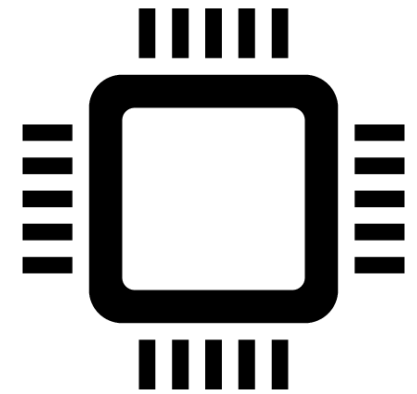


# Programowanie struktur cyfrowych



Modelowanie w VHDL.  
Wprowadzenie

dr Aleksander Lamża

Uniwersytet J. Kochanowskiego w Kielcach  
Uniwersytet Śląski w Katowicach

[aleksander.lamza@us.edu.pl](mailto:aleksander.lamza@us.edu.pl)

# Proces modelowania układu cyfrowego

Modelowanie w VHDL (a także każdym innym języku HDL) różni się od tradycyjnego procesu budowania prototypu. Przede wszystkim nie jest wymagane tworzenie fizycznego urządzenia, więc sam proces budowania, jak i jego poprawiania są dużo mniej kosztowne (czasowo i finansowo).

Układ modeluje się za pomocą języka (czyli w formie tekstu) w postaci strukturalnej, behawioralnej (funkcjonalnej) lub mieszanej. Tak przygotowany model można przetestować w procesie symulacji. Jeżeli są wymagane poprawki, naprowadza się je w kodzie i znów przeprowadza symulację. Cały proces powtarza się aż do uzyskania zadowalających wyników.

Dopiero później dochodzi do realizacji sprzętowej, która jest czasowo i kosztochłonna.

# Struktura modelu: **entity**

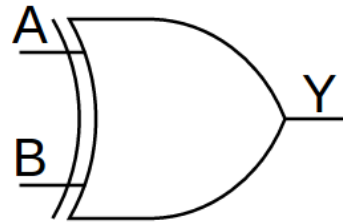
Podstawowym elementem modelowania jest jednostka projektowa (**entity**). Reprezentuje ona pewien element (układ) cyfrowy. Pierwszą częścią jest deklaracja jednostki zawierająca m.in. definicję portów wejścia-wyjścia i parametrów. W notacji BNF wygląda to tak:

```
deklaracja_jednostki := entity id_jednostki is  
                        [port (lista_portów);]  
                        [generic (lista_parametrów);]  
                        [inne_deklaracje]  
                        [begin  
                          instrukcje]  
                        end [entity] [nazwa];
```

Ten element jest nazywany też **interfejsem**, ponieważ definiujemy w nim m.in. sposób podłączenia jednostki z innymi elementami.

# Przykład: model bramki XOR

Najlepiej będzie to omówić na prostym przykładzie, np. bramki XOR.



Bramka ma **dwa wejścia** i **jedno wyjście** – to jest jej interfejs.  
W deklaracji jednostki projektowej wyglądałoby to tak:

```
entity XOR_gate is
  port (
    a    : in bit;
    b    : in bit;
    y    : out bit
  );
end
```

# Przykład: model bramki XOR

```
entity XOR_gate is
  port (
    a    : in bit;
    b    : in bit;
    y    : out bit
  );
end;
```

Widać tu definicję linii wejściowych (**in**) i linii wyjściowej (**out**).  
Typ tych linii jest określony na **bit**, czyli wartość **0** albo **1**.

Zwróćcie uwagę, że definicje portu są **oddzielane średnikami**,  
a więc po ostatniej definicji średnika nie ma!

Poprawność składniową sprawdzimy za pomocą polecenia `-s` (syntax)  
narzędzia GHDL:

A terminal window with a red title bar containing the text "/bin/bash 77x8". The terminal shows a user prompt "olek@lap" followed by the command "ghdl -s xor\_gate.vhdl" and a subsequent prompt "olek@lap".

```
olek@lap ~/Projekty/fpga/5 $ ghdl -s xor_gate.vhdl
olek@lap ~/Projekty/fpga/5 $
```

W tym przypadku brak informacji jest dobrą informacją – błędów nie ma.

# Przykład: model bramki XOR

Dla tych, którzy nie wierzą, że to działa, wprowadzę błąd – średnik po definicji ostatniego portu:

```
entity XOR_gate is
  port (
    a    : in bit;
    b    : in bit;
    y    : out bit;
  );
end;
```

Wynik działania polecenia `ghdl -s`:

A terminal window titled "/bin/bash" with a red header bar. The prompt is "olek@lap ~/Projekty/fpga/5". The command "ghdl -s xor\_gate.vhdl" has been executed. The output shows a syntax error: "xor\_gate.vhdl:5:33: extra ';' at end of interface list". The prompt is now "olek@lap ~/Projekty/fpga/5 \$".

```
x - /bin/bash
/bin/bash 77x8
olek@lap ~/Projekty/fpga/5 $ ghdl -s xor_gate.vhdl
xor_gate.vhdl:5:33: extra ';' at end of interface list
olek@lap ~/Projekty/fpga/5 $
```

No dobra, teraz widać, że sprawdzanie działa.

# Struktura modelu: **architecture**

Zdefiniowaliśmy interfejs bramki, ale nigdzie nie napisaliśmy, jak ma **działać**.

Za to odpowiada tzw. architektura (**architecture**).

Architektura jest przypisana do jednostki (tak naprawdę, do jednostki można przypisać wiele architektur, ale o tym za moment).

Składnia architektury w notacji BNF:

```
architektura := architecture id of nazwa_jednostki is  
               część_deklaracyjna  
               [begin  
                 instrukcje;  
               end [architecture] [nazwa];
```

Zanim przejdziemy do definiowania struktury lub zachowania jednostki, kilka uwag na temat tzw. dobrych praktyk dotyczących definiowania jednostek i architektur.

# Dobre praktyki definiowania jednostek i architektur

Modelując układy w VHDL-u najlepiej przestrzegać kilku zasad. Niektóre są silniejsze, niektóre słabsze, ale na pewno warto rozważyć ich stosowanie.

**Twórz tylko jedną architekturę dla jednostki.** Chociaż można ich zdefiniować wiele, nie sprzyja to przejrzystości projektu.

Umieszczaj deklarację architektury **w tym samym pliku**, co deklaracja jednostki.

Używaj tylko **jednego pliku** dla pary jednostka-architektura.

**Plik z jednostką nazywaj tak samo, jak jednostkę.** Jeżeli jednostka to *xor\_gate*, plik powinien się nazywać *xor\_gate.vhdl*.

**Nie używaj polskich znaków** w kodzie (nawet w komentarzach!).



# Przykład: architektura behawioralna modelu bramki XOR

Wróćmy do naszego modelu bramki.

Jak wiadomo, funkcję XOR można zrealizować za pomocą funkcji negacji, iloczynu i sumy:

$$Y = A \cdot \bar{B} + \bar{A} \cdot B$$

W VHDL-u wygląda to tak:

```
architecture xor_gate_arch of xor_gate is  
begin  
    y <= (a and (not b)) or ((not a) and b);  
end;
```

Jak widać, dochodzi tu do przekazania ( $\leq$ ) wartości logicznej do wyjścia  $y$ . Wartość jest obliczana za pomocą operatorów `and`, `or` i `not`.

W VHDL-u mamy do dyspozycji sporo różnych operatorów działających zarówno na pojedynczych bitach, jak i wektorach bitów. Co ciekawe, jest wśród nich operator `xor`, który...

Jednym słowem, definiowanie modelu bramki XOR jest zbędne :)

# Przykład: architektura behawioralna modelu bramki XOR

Pozostaniemy jednak jeszcze na chwilę przy tym przykładzie.

Czasem wygodniej byłoby odwzorować zachowanie układu kombinacyjnego na podstawie tabeli prawdy, a nie funkcji.

Tabela prawdy dla modelowanego elementu przedstawia się następująco:

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Spróbujemy odwzorować tę postać zależności stanu wyjścia od stanu wejść w architekturze. Przy okazji poznamy kolejne elementy i konstrukcje języka VHDL.

# Przykład: architektura behawioralna modelu bramki XOR

Skorzystamy z instrukcji **with-select**, czyli selektywnego przypisania sygnału:

```
with wyrażenie  
select cel_przypisania <= wartość_1 when wybór_1,  
                                wartość_2 when wybór_2,  
                                ...  
                                wartość_n when wybór_n;
```

Zaraz zobaczymy tę instrukcję w akcji, ale najpierw przygotujemy wejścia naszego modelu. W tej chwili są to dwa niezależne sygnały (a i b). Na potrzeby instrukcji **with-select** lepiej będzie je połączyć w jeden **wektor** (tablicę).

Najpierw w części deklaracyjnej architektury zadeklarujemy wewnętrzny sygnał będący dwuelementowym wektorem:

```
signal inputs: bit_vector(0 to 1);
```

Następnie, w ciele architektury, ustawimy oba elementy wektora:

```
inputs(0) <= a;  
inputs(1) <= b;
```

# Przykład: architektura behawioralna modelu bramki XOR

Zgodnie z tabelą prawdy funkcji XOR instrukcja `with-select` przyjmie następującą postać:

```
with inputs
select y <= '0' when "00",
           '1' when "01",
           '1' when "10",
           '0' when "11";
```

Cała architektura wygląda więc tak:

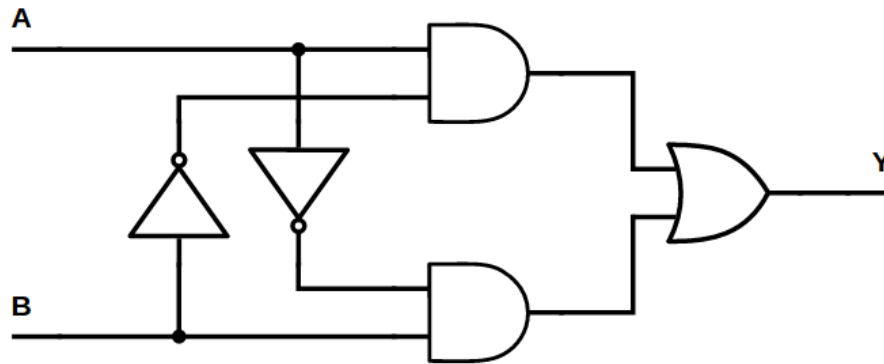
```
architecture xor_gate_arch of xor_gate is
  signal inputs: bit_vector(0 to 1);
begin
  inputs(0) <= a;
  inputs(1) <= b;

  with inputs
  select y <= '0' when "00",
           '1' when "01",
           '1' when "10",
           '0' when "11";
end;
```

## Przykład: architektura strukturalna modelu bramki XOR

Do tej pory traktowaliśmy modelowany układ z perspektywy jego **działania**. Teraz spróbujemy go zamodelować, skupiając się na jego **strukturze**.

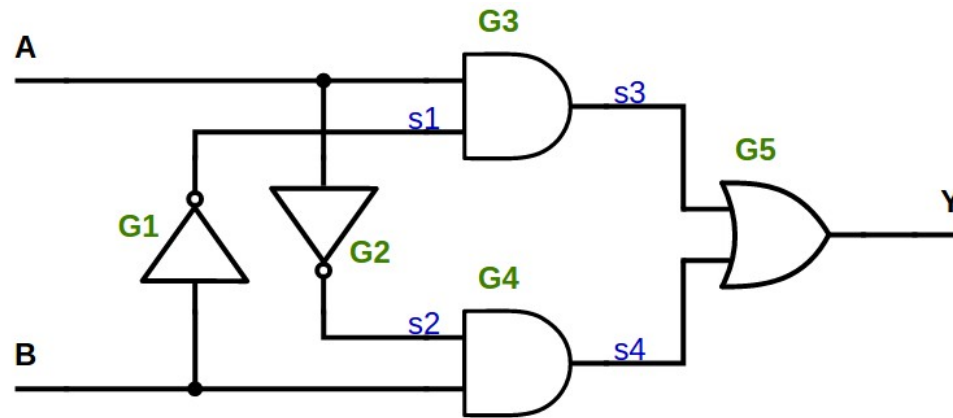
Funkcja XOR w postaci  $Y = A \cdot \bar{B} + \bar{A} \cdot B$  zrealizowana na bramkach wygląda tak:



Spróbujemy odwzorować ten układ połączeń w VHDL-u.

# Przykład: architektura strukturalna modelu bramki XOR

Zaczniemy od oznaczenia wszystkich wewnętrznych połączeń (sygnałów) i bramek (komponentów).



W części deklaracyjnej architektury musimy zadeklarować te sygnały:

```
signal s1, s2, s3, s4: bit;
```

Musimy tu również umieścić definicje komponentów (czyli bramek). W tym przypadku mamy trzy komponenty: bramkę NOT, AND i OR.

# Przykład: architektura strukturalna modelu bramki XOR

Oto definicje komponentów:

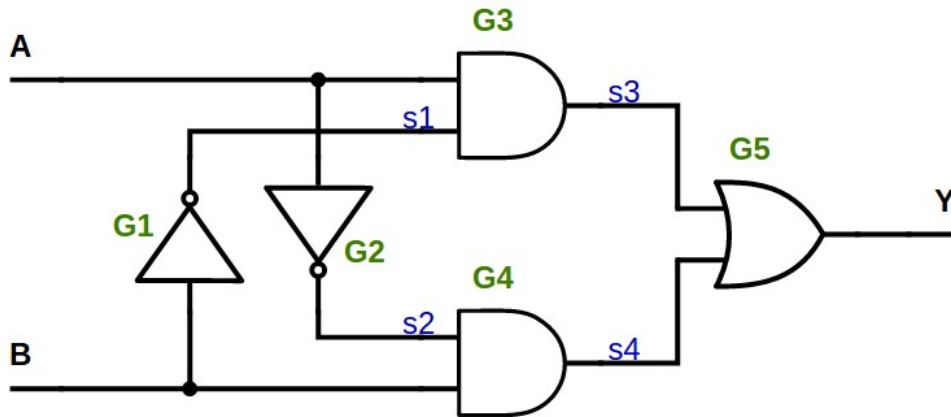
```
component not_gate
  port (
    a: in bit;
    y: out bit
  );
end component;

component and_gate
  port (
    a, b: in bit;
    y: out bit
  );
end component;

component or_gate
  port (
    a, b: in bit;
    y: out bit
  );
end component;
```

# Przykład: architektura strukturalna modelu bramki XOR

A teraz najciekawsze. W ciele architektury opiszemy elementy i połączenia między nimi.



```
begin
```

```
g1: not_gate  
    port map(b, s1);
```

```
g2: not_gate  
    port map(a, s2);
```

```
g3: and_gate  
    port map(a, s1, s3);
```

```
g4: and_gate  
    port map(b, s2, s4);
```

```
g5: or_gate  
    port map(s3, s4, y);
```

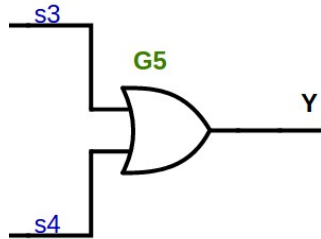
```
end architecture;
```



# Przykład: architektura strukturalna modelu bramki XOR

Wyjaśnienia wymaga instrukcja `port map`. Służy ona do połączenia wejść i wyjść komponentów z sygnałami.

Przypatrzmy się bramce G5. Jest to komponent o dwóch wejściach (a i b) oraz wyjściu y:



Do wejścia a podłączamy sygnał s3, a do wejścia b sygnał s4. Wyjście łączymy z y:

Przypisanie do poszczególnych wejść można pominąć i zastosować skrócony zapis:

Ostatecznie wystąpienie bramki G5 ma w VHDL-u następującą postać:

```
component or_gate
    port (
        a, b: in bit;
        y: out bit
    );
end component;
```

```
port map(
    a => s3,
    b => s4,
    y => y
);
```

```
port map(s3, s4, y);
```

```
g5: or_gate
    port map(s3, s4, y);
```

# Przykład: architektura strukturalna modelu bramki XOR

Opis układu wygląda więc tak:

```
-- Deklaracja jednostki
entity xor_gate is
    port (
        a: in bit;
        b: in bit;
        y: out bit
    );
end entity;
```

Czy na pewno wszystko już mamy?

A gdzie definiujemy sposób działania  
bramek NOT, AND i OR?

Mamy tylko definicje interfejsów  
komponentów, ale brak ich jednostek  
i architektury.

```
-- Deklaracja architektury
architecture xor_gate_arch of xor_gate is

    signal s1, s2, s3, s4: bit;

    component not_gate
        port (
            a: in bit;
            y: out bit
        );
    end component;

    component and_gate
        port (
            a, b: in bit;
            y: out bit
        );
    end component;

    component or_gate
        port (
            a, b: in bit;
            y: out bit
        );
    end component;

    begin
        u1: not_gate port map(b, s1);
        u2: not_gate port map(a, s2);
        u3: and_gate port map(a, s1, s3);
        u4: and_gate port map(b, s2, s4);
        u5: or_gate port map(s3, s4, y);
    end architecture;
```

# Przykład: architektura strukturalna modelu bramki XOR

Deklaracje bramek wyglądają następująco:

```
entity and_gate is
    port (
        a: in bit;
        b: in bit;
        y: out bit
    );
end;

architecture and_gate_arch of and_gate is
begin
    y <= a and b;
end;
```

```
entity not_gate is
    port (
        a: in bit;
        y: out bit
    );
end;

architecture not_gate_arch of not_gate is
begin
    y <= not a;
end;
```

```
entity or_gate is
    port (
        a: in bit;
        b: in bit;
        y: out bit
    );
end;

architecture or_gate_arch of or_gate is
begin
    y <= a or b;
end;
```

Pamiętajcie, że deklaracje te, zgodnie z „dobrymi praktykami” powinny się znaleźć w osobnych plikach nazwanych tak jak jednostki, czyli *and\_gate.vhdl*, *or\_gate.vhdl* i *not\_gate.vhdl*.

# Symulowanie układu

Na razie skupiliśmy się na tworzeniu modelu układu. Do tej pory ani razu nie sprawdziliśmy, czy to działa. W praktyce trudno w ten sposób pracować... Zaczniemy od przygotowania jednostki testowej:

```
entity test is  
  -- tu nic nie trzeba umieszczać  
end;
```

Potrzebna też będzie architektura wraz z komponentem, który chcemy testować, czyli naszą bramkę XOR (`xor_gate`):

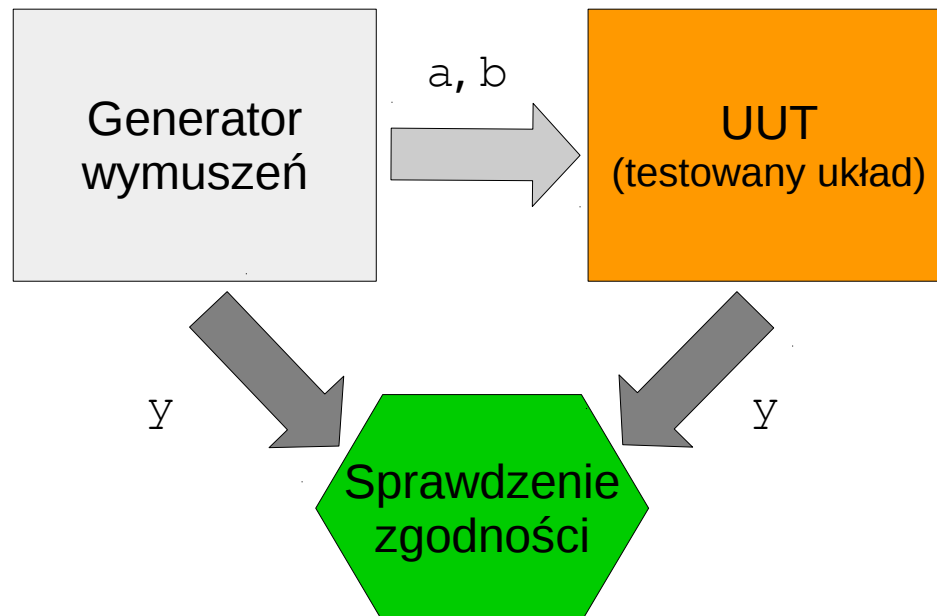
```
architecture test_arch of test is  
  
  -- testowany komponent  
  component xor_gate  
    port(  
      a: in bit;  
      b: in bit;  
      y: out bit  
    );  
  end component;  
  
  signal a, b, y: bit;  
  
begin  
  -- tu umieścimy kod testujący  
end architecture;
```

# Symulowanie układu

Teraz, w celu architektury, tworzymy egzemplarz komponentu i „podpinamy” go pod sygnały  $a$ ,  $b$  i  $y$ :

```
uut: xor_gate port map(a, b, y);
```

Testowanie będzie polegało na zasymulowaniu działania układu. Podamy po kolei wszystkie możliwe stany na wejścia i sprawdzimy, czy na wyjściu jest stan, którego oczekujemy.



# Symulowanie układu

Ponieważ symulacja jest procesem sekwencyjnym (stany nie są podawane równocześnie, tylko jeden po drugim), instrukcje testujące musimy umieścić w bloku `process`:

```
process
begin
  -- tu znajdzie się kod testujący
end process;
```

Jeden przypadek testowy, np. dla  $a=0$  i  $b=0$  wygląda następująco:

```
a <= '0';
b <= '0';
wait for 1 ns;
assert (y='0') report "FAIL" severity error
```

O ile dwa pierwsze wiersze nie wymagają wyjaśnienia, kolejne już raczej tak...

```
wait for 1 ns;
```

**Instrukcja oczekiwania** `wait` powoduje wstrzymanie wykonywania procesu. W tym przypadku czekamy 1 ns, aż ustali się stan wyjścia testowanego układu.

# Symulowanie układu

```
assert (y='0') report "FAIL" severity error
```

**Instrukcja założenia `assert`** sprawdza prawdziwość warunku i zgłasza błąd, jeżeli warunek nie jest spełniony.

Po opcjonalnym słowie kluczowym `report` można umieścić komunikat, który pojawi się w oknie symulatora.

Jeżeli w instrukcji złożenia umieścimy słowo kluczowe `severity`, możemy określić wagę zgłoszonego komunikatu. Możliwe wartości to:

- `note` (notatka),
- `warning` (ostrzeżenie),
- `failure` (usterka),
- `error` (błąd).

# Symulowanie układu

Pełny kod testujący wygląda tak:

Zwróćcie uwagę na instrukcję `wait` umieszczoną na końcu procesu. W takiej postaci spowoduje ona zawieszenie procesu do końca symulacji.

```
process
begin
  a <= '0';
  b <= '0';
  wait for 1 ns;
  assert (y='0') report "FAIL 00" severity error;

  a <= '0';
  b <= '1';
  wait for 1 ns;
  assert (y='1') report "FAIL 01" severity error;

  a <= '1';
  b <= '0';
  wait for 1 ns;
  assert (y='1') report "FAIL 10" severity error;

  a <= '1';
  b <= '1';
  wait for 1 ns;
  assert (y='0') report "FAIL 11" severity error;

  -- Ustawienie stanu początkowego
  a <= '0';
  b <= '0';
  assert false report "Test OK." severity note;

  wait;
end process;
```



# Uruchomienie symulacji w EDA playground

Uruchomimy symulację w narzędziu EDA playground.  
Najpierw musimy je odpowiednio skonfigurować:



The screenshot shows the EDA Playground configuration interface. It has a blue header with the 'EDA playground' logo. Below the header, there are two main sections: 'Languages & Libraries' and 'Tools & Simulators'. In the 'Languages & Libraries' section, there is a 'Testbench + Design' dropdown menu set to 'VHDL', a 'Libraries' dropdown menu with options 'None', 'OVL 2.8.1', and 'OSVVM 2014.01', and a 'Top entity' text input field containing 'test'. In the 'Tools & Simulators' section, there is a dropdown menu set to 'Aldec Riviera Pro 2015.06', a 'Compile & Run Options' section with 'Compile Options' and 'Run Options' buttons, a 'Run Time' input field set to '10 ms', and two checkboxes: 'Open EPWave after run' and 'Download files after run', both of which are unchecked.

← Przede wszystkim musimy ustawić język: **VHDL**.

← Główna jednostka to **test**.

← Silnikiem uruchamiającym kod będzie **Aldec Riviera Pro**.

I tyle wystarczy. Uruchamiamy!

# Uruchomienie symulacji w EDA playground

The screenshot displays the EDA Playground web interface. On the left, a sidebar contains sections for 'Languages & Libraries' (with a dropdown for 'VHDL'), 'Tools & Simulators' (with a dropdown for 'Aldec Riviera Pro 2015.06'), and 'Examples'. The main area is split into two panels: 'testbench.vhd' and 'design.vhd'. The 'testbench.vhd' panel shows a VHDL testbench for an XOR gate. The 'design.vhd' panel shows a VHDL design for an AND gate. Below the code panels is a 'Log' panel with a 'Log' button and a 'Share' button. The log text includes messages from the ELAB2, KERNEL, and VSIM, indicating a successful simulation. A red circle highlights the message 'EXECUTION:: NOTE : Test OK.' in the log. A yellow text box is overlaid on the log, containing two paragraphs of Polish text.

EDA playground

Run Save Copy Collaborate beta Forum ?

Aleksander Lamz

Languages & Libraries

Testbench + Design

VHDL

Libraries

None

OV 2.8.1

OSVVM 2014.01

Top entity

test

Tools & Simulators

Aldec Riviera Pro 2015.06

Compile & Run Options

Compile Options

Run Options

Run Time: 10 ms

Open EPWave after run

Download files after run

Examples

VHDL

Verilog/SystemVerilog

UVM

EasierUVM

SVAUnit

testbench.vhd

VHDL Testbench

```
1
2 entity test is
3 end;
4
5 architecture test_arch of test is
6
7     -- testowany komponent
8     component xor_gate
9     port(
10         a: in bit;
11         b: in bit;
12         y: out bit
13     );
14 end component;
15
16 signal a,
17
18 begin
```

design.vhd

VHDL Design

```
1 -- Bramki
2 -- AND
3 entity and_gate is
4     port(
5         a: in bit;
6         b: in bit;
7         y: out bit
8     );
9 end;
10
11 architecture and_gate_arch of and_gate
12 is
```

Log

Share

# ELAB2: Elaboration done.

# KERNEL: Warning: ...

# KERNEL: Warning: ...

# KERNEL: Kernel ...

# Allocation: Simulator allocated 7004 kB (elbread=1024 elab2=5830 kernel=150 sdf=0)

# KERNEL: ASDB file was created in location /home/runner/dataset.asdb

run -all,

EXECUTION:: NOTE : Test OK.

# EXECUTION:: Time: 4 ns, Iteration: 0, Instance: /test, Process: line\_24.

# KERNEL: Simulation has finished. There are no more test vectors to simulate.

exit

# VSIM: Simulation has finished.

Done

W panelu Log pojawi się mnóstwo komunikatów. Większość z nich to standardowe logi kompilatora i innych narzędzi.

Nas interesuje to, co sami wyświetlamy, czyli komunikaty asercji. I jest: „Test OK”.

# Podsumowanie

W tej prezentacji pojawiło się całkiem sporo nowych zagadnień. Trzeba to z pewnością na spokojnie przemyśleć i „przetrawić”, a w trawieniu pomoże zadanie :)

Zadanie polega na zasymulowaniu w EDA playground dwóch omówionych wcześniej wersji bramki: (1) z wartością obliczaną z funkcji i (2) z instrukcją `with-switch`.

Rozwiązaniem są zrzuty ekranu z wykonanych symulacji (2 sztuki).