^0^1^0 = 1'b0
```

  - Redukcja **xor** lub **xnor** może być wykorzystana przy obliczaniu ilości bitów parzystych lub nieparzystych w wektorze.

# Wyrażenia i operatory (cd.)

- Operatory przesuwania
  - >> - przesunięcie w prawo
  - << - przesunięcie w lewo
  - Nowe pozycje są wypełniane <u>zerami</u> (ostatni bit nie przechodzi na początek!).

```
Y=X<<2 //przesuń X 2 bity w lewo
```

# Wyrażenia i operatory

- Operator łączenia
  - Umożliwia łączenie poszczególnych argumentów o znanych rozmiarach. Argumentami mogą być skalary (sieci, rejestry), wektory (sieci, rejestry), części wektorów (zakres bitów) i stałe o znanym rozmiarze.

```
// A=1'b1, B = 2'b00 C = 2'b10, D = 3'b110
Y={B, C} // wynik: Y=4'B0010
Y={A, B, C, D, 3'B001} // wynik:
                        // Y=11'b10010110001
Y={A, B[0], C[1]} // wynik: 3'b101
```

# Wyrażenia i operatory (cd.)

- Operator replikacji
  - Wielokrotne łączenie tego samego argumentu może być wyrażone za pomocą stałej oraz elementu powtarzanego w nawiasach klamrowych:

```
reg A;
reg [1:0] B, C;
reg [2:0] D;
A= 1'b1; B=2'b00; C=2'b01;
Y={4{A}}; // wynik: Y = 4'b1111
Y={ {4{A}}, {2{B}} }; // wynik: Y=8'b11110000
Y={ {4{A}}, {2{C}}, B};// wynik: Y=8'b1111010100
```

# Wyrażenia i operatory (cd.)

- Operator warunkowy
  - wymaga podania trzech argumentów:

    **warunek ? wyrażenie_true : wyrażenie_false;**
  - Najpierw obliczany jest warunek - jeśli wynik obliczeń jest **1**, to wykonywane jest **wyrażenie_true**, w przeciwnym wypadku **wyrażenie_false**. Jeśli wynik wyrażenia jest **x**, to obliczane są **wyrażenie_true** oraz **wyrażenie_false**, a następnie są porównywane bit po bicie. Jeśli bity się nie zgadzają, to w wyniku na tym miejscu będzie wartość bitu **x**.
  - Operator ten przypomina w pracy multiplekser lub polecenie *if-then-else*. Wykorzystuje się go często do przypisań warunkowych.

# Wyrażenia i operatory (cd.)

- Operator warunkowy (cd.)

```
// model bufora trójstanowego
assign addr_buf = drive_enable ? addr_out :
  36'bz;

// model multipleksera 2-do-1
assign out = control ? in1 : in0;
```

- Operatory warunkowe moga być zagnieżdżane:

```
assign out = (A==3) ? (control ? x : y) :
  (control ? m : n);
```

# Wyrażenia i operatory (cd.)

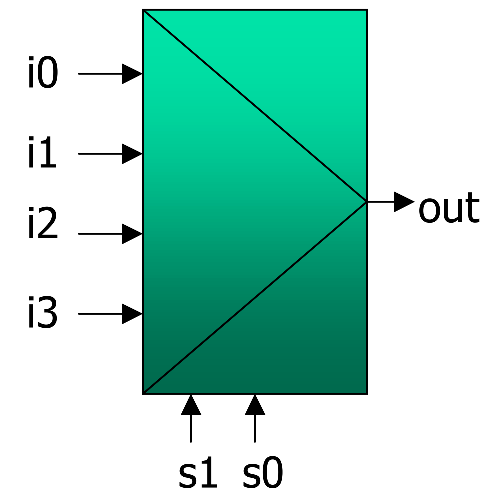
- Priorytet operatorów

| Operator | Symbole | Priorytet |
|---|---|---|
| jednoargumentowe:<br>mnożenie, dzielenie, modulo | + - ! ~<br>* / % | najwyższy |
| dodawanie, odejmowanie<br>przesunięcie | + -<br><< >> | |
| porównanie<br>równość | < <= > >=<br>== != === !== | |
| redukcja<br><br><br>logiczne | & ~&<br>^ ^~<br>\| ~\|<br>&&<br>\|\| | |
| Warunkowe | ?: | najniższy |

# Wyrażenia i operatory (cd.)

- Multiplekser 4-do-1 (Sposób 1: równania logiczne)

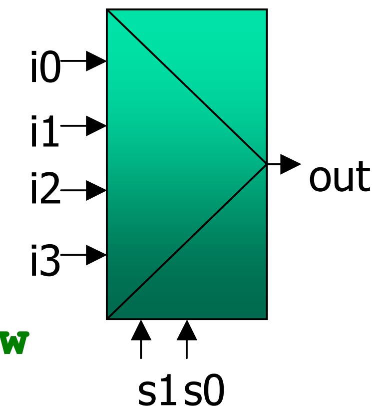
```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
  // Deklaracja portów:
  output out;
  input i0, i1, i2, i3;
  input s1, s0;
  // Równania logiczne:
  assign out = (~s1 & ~s0 & i0) |
               (~s1 & s0 & i1) |
               (s1 & ~s0 & i2) |
               (s1 & s0 & i3);
endmodule
```



# Wyrażenia i operatory (cd.)

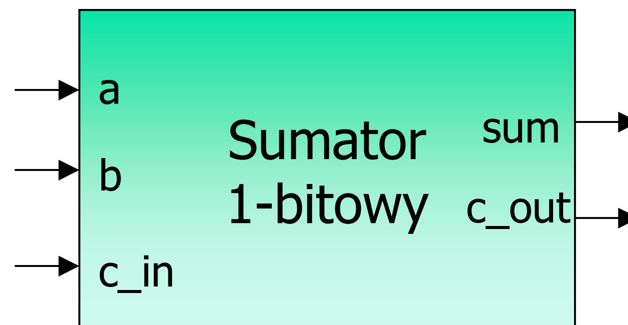
- Multiplekser 4-do-1 (Sposób 2: operator warunkowy)

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
  // Deklaracja portów:
  output out;
  input i0, i1, i2, i3;
  input s1, s0;
  // Wykorzystanie zagnieżdżonych operatorów
  // warunkowych:
  assign out = s1 ? (s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```



# Wyrażenia i operatory (cd.)

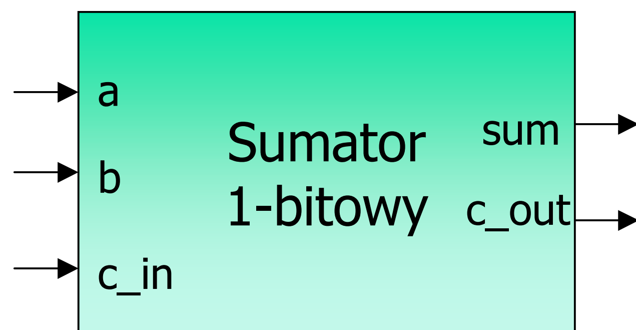
- Przykład – sumator:

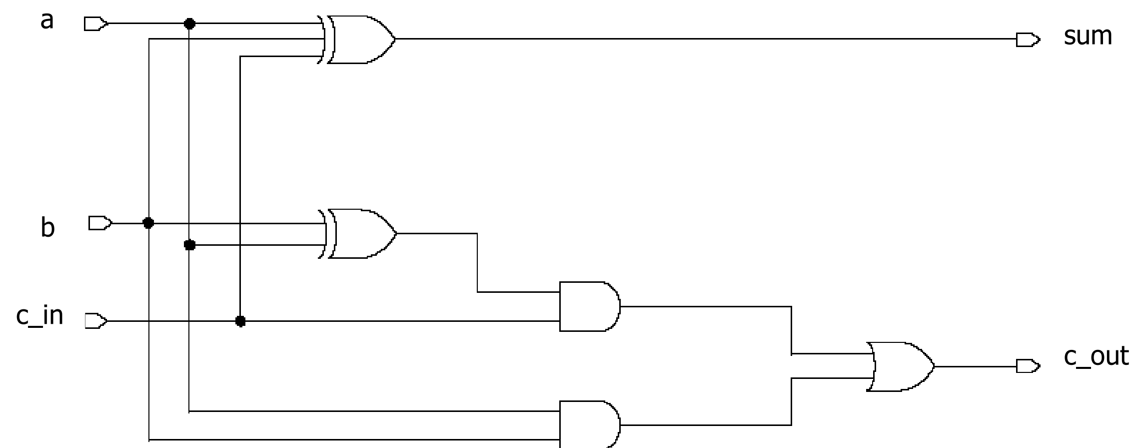



- Jak zrealizować 1-bitowy sumator?

# Wyrażenia i operatory (cd.)

- Przykład – sumator (cd.)

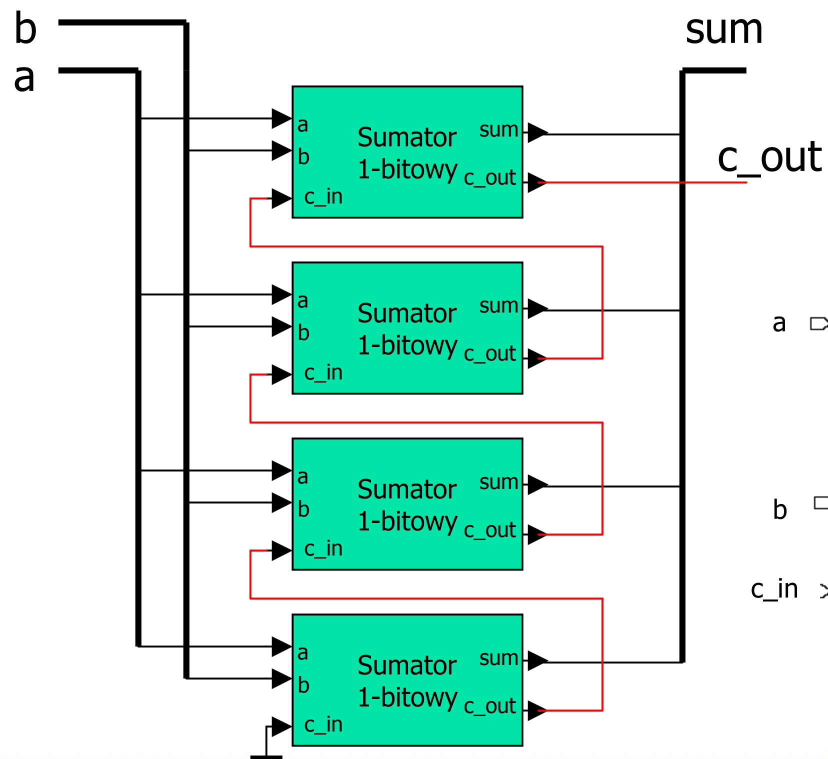


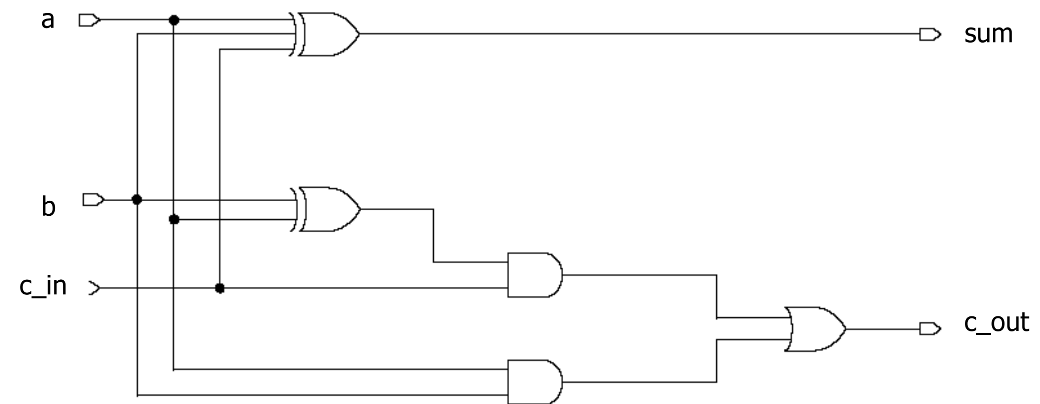




- sum=a⊕b⊕c_in
- c_out = (a⊕b)*c_in + (a*b)

# Wyrażenia i operatory (cd.)

- Przykład – sumator 4-bitowy



# Wyrażenia i operatory (cd.)

- Sumator 4-bitowy (operatory *dataflow*):

```
module fulladd4 (sum, c_out, a, b, c_in);

  // Porty I/O:
  output [3:0] sum;
  output c_out;
  input [3:0] a, b;
  input c_in;

  // Określenie funkcji układu:
  assign {c_out, sum} = a + b + c_in;
endmodule
```

# Wyrażenia i operatory (cd.)

- Sumator 4-bitowy (*carry look ahead*):
  - W tym sumatorze przepełnienie pojawia się na wyjściu szybciej niż dane.

```
module fulladd4(sum, c_out, a, b, c_in);
// Porty I/O:
output [3:0] sum;
output c_out;
input [3:0] a, b;
input c_in;
// linie wewnętrzne:
wire p0, g0, p1, g1, p2, g2, p3, g3;
wire c4, c3, c2, c1;
```

*cd. na następnej planszy ....*

# Wyrażenia i operatory (cd.)

- Sumator 4-bitowy (*carry look ahead*):

```
// Obliczanie p dla każdego stopnia:
assign p0 = a[0] ^ b[0],
       p1 = a[1] ^ b[1],
       p2 = a[2] ^ b[2],
       p3 = a[3] ^ b[3];

// Obliczanie g dla każdego stopnia:
assign g0 = a[0] & b[0],
       g1 = a[1] & b[1],
       g2 = a[2] & b[2],
       g3 = a[3] & b[3];
```

*cd. na następnej planszy ....*

# Wyrażenia i operatory (cd.)

- Sumator 4-bitowy (*carry look ahead*):

```
// Obliczanie przeniesienia dla
// każdego stopnia:
assign c1 = g0 | (p0 & c_in),
       c2 = g1 | (p1 & g0) | (p1 & p0 & c_in),
       c3 = g2 | (p2 & g1) | (p2 & p1 & g0) |
            (p2 & p1 & p0 & c_in),
       c4 = g3 | (p3 & g2) | (p3 & p2 & g1) |
            (p3 & p2 &p1 &g0)| (p3 & p2 & p1 &
       p0 &c_in);
```

*cd. na następnej planszy ....*

# Wyrażenia i operatory (cd.)

- Sumator 4-bitowy (*carry look ahead*):

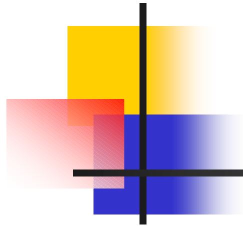
```
// Obliczenie sumy:
assign sum[0] = p0 ^ c_in,
       sum[1] = p1 ^ c1,
       sum[2] = p2 ^ c2,
       sum[3] = p3 ^ c3;

// Przypisanie przeniesienia:
assign c_out = c4;

endmodule
```

# Wyrażenia i operatory (cd.)

- Realizacje sumatorów
  - Brent-Kung *najszybszy*
  - Carry Look Ahead
  - Carry Look Forward
  - Ripple Carry *najmniejszy*



# Część 7. (Verilog)

Modelowanie na poziomie behawioralnym

# Procedury strukturalne

- Bloki **always** i **initial**
  - Polecenia w Verilog'u wykonywane są jednocześnie.
  - Każdy z bloków **always** i **initial** reprezentują osobny ciąg poleceń,
  - Rozpoczynają się w czasie **0**,
  - Nie mogą być zagnieżdżane.

# Procedury strukturalne (cd.)

- Blok **initial**
  - Wykonuje się tylko jeden raz podczas symulacji, począwszy od czasu **0**.
  - Działanie każdego bloku **initial** kończy się niezależnie od pozostałych.
  - Instrukcje wewnątrz bloku muszą być zgrupowane za pomocą **begin** i **end**.
  - Jeśli w bloku występuje tylko jedno polecenie, grupowanie jest zbędne (podobnie jak w Pascalu *begin* i *end*, a w C { }).

*cd. na następnej planszy ....*

# Procedury strukturalne (cd.)

- Blok **initial** (cd.):
  - Polecenia wykonywane są po kolei.
  - Opóźnienie czasowe **#<czas>** oznacza opóźnienie względem **aktualnego** czasu symulacji.
  - Bloki **initial** wykorzystuje się m.in. do inicjalizacji, monitorowania.

# Procedury strukturalne (cd.)

- Blok **initial** (cd.):

```
module stimulus;
  reg x, y,  a, b,  m;
  initial
    m = 1'b0;

  initial
  begin
    #5 a=1'b1;
    #25 b=1'b0;
  end
endmodule
```

| czas | wykonane polecenie |
|---|---|
| 0 | m=1'b0; |
| 5 | a=1'b1; |
| 30 | b=1'b0; |

# Procedury strukturalne (cd.)

- Blok **always**
  - Wszystkie bloki **always** rozpoczynają działanie równocześnie w czasie **0**.
  - Są od siebie niezależne.
  - Wykonują się w sposób ciągły (po zakończeniu ostatniej instrukcji wykonuje się pierwsza).

# Procedury strukturalne (cd.)

- Blok **always** - przykład

```
module clock_gen;
      reg clock;

      initial // = inicjalizacja zegara w czasie 0
        clock = 1'b0;

      always //przełączanie zegara co pół okresu:
        #10 clock = ~clock;

      initial
        #1000 $finish;
endmodule
```

# Przypisanie proceduralne

- Uaktualnia wartości zmiennych typu **reg**, **integer**, **real** lub **time**.
- Wartości zmiennych nie ulegną zmianie, aż do następnego przypisania proceduralnego.
- Przypisanie odbywa się jednorazowo - w odróżnieniu od przypisania **assign**, które działa w sposób ciągły.
- **<lvalue> = <wyrażenie>** lub **<lvalue> <= <wyrażenie>**
- Lewa część przypisania może być:
  - zmienną typu **reg**, **integer**, **real**, **time**;
  - wybranym bitem z powyższych zmiennych;
  - zestawem bitów;
  - połączeniem powyższych elementów.
- Wyróżnia się dwa typy przypisań: **blocking** i **nonblocking**.

# Przypisanie proceduralne (cd.)

- Przypisanie typu **blocking (=)**
  - Wykonywane według kolejności ich umieszczenia w programie.
  - Nie wstrzymują one działania innych, równolegle działających, bloków.
  - <u>Przypomnienie:</u> Wszystkie wyrażenia behawioralne muszą być w bloku **initial** lub **always**.

# Przypisanie proceduralne (cd.)

- Przypisanie typu **blocking (=)** (cd.)

```
module test;
  reg x, y, z;
  initial
    begin
      x = 1'b0;
      y = #100 1'b1;
      z = #100 1'b1;
    end
endmodule
```

| Name | Value | 50 · 100 · 150 · 200 · 250 |
|---|---|---|
| x | 0 | |
| y | 1 | ?????????????? |
| z | 1 | ????????????????????????????????? |

200 ps

- Przypisanie typu **nonblocking (<=)**
  - Pozwala na wykonanie przypisań bez wstrzymywania działania pozostałych instrukcji.
  - Symbol **<=** jest taki sam, jak operator warunkowy "mniejsze lub równe" - znaczenie rozstrzygane jest w zależności od kontekstu.
  - Wszystkie przypisania typu **nonblocking** będą wykonane równocześnie - tj. zostaną zaplanowane do wykonania po odpowiednim opóźnieniu. Zazwyczaj symulator wykonuje wszystkie przypisania **nonblocking** na końcu danego kroku czasowego.

# Przypisanie proceduralne (cd.)

- Przypisanie **nonblocking (<=)** - przykład

```
module test;
  reg x, y, z;
  initial
    begin
      x <= 1'b0;
      y <= #100 1'b1;
      z <= #100 1'b1;
    end
endmodule
```

| Name | Value | 50 · 100 · 150 · 200 · 250 (200 ps) |
|---|---|---|
| x | 0 | |
| y | 1 | XXXXXXXXXXXXX |
| z | 1 | XXXXXXXXXXXXX |

# Przypisanie proceduralne (cd.)

- Różnica pomiędzy przypisaniami typu **blocking** i **nonblocking**.

```
//Przykład I: przypisania blocking
always @(posedge clock)
   a = b;

always @(posedge clock)
   b = a;
```

HAZARD! a i b będą miały taką samą wartość.

```
//Przykład II: przypisania nonblocking
always @(posedge clock)
   a <= b;

always @(posedge clock)
   b <= a;
```

Zamiana a i b odbędzie się prawidłowo.

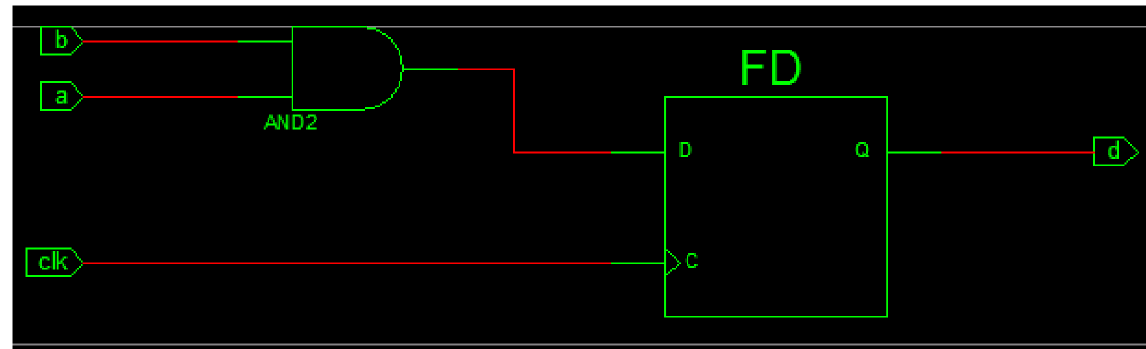
# Przypisanie proceduralne (cd.)

- Różnica pomiędzy przypisaniami typu **blocking** i **nonblocking (cd)**.

```
module test(clk, a, b, d);
  input a;
  input b;
  input clk;
  output d;

  reg c;
  reg d;

  always @(posedge clk)
      begin
            c = a & b;
            d = c;
      end
endmodule
```



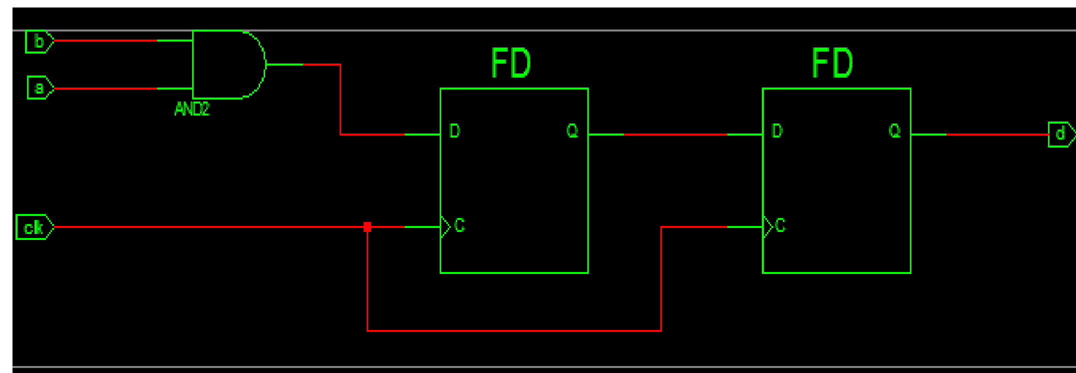
# Przypisanie proceduralne (cd.)

- Różnica pomiędzy przypisaniami typu **blocking** i **nonblocking (cd)**.

```
module test(clk, a, b, d);
   input a;
   input b;
   input clk;
   output d;

   reg c;
   reg d;

   always @(posedge clk)
      begin
             c <= a & b;
             d <= c;
      end
endmodule
```



# Sterowanie wykonaniem instrukcji

- Opóźnienia
  - **#** = czas pomiędzy „napotkaniem” wyrażenia przez symulator a jego wykonaniem.
  - **# <NUMBER>**
  - **# <identyfikator>**
  - **#(<min>:<typ>:<max>)**
  - Dla wyrażeń proceduralnych wyróżnia się trzy typy opóźnień:
    - zwykłe,
    - wewnętrzne,
    - zerowe.

# Sterowanie wykonaniem instrukcji (cd.)

- Opóźnienie zwykłe

```
parameter opoznienie = 20;
parameter delta = 2;
reg x, y, z, p, q;
initial
begin
   x=0; // brak opóźnienia
   #10 y=1; // opóźnienie o 10 jedn.czasowych
   #opoznienie z=0; // opóźnienie określone parametrem
   #(opoznienie+delta) p=1; // opóźn.określ.wyrażeniem
   #y x=x+1 // opóźnienie określone zmienną
   #(4:5:6) q=0 // mimnalne, typowe i maksymalne
end
```

  - Dla grupy **begin-end**, opóźnienie jest odliczane zawsze względem czasu zakończenia poprzedniego polecenia w bloku.

# Sterowanie wykonaniem instrukcji (cd.)

- Opóźnienie wewnętrzne
  - opóźnienie przypisane do prawej części przypisania.

```
reg x, y, z; // definicja rejestrów

//opóźnienia wewnętrzne:
initial
begin
  x=0;
  z=0;
  y = #5 x+z; // pobierz wartość x i z w czasie=0,
              // oblicz x+z i poczekaj 5 jedn.
              // czasu a następnie zapisz wynik do y.
end
```

*cd. na następnej planszy ....*

# Sterowanie wykonaniem instrukcji (cd.)

- Opóźnienie wewnętrzne (cd.)

```
// odpowiednik z wykorzystaniem opóźnień zwykłych oraz
// zmiennych tymczasowych
initial
begin
   x=0; z=0;
   temp_xz = x+z;
   #5 y=temp_xz; // pobierz wart. x+z w czasie bieżącym
                 // i zapisz w zmiennej tymczasowej.
end
```

```
y = #5 x+z;
```

# Sterowanie wykonaniem instrukcji (cd.)

- Opóźnienie zerowe
  - Gdy wyrażenia proceduralne w różnych blokach **always/initial** będą wykonane w tym samym kroku czasu => kolejność ich wykonania nie jest określona.
  - Opóźnienie zerowe = dana instrukcja wykona się jako ostatnia w danym kroku czasu.
  - Jeśli będzie więcej takich instrukcji, to będą one wykonane jako ostatnie, ale kolejność między nimi nie jest określona.

# Sterowanie wykonaniem instrukcji (cd.)

- Opóźnienie zerowe (cd.)

```
initial
begin
    x = 0;
    y = 0;
end

initial
begin
    #0 x = 1;
    #0 y = 1;
end
```

  - Pod koniec czasu **0** zmienne **x** i **y** będą miały wartość **1**.

# Pytanie

- Dane są dwa moduły:

```
module test;
  reg a, b, c;
  initial
    begin
      a = 1'b0;
      b = #100 a;
      c = #100 a;
    end
endmodule
```

```
module test;
  reg a, b, c;
  initial
    begin
      a <= 1'b0;
      b <= #100 a;
      c <= #100 a;
    end
endmodule
```

- Jaki będzie wynik ich działania?

# Odpowiedź

# Sterowanie wykonaniem instrukcji

- Sterowanie przy użyciu zdarzeń
  - Zdarzenie = zmiana wartości rejestru lub sieci.
  - Zdarzenia mogą być wykorzystane do wyzwolenia wykonania bloku instrukcji.
  - Są 4 typy sterowania zdarzeniem:
    - zwykłe sterowanie przy użyciu zdarzeń,
    - sterowanie przy użyciu nazwanych zdarzeń,
    - sterowanie przy użyciu zdarzeń typu OR,
    - sterowanie poziomem sygnału.

# Sterowanie wykonaniem instrukcji (cd.)

- Zwykłe sterowanie przy użyciu zdarzeń
  - Do sterowania zdarzeniami służy symbol @.
  - Instrukcje mogą reagować na:
    - zmianę wartości sygnału,
    - rosnące (**posedge**) zbocze,
    - opadające (**negedge**) zbocze.

```
@clock q = d; // poczekaj na zmianę clock i przypisz q=d
@(posedge clock) q = d; // q=d gdy clock zmienia się:
                        // 0->1,0->x,0->z,x->1,z->1
@(negedge clock) q = d; // q=d gdy clock zmienia się:
                        // 1->0,1->x,1->z,x->0,z->0
q=@(posedge clock) d;   // d jest oblicz.natychmiast,
                        // a wart.jest przypis.do q
                        // podczas rosn.zbocza clock.
```

# Sterowanie wykonaniem instrukcji (cd.)

- Sterowanie przy użyciu nazwanych zdarzeń
  - Wyrażenie nazywa się słowem kluczowym **event**.
  - Wyzwalanie zdarzenia: **->**.
  - Wykrywanie wyzwolenia zdarzenia: **@**.

```
event received_data;     // definicja zdarzenia

always @(posedge clock) // wyzwolenie zdarzenia
begin
   if(last_data_packet)   // jeśli to ostatni pakiet
     ->received_data;     // to wyzwól zdarzenie
end

always @(received_data) // oczekuj na zdarzenie
  data_buf={data_pkt[0], data_pkt[1], data_pkt[2],
   data_pkt[3]};
```

# Sterowanie wykonaniem instrukcji (cd.)

- Sterowanie przy użyciu zdarzeń typu OR
  - wyzwolenie może nastąpić poprzez jeden z kilku sygnałów:

```
// zatrzask wyzwalany poziomem z asynchr.resetem
always @(reset or clock or d)
   // czekaj na zmianę sygnału reset, clock lub d
begin
   if (reset) //jeśli reset ma stan wysoki, ustaw q=0
       q = 1'b0;
   else if (clock) //jeśli clock ma stan wysoki
       q = d;
end
```

# Sterowanie wykonaniem instrukcji (cd.)

- Sterowanie poziomem sygnału
  - **wait** - oczekiwanie na określoną wartość sygnału.

*czeka, aż to wyrażenie będzie spełnione (true)*

```
always
  wait (count_enable) #20 count = count + 1;
```

  - **count_enable** jest monitorowane cały czas.
  - Jeśli **count_enable** = 0, to wyrażenie nie jest wykonywane.
  - Jeśli **count_enable** = 1 to **count = count + 1** zostanie wykonane po 20 jednostkach czasu i będzie powtarzane co 20 jednostek, dopóki wartość **count_enable** ma wartość **1**.

# Wyrażenie warunkowe

- typ I - brak **else**:

```
if (<wyrażenie>) polecenie_true;
```

- typ II - jedno **else**:

```
if (<wyrażenie>) polecenie_true;  else
  polecenie_false;
```

- typ III – skomplikowane:

```
if (<wyrażenie1>) polecenie_true1;
else if (<wyrażenie2>) polecenie_true2;
else if (<wyrażenie3>) polecenie_true3;
else polecenie_domyślne;
```

# Wyrażenia typu **case**

- Wyrażenie **case**

```
case (wyrażenie)
  alternatywa_1: polecenie1;
  alternatywa_2: polecenie2;
  alternatywa_3: polecenie3;
  ...
  ...
  default: polecenie_domyślne;
endcase
```

# Wyrażenia typu **case** (cd.)

- Wyrażenie **case** (cd.)
  - Pierwsze dopasowanie spowoduje uruchomienie odpowiedniego polecenia.
  - Wartości porównywane są bit po bicie, także wartości **x** i **z** są brane pod uwagę.
  - Jeśli nie zgadza się ilość bitów, to krótsze wyrażenie uzupełniane jest zerami, aby długość była identyczna.
  - Dopuszcza się oddzielenie przecinkiem kilku alternatyw.
  - Wielokrotne użycie części **default** nie jest dozwolone.
  - Wyrażenia **case** mogą być zagnieżdżone.

# Wyrażenia typu **case** (cd.)

- Przykład wyrażenia **case**:

```
reg [1:0] alu__control;
...
...
case (alu_control)
  2'd0        : y = x + z;
  2'd1, 2'd2 : y = x - z;
  default : $display("Invalid ALU ctr");
endcase
```

# Wyrażenia typu **case** (cd.)

- Wyrażenie **casex** i **casez**
  - Są dwie modyfikacje wyrażenia case:
    - **casez** traktuje wszystkie wartości **z** jako nieznaczące. Wszystkie bity o wartości **z** można też zapisać za pomocą **?**.
    - **casex** traktuje wszystkie wartości **x** i **z** jako nieznaczące.

# Pętle

- Pętla **while**
  - Uruchamia się dopóki **wyrażenie** jest prawdziwe.
  - Jeśli **wyrażenie** jest fałszywe od początku, to pętla **while** nie uruchomi się w ogóle.

```
while (count < 128)
begin
  $display("count=%d", count);
  count = count + 1;
end
while ((i<16) && continue)
begin
...
end
```

# Pętle (cd.)

- Pętla **for**
  - Pętla for zawiera trzy części:
    - warunek początkowy;
    - warunek trwania w pętli;
    - przypisanie zmieniające zmienną pętli.

```
integer count;

initial
   for (count=0; count<128; count=count+1)
      $display("Count=%d",count);
```

# Pętle (cd.)

- Pętla **repeat**
  - Wykonuje polecenie określoną ilość razy.
  - Musi zawierać liczbę powtórzeń, (stała, zmienna lub sygnał).
  - Ilość powtórzeń jest obliczana tylko raz na początku uruchamiania polecenia **repeat**.

```
repeat(128)
begin
   $diplay("Count=%d", count);
   count=count+1;
end
```

# Pętle (cd.)

- Pętla **forever**
  - Wykonuje się aż do napotkania polecenia **$finish**.
  - Jest równoważna pętli **while(1)**.
  - Pętlę **forever** najczęściej używa się wraz z poleceniami kontrolującymi czas symulacji.
  - Gdyby ich nie było, pętla wykonywała by się w nieskończoność, wstrzymując pozostałe polecenia.

```
initial
begin
   clock = 1'b0;
   forever #10 clock = ~clock;
end
```

# Bloki sekwencyjne i równoległe

- Typy bloków: **sekwencyjne** i **równoległe**.
- Bloki **sekwencyjne**
  - Zgrupowanie poleceń w blok: **begin** i **end**.
  - Polecenia wykonywane są w kolejności, w jakiej są tam umieszczone.
  - Następne polecenie jest wykonywane, gdy zakończy się poprzednie (z wyjątkiem przypisań **nonblocking** w wewnętrznym opóźnieniem).
  - Jeśli określone jest opóźnienie lub kontrola wydarzeniem, jest ono określane względem czasu zakończenia poprzedniego polecenia w bloku.

# Bloki sekwencyjne i równoległe (cd.)

- Bloki **równoległe**
  - Zgrupowanie poleceń w blok: **fork** i **join**.
  - Wyrażenia wykonywane są jednocześnie;
  - Kolejność wykonywania poleceń jest określona poprzez opóźnienia lub wydarzenia związane z konkretnymi poleceniami.
  - Opóźnienia liczone są względem czasu wejścia do bloku.

# Bloki sekwencyjne i równoległe (cd.)

- Bloki **równoległe** - przykład

```
reg x, y;
reg [1:0] z, w;

initial
fork
   x=1'b0;          // wykonuje się w czasie=0
   #5 y=1'b1;       // wykonuje się w czasie=5
   #10 z={x,y};     // wykonuje się w czasie=10
   #20 w={y,x};     // wykonuje się w czasie=20
join
```

- Należy uważać na zjawisko hazardu, mogące wystąpić w bloku równoległym.

# Bloki sekwencyjne i równoległe (cd.)

- Cechy specjalne bloków
  - Bloki mogą być zagnieżdżane.
  - Można mieszać bloki sekwencyjne i równoległe.

```
// zagnieżdżanie bloków
initial
begin
   x=1'b0;
   fork
       #5 y=1'b1;
       #10 z={x,y};
   join
   #20 w={y,x};
end
```

# Bloki sekwencyjne i równoległe (cd.)

- Bloki z nazwą
  - Bloki mogą mieć nadane nazwy.
  - Można deklarować zmienne lokalne dla bloków z nazwą.
  - Bloki z nazwą są częścią hierarchii projektu. Zmienne bloków mogą być dostępne przy wykorzystaniu pełnej nazwy hierarchicznej.
  - Bloki z nazwą mogą być dezaktywowane, tj. ich wykonywanie może być zatrzymane.

# Bloki sekwencyjne i równoległe (cd.)

- Bloki z nazwą - przykład

```
//bloki z nazwą:
module top;
initial
begin: block1 // blok sekwencyjny block1
  integer i; //i jest zmienną statyczną i lokalną
             //dostęp do niej: top.block1.i
  ...
end

initial
fork: block2 // blok równoległy nazwany block2
  reg i; // i jest zmieną statyczna i lokalną
  ...    // dostęp do niej: top.block2.i
join
```

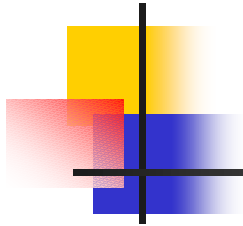
# Bloki sekwencyjne i równoległe (cd.)

- Dezaktywacja bloków z nazwą
  - Słowo **disable** powoduje zakończenie wykonywania poleceń bloku lub zadania.
  - W ten sposób można wychodzić z pętli, obsługiwać błędy, itp.
  - Wyłączenie bloku powoduje przejęcie kontroli nad programem przez polecenia występujące bezpośrednio za blokiem.
  - Wyłączenie bloku jest podobne do polecenia **break** w C.
  - Różnicą jest, że **break** powoduje przerwanie aktualnej pętli, a polecenie **disable** może przerwać wykonywanie dowolnego bloku.
  - Polecenie **disable** nie może kończyć działania funkcji (tylko zadanie).

# Bloki sekwencyjne i równoległe (cd.)

- Przykłady **disable**:

```
module test;
  integer a, i;
  initial
    begin : blok_testowy1
      a = 1; // - ta instrukcja się wykona,
      disable blok_testowy1;
      a = 2; // - nigdy się nie wykona!
    end
  initial
    begin : blok_testowy2
      i = 0;
      forever
        begin
          if (i==a) disable blok_testowy2;
          #1 i = i + 1;
        end
    end
endmodule
```



# Część 8. (Verilog)

## Zadania i funkcje

# Różnice pomiędzy zadaniami i funkcjami

| **Funkcje** | **Zadania** |
|---|---|
| Funkcja może uruchamiać inną funkcję, nie może uruchamiać zadania. | Zadanie może uruchamiać inne zadania i funkcje |
| Czas wykonywania funkcji jest zawsze zerowy | Zadanie może wykonywać się przez czas niezerowy. |
| Funkcja nie może zawierać opóźnień, wydarzeń ani innych struktur kontroli czasowej | Zadanie może zawierać opóźnienia, wydarzenia i inne struktury kontroli czasowej |
| Funkcja musi posiadać przynajmniej jeden argument (może być więcej) typu **input**. | Zadanie może mieć zero lub więcej argumentów typu **input**, **output** lub **inout**. |
| Funkcja zawsze zwraca pojedynczą wartość. | Zadanie nie zwraca wartości, ale może zwracać wartości poprzez argumenty typu **output** lub **inout**. |

# Cechy zadań i funkcji

- Muszą być zdefiniowane w module
- Są lokalne dla danego modułu.
- Mogą zawierać lokalne zmienne, rejestry, zdarzenia itp.
- Nie mogą zawierać zmiennych typu **wire**.
- Zawierają tylko wyrażenia behawioralne.
- Nie zawierają bloków **initial**/**always**, ale często są wywoływane przez te bloki.

# Funkcje

- Deklaruje się za pomocą słów kluczowych **function** i **endfunction**.
- Funkcje wykorzystuje się, gdy:
  - nie ma opóźnień lub zdarzeń w procedurze;
  - procedura zwraca dokładnie jedną wartość;
  - procedura posiada przynajmniej jeden argument wejściowy.

# Funkcje (cd.)

- Deklaracja funkcji i jej wywołanie
  - Gdy zdeklarowano funkcję, automatycznie tworzona jest zmienna typu **reg** o takiej samej nazwie jak nazwa funkcji.
  - Wartość funkcji jest zwracana w ten sposób, że ustawiana jest wspomniana zmienna typu **reg**.
  - Funkcja <u>nie może</u> wywoływać innych zadań.
  - Funkcja <u>może</u> wywoływać inne funkcje.

# Funkcje (cd.)

- Przykład:

```
module shifter;
  `define LEFT_SHIFT          1'b0
  `define RIGHT_SHIFT         1'b1
  reg [31:0] addr, left_addr, right_addr;
  reg control;

  // definicja f-cji shift (zwraca 32-bit.wart.)
  function [31:0] shift;
    input [31:0] address;
    input control;
    shift=(control==`LEFT_SHIFT)?
          (address<<1):(address>>1);
  endfunction
  // cd na następnej planszy...
```

# Funkcje (cd.)

- Przykład (cd.):

```
// ... cd z poprzedniej planszy

  always @(addr)
    begin
      // wywołaj funkcję:
      left_addr = shift(addr, `LEFT_SHIFT);
      right_addr = shift(addr, `RIGHT_SHIFT);
    end
endmodule
```

# Zadania

- Zadania deklaruje się za pomocą słów kluczowych **task** i **endtask**.
- Zadania wykorzystuje się, gdy:
  - potrzebne jest opóźnienie lub zdarzenie w procedurze;
  - procedura posiada zero lub więcej argumentów wyjściowych;
  - procedura nie posiada argumentów wejściowych.

# Zadania (cd.)

- Deklaracja zadania:

```
task <nazwa_zadania>;
  <deklaracja>*
  <polecenia>
endtask
```

  - Deklaracje argumentów: **input**, **output** lub **inout**.
  - Argumenty typu **input** i **inout** są przekazywane do zadania.
  - Argumenty typu **output** i **inout** są zwracane z powrotem po zakończeniu zadania.
  - Zadanie może wywoływać inne zadania lub funkcje.
  - Zadanie może operować na zmiennej typu **reg** zdefiniowanej w module.

# Zadania (cd.)

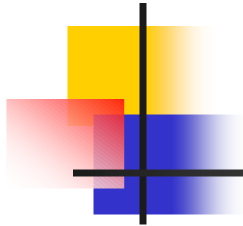
- Przykład:

```
module operation; // definicja modułu z zadaniem
parameter delay = 10;
reg [15:0] A, B;
reg [15:0] AB_AND, AB_OR, AB_XOR;

task bitwise_oper; //definicja zadania
  output [15:0] ab_and, ab_or, ab_xor;
  input [15:0] a, b; // argumenty wejściowe
  begin
    #delay ab_and = a & b;
    ab_or = a | b;
    ab_xor = a ^ b;
  end
endtask
// cd na następnej planszy...
```

- Przykład (cd.):

```
// ... cd. z poprzedniej planszy
always @(A or B) //gdy A lub b zmienią wartość
begin
  bitwise_oper(AB_AND, AB_OR, AB_XOR, A, B);
end
endmodule
```



# Część 9. (Verilog)

## Techniki modelowania

# Proceduralne przypisanie ciągłe

- *Przypisanie proceduralne przypisuje wartość do rejestru. Rejestr zachowuje daną wartość, aż do następnego wpisu.*
- Proceduralne przypisanie ciągłe - zmienne lub wyrażenia są w sposób ciągły podawane do rejestru przez określony okres czasu.
- Dwa typy przypisań
  - Typ 1: **assign** i **deassign**
  - Typ 2: **force** i **release**

# Proceduralne przypisanie ciągłe (cd.)

- Typ 1: **assign** i **deassign**
  - Lewa cześć przypisania może być tylko rejestrem lub połączeniem rejestrów.
  - Nie może być częścią bitów z sieci ani też macierzą rejestrów.
  - Przypisanie ciągłe ma większy priorytet niż zwykłe przypisania proceduralne.
  - Poniższy przykład pokazuje realizację przerzutnika **D** z asynchronicznym resetem.

# Proceduralne przypisanie ciągłe (cd.) – przykład *assign*

```
module dff (q, qbar, d, clk, rst);
output q, qbar;
input d, clk, rst;
reg q, qbar;

always @(negedge clk)
  begin
    q = d;
    qbar = ~d;
  end

always @(rst)
  if (rst)
    begin
      assign q = 1'b0;
      assign qbar = 1'b1;
    end
  else
    begin
      deassign q;
      deassign qbar;
    end
endmodule
```

# Proceduralne przypisanie ciągłe (cd.)

- Typ 2: **force** i **release**
  - Pozwalają zapisywać wartości do rejestrów i sieci.
  - Polecenia te wykorzystywane są głównie przy debuggingu.
  - Rejestry:
    - **force** ma większy priorytet niż przypisania proceduralne czy ciągłe.
    - Po **release** wartość będzie pamiętana, ale będzie już można ją zmienić.
  - Sieci:
    - Przypisanie **force** nadpisuje jakiekolwiek inne wartości.
    - Po **release** następuje powrót do normalnej wartości wymuszanej na sieci.

# Proceduralne przypisanie ciągłe (cd.) – przykład *force*

```
module stimulus;
   dff DFF1( Q, Qbar, D, CLK, RST);

   initial
        begin
                #50 force DFF1.q = 1'b1;
                #50 release DFF1.q;
        end
endmodule

module top;
   wire out, a, b, c;

   assign out = a & b & c;
   initial
        begin
                #50 force out = a | b & c;
                #50 release out;
        end
endmodule
```

# Definiowanie parametrów

- Parametry umożliwiają ustawienie pewnych wartości w modułach podczas ich powoływania do życia.
- Są dwie możliwości podawania parametrów:
  - poprzez **defparam**
  - podczas powoływania modułu

# Definiowanie parametrów (cd.)

- Polecenie **defparam**:
  - Polecenie **defparam** umożliwia ustawienie parametrów w dowolnym module:
  - W module może być wiele poleceń **defparam**.

```
module text;
   parameter id = 0;
   initial
       $display("This module number = %d", id);
endmodule

module top;
   defparam T1.id = 1, T2.id = 2;
   text T1();
   text T2();
endmodule
```

# Definiowanie parametrów (cd.)

- Ustawienie parametrów podczas powoływania do życia

```
module top;
        text #(1) T1();
        text #(2) T2();
endmodule
```

- W przypadku kilku parametrów, należy je wymienić po przecinku, w takiej kolejności, w jakiej zostały one zdefiniowane w definicji modułu:

```
text3 #(1,4,5) w1();
```

# Warunkowa kompilacja i uruchamianie

- Warunkowa kompilacja
  - Podanie słów kluczowych **`ifdef**, **`else** oraz **`endif**.
  - Warunki sprawdzane przez **`ifdef** definiuje się za pomocą polecenia **`define**.

```
`ifdef beh // zdefiniowano beh (`define)
  `include "file1_beh.v";
  `include "file2_beh.v";
`else // beh nie zdefiniowane
  `include "file1_synt.v";
  `include "file2_synt.v";
`endif
```

# Warunkowa kompilacja i uruchamianie (cd.)

- Skala czasu
  - W jednym module czas w *ns* a w innym w *ms*.
  - Verilog umożliwia definicję jednostki czasu dla modułu.

```
`timescale <jednostka_czasu>/<dokładność>
```

  - **<jednostka_czasu>** = jednostka czasu dla czasu symulacji oraz opóźnień.
  - **<dokładność>** = z jaką dokładnością zaokrąglane są opóźnienia podczas symulacji.
  - Do określenia obydwu tych wartości można wykorzystać tylko trzy wartości: 1, 10 i 100.
  - Przykłady:

```
`timescale 100 ns / 1 ns
`timescale 1 us / 10ns
```

# Zadania systemowe

- Pliki - otwieranie plików

```
$fopen("<nazwa_pliku>");
<file_handle> = $fopen("<nazwa_pliku>");
```

  - Zadanie **$fopen** zwraca deskryptor typu **integer**.
  - W deskryptorze tylko jeden bit jest ustawiony.
  - *Standard output* posiada deskryptor z ustawionym pierwszym bitem (tzw. kanał **0**) = zawsze otwarty.
  - Każde wywołanie funkcji **$fopen** otwiera nowy kanał z kolejnym bitem ustawionym na **1**, aż do **31**.
  - Zaletą takiego rozwiązania jest możliwość selektywnego zapisu do kilku plików jednocześnie.

# Zadania systemowe (cd.)

- Pliki - otwieranie plików - przykład

```
module file_text;
  integer handle1, handle2, handle3;

  initial
  begin
    handle1=$fopen("file1.out");
    handle2=$fopen("file2.out");
    handle3=$fopen("file3.out");
    $display ("Handle1 = %b, %d", handle1, handle1);
    $display ("Handle2 = %b, %d", handle2, handle2);
    $display ("Handle3 = %b, %d", handle3, handle3);
    $fdisplay (1, "Text on standard output");
  end
endmodule
```

# Zadania systemowe (cd.)

- Pliki - otwieranie plików – przykład (cd.)

```
# Simulation has been initialized
# Selected Top-Level: file_text (file_text)
run 100 ns
# KERNEL: Handle1 = 00000000000000000000000000000010,    2
# KERNEL: Handle2 = 00000000000000000000000000000100,    4
# KERNEL: Handle3 = 00000000000000000000000000001000,    8
# KERNEL: Text on standard output
# KERNEL: stopped at time: 100 ns
# KERNEL: Simulation has finished. There are no more test
  vectors to simulate.
```

# Zadania systemowe (cd.)

- Pisanie do pliku
  - Do pisania do pliku służą następujące zadania:
    - **$fdisplay**,
    - **$fmonitor**,
    - **$fwrite $fstrobe**.
  - Działają one podobnie jak **$display** i **$monitor**. Rozważone zostaną tylko **$fdisplay** i **$fmonitor**.

```
$fdisplay(<deskryptor>, p1, p2, ... ,pn);
$fmonitor(<deskryptor>, p1, p2, ... ,pn);
```

  - **p1**, **p2**,..., **pn** = zmienne, sygnały czy łańcuchy.
  - Deskryptor może być kombinacją kilku deskryptorów wielokanałowych - Verilog będzie pisał do wszystkich, które mają ustawioną **1** na odpowiednim miejscu.

# Zadania systemowe (cd.)

- Zamykanie pliku

```
$fclose(<handle>);
```

- Wyświetlanie hierarchii
  - Hierarchia na dowolnym poziomie może być wydrukowana przy użyciu **%m**.
- **Strobe**
  - Polecenie **$strobe** jest podobne do **$display**, ale wykonuje się ono zawsze pod koniec danego kroku czasu - jest wtedy pewność, że wykonały się już wszystkie przypisania.

# Verilog 2001 – nowe elementy języka

- Blok konfiguracji

```
/* nadanie nazwy bloku konfiguracyjnego: */
config cfg4
  /* określenie modułu toplevel: */
  design rtlLib.top
  /* domyślna kolejność przeszukiwania bibliotek: */
  default liblist rtlLib gateLib;
  /* wskazanie biblioteki dla konkretnych modułów:*/
  instance test.dut.a2 liblist gateLib;
endconfig
```

# Verilog 2001 – nowe elementy języka (cd.)

- Polecenie **generate**

```
genvar i;
generate
  for(i=0; i<SIZE; i=i+1)
    begin:addbit
      wire n1,n2,n3; //internal nets
      xor g1 ( n1, a[i], b[i]);
      xor g2 (sum[i],n1, c[i]);
      and g3 ( n2, a[i], b[i]);
      and g4 ( n3, n1, c[i]);
      or g5 (c[i+1],n2, n3);
    end
endgenerate
```

# Verilog 2001 – nowe elementy języka (cd.)

- Polecenie **generate** (cd.)

```
generate
  if((a_width < 8) || (b_width < 8))
    CLA_multiplier #(a_width, b_width) u1 (a, b, product);
  else
    WALLACE_multiplier #(a_width, b_width) u1 (a, b, product);
endgenerate
```

# Verilog 2001 – nowe elementy języka (cd.)

- Nowy sposób indeksowania wektorów

```
[base_expr +: width_expr] //positive offset
[base_expr -: width_expr] //negative offset

reg [63:0] word;
reg [3:0] byte_num; //a value from 0 to 7
wire [7:0] byteN = word[byte_num*8 -: 8];
```

- **Base** może być zmienne
- **Width** musi być stałe

# Verilog 2001 – nowe elementy języka (cd.)

- Tablice wielowymiarowe

```
//1-wymiarowa tablica 8-bitowych zmiennych reg
//(dozwolone w Verilog-1995 oraz Verilog-2001)
reg [7:0] array1 [0:255];
wire [7:0] out1 = array1[address];
//3-wymiarowa tablica 8-bitowych zmiennych wire
//(nowość w Verilog-2001)
wire [7:0] array3 [0:255][0:255][0:15];
wire [7:0] out3 = array3[addr1][addr2][addr3];
```

# Verilog 2001 – nowe elementy języka (cd.)

- Indeksowanie zakresów w tablicach

```
reg [31:0] array2 [0:255][0:15];

wire [7:0] out2 = array2[100][7][31:24];
```

# Verilog 2001 – nowe elementy języka (cd.)

- Operacje na liczbach **signed**
  - **reg** oraz **net** mogą być zadeklarowane jako **signed**
  - Funkcja może zwracać wartości **signed**
  - Liczby **integer** mogą być zadeklarowane jako **signed**
  - Możliwa jest konwersja **signed** ←→ **unsigned**
  - Dodano operatory przesunięcia arytmetycznego

# Verilog 2001 – nowe elementy języka (cd.)

- Operacje na liczbach **signed** (cd.)

```
reg signed [63:0] data;
wire signed [7:0] vector;
input signed [31:0] a;
function signed [128:0] alu;
```

```
16'hC501 //16-bitowa liczba hex unsigned
16'shC501 //16-bitowa liczba hex signed
```

```
reg [63:0] a; //wartość unsigned
always @(a) begin
  result1 = a / 2; //operacja na wartościach unsigned
  result2 = $signed(a) / 2;// operacja na wart. signed
end
```

```
// D=8'b10100011
D >> 3 //logical shift → 8'b00010100
D >>> 3 //arithmetic shift → 8'b11110100
```

# Verilog 2001 – nowe elementy języka (cd.)

- Operator potęgowania **
- Domyślne reagowanie na wszystkie zdarzenia

```
always @* //dla układów kombinacyjnych
  if (sel)
    y = a;
  else
    y = b;
```

- Wymienianie sygnałów po przecinku:

```
always @(a or b or c or d or sel)
// jest równoważne:
always @(a, b, c, d, sel)
```

# Verilog 2001 – nowe elementy języka (cd.)

- Nowe polecenia pracy z plikami
  - Można jednocześnie otworzyć $2^{20}$ plików
  - Polecenia:

    `$ferror, $fgetc, $fgets, $fflush, $fread, $fscanf, $fseek, $fscanf, $ftel, $rewind $ungetc.`
- Polecenia formatowania łańcucha:

  `$sformat, $swrite, $swriteb, $swriteh, $swriteo and $sscanf.`

# Verilog 2001 – nowe elementy języka (cd.)

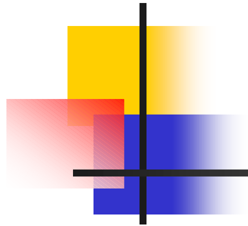
- Połączenie deklaracji portu z typem danych
  - Nie trzeba osobno deklarować **input** a potem **reg**

```
module mux8 (y, a, b, en);
  output reg [7:0] y;
  input wire [7:0] a, b;
  input wire en;
```

# Verilog 2001 – nowe elementy języka (cd.)

- Deklaracja portów podobna do ANSI-C

```
module mux8 (output reg [7:0] y,
             input wire [7:0] a,
             input wire [7:0] b,
             input wire en );
```



# Część 10 (VHDL)

## Wstęp

# Geneza powstania języka

- VHDL = VHSIC Hardware Description Language,
- Prace badawcze 1983-1985, IBM, Intermetics oraz Texas Instruments, sponsorowane przez Departament Obrony USA.
- 1987: standard IEEE 1076-1987
- 1993: znowelizowano standard 1076-1993.

# Geneza powstania języka (cd.)

- Rozszerzenia języka VHDL:
  - **1076.1** – VHDL-AMS
  - **1076.2** – **math_real**, **math_complex**
  - **1076.3** – **numeric_bit**, **numeric_std**
  - **1076.4** – **vital**
  - **1164** – **std_logic**

# Poziomy opisu

**Verilog**

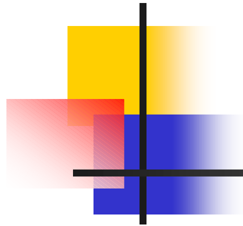
- *Poziom kluczy*
- Poziom bramek logicznych
- Poziom rejestrów (Dataflow)
- Poziom behawioralny

**VHDL:**

- Poziom strukturalny
- Poziom przesłań międzyrejestrowych RTL
- Abstrakcyjny poziom behawioralny

# Plan części wykładu dot. języka VHDL

- Składnia języka i typy danych
- Biblioteki
- Poziom strukturalny
- Poziom przesłań międzyrejestrowych RTL
- Poziom behawioralny
- Funkcje i procedury
- Pakiety
- Synteza układów kombinacyjnych
- Atrybuty
- Symulacja i testowanie
- Praca z plikami
- Uwagi dotyczące syntezy



# Część 11 (VHDL)

## Składnia języka i typy danych

# Konwencja nazw

- **Nie** są rozróżniane duże i małe litery
  - ENTITY = EntitY
- Komentarz:

```
RESET : in std_ulogic; --Zerowanie o szerokości
-- dwu mikrosekund
```

# Typy danych

- Typ wyliczeniowy

```
type <Nazwa_typu> is ( <wartość_1>, <wartość_2>,
   ..., <wartość_n>);
```

  - Przykłady:

```
type t_KOLOR is (NIEBIESKI, ZIELONY, CZERWONY);
type t_MOJTYP is ('0', '1', 'U', 'Z');
variable x_v : t_KOLOR;
signal a: t_MOJTYP;
...
x_v := NIEBIESKI;
a <= 'Z';
```

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

# Typy danych (cd.)

- Typy tablicowe - <u>Ograniczony</u> typ tablicowy

```
type <Nazwa_typu_tablicowego> is array
   (<Zakres_całkowity>) of <Typ_elementu_tablicy>
```

- Przykład:

```
type t_A is array (7 downto 0) of std_logic;
type t_B is array (0 to 7)     of std_logic;
signal reg_a : t_A;
signal reg_b : t_B;
```

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

# Typy danych (cd.)

- Typy tablicowe - Nieograniczony typ tablicowy

```
type <Nazwa_typu_tablicowego> is array
   (<Nazwa_typu_całkowitego> range <>) of
   <Typ_elementu_tablicy>
```

  - Przykłady:

```
type bit_vector is array (integer range <>) of bit;
type std_logic_vector is array (integer range <>) of
   std_logic;
...
signal moj_wektor1 : std_logic_vector (5 downto -5);
signal mój_wektor2 : std_logic_vector (1 to 11);
```

- Typy tablicowe - Dostęp do elementów tablicy

```
<identyfikator_tablicy>(<wyrażenie>)
```

  - Przykłady:

```
tab(6);
tab(x);
tab(3 to 6);
```

# Typy danych (cd.)

- Tablice wielowymiarowe
  - Syntezowalne tablice dwuwymiarowe :

```
type t_BAJT   is array (7 downto 0) of std_logic;
type t_WEKTOR is array (3 downto 0) of t_BAJT;
type t_WEKTOR is array (3 downto 0) of
   std_logic_vector(7 downto 0);
```

  - Tablice wielowymiarowe (niesyntezowalne):

```
type t_MULTI is array ( 7 downto 0, 255 downto 0)
   of std_logic;
```

# Typy danych (cd.)

- Typ rzeczywisty

```
variable x_v : real;
...
x_v := 1.234;
```

- Typ fizyczny

```
wait for 10 ns;
```

# Typy danych (cd.)

- Konwersja typów

```
type t_T1 is range 0 to 255;
signal a1 : t_T1;
signal a2 : integer range 0 to 255;
if a1 = a2 then -- =błąd składniowy (type mismatch,
...                 --operator not defined for such
                    --operands).

-- JAK ROZWIĄZAĆ TAKI PROBLEM?
```

# Typy danych (cd.)

- Konwersja typów (cd.)

```
-- ROZWIĄZANIE PROBLEMU:
type t_T1 is range 0 to 255;
signal a1 : t_T1;
signal a2 : integer range 0 to 255;

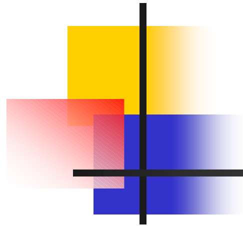
if a1 = t_T1(a2) then -- nie ma błędu!
...
```

# Typy danych (cd.)

- Aliasy

```
signal addr : std_logic_vector(31 downto 0);
alias top: std_logic_vector (3 downto 0) is addr (31
   downto 28);
...
  wait for 10 ns;
  top <= "1111";
  wait for 10 ns;
  addr <= (others => '0');
```

| Name | 0–10 | 10–20 | 20– |
|---|---|---|---|
| addr | UUUUUUUU | FUUUUUUU | 00000000 |
| top | U | F | 0 |



# Część 12 (VHDL)

Biblioteki

# Biblioteki (cd.)

```
library <Nazwa_biblioteki>;
use <Nazwa_biblioteki>.<Nazwa_pakietu>.all;
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
```

nazwa biblioteki

nazwa pakietu

wyrażenie "all" oznaczające wszystkie elementy pakietu lub nazwa składnika pakietu

```
library IEEE;

use IEEE.std_logic_1164.my_func
```

# Struktura bibliotek

- Biblioteka **std**:
  - Pakiet **standard**
  - Pakiet **textio**
- Biblioteka **IEEE**:
  - Pakiet **std_logic_1164**
  - Pakiet **numeric_std**
  - Pakiet **numeric_bit**
  - Pakiet **std_logic_arith**
  - Pakiet **std_logic_unsigned**
  - Pakiet **std_logic_signed**
  - Pakiet **std_logic_textio**

# Biblioteka **std**

- Pakiet **standard** – typy:
  - **bit**
  - **boolean**
  - **integer**
  - **character**
  - **real**
  - **time**
  - **string**
  - **bit_vector**

- Pakiet **std_logic_1164**:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

- **U** = stan niezainicjowany (*uninitialized*)
- **X** = stan nieznany (*unknown*)
- **0** = stan logiczny '0'
- **1** = stan logiczny '1'
- **Z** = stan wysokiej impedancji dla sygnału trzystanowego (*tri-state*)
- **W** = stan nieznany dla sygnału niskoobciążalnego (*weak unknown*)
- **L** = stan niskiej rezystancji (dla wyjść typu otwarty emiter) (*weak* '0')
- **H** = stan wysokiej rezystancji (dla wyjść typu otwarty kolektor) (*weak* '1')
- **-** = stan nieistotny (*don't care*)

# Pakiet **std_logic_1164** (kod źródłowy)

```
package std_logic_1164 is
-- logic state system (unresolved)
type std_ulogic is ('U', -- Uninitialized
                    'X', -- Forcing Unknown
                    '0', -- Forcing 0
                    '1', -- Forcing 1
                    'Z', -- High Impedance
                    'W', -- Weak Unknown
                    'L', -- Weak 0
                    'H', -- Weak 1
                    '-'  --Don't care
                   );

-- unconstrained array of std_ulogic for use
-- with the resolution function:

type std_ulogic_vector is array (natural range <> ) of std_ulogic;
```

# Biblioteka IEEE

- Pakiet **std_logic_1164** – funkcja arbitrażowa:

| | 'U' | 'X' | '0' | '1' | 'Z' | 'W' | 'L' | 'H' | '-' |
|---|---|---|---|---|---|---|---|---|---|
| 'U' | 'U' | 'U' | 'U' | 'U' | 'U' | 'U' | 'U' | 'U' | 'U' |
| 'X' | 'U' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' |
| '0' | 'U' | 'X' | '0' | 'X' | '0' | '0' | '0' | '0' | 'X' |
| '1' | 'U' | 'X' | 'X' | '1' | '1' | '1' | '1' | '1' | 'X' |
| 'Z' | 'U' | 'X' | '0' | '1' | 'Z' | 'W' | 'L' | 'H' | 'X' |
| 'W' | 'U' | 'X' | '0' | '1' | 'W' | 'W' | 'W' | 'W' | 'X' |
| 'L' | 'U' | 'X' | '0' | '1' | 'L' | 'W' | 'L' | 'W' | 'X' |
| 'H' | 'U' | 'X' | '0' | '1' | 'H' | 'W' | 'W' | 'H' | 'X' |
| '-' | 'U' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X |

# Biblioteka IEEE (cd.)

- Pakiet **std_logic_arith** (cd.)

```
library IEEE;
use IEEE.std_logic_arith.all;


type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range <>) of std_logic;
```

| ARG1 | op | ARG2 | UNSIGNED | SIGNED | bit_vector |
|---|---|---|---|---|---|
| "000" | = | "000" | true | true | true |
| "00" | = | "000" | true | true | false |
| "100" | = | "0100" | true | false | false |
| "000" | < | "000" | false | false | false |
| "00" | < | "000" | false | false | true |
| "100" | < | "0100" | false | true | false |

# Biblioteka IEEE (cd.)

- Pakiet **std_logic_arith** (cd.)

```
variable X: unsigned(1 to 8);
-- 8-bitowa liczba,
-- X(X'left) = X(1) - najbardziej znaczący bit
```

```
signal Y: unsigned(3 downto 0)
-- 4-bitowa liczba
-- Y(Y'left) = Y(3) - najbardziej znaczący bit
```

# Biblioteka IEEE (cd.)

- Pakiet **std_logic_arith** (cd.)

```
signed'("0101") -- +5
signed'("1011") -- -5
```

  - Funkcje konwersji:

```
integer
unsigned
signed
std_ulogic
```

```
conv_integer(◊) → integer;
conv_unsigned(◊; SIZE) → unsigned;
conv_signed(◊; SIZE) → signed;
conv_std_logic_vector(◊; SIZE) → std_logic_vector
```

*SIZE* = integer

# Biblioteka IEEE (cd.)

- Pakiet **std_logic_arith** (cd.)

```
conv_signed(signed'("110"),8) → "11111110"
conv_unsigned(unsigned'("1101010",3) → "010"
```

| **A** | | **B** | | **C** |
|---|---|---|---|---|
| *n bitów* | + | *m bitów* | = | *max(n,m) bitów* |

Wyjątek:

| **A** | | **B** | | **C** |
|---|---|---|---|---|
| *n bitów* | +/- | *m bitów* | = | *m+1 bitów* |
| **signed** | | **unsigned** | | **signed** |

$n \leq m$

# Biblioteka IEEE (cd.)

- Pakiet **std_logic_arith** (cd.)

```
signal U4: unsigned (3 downto 0);
signal U8: unsigned (7 downto 0);
signal S4: signed (3 downto 0);
signal S8: signed (7 downto 0);
```

Liczba bitów zwracana przez operację + lub -:

| + lub - | U4 | U8 | S4 | S8 |
|---|---|---|---|---|
| U4 | 4 | 8 | 5 | 8 |
| U8 | 8 | 8 | 9 | 9 |
| S4 | 5 | 9 | 4 | 8 |
| S8 | 8 | 9 | 8 | 8 |

# Biblioteka IEEE (cd.)

- Pakiet **std_logic_arith** (cd.)

```
process
  variable a_v, b_v, res_v: unsigned (7 downto
  0);
  variable sum_v: unsigned (8 downto 0);
  variable carry_v: std_logic;
begin
  sum_v := conv_unsigned(a_v, 9) + b_v;
  res_v := sum_v(7 downto 0);
  carry_v := sum_v(8);
end process;
```

# Biblioteka IEEE (cd.)

- Pakiet **std_logic_arith** (cd.)

```
function shl(ARG: unsigned;
             COUNT: unsigned) return unsigned;
function shl(ARG: signed;
             COUNT: unsigned) return signed;
function shr(ARG: unsigned;
             COUNT: unsigned) return unsigned;
function shr(ARG: signed;
             COUNT: unsigned) return signed;
```

# Biblioteka IEEE (cd.)

- Pakiet **std_logic_unsigned**
  - Umożliwia wykonywanie operacji
    - arytmetycznych,
    - konwersji
    - porównywania

    dla wartości typu **std_logic** traktowanych jako liczby całkowite bez znaku (**unsigned**).

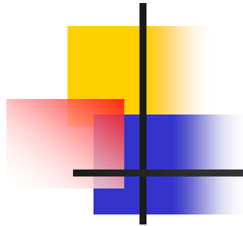
# Biblioteka IEEE (cd.)

- Pakiet **std_logic_signed**
  - Umożliwia wykonywanie operacji:
    - arytmetycznych,
    - konwersji
    - porównywania

    dla wartości typu **std_logic** traktowanych jako liczby całkowite ze znakiem (**signed**) zakodowanych jako uzupełnienie do 2.

# Biblioteka IEEE (cd.)

- Pakiet **std_logic_textio**
  - Rozszerzenie pakietu **std.textio**
  - Pakiet ten zawiera m.in. funkcje **hread** i **hwrite**

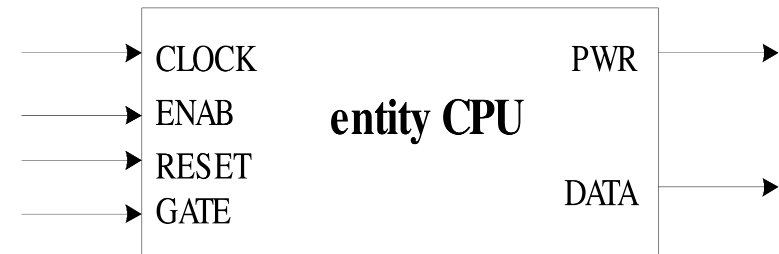


# Część 13 (VHDL)

## Poziom strukturalny

# Jednostki projektowe (**entity**)

```
entity CPU is
   port(CLOCK : in bit;
         ENAB : in bit;
        RESET : in bit;
         GATE : in bit;
          PWR : out bit;
         DATA : out bit_vector(7 downto 0)
       );
end entity;
-- lub: end CPU;
-- lub: end entity CPU;
```



# Architektury jednostek (**architecture**)

```
architecture <Nazwa_architektury> of
   <Nazwa_entity> is
     <Deklaracje>
         ......
begin
     <Polecenia_współbieżne>
               ......
end architecture;
-- lub end <Nazwa_architektury>
-- lub end architecture <Nazwa_architektury>
```

# Osadzanie komponentów

- Deklaracja komponentu:

```
component <Nazwa_komponentu>
  port (<Deklaracje_portów>);
end component;
```

- Przykład deklaracji komponentu – bramki **nand2**:

```
component nand2 is
  port (a : in bit;
        b : in bit;
        y : out bit);
end component;
```

```
entity NAND2
  port (A, B : in bit,
        Y    : out bit);
end entity;

architecture beh of NAND2 is
...
begin
...
end architecture;
```

# Osadzanie komponentów (cd.)

- Mapowanie portów
  - Sposób uproszczony (kolejności końcówek jak w deklaracji):

```
B1 : nand2 port map (in1, in2, out1);
```

  - Sposób II (kolejność jest nieistotna):

```
B1 : nand2
  port map (y => out1, a => in1, b => in2);
```

port komponentu

sygnał

# Osadzanie komponentów (cd.)

```
entity main is port (
      a   : in bit;
      b   : in bit;
      c   : in bit;
      d   : out bit
      );
end entity;

architecture beh of main is
   signal internal : bit;
   component nand2 is
     port (x, y : in bit;
           z    : out bit);
   end component;
begin
   BRAMKA1: nand2 port map (x => a, y => b, z => internal);

   BRAMKA2: nand2 port map (x => internal, y => c, z => d);
end architecture;
```

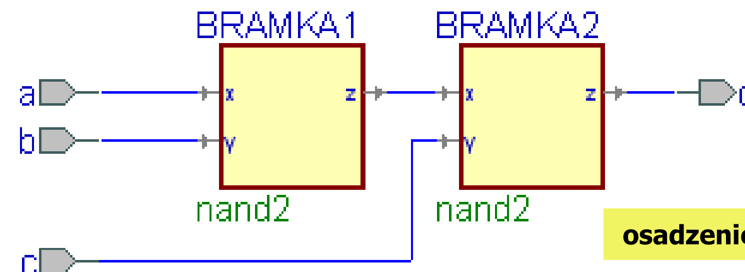
deklaracja komponentu

część deklaracyjna

część współbieżna

```
entity nand2 is
   port (x, y : in bit;
         z    : out bit);
end entity;

architecture beh of nand2 is
begin
   z <= x and y;
end architecture;
```




osadzenie komponentu

# Polecenie **generate**

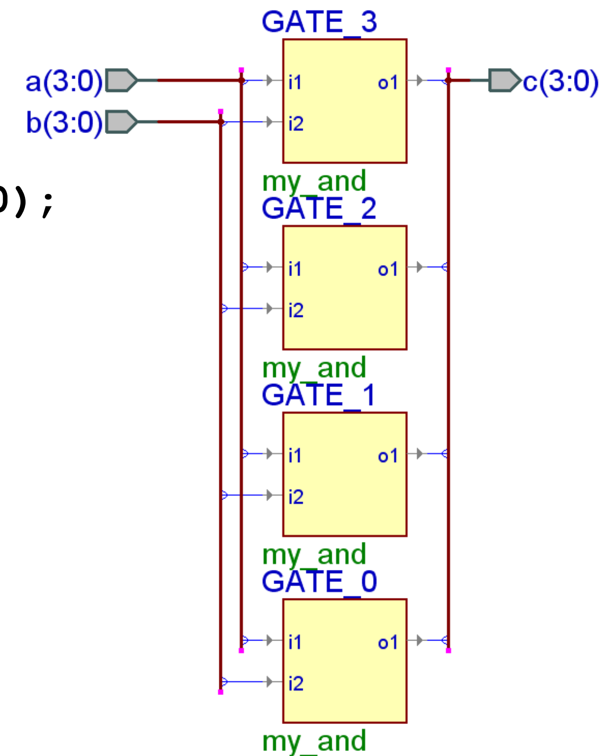
- Polecenie **for-generate**
- Polecenie **if-generate**

# Polecenie **generate** (cd.)

- Polecenie **for-generate**

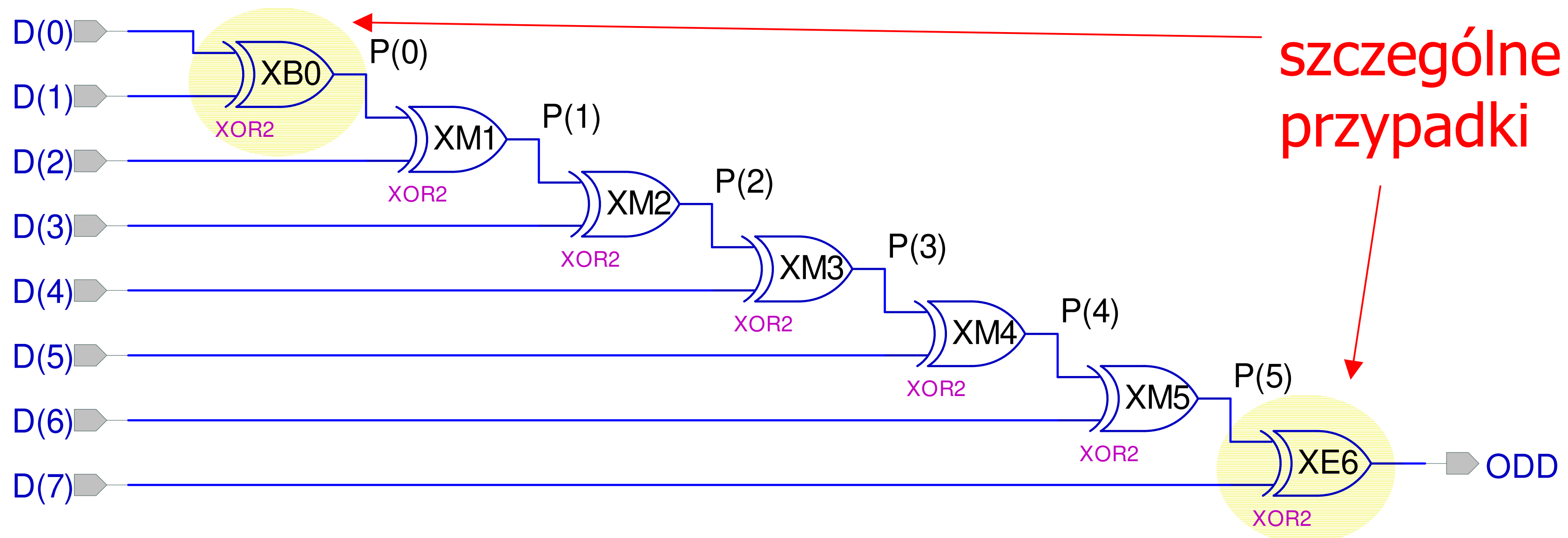
```
entity test is
  port (a, b : in  bit_vector(3 downto 0);
    c : out bit_vector(3 downto 0) );
end entity;

architecture rtl of test is
  component my_and
    port (
      i1 : in  bit;
      i2 : in  bit;
      o1 : out bit);
  end component;
begin
  AND_BLOCK: for i in 0 to 3 generate
    GATE : my_and port map (a(i), b(i), c(i) );
  end generate;
end architecture;
```



# Polecenie **generate** (cd.)

- Polecenie **if-generate**

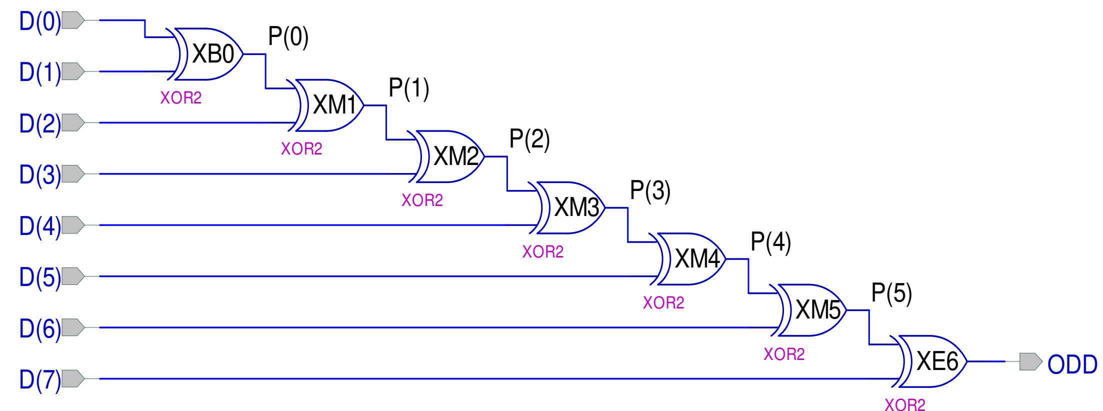


# Polecenie **generate** (cd.)

- Polecenie **if-generate**

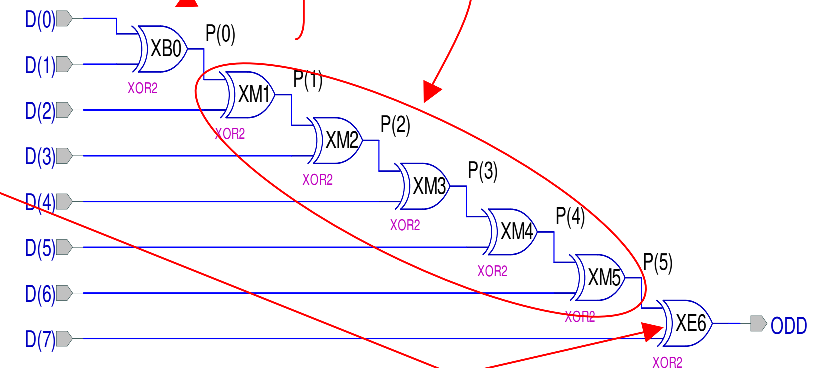
```
entity parity8 is
  port (d : in  bit_vector(0 to 7);
    odd : out bit );
end entity;

architecture rtl of parity8 is
  component xor2
    port (a, b : in  bit;
      y : out bit);
  end component;
  signal p: bit_vector(d'low to d'high - 2);
...
```



# Polecenie **generate** (cd.)

```
begin
  G: for i in d'low to d'high-1  generate
    G_BEG: if i=d'low generate
      XB: xor2 port map (a => d(d'low),
        b => d(d'low+1),
        y => p(d'low) );
    end generate;
    G_MIG: if i > d'low and i < d'high-1 generate
      XM: xor2 port map (a => p(i-1),
        b => d(i+1),
        y => p(i) );
    end generate;
    G_END: if i = d'high-1 generate
      XE: xor2 port map (a=>p(i-1),
        b=>d(i+1),
        y => odd );
    end generate;
  end generate;
end architecture;
```



# Parametry bloku **entity** (**generic**)

- Przykład:

```
entity ANDX is
  generic (W : positive);
  port( A : in  bit_vector(W-1 downto 0);
        B : in  bit_vector(W-1 downto 0);
        Y : out bit_vector(W-1 downto 0) );
end entity;

architecture RTL of ANDX is

begin
  G: for I in W-1 downto 0 generate
    Y(I) <= A(I) and B(I);
  end generate;
end architecture;
```

# Parametry bloku **entity** (**generic**) (cd.)

```
entity AND8 is
  port (A8 : in  bit_vector(7 downto 0);
        B8 : in  bit_vector(7 downto 0);
        Y8 : out bit_vector(7 downto 0) );
end entity;

architecture RTL of AND8 is
  component ANDX
    generic (W : positive);
    port( A : in  bit_vector(W-1 downto 0);
          B : in  bit_vector(W-1 downto 0);
          Y : out bit_vector(W-1 downto 0) );
  end component;
begin
  A1: ANDX
        generic map (W => 8)
        port map (A => A8, B => B8, Y => Y8);
end architecture;
```

deklaracja komponentu

osadzenie modułu **ANDX** z parametrem W=8

# Parametry bloku **entity** (**generic**) (cd.)

```
-- wersja 1-bitowa:
entity dff1 is
   port (
        clk, rst : in std_logic;
        d : in std_logic;
        q: out std_logic);
end entity;
...
-- wersja wielobitowa:
entity dff is
   generic (W : integer := 2);
   port (
        clk, rst : in std_logic;
        d : in std_logic_vector (W-1 downto 0);
        q: out std_logic_vector (W-1 downto 0));
end entity;
```

VHDL

# Konfiguracja osadzanych elementów

Realizacja **L1** = blok **licznik** z domyślną architekturą.

```
architecture beh of tb is
   component licznik
     port (...); -- pominięto szczegóły
   end component licznik;

   for L1: licznik use entity work.licznik; --
   domyślna
                                                   --
   architektura
   for L2: licznik use entity work.licznik(active_1);
begin
   ...
   L1: licznik port map(Q1,Reset,Clk,En);
   L2: licznik port map(Q2,Reset,Clk,En);
   ...
end TB;
```

konfigu-racja

osadzenie

Realizacja **L2** = blok **licznik** z architekturą **active_1**.

# Konfiguracja osadzanych elementów

VHDL

```
architecture beh of tb is
component licznik
  port (...); -- pominięto szczegÓły
end component licznik;
  for L1: licznik use entity work.counter; -- domyślna
                                           -- archit.
  for L2: licznik use entity work.counter(active_1);
begin
  ...
  L1: licznik port map(Q1,Reset,Clk,En);
  L2: licznik port map(Q2,Reset,Clk,En);
  ...
end TB;
```

VHDL

# Konfiguracja osadzanych elementów

```
for all: licznik use entity work.licznik(active_1);
```

```
for all: licznik use entity work.licznik(active_1);
for L1:  licznik use entity work.licznik;
for L2:  licznik use entity work.licznik(active_1);
```

# Blok konfiguracji (**configuration**)

```
configuration <nazwa_konfiguracji> of <nazwa_entity>
   is
   for <nazwa_architektury>
     for <etykieta1> :<nazwa_komponentu1>
       use entity <lib>.<nazwa_kompon>(<nazwa_arch>);
     end for;
     for <etykieta2> :<nazwa_komponentu2>
       use configuration <lib>.<nazwa_konfiguracji>;
     end for;
     ...
   end for;
end configuration;
```

# Blok konfiguracji (**configuration**)

VHDL

- Przykład deklaracji bloku:

```
configuration config_1 of tb is
  for beh -- beh = architektura
    for L1, L3: licznik use entity work.counter
          generic map( Trise =>  10 ns);
    end for;
    for L2: licznik use entity
          work.counter(active_1);
    end for;
  end for;
end configuration config_1;
```

Ta linia identyfikuje konkretną architekturę, której dotyczy konfiguracja.

Tu wstawiane są bloki **for...end for** deklarujące powiązania pomiędzy wstawianymi komponentami a konkretnymi ich architekturami.

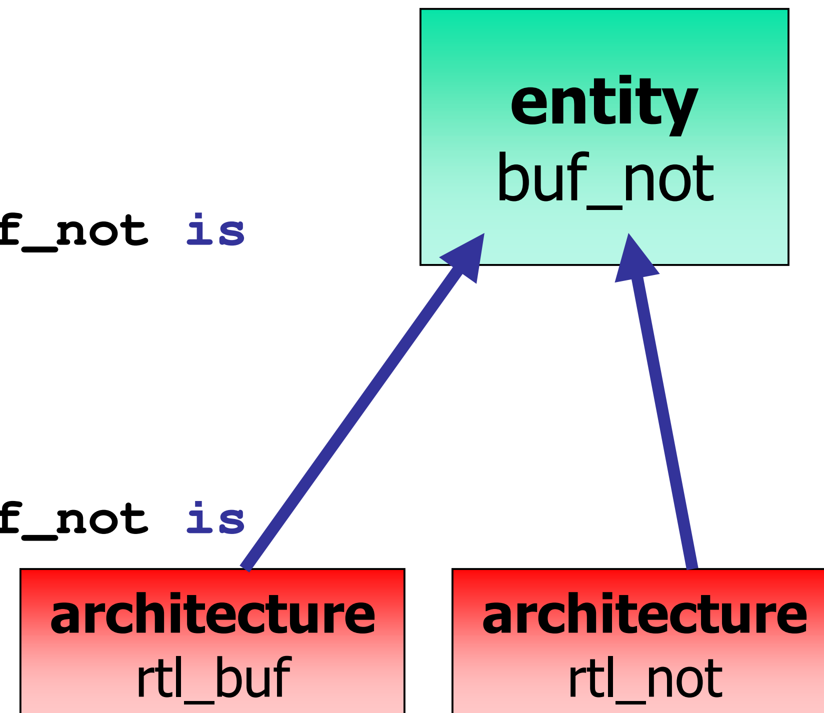
~~for all: licznik use entity counter ...,~~

# Blok konfiguracji – zaawansowany przykład

```
entity buf_not is
  port( A : in  bit;
        Y : out bit);
end entity;

architecture rtl_buf of buf_not is
begin
  Y <= A;
end architecture;

architecture rtl_not of buf_not is
begin
  Y <= not A;
end architecture;
```



# Blok konfiguracji – zaawansowany przykład (cd.)

```
entity buf_not2 is
  port (A2 : in  bit_vector(1 downto 0);
        Y2 : out bit_vector(1 downto 0) );
end entity;

architecture rtl of buf_not2 is
  component buf_not
    port( A : in bit;
          Y : out bit);
  end component;
begin
  C1 : buf_not port map (A => A2(1), Y => Y2(1));
  C2 : buf_not port map (A => A2(0), Y => Y2(0));
end architecture;
```

buf_not2(rtl)

C1:buf_not

C2:buf_not

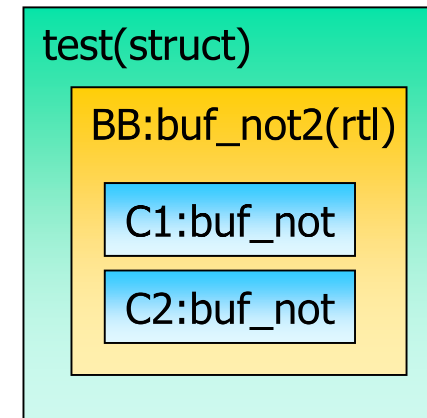
VHDL

# Blok konfiguracji – zaawansowany przykład (cd.)

```
entity test is
  port (Ain   : in  bit_vector(1 downto 0);
        Yout  : out bit_vector(1 downto 0) );
end entity;

architecture struct of test is
  component buf_not2 is
    port (A2 : in  bit_vector(1 downto 0);
    Y2  : out bit_vector(1 downto 0) );
  end component;

begin
  BB: buf_not2 port map (A2 => Ain, Y2 => Yout);
end architecture;
```



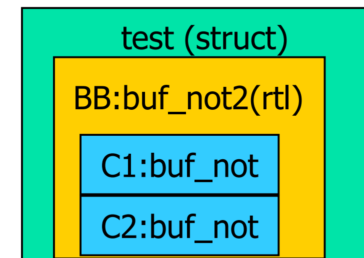
# Blok konfiguracji – zaawansowany przykład (cd.)

```
configuration bufor of test is
  for struct
    for BB:buf_not2 use entity work.buf_not2(rtl);
      for rtl
        for C1  : buf_not
                  use entity buf_not(rtl_buf);
        end for;
        for C2  : buf_not
                  use entity buf_not(rtl_buf);
        end for;
      end for;
    end for;
  end for;
end configuration;
```

**Nazwa własna** bloku konfiguracji

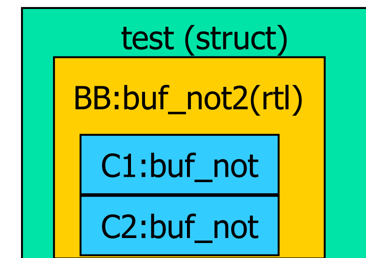
**Entity**, którego dotyczy konfiguracja

Architektura



# Blok konfiguracji – zaawansowany przykład (cd.)

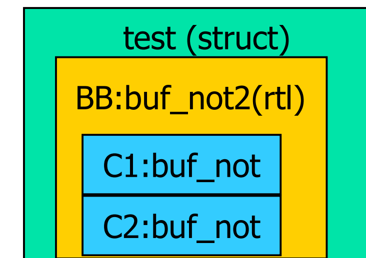
```
configuration config_buf of buf_not2 is
   for rtl
        for C1: buf_not
          use entity work.buf_not(rtl_buf) ;
        end for;
        for C2: buf_not
          use entity work.buf_not(rtl_buf) ;
        end for;
     end for;
end configuration;
```

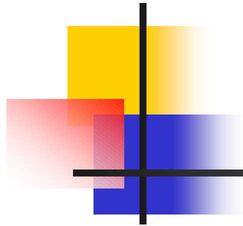


# Blok konfiguracji – zaawansowany przykład (cd.)

```
configuration config_not of buf_not2 is
   for rtl
      for C1: buf_not
       use entity work.buf_not(rtl_not) ;
      end for;
      for C2: buf_not
       use entity work.buf_not(rtl_not) ;
      end for;
    end for;
end configuration;

configuration bufor of test is
   for struct
     for BB:buf_not2
       use configuration work.config_buf;
     end for;
   end for;
end configuration;
```





# Część 14 (VHDL)

## Poziom przesłań międzyrejestrowych RTL

# Współbieżne przypisanie proste

- Składnia:

```
<Odbiorca> <= <Wyrażenie>;
```

- Przykład:

```
Y <= A and B;
```

# Współbieżne przypisanie warunkowe **when ... else**

- Składnia przypisania:

```
<Odbiorca> <= <Wyrażenie1> when <Warunek1> else
              <Wyrażenie2> when <Warunek2> else
              ...
              <WyrażenieN>;
```

- Przykład:

```
Y <= A when SELECT_A = '1' else
     B when SELECT_B = '1' else
     C;

Y <= A when ENABLE = '1' else 'Z';
```

# Przypisanie współbieżne **select ... when**

- Składnia przypisania **select**:

```
with <Wyrażenie_wyboru> select
     <Odbiorca> <= <Wyrażenie1> when <Wybór1>,
                   <Wyrażenie2> when <Wybór2>,
                   ...
                   <WyrażenieN> when <WybórN>;
```

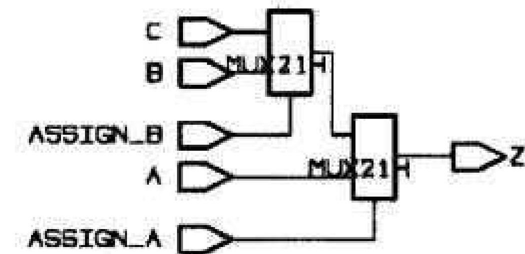
- Przykład:

```
signal A, B, C, D, Y : bit;
signal SEL: bit_vector(1 downto 0);
...
with SEL select
  Y <= A when "00",
       B when "01",
       C when "10",
       D when others;
```

# Różnice pomiędzy **when ...else** i **select...when** (cd.)

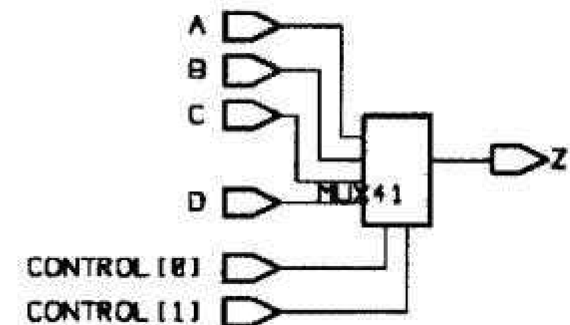
- Przypisanie warunkowe:

```
Z <= A when ASSIGN_A = '1' else
     B when ASSIGN_B = '1' else
     C;
```




- Przypisanie **select**:

```
with CONTROL select
  Z <= A when "00",
       B when "01",
       C when "10",
       D when "11";
```



# Operatory porównania

```
"101011" < "1011" -- warunek spełniony

"101" < "101000" -- warunek spełniony
```

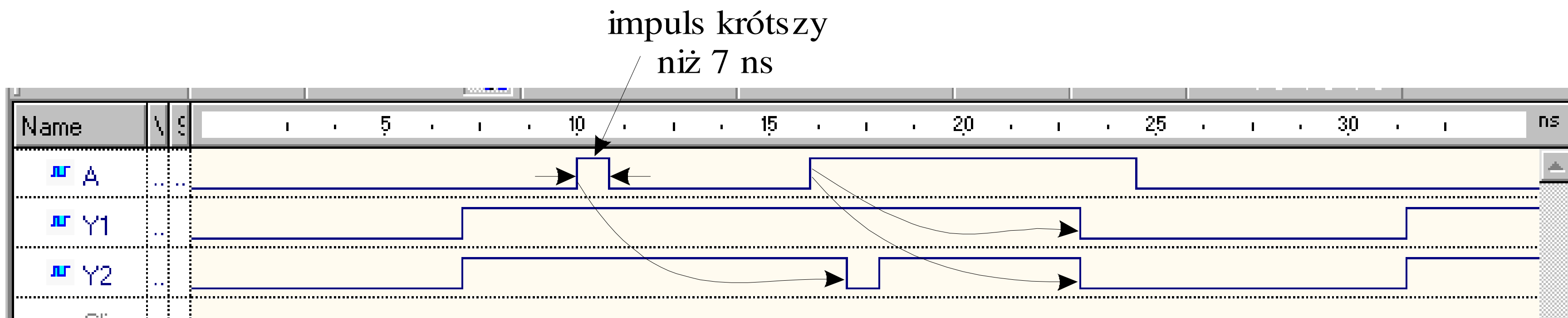
# Opóźnienia

- Opóźnienie inercyjne
- Opóźnienia typu **transport**

```
Y1 <= inertial  not A after 7 ns;
Y2 <= transport not A after 7 ns;
```

# Opóźnienia (cd.)

```
Y1 <= inertial  not A after 7 ns;
Y2 <= transport not A after 7 ns;
```



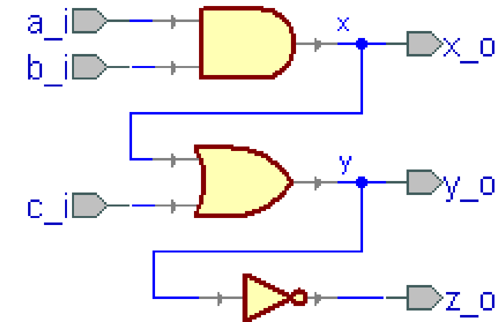

```
Y1<= reject 3 ns inertial not A after 7 ns;
```

# Opóźnienia (cd.)

```
Y <= inertial A after 7 ns when SEL="00" else
              B after 11 ns when SEL="01" else
              C when SEL="10" else
              D ;

with SEL select
  Y <= transport A after 4 ns when "00",
                 B after 3 ns when "01",
                 C after 6 ns when "10",
                 D when others;
```

# Operacje współbieżne oraz czasowe (cd.)




- Przykład działania symulatora

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

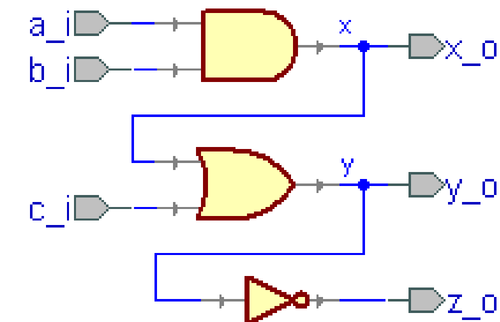
entity tst is
  port(
    a_i : in std_logic;
    b_i : in std_logic;
    c_i : in std_logic;
    x_o : out std_logic;
    y_o : out std_logic;
    z_o : out std_logic
    );
end entity;
```

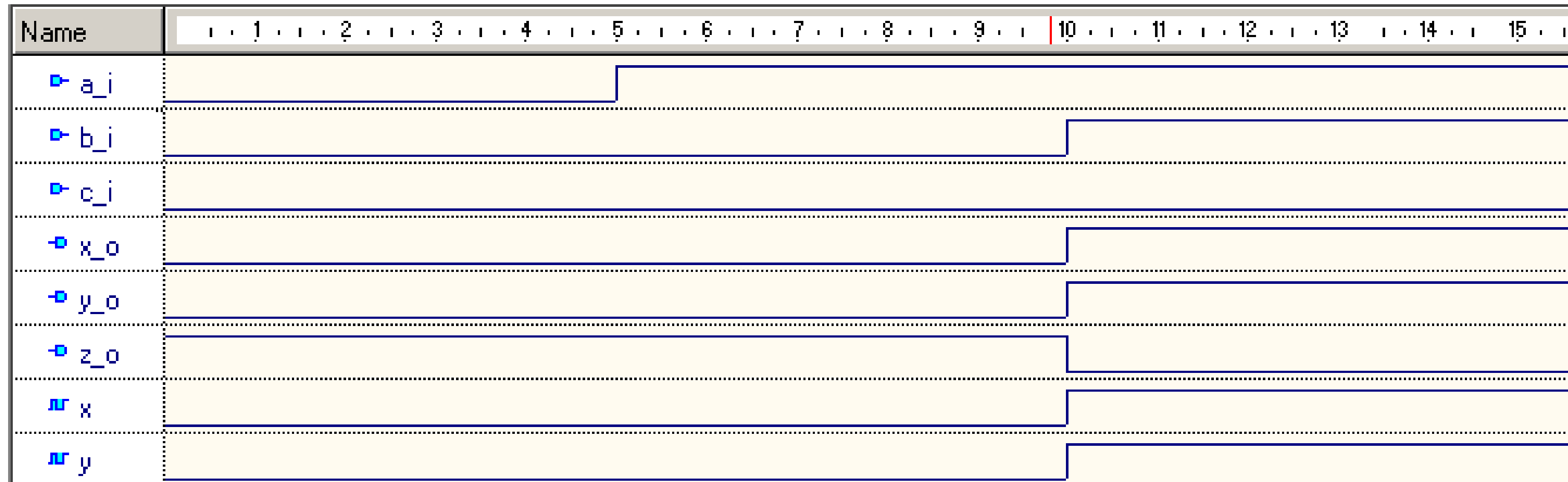
```
architecture tst of tst is
  signal x: std_logic;
  signal y: std_logic;
begin
  x <= a_i and b_i;
  y <= c_i or x;
  z_o <= not y;

  x_o <= x;
  y_o <= y;
end architecture;
```

# Operacje współbieżne oraz czasowe (cd.)

- Przykład działania symulatora (cd.)

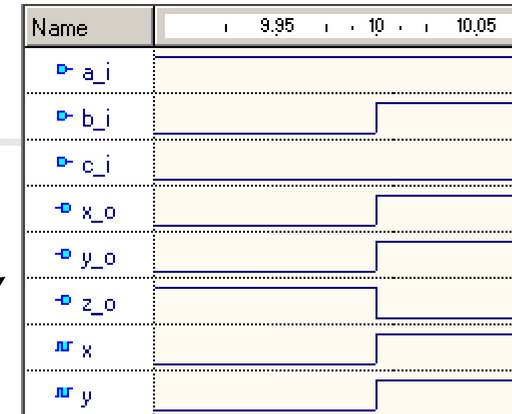


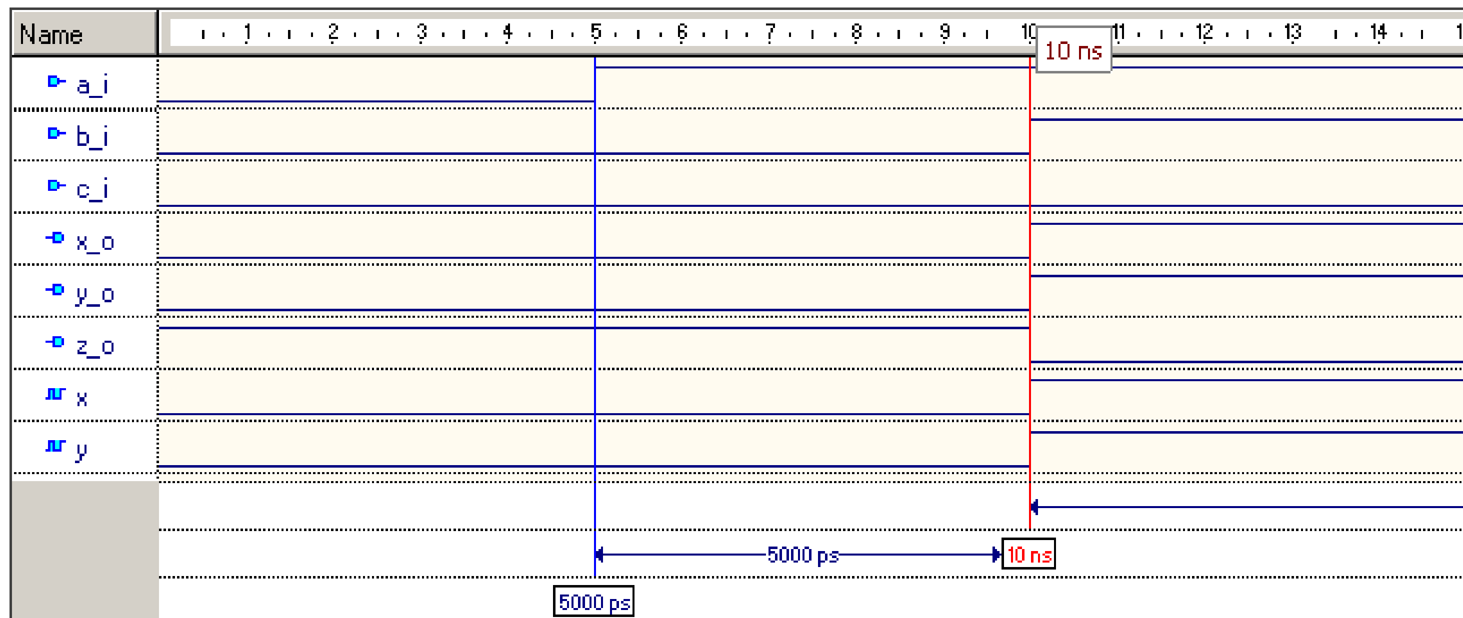



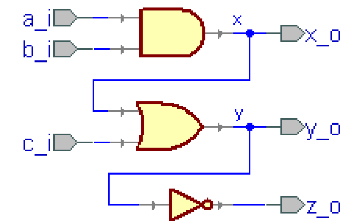
# Operacje współbieżne oraz czasowe (cd.)

- Przykład działania symulatora (cd.)

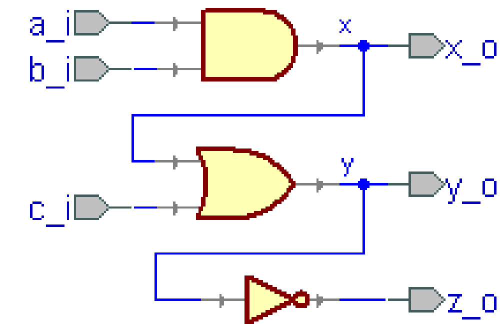
powiększenie



# Operacje współbieżne oraz czasowe (cd.)




- Przykład działania symulatora (cd.)

| Time | Delta | a_i | b_i | c_i | x | y | x_o | y_o | z_o |
|---|---|---|---|---|---|---|---|---|---|
| 0.000 | 0 | 0 | 0 | 0 | U | U | U | U | U |
| 0.000 | 1 | 0 | 0 | 0 | 0 | U | U | U | U |
| 0.000 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | U | U |
| 0.000 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5.000 ns | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 10.000 ns | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 10.000 ns | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 10.000 ns | 2 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 10.000 ns | 3 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |

# Instrukcje współbieżne i sekwencyjne

- Instrukcje współbieżne:
  - przypisania wartości sygnału (**<=**),
  - kontroli (**assert**),
  - procesu (**process**),
  - procedury,
  - funkcji,
  - blokowa (**block**),
  - wstawiania składnika (**component**),
  - powielania (**generate**).
- Instrukcje sekwencyjne.
  - oczekiwania,
  - przypisania wartości sygnału,
  - przypisania wartości zmiennej,
  - kontroli (**assert**),
  - raportu,
  - wywołania procedury,
  - instrukcje warunkowe (**if**, **case**),
  - instrukcje pętli,
  - instrukcja wyjścia,
  - instrukcja powrotu,
  - instrukcja pusta.

# Procesy

- Proces - wykonywanie instrukcji jest sekwencyjne.

```
<nazwa_procesu> : process (<lista_czułości>)
  <część_deklaracyjna>
begin
  <część_sekwencyjna>
end process;
```

- Proces jest traktowany jako jedno polecenie współbieżne.
- Jeśli w architekturze zdefiniowano kilka procesów, wszystkie one wykonują się współbieżnie względem siebie, a także niezależnie od siebie.

# Przypisanie sekwencyjne

- Składnia przypisania dla sygnału:

```
<sygnał> <= <wyrażenie>;
```

- Składnia przypisania dla zmiennej (zadeklarowanej jako **variable**) :

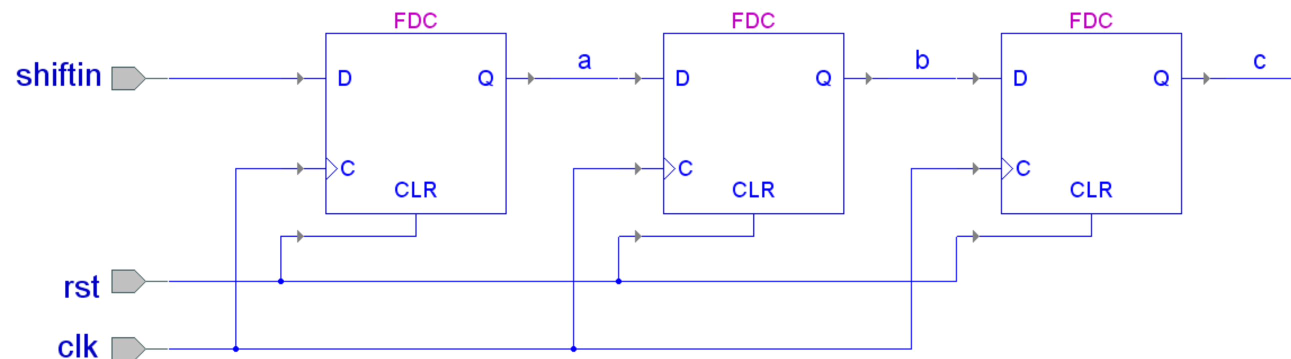
```
<zmienna> := <wyrażenie>;
```

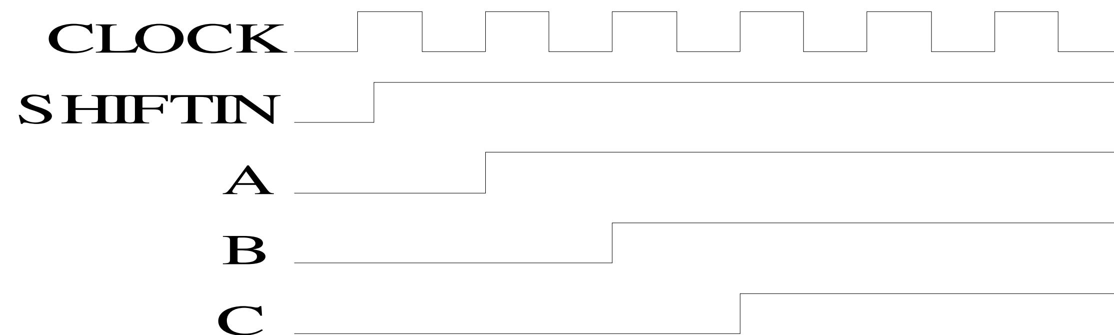
# Różnice pomiędzy sygnałem i zmienną

```
process ...
   ...
   data <= (others => '0');
   ...
   if ... then
      data(6) <= '1';
   end if;
   ...
end process;
```

# Różnice pomiędzy sygnałem i zmienną (cd.)

```
SHIFTREG :
process(clk,rst)
begin
   if rst='1' then
      a <= '0';
      b <= '0';
      c <= '0';
   elsif rising_edge(clk)
then
      a <= shiftin;
      b <= a;
      c <= b;
   end if;
end process;
```

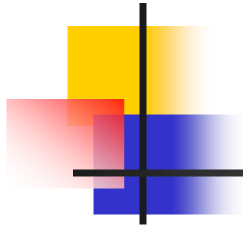


# Różnice pomiędzy sygnałem i zmienną (cd.)

- Następujący kod dałby identyczne rezultaty:

```
...
elsif rising_edge(clk)
then
    c <= b;
    b <= a;
    a <= shiftin;
end if;
end process;
```



# Część 15 (VHDL)

## Poziom behawioralny

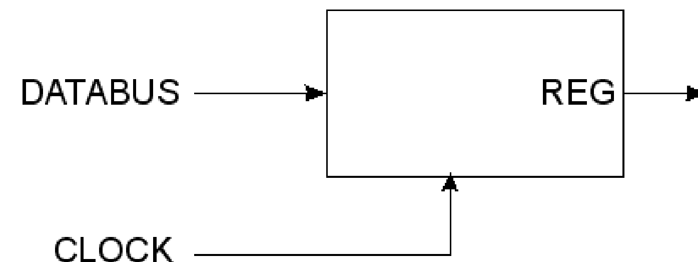
# Podstawowe rodzaje procesów

- Procesy zegarowane
- Procesy kombinacyjne
- Procesy zatrzaskiwane

# Podstawowe rodzaje procesów (cd.)

- Procesy zegarowane (cd.)

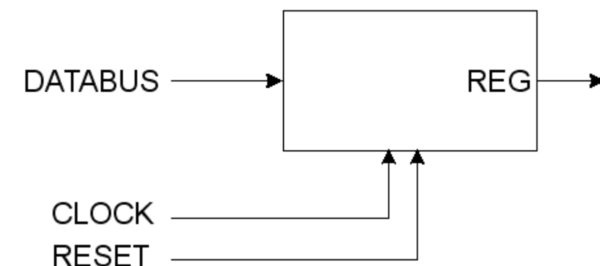
```
LOADREG : process (CLOCK)
begin
  if rising_edge(CLOCK) then
    REG<=DATABUS;
  end if;
end process;
```



# Podstawowe rodzaje procesów (cd.)

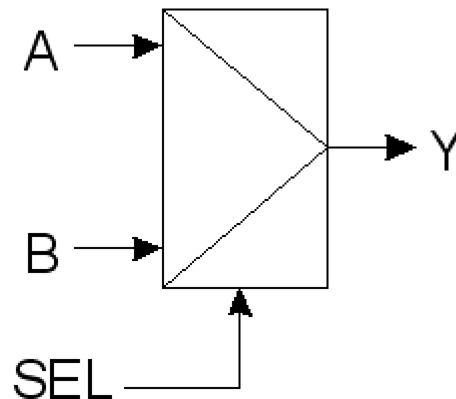
- Procesy zegarowane (cd.)

```
LOADREG : process (CLOCK, RESET)
begin
  if RESET='1' then
    REG <= "00000000"; --wyczyść rejestr REG
  elsif rising_edge(CLOCK) then
    REG <= DATABUS;
  end if;
end process;
```



# Podstawowe rodzaje procesów (cd.)

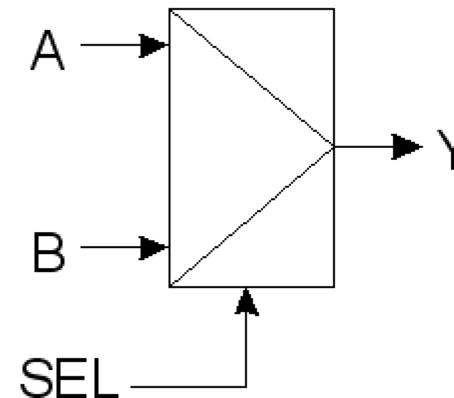
- Procesy kombinacyjne
  - Logika kombinacyjna może zostać opisana:
    - poza procesem,
    - poprzez proces kombinacyjny.



# Podstawowe rodzaje procesów (cd.)

- Procesy kombinacyjne (cd.)

```
MUX2_2 : process(A, B, SEL) -- multiplekser 2 na 1
begin
  if SEL='0' then
    Y<=A;
  else
    Y<=B;
  end if;
end process;
```



# Podstawowe rodzaje procesów (cd.)

- Procesy kombinacyjne (cd.)

  - Należy zauważyć, że współbieżne przypisanie ciągłe, np.:

```
architecture beh of test
   is
begin

  y <= a and b;

end architecture;
```

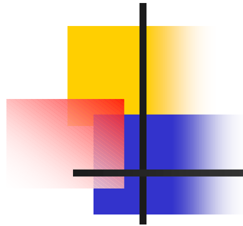
  - jest równoważne następującemu procesowi kombinacyjnemu:

```
architecture beh of test
   is
begin
  process (a, b)
  begin
    y <= a and b;
  end process;
end architecture;
```

# Podstawowe rodzaje procesów (cd.)

- Procesy zatrzaskiwane

```
load_latch : process(ENABLE, RESET, DATABUS)
begin
   if RESET='1' then --zerowanie zatrzasku
       LATCH<="00000000";
   elsif ENABLE='1' then --zatrzask aktywny
       LATCH<=Databus;
   end if;
end process;
```



# Część 16 (VHDL)

## Funkcje i procedury

# Funkcje (cd.)

```
function rising_edge(signal CLK: std_logic)
   return boolean is
--
-- część deklaracyjna
--
begin
   --
   -- ciało funkcji
   --
   return (VALUE);
end function rising_edge;
```

- Przykład wywołania funkcji

```
      rising_edge(ENABLE);
```

# Funkcje (cd.)

- Funkcje i procedury mogą być zagnieżdżone.
- Wyrażenia **wait** nie są dopuszczalne w ciele funkcji (w procedurach są!) i dlatego funkcje są wykonywane w zerowym czasie.
- Z tego powodu wyrażenie **wait** nie może znajdować się również w procedurze wywoływanej przez funkcję.
- Funkcje mogą być wywoływane rekurencyjnie.
- Można definiować funkcje o nazwach zarezerwowanych dla operatorów lub innych funkcji – nazywa się to przeciążaniem operatora lub funkcji.
- VHDL'93 obsługuje 2 typy funkcji: czyste (ang. **pure**) i nieczyste (ang. **impure**).

# Funkcje (cd.)

```
library IEEE:
use IEEE.std_logic_1164.all;

entity DFF is port (D, CLK : in std_logic;
                    Q, QBAR: out std_logic);
end entity DFF:
-- cdn...
```

# Funkcje (cd.)

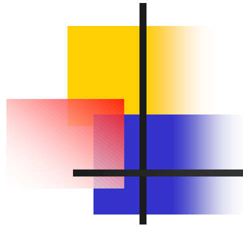
```
-- ... cd.
architecture BEHAVIORAL of DFF is
   function rising_edge (signal CLOCK : std_logic)
            return boolean is
      variable EDGE : boolean:= FALSE;
   begin
      EDGE := (CLOCK = '1' and CLOCK'event);
      return (EDGE);
   end function rising_edge;
begin
   OUTPUT: process
   begin
      wait until (rising_edge(CLK));
      Q <= D after 5 ns;
      QBAR <= not D after 5 ns;
   end process OUTPUT;
end architecture BEHAVIORAL;
```

# Procedury (cd.)

```
procedure QDR_read (
        read_addr: in std_logic_vector(7 downto 0);
        signal read_phase  : in std_logic;
        signal write_phase : in std_logic;
        signal QDR_rps_n_i : out std_logic;
        signal QDR_a_read: out std_logic_vector(23 downto 0)) is
begin
        QDR_rps_n_i<='Z';
        QDR_a_read  <= (others => 'Z');
        wait until read_phase='1';
        QDR_rps_n_i<='0';
        QDR_a_read  <= (others => '0');
        QDR_a_read(7 downto 0) <=  read_addr;
        wait until read_phase='0';
        QDR_rps_n_i <= 'Z';
        QDR_a_read  <= (others => 'Z');
end procedure;
```

# Procedury (cd.)

```
process
...
begin
   ...
   for i in 0 to 3 loop
     QDR_read (address_v, read_phase, write_phase,
               QDR_rps_n_i, QDR_a_read);
     address_v := address_v + 1;
     random_delay(seed_v);
   end loop;
   ...
end process;
```



# Część 17 (VHDL)

## Pakiety

# Tworzenie pakietu

```
package moj_pakiet is
...
-- deklaracje typów, stałych,
-- deklaracje funkcji i procedur
...
end package moj_pakiet;

package body moj_pakiet is
...
-- deklaracje i definicje (implementacje)
-- funkcji i procedur

end package moj_pakiet;
```

# Tworzenie pakietu – dobry zwyczaj

```
library ieee;
use ieee.std_logic_1164.ALL;

entity qdr_main is
  ...
end entity;

architecture beh of qdr_main is
 component fifo
   ... długa lista portów !!!
 end component
 component fifo2
   ... długa lista portów !!!
 end component
...
```

# Tworzenie pakietu – dobry zwyczaj (cd.)

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

entity fifo is
.... – długa lista portów
end entity;
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package fifo_pkg is
   component fifo
.... – długa lista portów
   end component;
end package;
```

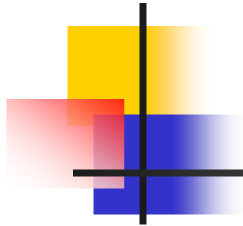
# Tworzenie pakietu – dobry zwyczaj (cd.)

```
library ieee;
use ieee.std_logic_1164.ALL;

library WORK;
use WORK.fifo_pkg.all;

entity qdr_main is
 ...
end entity;

architecture beh of qdr_main is
...
```



# Część 18 (VHDL)

Synteza układów kombinacyjnych
i synchronicznych

# Podstawowy licznik binarny

- Prosty licznik binarny bez dodatkowych wejść funkcyjnych:

# Podstawowy licznik binarny (cd.)

- Prosty licznik binarny bez dodatkowych wejść funkcyjnych (cd.):

```
signal upcount : integer range 0 to 255;

signal upcount: unsigned(7 downto 0);

signal upcount: std_logic_vector(7 downto 0);
```

# Podstawowy licznik binarny (cd.)

- Prosty licznik binarny bez dodatkowych wejść funkcyjnych (cd.):

```
upcount <= upcount + 1;
upcount <= upcount + '1';
upcount <= upcount + "00000001";
```

# Podstawowy licznik binarny (cd.)

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

entity licz is port (
    clk_i      : in std_logic;
    upcount_o  : out std_logic_vector(7 downto 0)
    );
end entity;
-- cd. na nast. planszy
```

# Podstawowy licznik binarny (cd.)

```
architecture beh of licz is
signal upcount : integer range 0 to 255;
begin
  UPCOUNTER: process(clk_i)
  begin
    if rising_edge(clk_i) then
      upcount <= upcount + 1;
    end if;
  end process;
  upcount_o <= upcount;
end architecture;
```

błąd konwersji

# Podstawowy licznik binarny (cd.)

```
library ieee;
use ieee.std_logic_arith.all;
architecture beh of licz is
signal upcount : integer range 0 to 255;
begin
   UPCOUNTER: process (clk_i)
   begin
      if rising_edge(clk_i) then
         upcount <= upcount + 1;
      end if;
   end process;
   upcount_o <= conv_std_logic_vector(upcount, 8);
end architecture;
```

# Podstawowy licznik binarny (cd.)

```
library ieee;
use ieee.std_logic_arith.all;
architecture beh of licz is
signal upcount : unsigned(7 downto 0);
begin
  UPCOUNTER: process(clk_i)
  begin
    if rising_edge(clk_i) then
      upcount <= upcount + 1;
    end if;
  end process;
  upcount_o <= upcount;
end architecture;
```

błąd konwersji

# Podstawowy licznik binarny (cd.)

```
library ieee;
use ieee.std_logic_arith.all;
architecture beh of licz is
signal upcount : unsigned(7 downto 0);
begin
  UPCOUNTER: process(clk_i)
  begin
    if rising_edge(clk_i) then
      upcount <= upcount + 1;
    end if;
  end process;
  upcount_o <= conv_std_logic_vector(upcount, 8);
end architecture;
```

# Podstawowy licznik binarny (cd.)

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

entity licz is
  generic (
    WIDTH : integer := 8
    );
  port (
    clk_i       : in std_logic;
    upcount_o : out std_logic_vector(WIDTH-1 downto 0)
    );
end entity; -- cd na następnej planszy
```

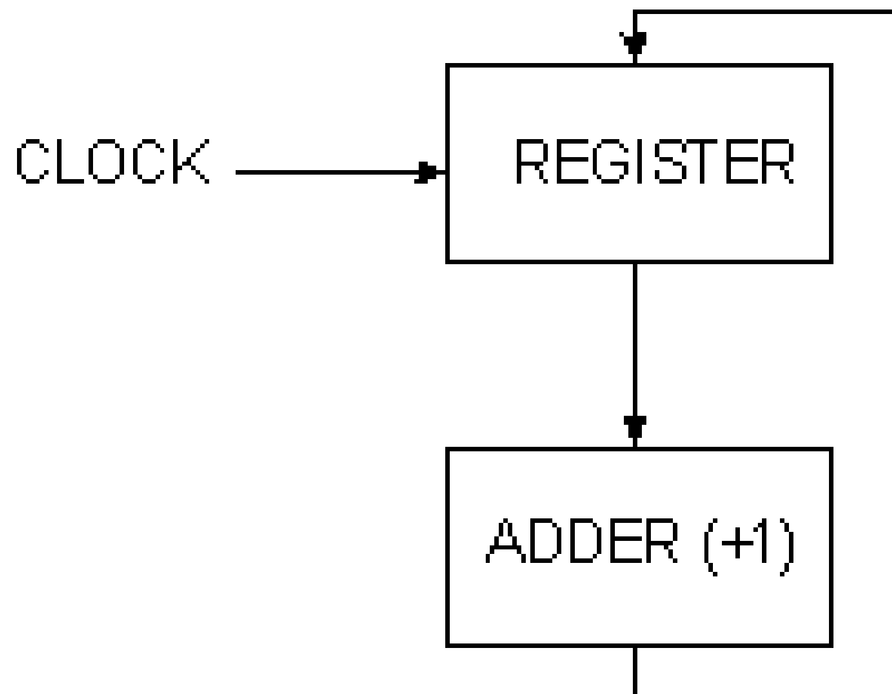
# Podstawowy licznik binarny (cd.)

```
library ieee;
use ieee.std_logic_arith.all;
architecture beh of licz is
  signal upcount : unsigned(WIDTH-1 downto 0);
begin
  UPCOUNTER: process(clk_i)
  begin
    if rising_edge(clk_i) then
      upcount <= upcount + 1;
    end if;
  end process;
  upcount_o <= conv_std_logic_vector(upcount, WIDTH);
end architecture;
```

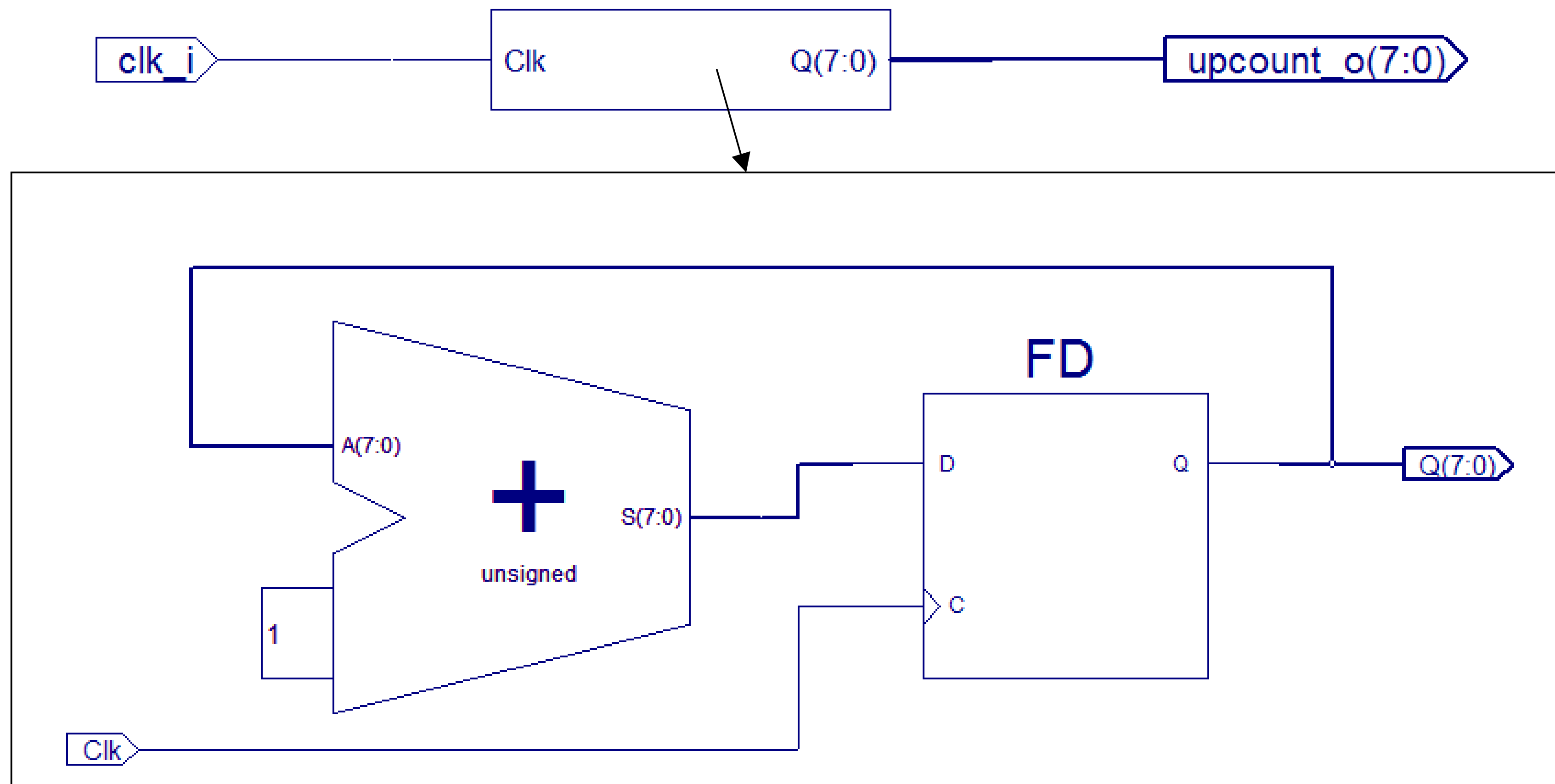
# Podstawowy licznik binarny (cd.)

```
library ieee;
use ieee.std_logic_unsigned.all;
architecture beh of licz is
  signal upcount : std_logic_vector(WIDTH-1 downto 0);
begin
  UPCOUNTER: process(clk_i)
  begin
    if rising_edge(clk_i) then
      upcount <= upcount + 1;
    end if;
  end process;
  upcount_o <= upcount;
end architecture;
```

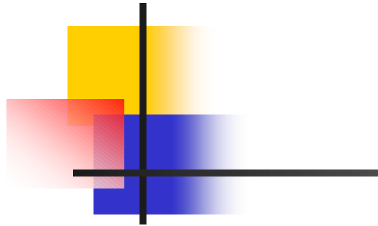
# Jaki jest wynik procesu syntezy logicznej?



# Jaki jest wynik procesu syntezy logicznej ?(cd.)



VHDL



Wynik syntezy logicznej licznika

# Dodanie sygnału *reset*

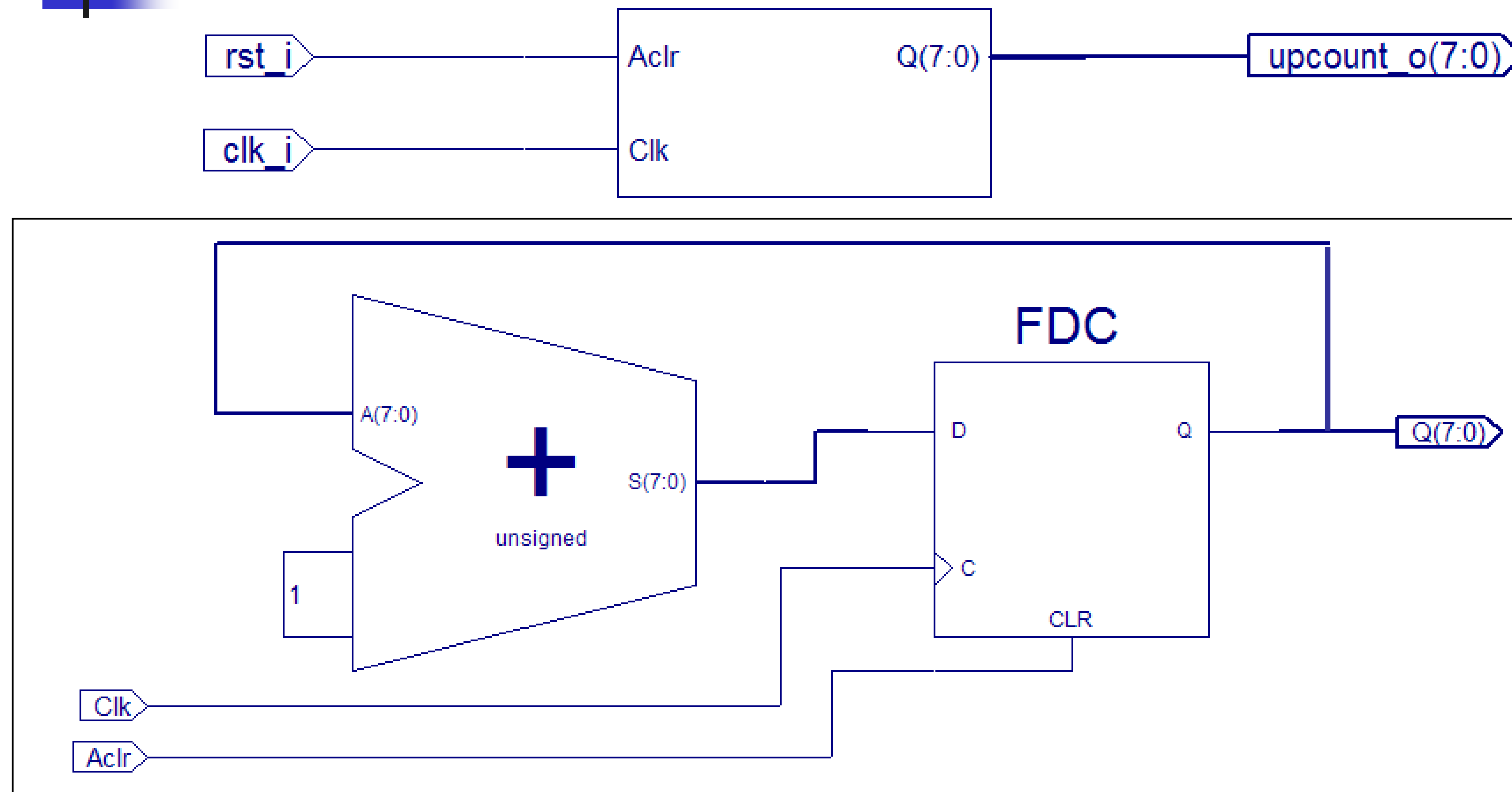
```
library ieee;
use ieee.std_logic_1164.all;

entity licz2 is
  generic (
    WIDTH : integer := 8
    );
  port (
    clk_i       : in  std_logic;
    rst_i       : in  std_logic;
    upcount_o : out std_logic_vector(WIDTH-1 downto 0)
    );
end entity; -- cd. na następnej planszy
```

# Dodanie sygnału *reset* (cd.)

```
library ieee;
use ieee.std_logic_unsigned.all;
architecture beh of licz2 is
  signal upcount : std_logic_vector(WIDTH-1 downto 0);
begin
  UPCOUNTER: process(clk_i, rst_i)
  begin
    if rst_i = '1' then
      upcount <= (others => '0');
    elsif rising_edge(clk_i) then
      upcount <= upcount + 1;
    end if;
  end process;
  upcount_o <= upcount;
end architecture;
```

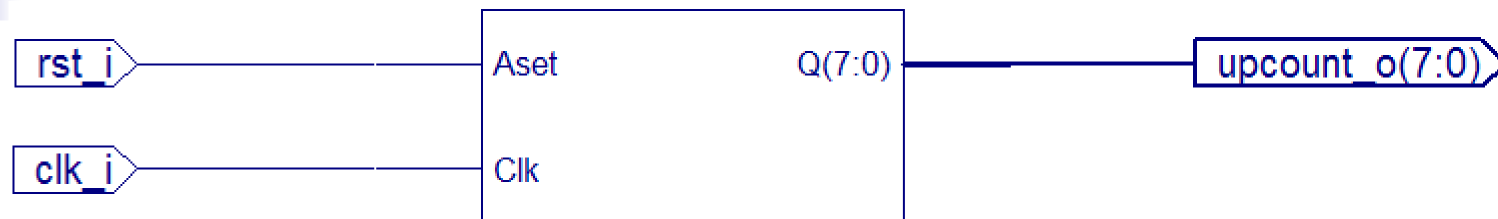
# Dodanie sygnału *reset* (cd.)

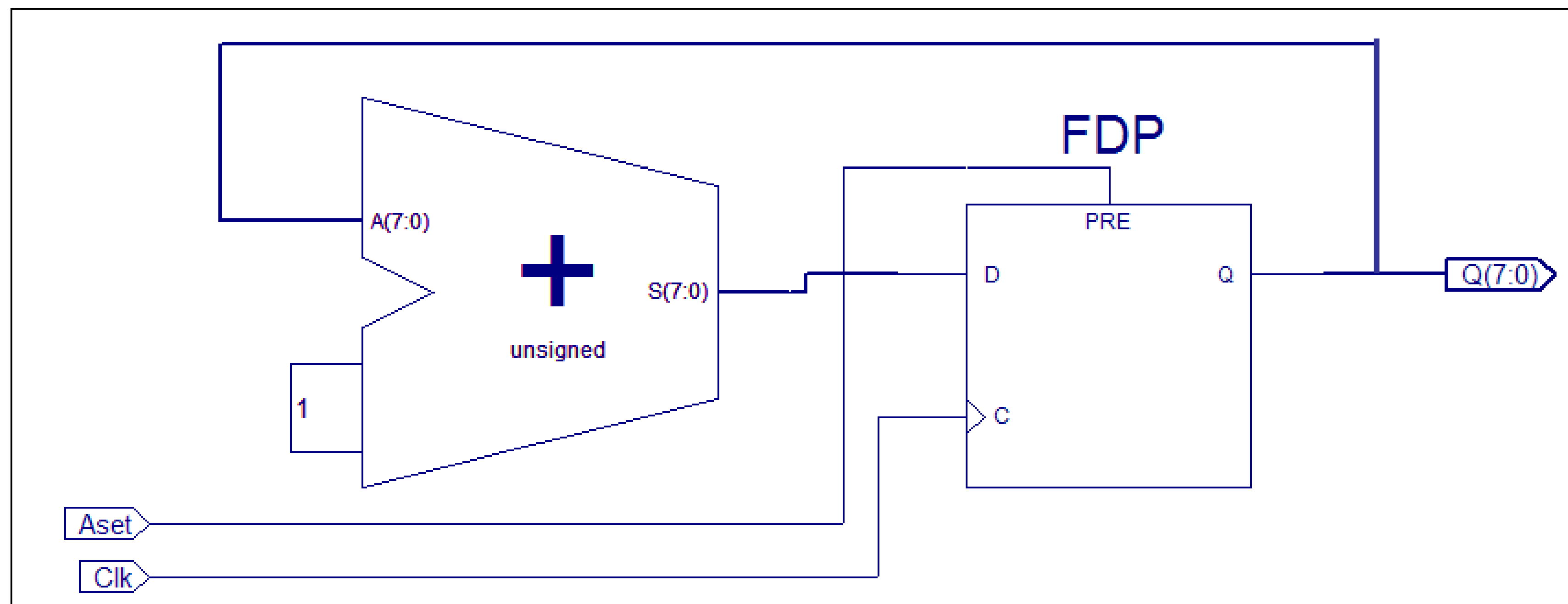


# Dodanie sygnału *preset* (cd.)

```
library ieee;
use ieee.std_logic_unsigned.all;
architecture beh of licz2 is
  signal upcount : std_logic_vector(WIDTH-1 downto 0);
begin
  UPCOUNTER: process(clk_i, rst_i)
  begin
    if rst_i = '1' then
      upcount <= (others => '1');
    elsif rising_edge(clk_i) then
      upcount <= upcount + 1;
    end if;
  end process;
  upcount_o <= upcount;
end architecture;
```

# Dodanie sygnału *reset* (cd.)



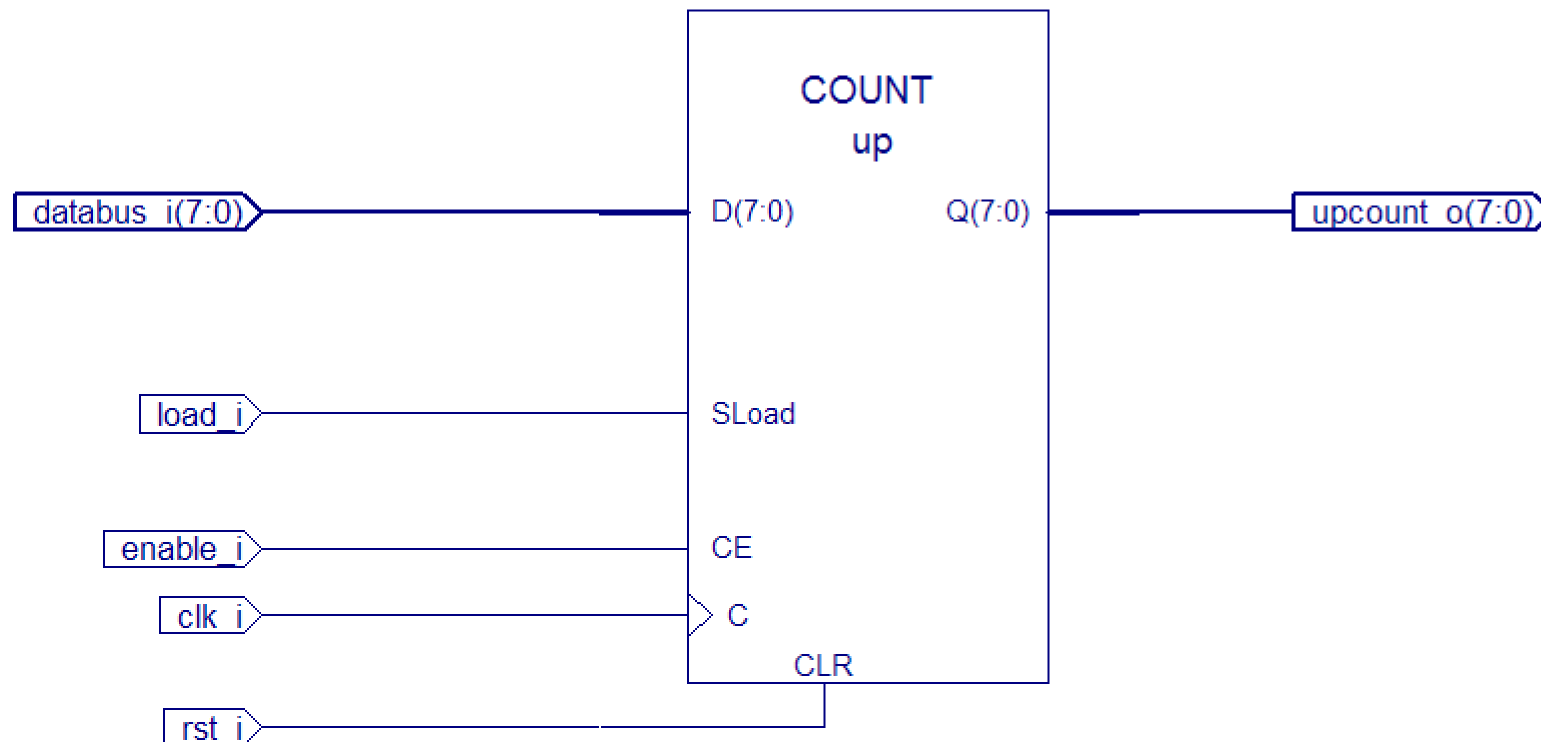
# Dodanie wejść ENABLE oraz LOAD

```
library ieee;
use ieee.std_logic_1164.all;
entity licz3 is
  generic (
    WIDTH : integer := 8
    );
  port (
    clk_i       : in  std_logic;
    rst_i       : in  std_logic;
    databus_i   : in  std_logic_vector(WIDTH-1 downto 0);
    enable_i    : in  std_logic;
    load_i      : in  std_logic;
    upcount_o   : out std_logic_vector(WIDTH-1 downto 0)
    );
end entity; -- cd na następnej planszy
```

# Dodanie wejść ENABLE oraz LOAD (cd.)

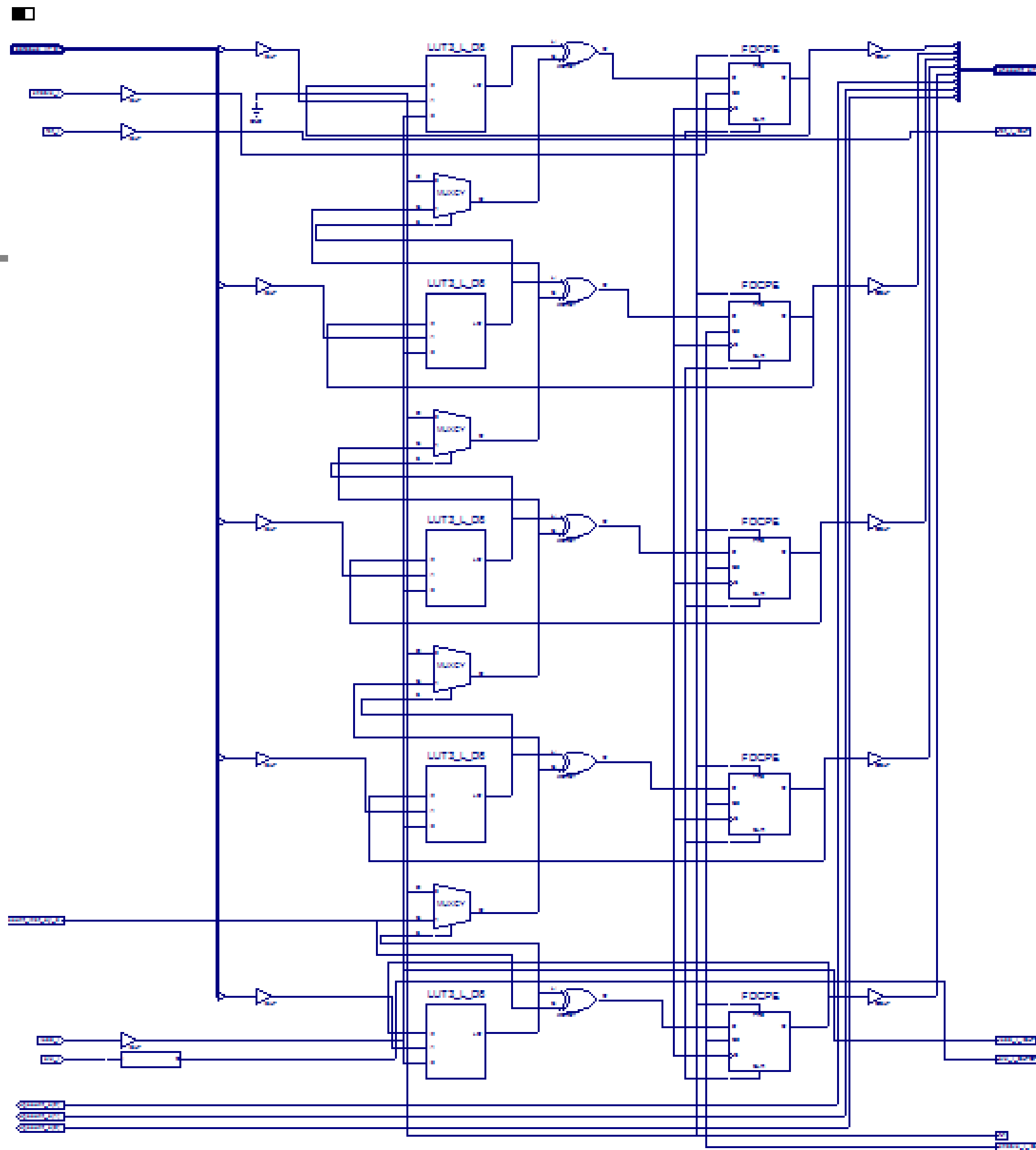
```
...
UPCOUNTER: process(clk_i, rst_i)
  begin
    if rst_i = '1' then
      upcount <= (others => '0');
    elsif rising_edge(clk_i) then
      if enable_i = '1' then
        if load_i = '1' then
          upcount <= databus_i;
        else -- normalne zliczanie
          upcount <= upcount + 1;
        end if;
      end if;
    end if;
  end process;
...
```

# Dodanie wejść ENABLE oraz LOAD (cd.)



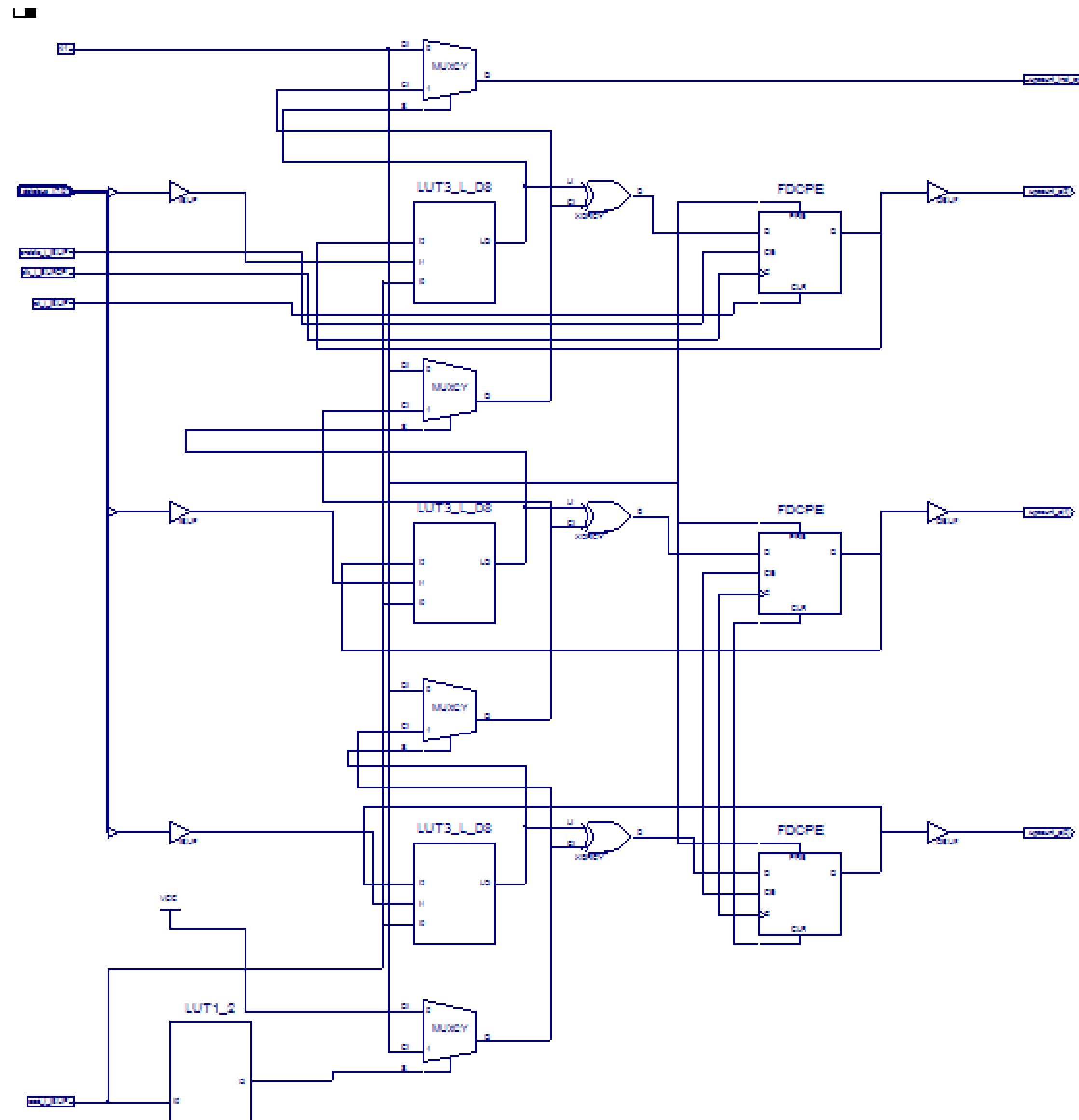
Schemat licznika z wejściem **load** i **enable**

(cz. 1 z 2)



Schemat licznika z wejściem **load** i **enable**

(cz. 2 z 2)



# Dodanie wejść ENABLE oraz LOAD (cd.)

- Modyfikacja:

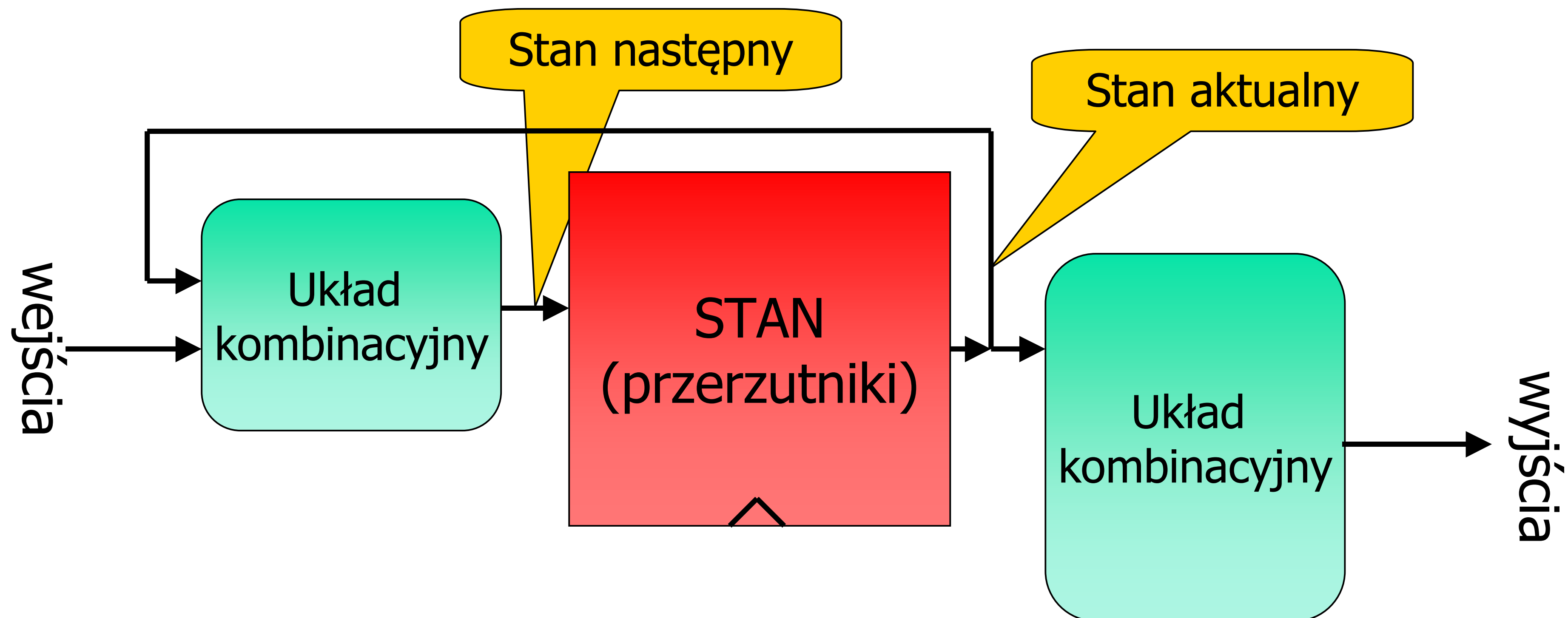
```
if load_i = '1' then
  upcount <= databus_i;
elsif enable_i = '1' then
  upcount <= upcount + 1;
end if;
```

# Dodanie wejść ENABLE oraz LOAD (cd.)

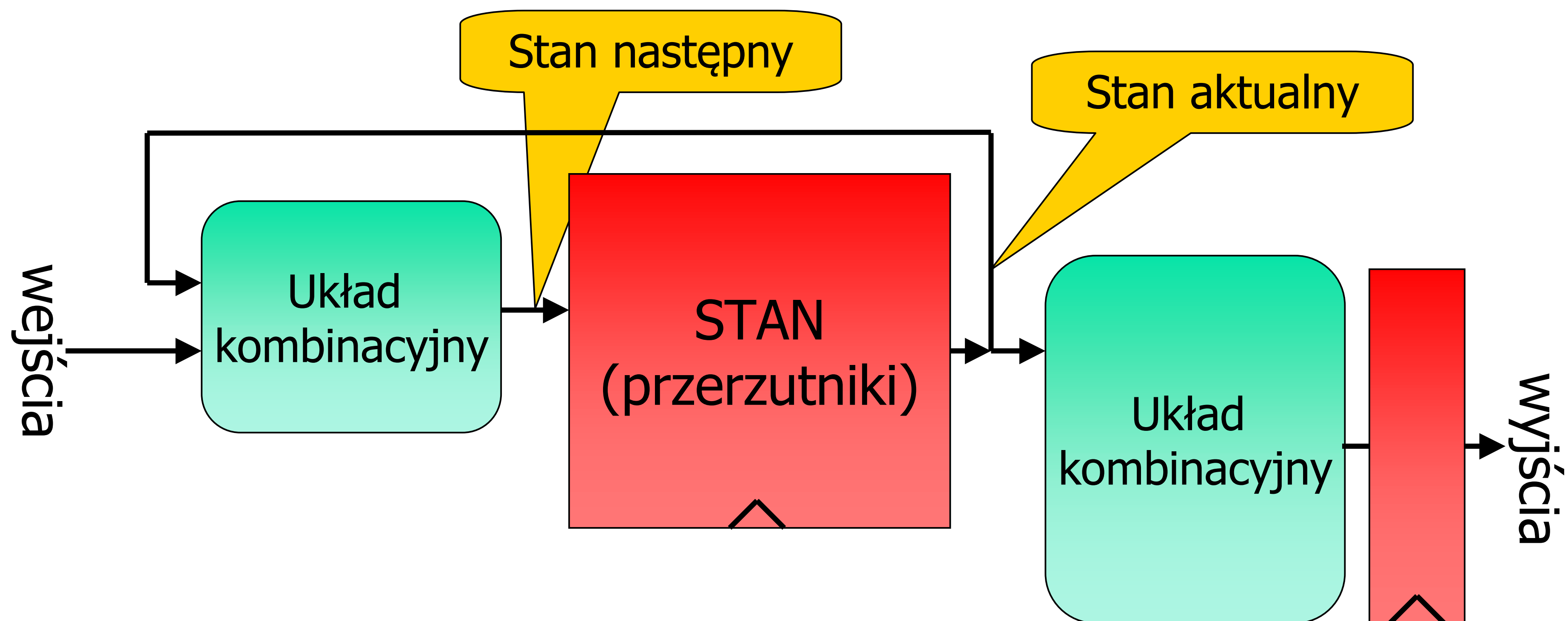
```
if load_i = '1' then
  upcount <= databus_i;
end if;

if enable_i = '1' then
  upcount <= upcount + 1;
end if;
```

# Maszyny stanów

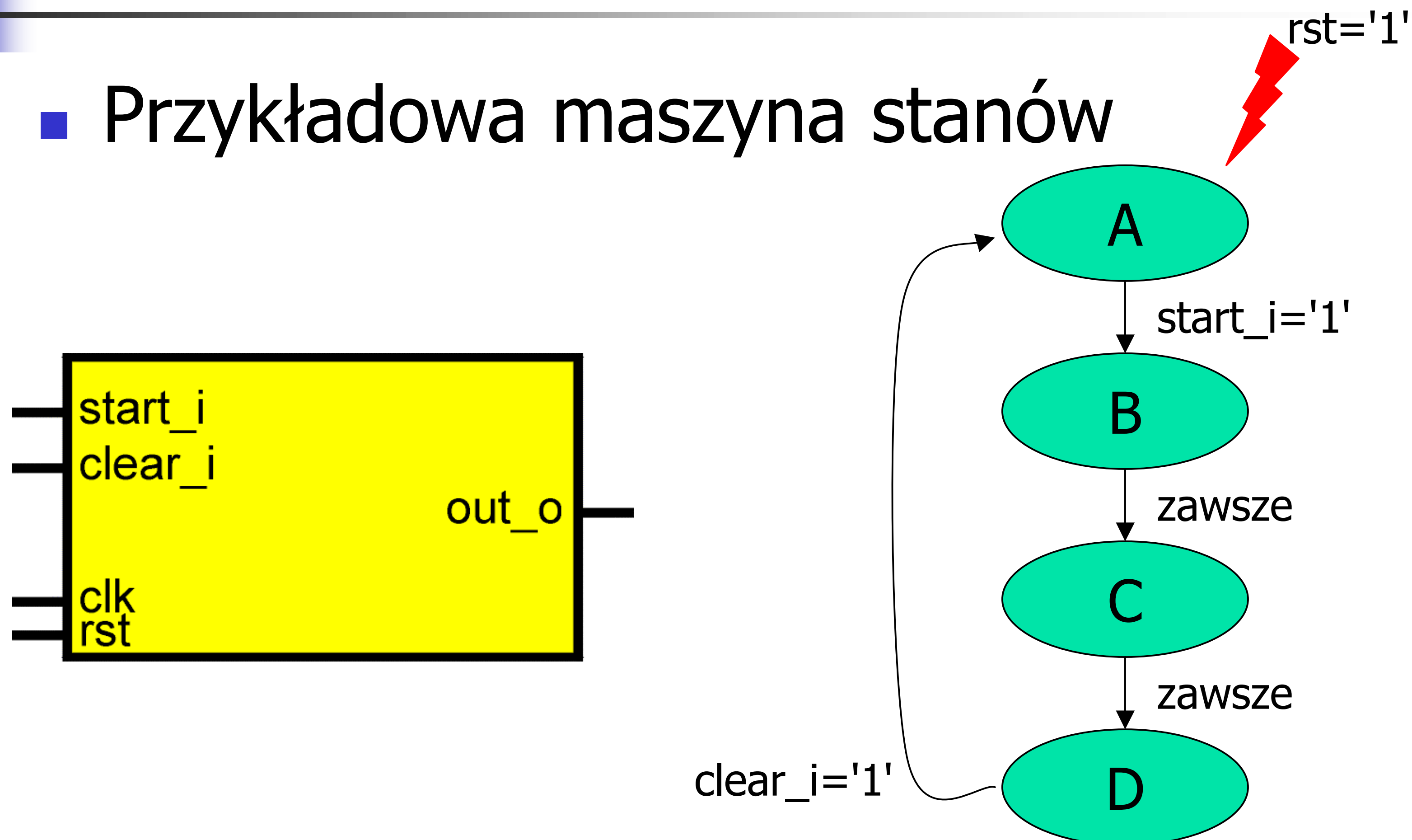


# Maszyny stanów



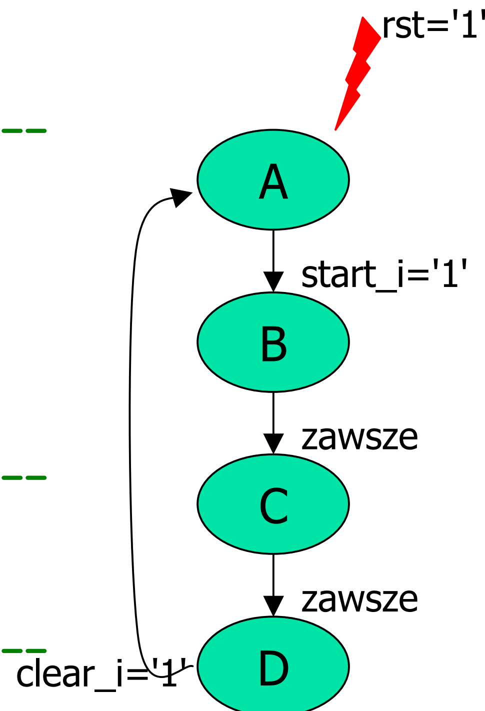
# Synteza maszyn stanów

- Przykładowa maszyna stanów



# Maszyna stanu (2 procesy)

```
COMB: process (state,
 start_i, clear_i)
begin
  case state is
    ------------------------
    when A =>
    out_o <= '0';
    if start_i = '1' then
      state_next <= B;
    else
      state_next <= A;
    end if;
    ------------------------
    when B =>
    out_o <= '0';
    state_next <= C;
    ------------------------
    when C =>
    out_o <= '0';
    state_next <= D;
```

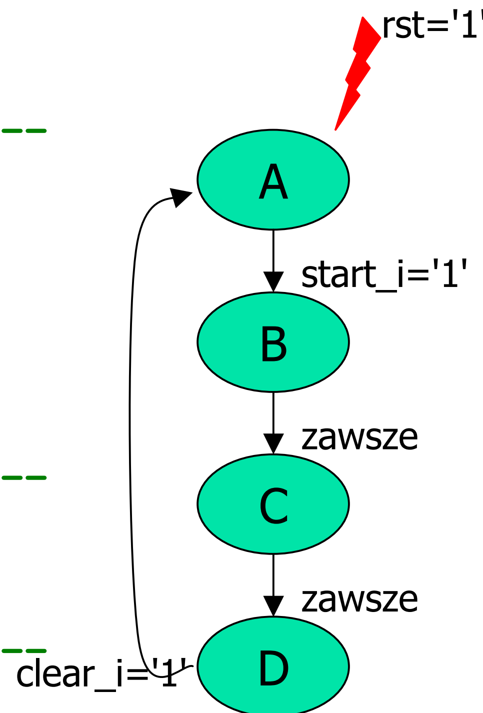



```
    when D =>
    out_o <= '1';
    if clear_i = '1' then
      state_next <= A;
    else
      state_next <= D;
    end if;
    ------------------------
  end case;
end process;

SEKW: process (clk, rst)
begin
  if rst = '1' then
    state<= A;
  elsif rising_edge(clk)
 then
    state <= state_next;
  end if;
end process;
```

# Maszyna stanu (2 procesy) - ten kod można uprościć!

```
COMB: process (state,
 start_i, clear_i)
begin
  case state is
    -----------------------
    when A =>
    out_o <= '0';
    if start_i = '1' then
      state_next <= B;
    else
      state_next <= A;
    end if;
    -----------------------
    when B =>
    out_o <= '0';
    state_next <= C;
    -----------------------
    when C =>
    out_o <= '0';
    state_next <= D;
```




```
    when D =>
    out_o <= '1';
    if clear_i = '1' then
      state_next <= A;
    else
      state_next <= D;
    end if;
    -----------------------
  end case;
end process;

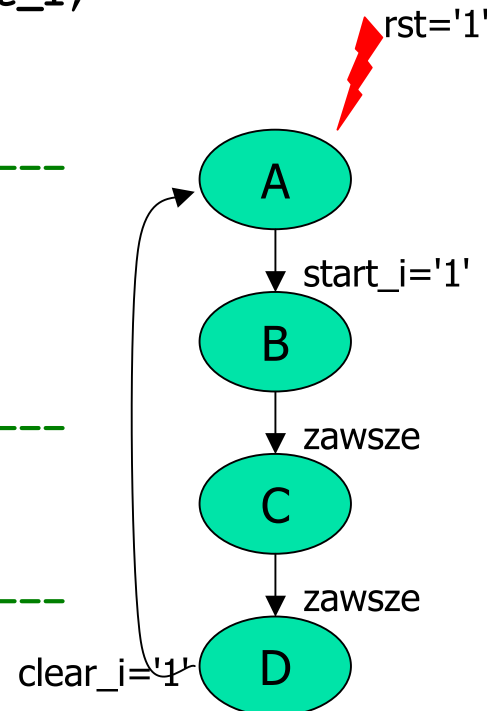
SEKW: process (clk, rst)
begin
  if rst = '1' then
    state<= A;
  elsif rising_edge(clk)
  then
    state <= state_next;
  end if;
end process;
```

# Maszyna stanu (2 procesy) – uproszczenie kodu

```
COMB: process (state, start_i,
  clear_i)
 begin
   case state is
     -------------------------
     when A =>
     out_o <= '0';
     if start_i = '1' then
       state_next <= B;
     end if;
     -------------------------
     when B =>

     state_next <= C;
     -------------------------
     when C =>
     out_o <= '0';
     state_next <= D;
```

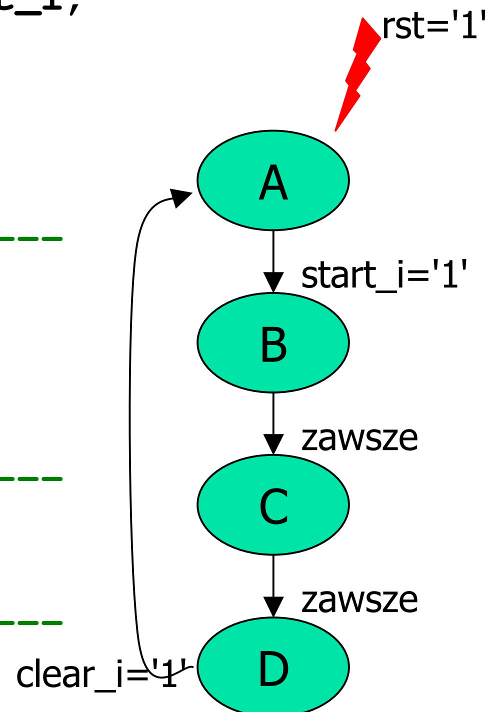
*Błąd: tu brak przypi-sania!* ***out_o****!!*



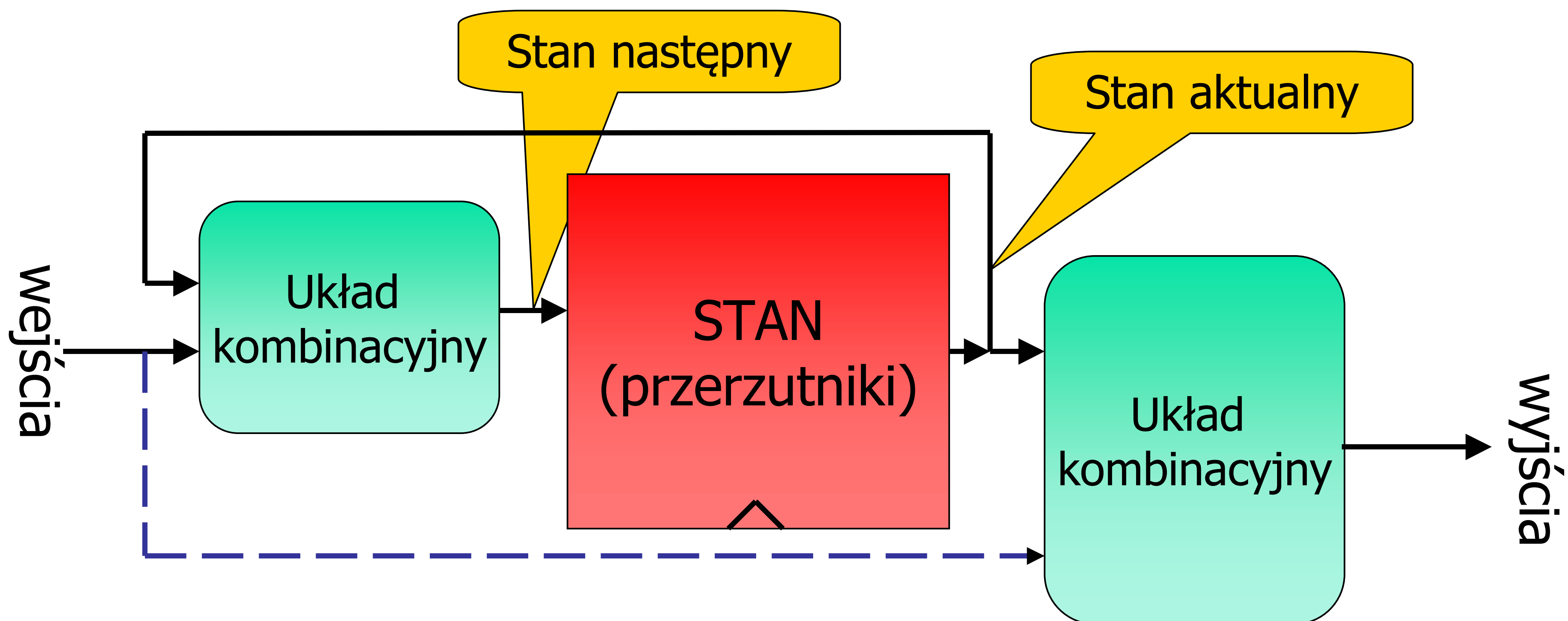

```
     ---------------------
     when D =>
     out_o <= '1';
     if clear_i = '1' then
       state_next <= A;
     end if;
     ---------------------
   end case;
 end process;
```

# Maszyna stanu (2 procesy) – uproszczenie kodu

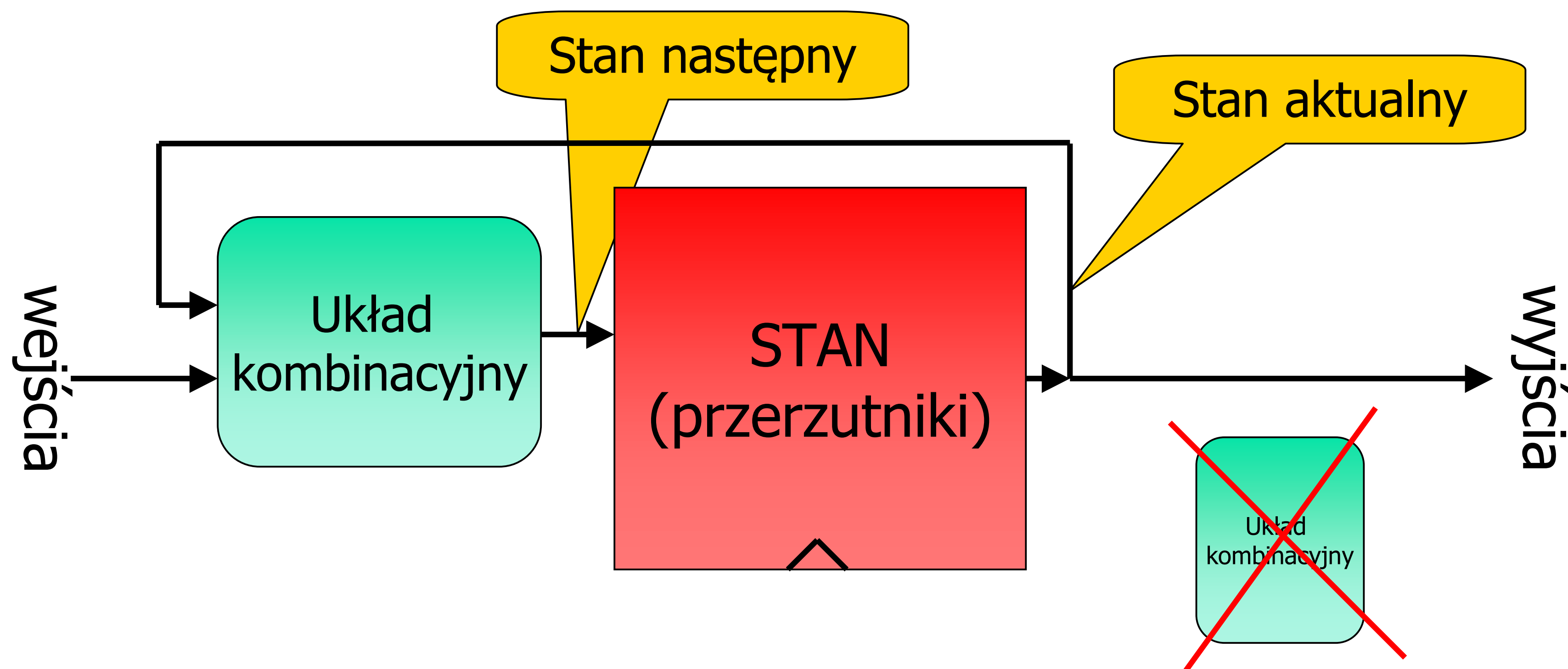
```
COMB: process (state, start_i,
  clear_i)
 begin
   out_o <= '0';
   case state is
     -----------------------
     when A =>
     if start_i = '1' then
       state_next <= B;
     end if;
     -----------------------
     when B =>
     state_next <= C;
     -----------------------
     when C =>
     state_next <= D;
     -----------------------
     when D =>
     out_o <= '1';
     if clear_i = '1' then
       state_next <= A;
     end if;
     -----------------------
   end case;
 end process;
```



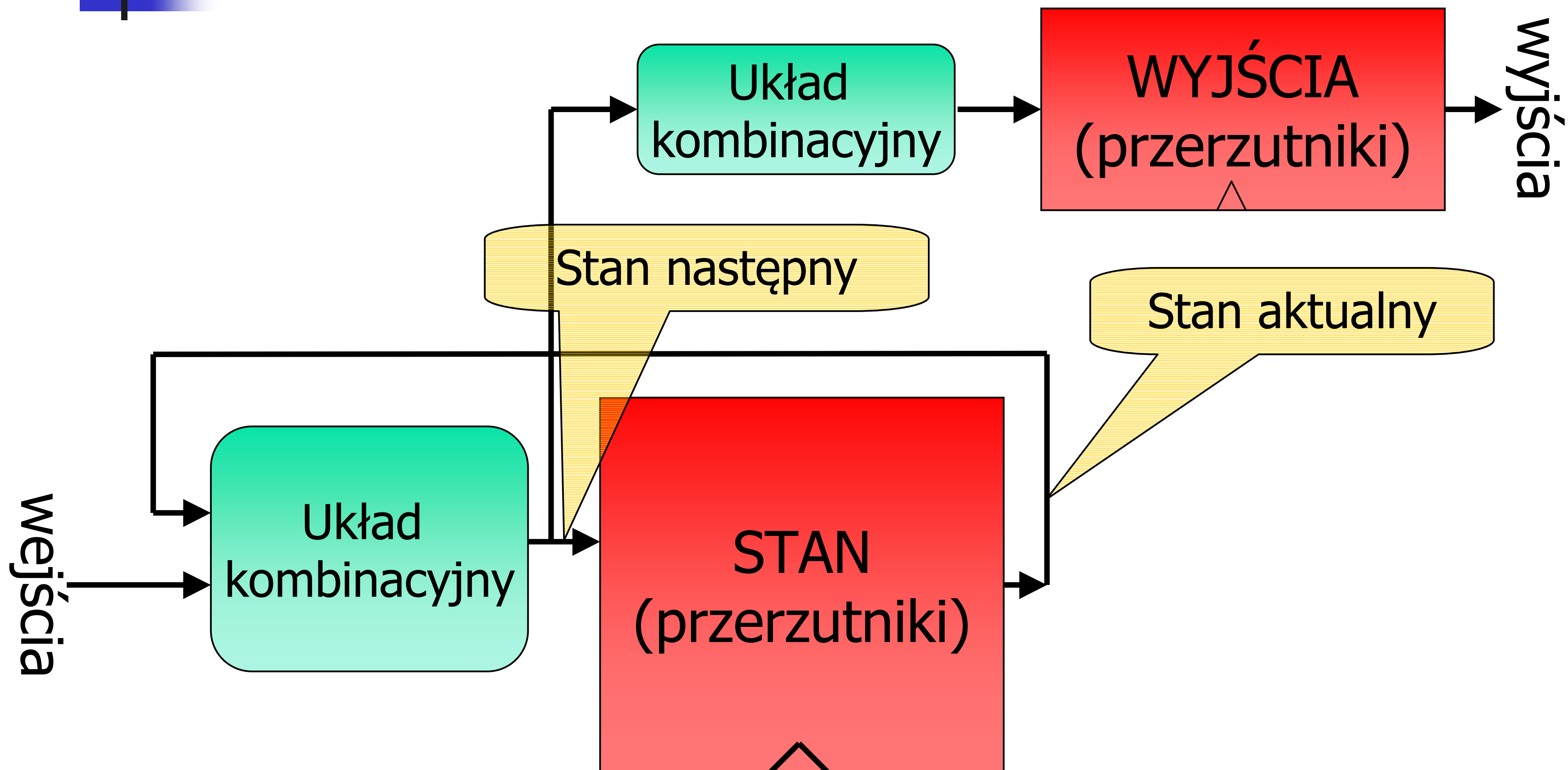
# Maszyny stanów - modyfikacje



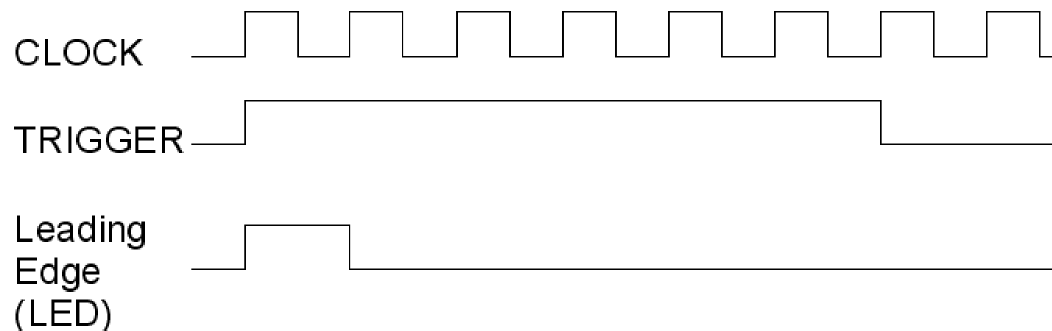
# Maszyny stanów - modyfikacje



# Maszyny stanów - modyfikacje



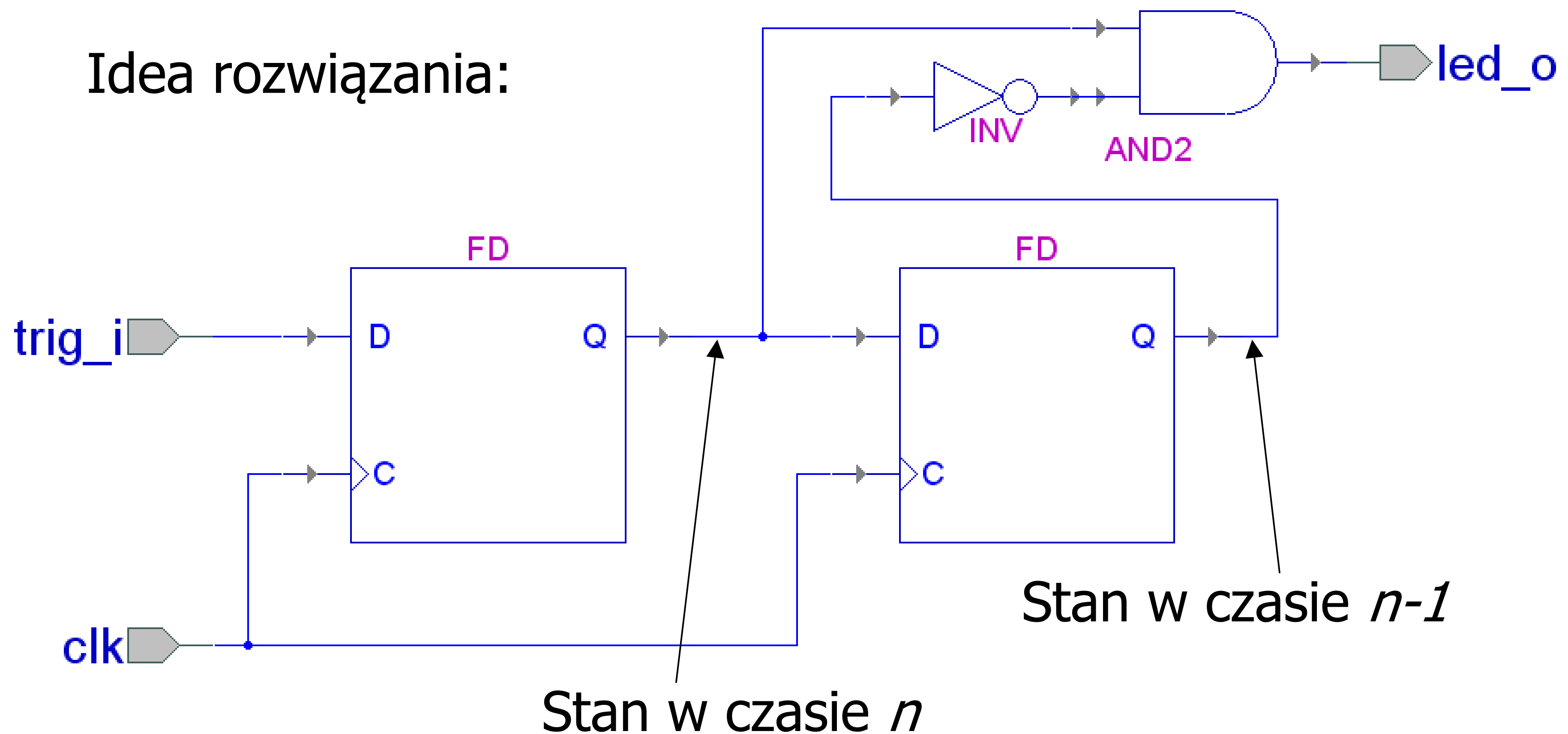
# Detektor zbocza narastającego




- Wyjście: sygnał o długości jednego cyklu zegarowego pojawiającym się po narastającym zboczu sygnału wejściowego **TRIGGER**.
- Funkcja dwustanowa:
  - stan pierwszy oczekuje na pojawienie się zbocza i kiedy się on pojawi, ustawiane jest wyjście **LED**
  - stan drugi ustawia **LED** do stanu niskiego i następnie oczekuje na zmianę do stanu niskiego wejścia **TRIGGER**.

# Detektor zbocza narastającego

Idea rozwiązania:



# Detektor zbocza narastającego (cd.)

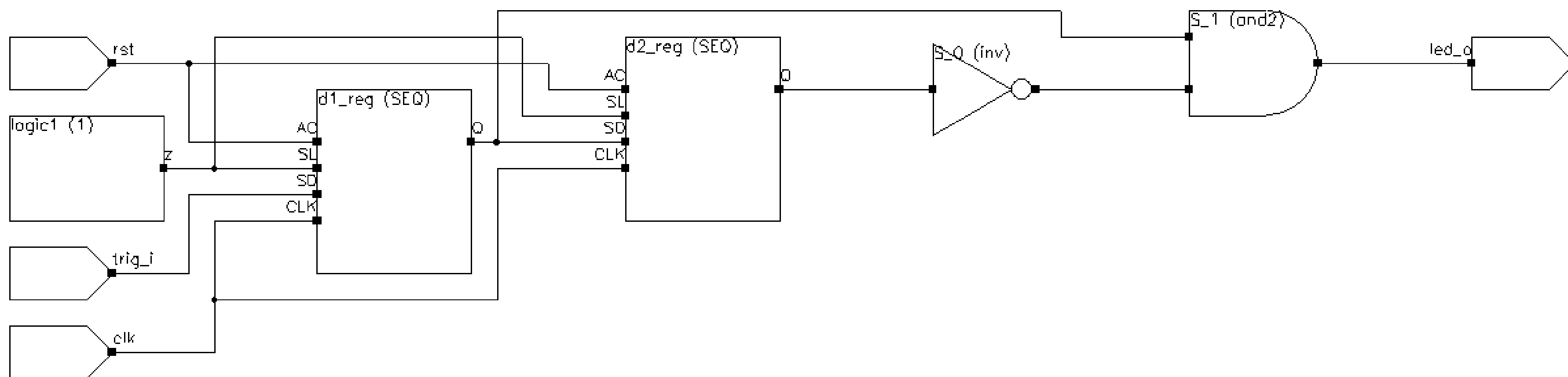
```
architecture beh of sm_ref is
  type t_STATE is (S0, S1);
  signal state, next_state : t_STATE;

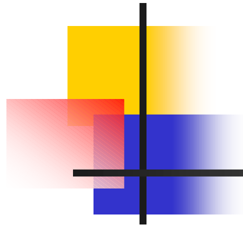
  signal d1, d2 : std_logic;
begin

  SEQW: process (clk, rst)
  begin
    if rst = '1' then
      d1 <= '0';
      d2 <= '0';
    elsif Rising_Edge(clk) then
      d1 <= trig_i;
      d2 <= d1;
    end if;
  end process;

  led_o <= '1' when d1='1' and d2 = '0' else '0';
end architecture;
```

# Detektor zbocza narastającego (cd.)





# Część 19 (VHDL)

## Atrybuty

# Typy danych (cd.)

- Typy tablicowe - Atrybuty tablic

```
signal moj_wektor : std_logic_vector (5 downto –5);
```

| **Wyrażenie zawierające atrybut** | **Wartość** |
|---|---|
| moj_wektor’**left** | 5 |
| moj_wektor’**right** | -5 |
| moj_wektor’**high** | 5 |
| moj_wektor’**low** | -5 |
| moj_wektor’**length** | 11 |
| moj_wektor’**range** | (5 downto –5) |
| moj_wektor’**reverse_range** | (-5 to 5) |

VHDL

# Atrybuty dotyczące sygnałów

- atrybut **'event** – zwraca **TRUE**, gdy w danym kroku symulacji nastąpiła zmiana wartości danego sygnału:

```
proces (RST, CLK)
begin
  if RST = '1' then
    Q <= '0';
  elsif CLK'event and CLK = '1' then
    -- wykrywanie dodatniego zbocza sygnału clk
    Q <= D;
  end if;
end process;
```

# Atrybuty dotyczące sygnałów (cd.)

- atrybut **'active** – zwraca **TRUE**, gdy w danym kroku symulacji nastąpiło zaplanowanie jakiejś operacji dla danego sygnału:

```
process
variable A, B : boolean;
begin
  Q <= D after 10 ns;
  A := Q'active; -- A = TRUE
  B := Q'event;  -- B = FALSE
  ...
end process;
```

# Atrybuty dotyczące sygnałów (cd.)

- atrybut '**last_event** – zwraca czas, jaki upłynął od czasu ostatniej zmiany sygnału:

```
process
  variable T : time;
begin
  Q <= D after 10 ns;
  wait for 15 ns;
  T := Q'last_event; -- T = 5ns
...
end process;
```

# Atrybuty dotyczące sygnałów (cd.)

VHDL

- atrybut '**last_active** – zwraca czas, jaki upłynął od czasu ostatniego zaplanowania jakiejś operacji dla danego sygnału:

```
process
  variable T : time;
begin
  Q <= D after 40 ns;
  wait for 10 ns;
  T := Q'last_active; -- T = 10 ns
...
end process;
```

# Atrybuty dotyczące sygnałów (cd.)

VHDL

- atrybut '**last_value** – zwraca wartość, jaką miał sygnał przed ostatnim przypisaniem:

```
process
  variable V : bit;
begin
  Q <= '1';
  wait for 10 ns;
  Q <= '0';
  wait for 10 ns;
  V := Q'last_value; -- V = '1'
...
end process;
```

# Atrybuty dotyczące sygnałów (cd.)

- Uwaga: atrybuty **'active**, '**last_event**, '**last_value** i '**last_active** nie są zazwyczaj syntezowalne – przypadku opisu układu do syntezy powinno się używać tylko atrybutu **'event**.
- Atrybuty **'active**, '**last_event**, '**last_value** i '**last_active** powinny być używane tylko do symulacji i testowania układu – np. do sprawdzania czasów ustalania się sygnałów (**setup time**, **hold time**).

# Atrybuty konwersji pomiędzy zmienną i łańcuchem

- atrybut <**nazwa_typu**>**'image**(<**wyrażenie**>) – zwraca łańcuch string reprezentujący wyrażenie <**wyrażenie**>, które musi być typu <**nazwa_typu**>. Typ <**nazwa_typu**> musi być typem skalarnym.

```
variable X :integer;
...
assert (ERROR = FALSE)
  report "Błąd symulacji, wartość X=" &
  integer'image(X);
```

# Atrybuty konwersji pomiędzy zmienną i łańcuchem (cd.)

- atrybut <**nazwa_typu**>'**value**(<**łańcuch**>) – zamienia łańcuch <**łańcuch**> na wartość typu <**nazwa_typu**>:

```
next_state <= STATE'value(text_line);
-- zmienna text_line jest typu string
-- zmienna next_state jest typu STATE
```

# Atrybuty tworzące nowe sygnały

- atrybut <**sygnał**>**'delayed**(<**czas**>) – tworzy nowy sygnał, identyczny jak sygnał <**sygnał**>, ale opóźniony o czas <**czas**>:

```
process (clk'delayed(hold))
```

VHDL

# Atrybuty tworzące nowe sygnały (cd.)

- atrybut <**sygnał**>'**stable**(<**czas**>) – tworzy nowy sygnał typu **boolean** (wartość **TRUE**, gdy <**sygnał**> stały przez <**czas**>):

```
process
  variable A: boolean;
begin
  Q <= D after 30 ns;
  wait for 10 ns;
  A := Q'stable(20 ns); -- A=TRUE, zdarzenie jeszcze
                        --              nie nastąpiło
  wait for 30 ns;
  A := Q'stable(20 ns); -- A=FALSE, zdarzenie nastąpiło
                        --        10 ns wcześniej
  ...
end process;
```

# Atrybuty tworzące nowe sygnały (cd.)

- atrybut <**sygnał**>**'quiet**(<**czas**>) – tworzy nowy sygnał typu **boolean**, (**TRUE**, gdy dla <**sygnał**> nie zaplanowano operacji <**czas**>):

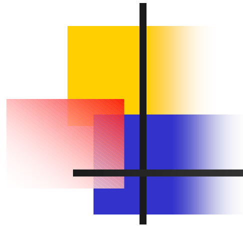
```
process
  variable A: boolean;
begin
  Q <= D after 30 ns;
  wait for 10 ns;
  A := Q'quiet(20 ns); -- A=FALSE, nastąpiło zaplanowanie
                       -- przypisania 10ns wcześniej
  wait for 40 ns;
  A := Q'quiet(20 ns); -- A = TRUE, ostatnie zdarzenie
                       -- nastąpiło 20 ns wcześniej
  ...
end process;
```

# Atrybuty tworzące nowe sygnały (cd.)

- atrybut <**sygnał**>**'transaction** – tworzy nowy sygnał typu **bit**, który zmienia swą wartość za każdym razem, gdy następuje zmiana sygnału <**sygnał**> lub dla tego sygnału następuje zaplanowanie jakiejś operacji.
  - Uwaga: atrybutów tworzących nowe sygnały można używać w każdym miejscu, gdzie dopuszczalne jest użycie zwykłego sygnału odpowiedniego typu.

# Pozostałe atrybuty

- **'structure -** zwraca **TRUE**, jeśli architektura do której się odnosi zawiera osadzone inne komponenty;
- **'behavior** – zwraca **TRUE**, jeśli architektura do której się odnosi nie zawiera osadzonych innych komponentów;
- '**simple_name** – zwraca prostą nazwę obiektu jako łańcuch;
- '**instance_name** – zwraca pełną nazwę obiektu jako łańcuch;
- '**path_name** – zwraca pełną ścieżkę do obiektu;
- '**pos**(<**wartość**>) – zwraca pozycję (indeks) podanej wartości na liście wszystkich wartości typu;
- '**val**(<**indeks**>) – zwraca wartość danego typu na podstawie indeksu;
- **'succ**(<**wartość**>) – zwraca wartość o następnym indeksie po wartości<**wartość**>;
- **'pred**(<**wartość**>) – zwraca wartość o poprzednim indeksie przed wartością<**wartość**>;
- **'leftof**(<**wartość**>) – zwraca wartość po lewej stronie wartości <**wartość**>;
- **'rightof**(<**wartość**>) – zwraca wartość po prawej stronie wartości<**wartość**>;
- atrybut **'base** – zwraca typ podstawowy dla danego typu
- Atrybuty dotyczące tablic wielowymiarowych, zwracają cechy dla wymiarui <wymiar>:
  - **'left**(<**wymiar**>), **'right**(<**wymiar**>), **'high**(<**wymiar**>), **'low**(<**wymiar**>) **'range**(<**wymiar**>), '**reverse_range**(<**wymiar**>), **'length**(<**wymiar**>), **'ascending**(<**wymiar**>);

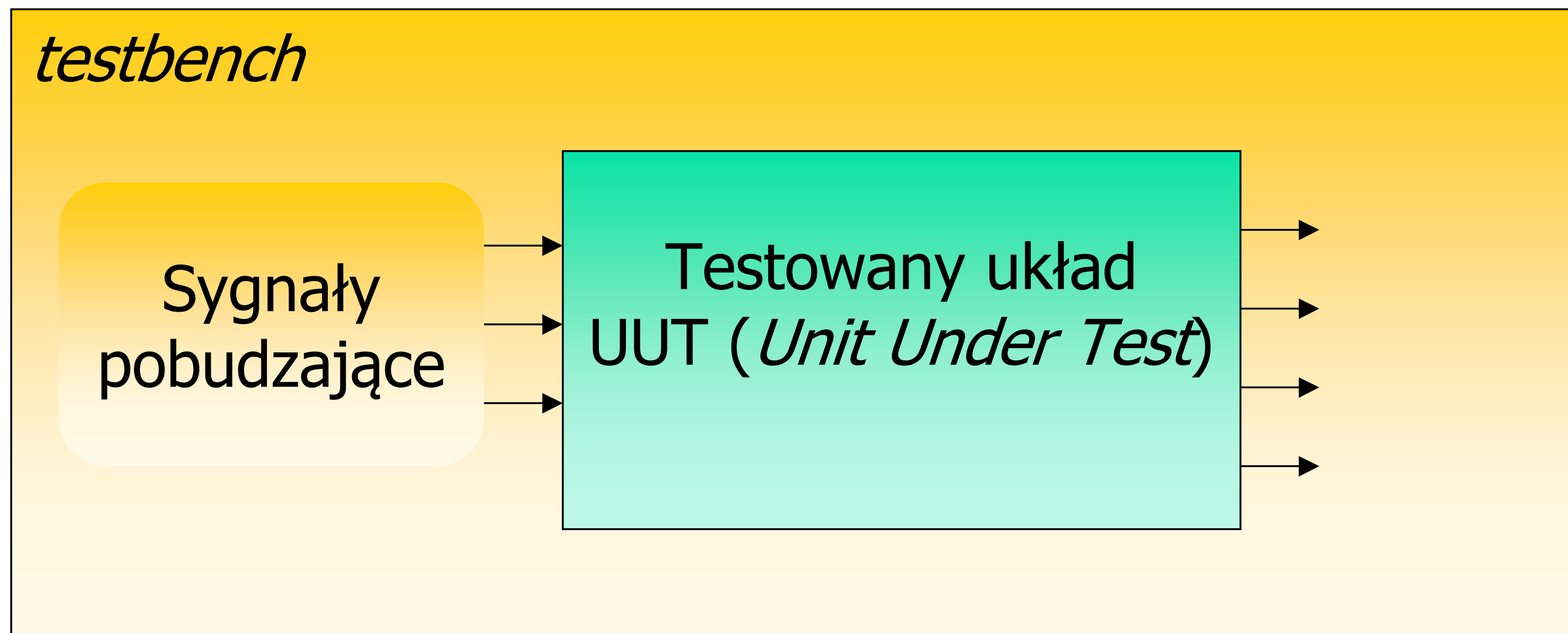


# Część 20 (VHDL)

## Symulacja i testowanie

# Struktura *testbench*

- Najczęściej wykorzystywana struktura:



# Sygnały pobudzające

- Przypisanie wartości początkowych:

```
signal clk : std_logic:='1';
signal vec : std_logic_vector(7 downto 0) := "00000000";
```

  - Nie zawsze syntezowalne...

- Sygnały stałe

```
sig <= '1';
data <= "11001";
```

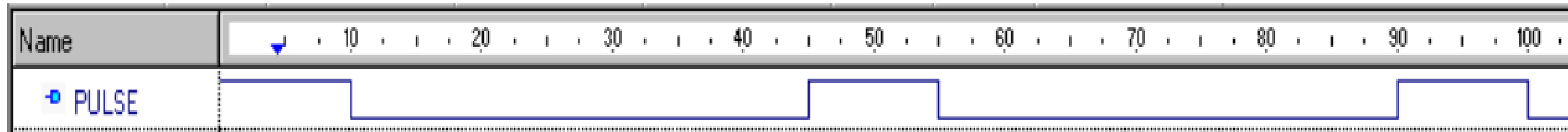
# Sygnały pobudzające (cd.)

- Sygnały powtarzalne

```
signal clk : std_logic :='0';
...
process
begin
  clk <= not clk after 25 ns;
end process;
```

```
CLK_PROC: process
begin
  clk <= '0';
  CLK_LOOP: loop
    wait for CLK_PERIOD/2;
    clk <= not clk;
  end loop;
end process;
```

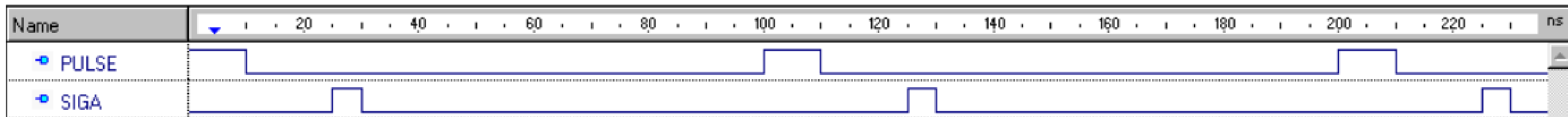
```
constant CLK_PERIOD: time := 50 ns;
clk <= not clk after CLK_PERIOD/2;
```




# Sygnały pobudzające (cd.)

- Niesymetryczne sygnały powtarzalne

```
PULSE1: process
constant ZERO_TIME : time := 35 ns;
begin
  pulse <= '1';
  wait for 10 ns;
  pulse <= '0';
  wait for ZERO_TIME;
end process PULSE1;
```




# Sygnały pobudzające (cd.)

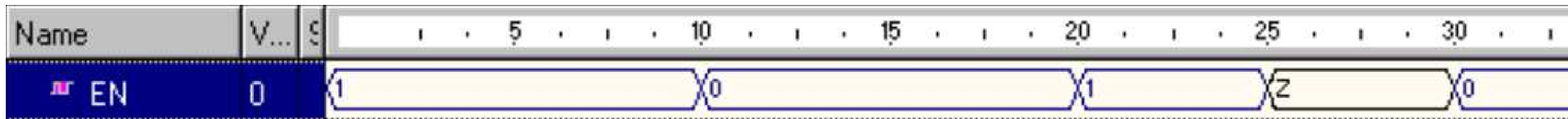
- Niesymetryczne sygnały powtarzalne (cd.)

```
PULSE1: process
begin
  pulse <= '1';
  wait for 10 ns;
  pulse <='0';
  wait for 15 ns;
  sig_a <='1';
  wait for 5 ns;
  sig_a <='0';
  wait for 70 ns; -- całkowity okres wynosi 100ns
end process PULSE1;
```

# Sygnały pobudzające (cd.)

- Proste sygnały niepowtarzalne
  - Proste sygnały niepowtarzalne występują tylko jednokrotnie (np. **reset**):

```
constant RESET_WIDTH : time:=140 ns;
...
reset <= '0', '1' after RESET_WIDTH; -- reset
                                     -- aktywowany
                                     -- zerem
```




# Sygnały pobudzające (cd.)

- Proste sygnały niepowtarzalne (cd.)

```
en<='1' , '0' after 10 ns, '1' after 20 ns, 'Z' after
   25 ns, '0' after 30 ns;
```

- Regularne wzory danych

```
signal databus : unsigned(7 downto 0) := "00000000";
process
begin
  databus <= databus + 1 after 75 ns;

end process;
```

# Sygnały pobudzające (cd.)

- Regularne wzory danych (cd.)
  - Dla przejrzystości kodu można wprowadzić stałą będącą okresem zmian:

```
signal databus : unsigned(7 downto 0) := "00000000";
constant DELAYTIME : time := 75 ns;
process
begin
  databus <= databus + '1' after DELAYTIME;
end process;
```

  - Wzór kroczącej jedynki można uzyskać poprzez dodawanie wartości do samej siebie lub inaczej mnożenie przez **2**:

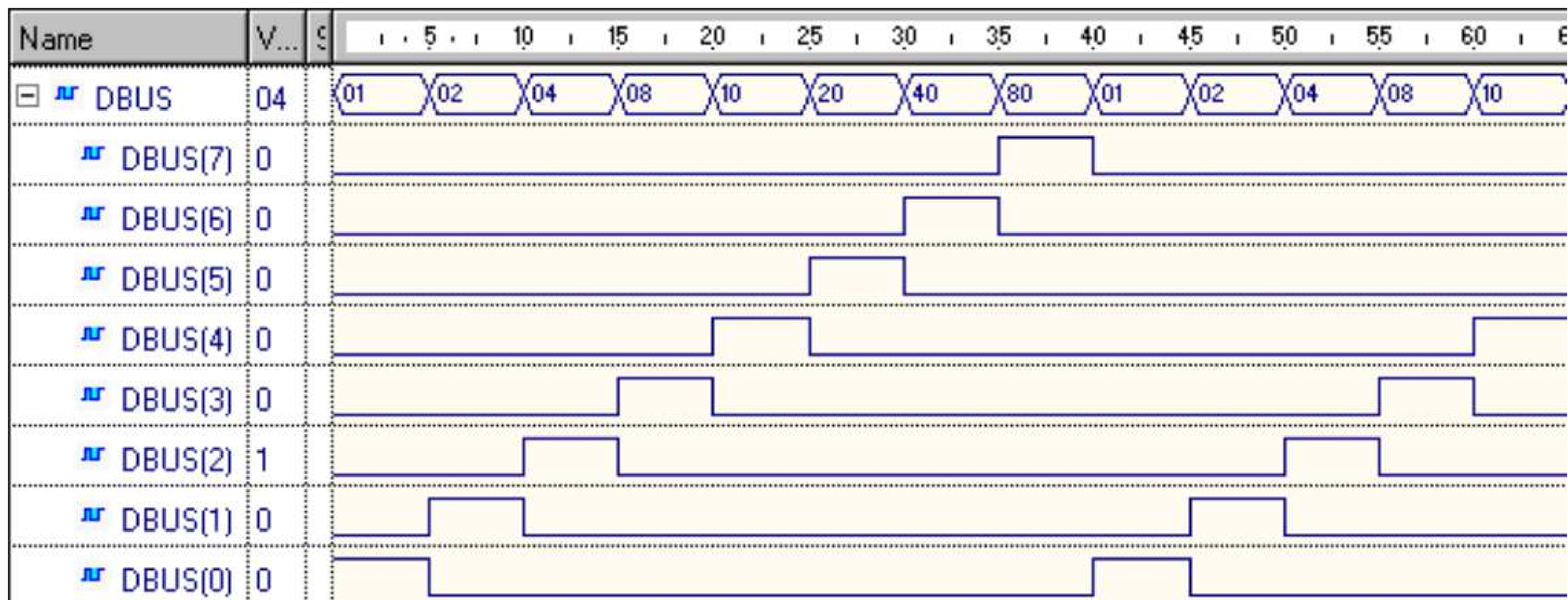
# Sygnały pobudzające (cd.)

- Regularne wzory danych (cd.)

```
architecture BEH of PULSE is
  signal DBUS : unsigned (7 downto 0);
begin
  WALK1: process
  begin
    DBUS <= "00000001";
    LOOP1: for I in 0 to DBUS'length-1 loop
      wait for 5 ns;
      DBUS <= DBUS + DBUS;
    end loop LOOP1;
  end process WALK1;
end architecture;
```

# Sygnały pobudzające (cd.)

- Regularne wzory danych (cd.)



# Sygnały pobudzające (cd.)

- Wzory danych odczytywane z tablic

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

entity TB is
end entity;

architecture BEH of TB is
  type FREQ is array (1 to 6) of integer;
  constant FREQVAL : FREQ := (10230,
                              10231,
                              10232,
```

# Sygnały pobudzające (cd.)

- Wzory danych odczytywane z tablic (cd.)

```
                                          10233,
                                          10234,
                                          10235);
  constant SIZE : integer := 16;
  signal dbus   : integer;
  signal dbus_v : std_logic_vector(SIZE-1 downto 0);
  constant BUS_DELAY: time:=10 ns;
begin
  TABLE_GEN: process
  begin
    LOOP1: for I in 1 to 6 loop
      dbus   <= FREQVAL(I);
      dbus_v <= conv_std_logic_vector(FREQVAL(I),SIZE);
      wait for BUS_DELAY;
    end loop LOOP1;
  end process;
end architecture;
```

# Sygnały pobudzające (cd.)

- Wzory danych odczytywane z tablic (cd.)

```
-- type FREQ is array (1 to 6) of integer;
type FREQ is array (natural range <>) of
  integer;

--LOOP1: for I in 1 to 6 loop
LOOP1: for I in FREQVAL'low to FREQVAL'high
  loop
```

# Instrukcja kontroli **assert**

```
assert PULSE = '0'
   report "Pulse has gone high"
   severity WARNING;
```

- Typy ważności komunikatu mogą być **FAILURE**, **ERROR**, **WARNING** oraz **NOTE**.
- Polecenie **assert** może występować zarówno w części współbieżnej jak i sekwencyjnej kodu VHDL.
- Drukowanie wartości zmiennych:

```
variable X :integer;
...
assert (ERROR = FALSE)
  report "Błąd symulacji, wartość X=" & integer'image(X);
```

Praca z plikami

(pakiet **textio**)

# Pakiet **textio**, biblioteka **std**

```
use std.textio.all;
```

- Pakiet **textio** - typy danych:
  - Typ **line**:

```
type line is access string;
```

  - Typ **text**:

```
type text is file of string;
```

  - Typ **side**:

```
type side is (RIGHT,LEFT)
```

```
use std.textio.all;
```

VHDL

# Pakiet **textio** (cd.)

- Pakiet **textio** - typy danych (cd.):
  - Podtyp **width**:

```
subtype width is natural;
```

  - Standardowe zmienne plikowe:

```
file input: text open READ_MODE is "std_input";
file output: text open WRITE_MODE is "std_output"
```

```
use std.textio.all;
```

VHDL

# Pakiet **textio** (cd.)

- Pakiet **textio** – procedury i funkcje:

```
procedure readline(file F: text; L: out line);

procedure writeline(file F: text; L: inout line);
```

```
use std.textio.all;
```

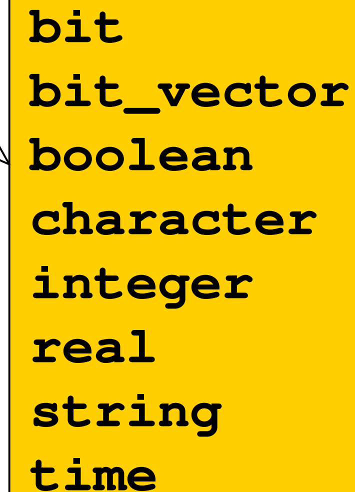
VHDL

# Pakiet **textio** (cd.)

- Pakiet **textio** – procedury i funkcje (cd.):

```
procedure read(L:inout line; VALUE: out ?; GOOD: out);
```

```
procedure read(L:inout line; VALUE: out ?);
```



```
bit
bit_vector
boolean
character
integer
real
string
time
```

```
use std.textio.all;
```

VHDL

# Pakiet **textio** (cd.)

- Pakiet **textio** – procedury i funkcje (cd.):

```
bit
bit_vector
boolean
character
integer
string
```

```
procedure write(L: inout line; VALUE : in  ? ; JUSTIFIED: in
   side := RIGHT; FIELD: in width := 0);

procedure write(L: inout line; VALUE : in real; JUSTIFIED: in
   side := RIGHT; FIELD: in width := 0; DIGITS: in natural :=
   0);

procedure write(L: inout line; VALUE : in time; JUSTIFIED: in
   side := RIGHT; FIELD: in width := 0; UNIT: in time := ns);
```

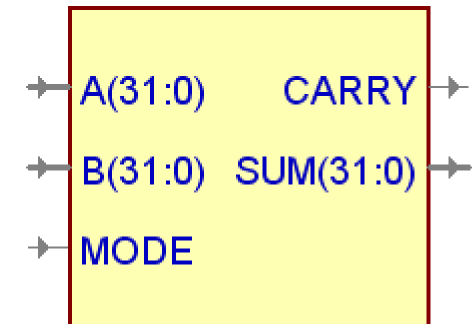
```
use std.textio.all;
```

VHDL

# Pakiet **textio** - przykład

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

entity ALU is
  port(
        MODE   : in std_ulogic;
        A, B   : in std_ulogic_vector(31 downto 0);
        SUM    : out std_ulogic_vector(31 downto 0);
        CARRY  : out std_ulogic
        );
end entity;
```

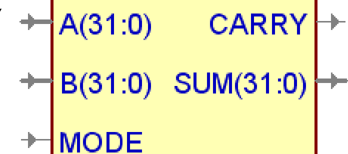


```
use std.textio.all;
```

VHDL

# Pakiet **textio** – przykład (cd.)

```
architecture BEH of ALU is
   signal A_INT, B_INT : signed(32 downto 0);
   signal SUM_INT : signed (32 downto 0);
begin
   A_INT <= resize(signed(A), 33);
   B_INT <= resize(signed(B), 33);
   SUM_INT <= A_INT + B_INT when MODE = '0' else
     A_INT - B_INT;
   SUM <= std_ulogic_vector(SUM_INT(31 downto 0));
   CARRY <= std_ulogic(SUM_INT(32));
end architecture;
```




ALU

```vhdl
use std.textio.all;
```

# Pakiet **textio** – przykład (cd.)

- należy napisać plik z wektorami testowymi, np.:

```
0   00000000000000000000000000000000 00000000000000000000000000000000 00000000000000000000000000000000 0
0   00000000000000000000000000000001 00000000000000000000000000000001 00000000000000000000000000000010 0
1   00000000000000000000000000000100 00000000000000000000000000000011 00000000000000000000000000000001 0
0   00000000000000000000000000000100 11111111111111111111111111111101 00000000000000000000000000000001 0
0   11111111111111111111111111111111 11111111111111111111111111111111 11111111111111111111111111111110 1
1   00000000000000000000000000000001 00000000000000000000000000000011 11111111111111111111111111111110 1
```

- a następnie wykorzystać program w VHDL napisany jako *testbench*:

```
use std.textio.all;
```

VHDL

# Pakiet **textio** – przykład (cd.)

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

entity TB is
end entity;

architecture BEH of TB is
   component ALU port (
              MODE  : in std_ulogic;
              A, B  : in std_ulogic_vector(31 downto 0);
              SUM   : out std_ulogic_vector(31 downto 0);
              CARRY : out std_ulogic
              );
   end component;
```

```
use std.textio.all;
```

VHDL

# Pakiet **textio** – przykład (cd.)

```
   signal MODE : std_ulogic;
   signal A, B : std_ulogic_vector(31 downto 0);
   signal SUM : std_ulogic_vector(31 downto 0);
   signal CARRY : std_ulogic;
   constant PERIOD: time := 200 NS;
begin
   ALU01 : ALU port map (MODE, A, B, SUM, CARRY);
   READCMD: process
        file CMDFILE: text;
        variable L: LINE;
        variable GOOD: boolean;
        variable MODE_REF : bit;
        variable A_REF, B_REF: bit_vector(31 downto 0);
        variable SUM_REF: bit_vector(31 downto 0);
        variable CARRY_REF : bit;
        variable LINE_COUNT : integer := 0;
```

```
use std.textio.all;
```

VHDL

# Pakiet **textio** – przykład (cd.)

```
begin
    file_open(CMDFILE, "TST_ADD.DAT", READ_MODE);
    loop
        if endfile(CMDFILE) then
            assert FALSE
            report "Koniec pliku i symulacji."
            severity NOTE;
            exit;
        end if;
        readline(CMDFILE, L);
        LINE_COUNT := LINE_COUNT + 1;
        next when L'length = 0;
        read(L ,MODE_REF, GOOD);
        assert GOOD
        report "Błąd odczytu z pliku"
        severity ERROR;
```

```
use std.textio.all;
```

VHDL

# Pakiet **textio** – przykład (cd.)

```
read(L, B_REF, GOOD);
assert GOOD
report "Błąd odczytu z pliku"
severity ERROR;
read(L, SUM_REF, GOOD);
assert GOOD
report "Błąd odczytu z pliku"
severity ERROR;
read(L,CARRY_REF, GOOD);
assert GOOD
report "Błąd odczytu z pliku"
severity ERROR;
MODE <= to_stdulogic(MODE_REF);
A <= to_stdulogicvector(A_REF);
B <= to_stdulogicvector(B_REF);
```

```
use std.textio.all;
```

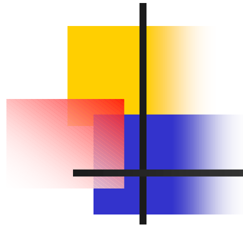
VHDL

# Pakiet **textio** – przykład (cd.)

```
                wait for PERIOD;

                assert (SUM = to_stdulogicvector(SUM_REF))
                report "Błąd - zły sygnał SUM. Wektor z linii nr:"
                                              &integer'image(LINE_COUNT)
                severity ERROR;

                assert (CARRY = to_stdulogic(CARRY_REF))
                report "Błąd - zły sygnał CARRY. Wektor z linii nr:"
                                               &integer'image(LINE_COUNT)
                severity ERROR;
        end loop;
        wait;
   end process;
end architecture;
```



# Część 22. (VHDL)

## Uwagi dotyczące syntezy

# Uwagi dotyczące syntezy

- Znajomość technologii
  - Pisząc kod VHDL, należy zawsze brać pod uwagę sposób implementacji tego kodu.
  - Nieznajomość *hardware'u* często sprawia, że projekt nie zostanie zsyntezowany lub jego jakość będzie słaba.
- Używanie standardowych konwencji zapisu
  - Aby projekt był przenośny i syntezował się w sposób przewidywalny, należy stosować sprawdzone schematy przy wprowadzaniu projektu - dotyczy to np. struktury procesów.

# Uwagi dotyczące syntezy (cd.)

- Testowanie stanu wysokiej impedancji
  - W wielu przypadkach podczas symulacji zachodzi potrzeba sprawdzenia, czy sygnał osiąga stan wysokiej impedancji **'Z'**. Osiąga się to np. poprzez instrukcje:

```
if sig = 'Z' then -- sig jest typu std_logic
  ...
end if;
```

  - W większości rzeczywistych układów nie istnieje układ sprawdzający, czy sygnał jest w stanie wysokiej impedancji, więc takie polecenie nie będzie zsyntezowane.

# Uwagi dotyczące syntezy (cd.)

- Opis strukturalny czy przypisania ciągłe?
  - Jest to szczególnie ważne dla bloków arytmetycznych, które są często bardziej zwarte w przypadku syntezy z poziomu strukturalnego – korzystając z bibliotek producenta-dostawcy.
  - Z drugiej strony używanie bibliotek producenta powoduje często trudności z przenoszeniem projektu do innych technologii i wymusza na projektancie wykonanie dodatkowych czynności.

# Uwagi dotyczące syntezy (cd.)

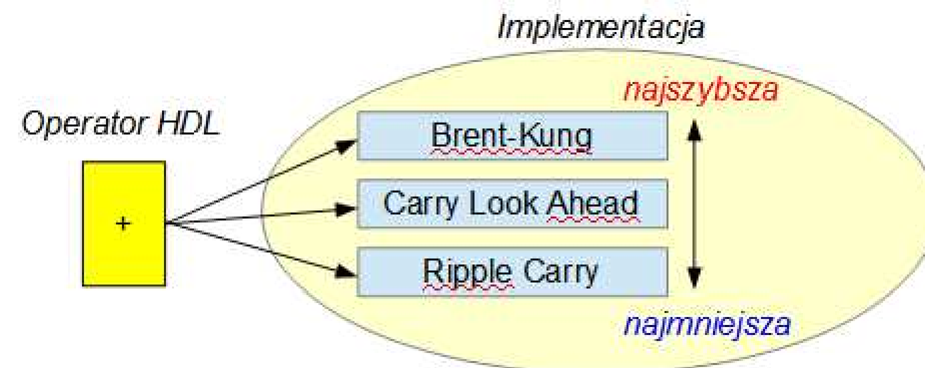
- Instrukcja **if** czy **case**?
  - Wyrażenia typu **if-then-else** są najczęściej syntezowane jako multipleksery z sygnałem sterującym zawartym w warunku instrukcji.
  - Jeżeli użyjemy zagnieżdżonych instrukcji **if-then-else**, nie prowadzi to do wykorzystania dużego multipleksera ze złożonym słowem sterującym. Zamiast tego użyty jest łańcuch małych multiplekserów mniej więcej oddający kolejność kodu VHDL.

# Uwagi dotyczące syntezy (cd.)

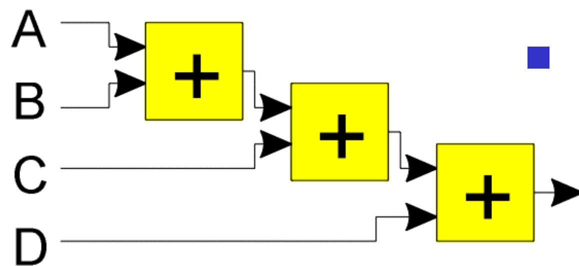
- Instrukcja **if** czy **case**? (cd.)
  - Wyrażenie **case** również prowadzi do syntezy multipleksera, jednakże w tym przypadku może to być to jeden element, niezależnie od liczby rozgałęzień.
  - Z tego powodu, jeśli chcemy mieć jeden duży multiplekser (prawdopodobnie szybki), powinniśmy użyć wyrażenia **case**.
  - Jeśli jednakże prędkość działania układu nie jest kluczowa, użycie zagnieżdżonych instrukcji **if** jest łatwiejsze do rozmieszczenia i trasowania (może dać mniejszą użytą powierzchnię).

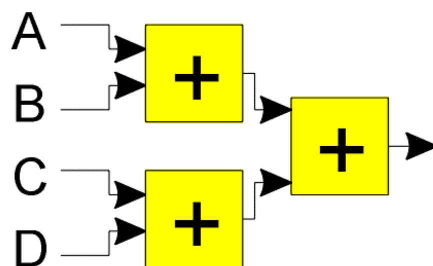
# Uwagi dotyczące syntezy (cd.)

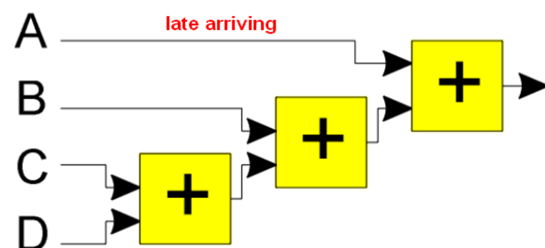
- Wybór implementacji
  - kompromis pomiędzy powierzchnią a szybkością działania
  - wybór automatyczny,
  - wybór poprzez ograniczenia (*constraints*)
  - wybór poprzez dyrektywy w kodzie HDL



# Uwagi dotyczące syntezy (cd.)



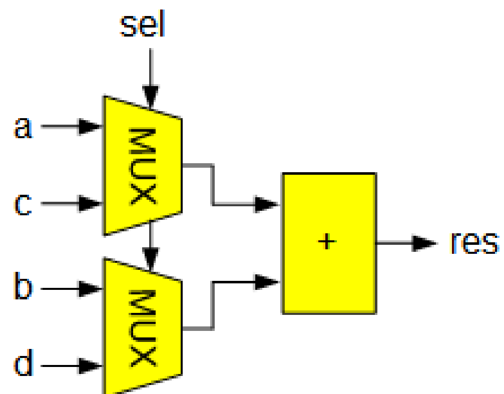



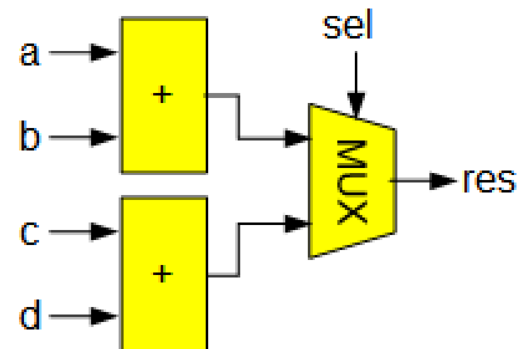

- Podział operacji
  - Większość operatorów jest dwuargumentowa i narzędzia syntezy logicznej biorą to pod uwagę.
  - Stosownie do tego wielokrotne operacje takie jak A+B+C+D prowadzą do syntezy jako sekwencja sumatorów tj. najpierw dodawane jest A do B, następnie do tej sumy C i do tego wyniku D.
  - W ten sposób stworzony jest łańcuch 3 sumatorów, a jeśli użyto by nawiasów (A+B)+(C+D) łańcuch wynosiłby 2 i układ byłby szybszy.

# Uwagi dotyczące syntezy (cd.)

- Współdzielenie zasobów

```
if (sel=0)
    res = a+b;
else
    res = c + d;
```

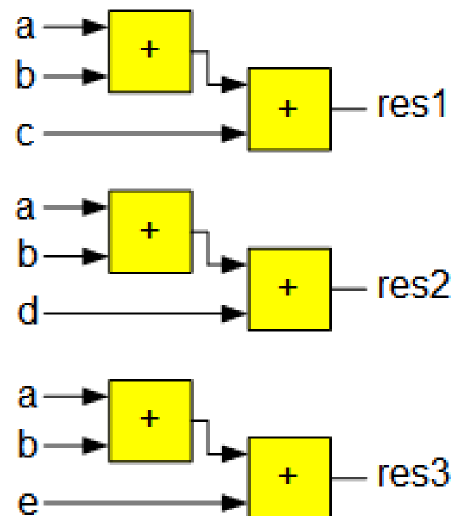


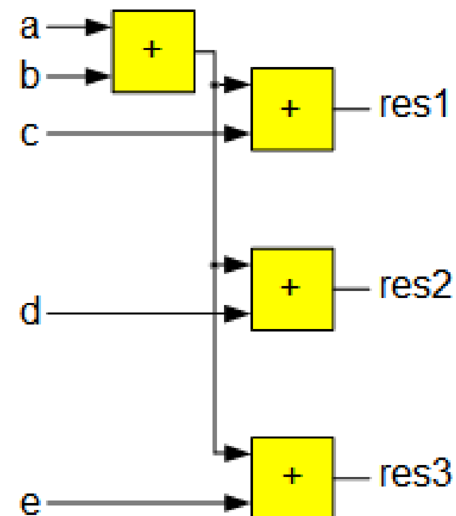



# Uwagi dotyczące syntezy (cd.)

- Współdzielenie zasobów

```
res1 <= a + b + c;
res2 <= a + b + d;
res3 <= a + b + e;
```



# Uwagi dotyczące syntezy (cd.)

- Przypisywanie wartości do wszystkich wyjść we wszystkich warunkach.
  - Stosownie do zasady - „Jeśli w procesie zegarowanym dla danego warunku nie specyfikuje się wartości sygnału oznacza to zapamiętanie jego poprzedniego stanu, czyli w zsyntezowanym układzie wstawienie przerzutnika” można łatwo spowodować wstawienie niepotrzebnych przerzutników lub nawet doprowadzić do nieprawidłowej pracy układu.

# Uwagi dotyczące syntezy (cd.)

- Uwaga na pętle
  - Pętle umożliwiają zwarty opis układu, w wyniku syntezy mogą jednak dawać długie połączenia.
  - Iteracje powodują bowiem powtarzanie w fizycznym układzie części logiki, rozbudowując ją.

# Uwagi dotyczące syntezy (cd.)

- Tryb portu **inout** czy **buffer**?
  - W przypadku konieczności odczytu wartości z portu wyjściowego, wiele niedoświadczonych osób często stosuje tryb **inout** dla tego portu, co jest w większości przypadków nieuzasadnione.
  - Gdy port jest wyjściem, tj. sygnały nie są podawane z zewnątrz do odczytu, ale potrzebna jest możliwość czytania wartości zapisanej już do portu, to należy stosować tryb **buffer**.
  - Tryb **inout** stosuje się tylko w przypadku rzeczywistego dwukierunkowego przepływu sygnałów przez port.

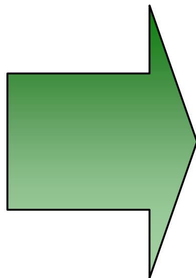
# Uwagi dotyczące syntezy (cd.)

- Tryb portu **inout** czy **buffer**? (cd.)
  - Wykorzystanie trybu **inout** może spowodować, że podczas implementacji zastosowana zostanie specjalna komórka wej/wyj, która ma bardziej złożoną strukturę niż komórka wyjściowa - dłuższe czasy propagacji większa powierzchnia krzemu.
  - Zastosowanie portu typu **buffer** może spowodować inne problemy – jeśli w bloku **entity** istnieje port typu **buffer**, a osadzony w tym module komponent posiada wyjście dołączone bezpośrednio do tego portu, to wyjście tego komponentu musi być także w trybie **buffer** (a nie **out**).

# Uwagi dotyczące syntezy (cd.)

- Tryb portu **inout** czy **buffer**? (cd.)
  - Dlatego można stosować dwa podejścia:
    - Systematycznie w definicji osadzanych komponentów pozamieniać porty trybu **out** na tryb **buffer**.

lub:



- Utworzyć dodatkowy sygnał wewnątrz modułu podstawowego, a port zadeklarować jako **out** – wtedy wszystkie operacje odczytu korzystają z dodatkowego sygnału, a przypisanie współbieżne na bieżąco przepisuje wartości tego sygnału do wyjścia pracującego w trybie **out**;

# Uwagi dotyczące syntezy (cd.)

- Wartości początkowe
  - Nadawanie wartości zmiennym już podczas ich deklaracji powoduje, że wynik implementacji może być inny niż symulacji.
  - Symulatory bez problemu nadają wartości początkowe zmiennym, takie operacje mogą być ignorowane podczas syntezy.
  - W poniższym przykładzie po syntezie (dot. układów ASIC) sygnałowi **out1** nigdy nie zostanie przypisana wartość **'0'**:

# Uwagi dotyczące syntezy (cd.)

- Wartości początkowe (cd.)

```
signal out1: bit :='0';
begin
  process (tst, start)
  begin
    if (tst = 2**6-1) then
      out1 <= '1';
    elsif (start = '1') then
      out1 <= '1';
    end if;
  end process;
...
```

# Uwagi dotyczące syntezy (cd.)

- Wartości początkowe (cd.)
  - Poprawny kod do syntezy:

```
signal out1: bit;
begin
  process (tst, start)
  begin
    if (tst = 2**6-1) then
      out1 <= '1';
    elsif (start = '1') then
      out1 <= '1';
    else
      out1 <= '0';
    end if;
  end process;
...
```

# Uwagi dotyczące syntezy (cd.)

- Wartości "*don't care*"
  - Wartość **'-'** typu **std_logic** jest jedną z 9 wartości tego typu i domyślnie nie oznacza dowolnej wartości! Dlatego dla:

```
a <= "00010";
```

porównanie:

```
a = "00---"
```

  - nigdy nie zwróci wartości TRUE po syntezie, gdyż syntezer uznaje, że wartość '-' nigdy w rzeczywistym układzie nie wystąpi.

# Uwagi dotyczące syntezy (cd.)

- Wartości "*don't care*" (cd.)
  - Aby uniknąć błędów, należy porównywać tylko odpowiednie części tablicy:

```
a(4 downto 3) = "00"
```

  - lub stosować funkcję **std_match** z pakietów **numeric_std** lub **std_arith**.
  - Funkcja ta traktuje wartości '**-**' jako wartości nieistotne, tj. porównanie '**-**' z **'0'** lub **'1'** daje zawsze wartość **TRUE**.

# Uwagi dotyczące syntezy (cd.)

- Syntezowanie niechcianych zatrzasków
  - W przypadku niepełnej instrukcji **if**, mogą powstać zatrzaski:

```
process (addr, strobe)
begin
  if strobe = '1' then
    decode_signal <= '1';
  end if;
end process;
```

  - Powyższy kod znaczy także, że gdy **strobe**='0', to **decode_signal** powinien przechowywać swą poprzednią wartość.

# Uwagi dotyczące syntezy (cd.)

- Syntezowanie niechcianych zatrzasków (cd.)
  - Poprawny kod powinien wyglądać następująco:

```
if strobe='1' then
  decode_signal <= '1';
else
  decode_signal <= '0';
end if;
```

  - Narzędzia do syntezy zazwyczaj informują o powstawaniu zatrzasków w syntezowanym układzie.

# Uwagi dotyczące syntezy (cd.)

- Różnice w syntezie przerzutników i zatrzasków
  - Aby w wyniku syntezy otrzymać przerzutnik (wyzwalany zboczem), należy na liście czułości procesu umieścić sygnał zegarowy, a w warunku **if** testować, czy nastąpiło narastające zbocze zegara:

```
-- przerzutnik wyzwalany zboczem narastającym:
process (clk)
  begin
    if (clk'event and clk = '1') then
      q <= d;
  end if;
end process;
```

# Uwagi dotyczące syntezy (cd.)

- Różnice w syntezie przerzutników i zatrzasków (cd.)
  - Aby otrzymać zatrzask (przerzutnik sterowany poziomem), należy dodać do listy czułości sygnał wejściowy i wyrzucić testowanie narastającego zbocza zegara:

```
-- zatrzask (przerzutnik sterowany poziomem):
process (clk, d)
begin
    if (clk = '1') then
            q <= d;
    end if;
end process;
```

# Uwagi dotyczące syntezy (cd.)

- Syntezowanie przerzutników
  - Aby powstał element pamiętający (w tym wypadku przerzutnik **D**), należy wykorzystać instrukcję **if**, ale bez części **else**:

```
if (clk'event and clk = '1') then
  q <= d;
end if;
```

  - co oznacza, że w przypadku, gdy nie jest spełniony warunek występujący po **if**, układ ma pamiętać poprzednią wartość wyjścia **q**.

# Uwagi dotyczące syntezy (cd.)

- Syntezowanie przerzutników (cd.)
  - Można także wyraźnie opisać, co ma się dziać w przypadku niespełnienia warunku.
  - Równoważny opis wykorzystujący część **else** wygląda następująco:

```
-- kod może się nie syntezować!!!
if (clk'event and clk = '1') then
  q <= d;
else
  q <= q;
end if;
```

# Uwagi dotyczące syntezy (cd.)

- Syntezowanie przerzutników (cd.)
  - Poniższy kod jest poprawny pod względem składniowym i symulacja przebiegnie zgodnie z oczekiwaniami, jednak syntezer może uznać, że wykonanie części **else** może być niejednoznaczne i wygenerować błąd.

```
-- kod niesyntezowalny!!!
if (clk'event and clk = '1') then
  q <= d;
else
  q <= a; -- = not (clk'event and clk = '1') ???
end if;
```

# Uwagi dotyczące syntezy (cd.)

- Synchroniczny i asynchroniczny **reset**
  - Zazwyczaj układy programowalne FPGA dysponują przerzutnikami z asynchronicznym resetem.
  - W przypadku syntezy układów z resetem synchronicznym, może się okazać, że zużyte zostaną dodatkowe zasoby układu:

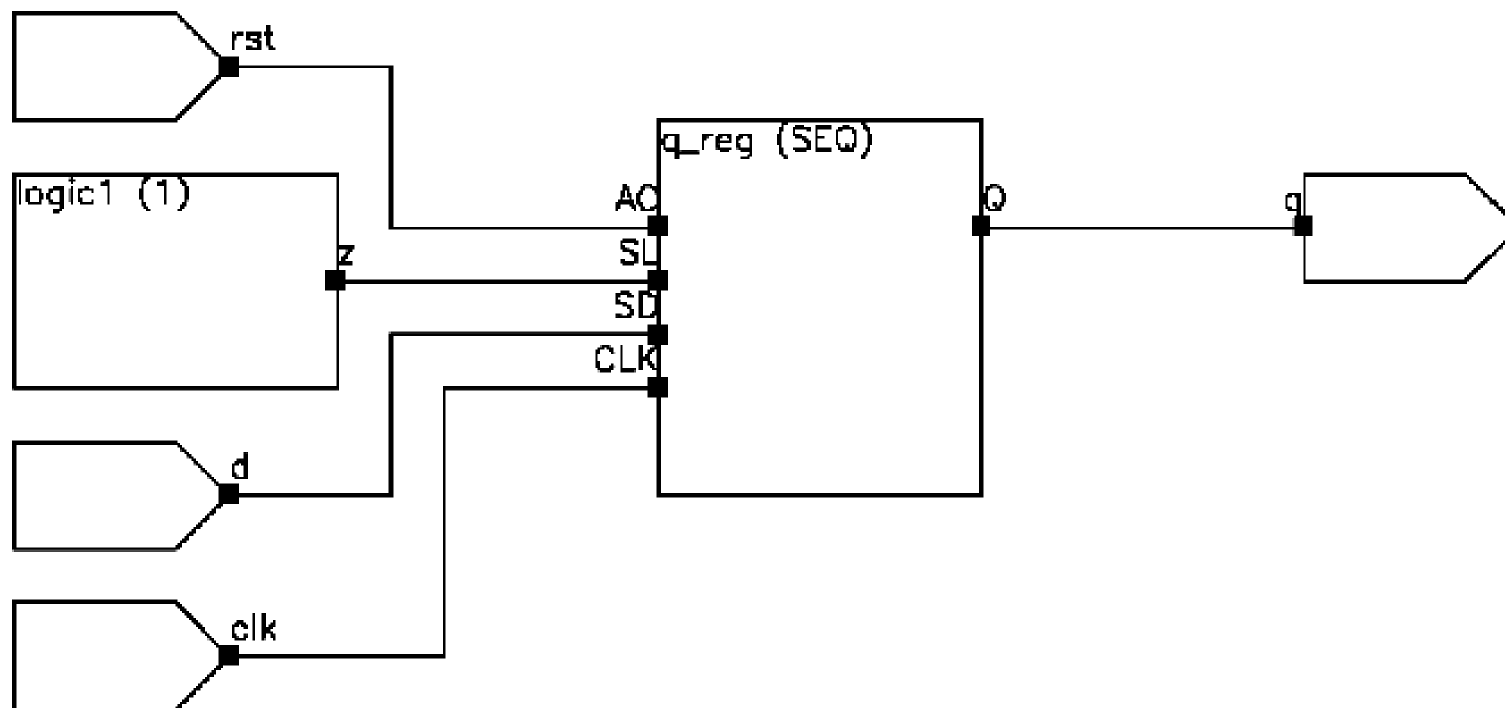
# Uwagi dotyczące syntezy (cd.)

- Synchroniczny i asynchroniczny **reset** (cd.)

```
-- Reset asynchroniczny:
process (clk, rst)
   begin
        if (rst = '1') then
                q <= '0';
        elsif (clk'event and clk = '1') then
                q <= d;
        end if;
   end process;
```

# Uwagi dotyczące syntezy (cd.)

- asynchroniczny **reset**:



# Uwagi dotyczące syntezy (cd.)

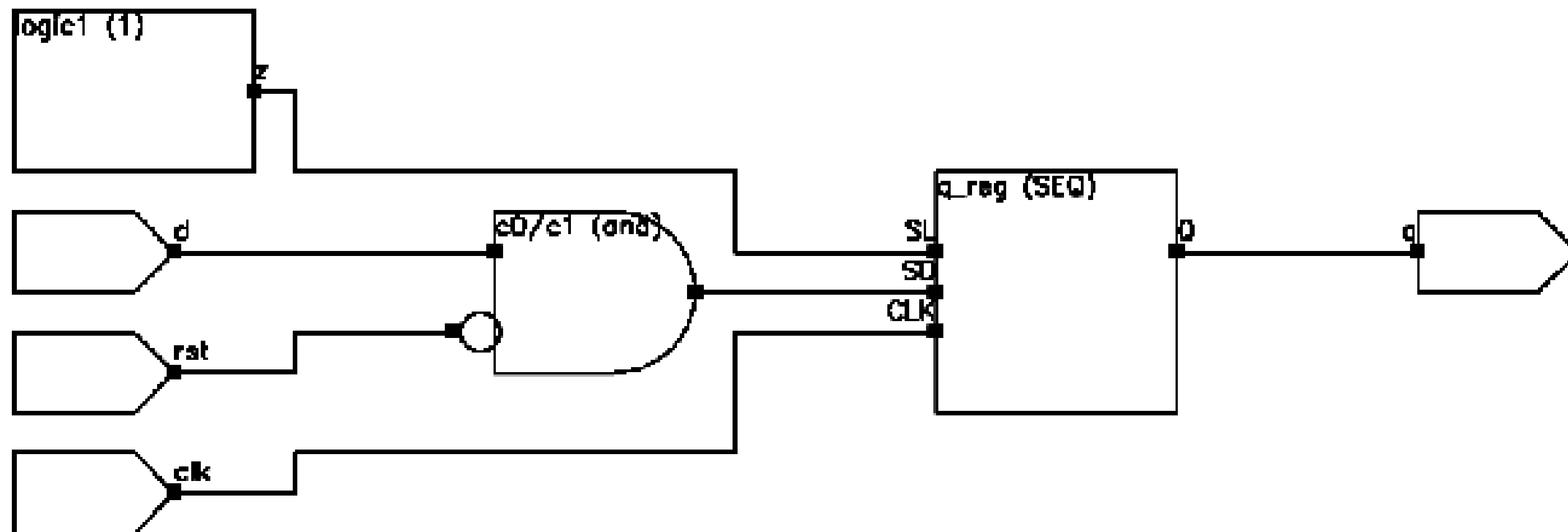
- Synchroniczny i asynchroniczny **reset** (cd.)

```
-- Reset synchroniczny:
process (clk, rst)
   begin
        if (clk'event and clk = '1') then
               if (rst = '1') then
                      q <= '0';
               else
                      q <= d;

               end if;
        end if;
   end process;
```

# Uwagi dotyczące syntezy (cd.)

- Synchroniczny **reset**:



# Uwagi dotyczące syntezy (cd.)

- Zwiększenie siły sygnału
  - Niektóre narzędzia syntezy rozpoznają wielokrotne podstawienia sygnału i umożliwiają zwiększenie siły sygnału, pod warunkiem, że instrukcje sterujące są identyczne:

```
architecture beh of test is
begin
  Y <= A and B;
  Y <= A and B;
end architecture;
```

  - Kod może spowodować powstanie dwóch buforów generujących sygnał **Y**.
  - Zaleca się jednak stosowanie w tym celu specjalnych dyrektyw programu syntezującego lub osadzanie buforów z biblioteki jako komponenty.

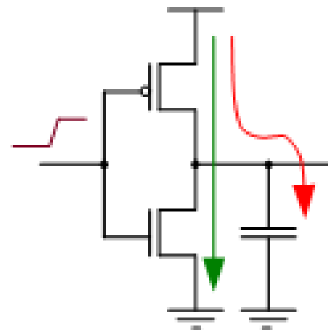
# Uwagi dotyczące strat mocy

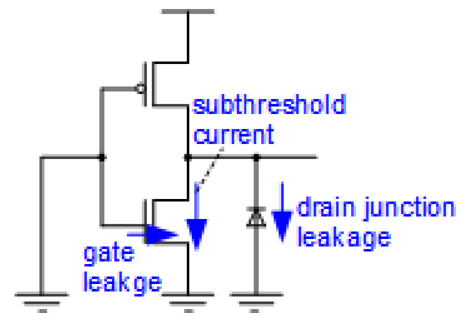
- Straty mocy w układach CMOS

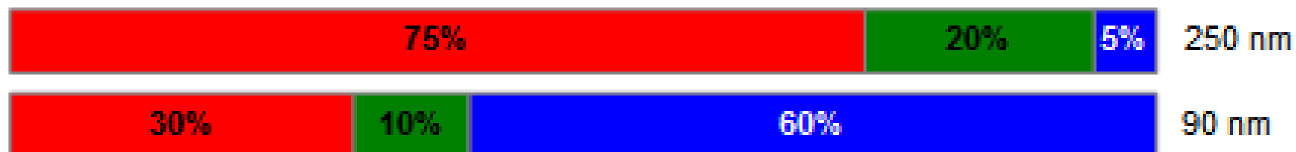
$$P_{tot} = C_L V_{DD}^2 f + t_{sc} V_{DD} I_{peak} f + V_{DD} I_{leakage}$$

*dynamic* *short circuit* *leakage*

$$I_{leakage} = I_{sub} + I_{gate\ oxide} + I_{junction}$$





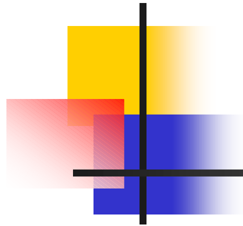



# Uwagi dotyczące strat mocy (cd.)

- Działania w celu:
  - zmniejszenia *leakage*
    - Biblioteki o wyższym Vth --> większe opóźnienia!
  - zmniejszenia mocy dynamicznej
    - Optymalizacja -> większe opóźnienia
    - *Clock gating* -> problemy z weryfikacją i P&R
    - *Operand isolation* -> problemy z P&R oraz większe opóźnienia
  - zmniejszenia mocy pobieranej przez caly blok/podukład
    - Obniżenie napięcia zasilania -> większe opóźnienia, komplikacja architektury
    - Wyłączenie bloku -> komplikacja architektury

# Techniki obniżania zużycia mocy

| Technika redukcji mocy | Oszczędności mocy | Wpływ na opóźnienia | Wpływ na zwiększenie powierzchni | Wpływ na sposób projektowania | Wpływ na sposób weryfikacji |
|---|---|---|---|---|---|
| **Optymalizacja powierzchni** | **małe** | brak | - | mały | brak |
| **Multi-Vt** | **średnie** | mały | mały | mały | brak |
| **Clock gating** | **średnie** | mały | mały | mały | mały |
| **Multi-supply voltage** | **duże** | średni | średni | średni | średni |
| **Power shot-off** | **duże** | mały | średni | duży | duży |
| **Dynamic and Adaptive Voltage Frequency Scaling** | **duże** | średni | średni | duży | duży |



# KONIEC