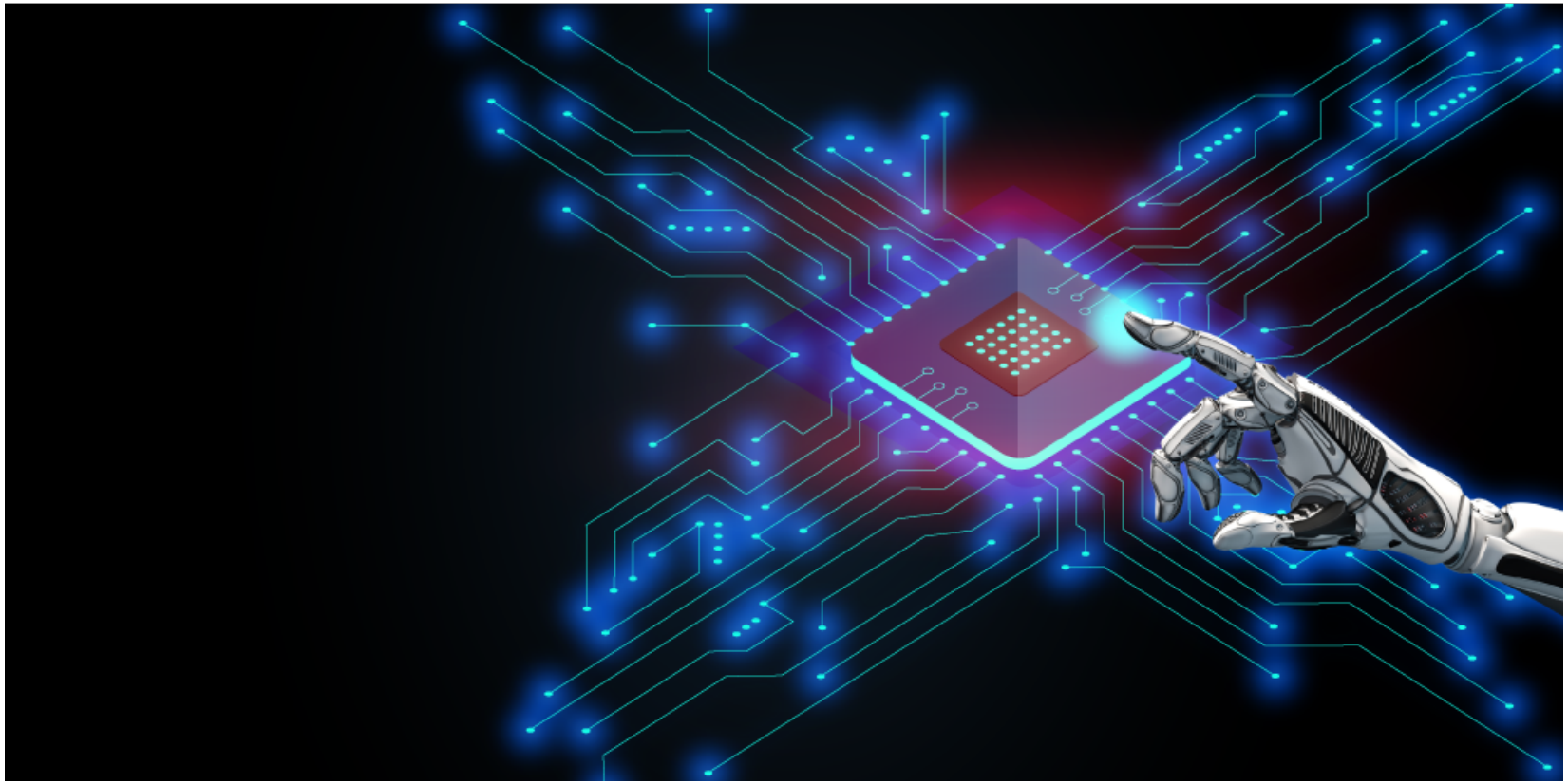
Verilog HDL

Opis podstawowych konstrukcji języka i składni



Czym są języki HDL i dlaczego powstały

Złożoność układów cyfrowych zgodnie z prawem Moore'a, podwaja się co około 18 miesięcy. Liczba tranzystorów znajdujących się w układzie scalonym sięga już dziesiątek, a nawet setek milionów. Przy takiej złożoności układów scalonych stosowane dotychczas narzędzia projektowe przestają być użyteczne. Operują one bowiem na zbyt niskim poziomie abstrakcji (bramki logiczne), co uniemożliwia projektowanie złożonych układów w rozsądnym czasie. Następstwem dynamicznego rozwoju techniki cyfrowej i rosnących trudności w projektowaniu coraz bardziej złożonych układów, jest powstanie języków opisu sprzętu (HDL – Hardware Description Language). Jest to zupełnie nowe podejście do projektowania – języki opisu sprzętu pozwalają na opis zachowania, a nie struktury układu (choć nie wykluczają także opisu struktury, jeśli zachodzi taka potrzeba).Język opisu sprzętu pozwala nam stworzyć abstrakcyjny model rzeczywistego urządzenia elektronicznego. Bardzo ważne jest właśnie pojęcie abstrakcji w opisywaniu działania urządzenia, a dokładniej poziom abstrakcji. Możemy wyróżnić trzy takie poziomy:

- poziom połączenia,
- poziom bramek,
- poziom transferu rejestrowego (RTL – register transfer level).

Historia języka Verilog

Język Verilog HDL powstał na przełomie 1983/1984 roku jako zastrzeżony produkt w firmie Gateway Design Automation. W roku 1985 ukazała się pierwsza wersja handlowa Veriloga. W roku 1986 powstał symulator używający tego języka – Verilog-XL. Rok 1988 przyniósł pierwsze narzędzie do syntezy sprzętu na podstawie opisu w Verilogu, wyprodukowane przez firmę Synopsys. Sukces języka pozwolił na szybki rozwój firmy, która została przejęta przez Cadence Design Systems. Od roku 1990 Verilog stał się językiem otwartym , nad którym opiekę sprawuje niezależna grupa Open Verilog International (OVI). Po powstaniu OVI szereg mniejszych firm rozpoczęło pracę nad własnymi symulatorami Veriloga. W rezultacie rynek narzędzi związanych z Verilogiem zaczął się znacząco rozwijać. W dniu dzisiejszym (2020 rok) praktycznie wszystkie narzędzia do syntezy i symulacji układów FPGA wydane przez producentów tych układów obsługują ten język (np. “Xilinx ISE”, “Vivado HLS”, “Intel Quartus Prime”, “Gowin EDA” itp.). Jako ciekawostkę można podać, że większość narzędzi projektowych do syntezy i weryfikacji specjalizowanych układów cyfrowych (ASIC) korzysta wyłącznie z języka Verilog (nie obsługują języka VHDL). W roku 1993 powołano grupę roboczą IEEE, która rozpoczęła prace nad przygotowaniem standardu IEEE dla Veriloga. Prace nad standardem zakończono w 1995 roku, co jeszcze umocniło jego pozycję na rynku.

Podstawowe cechy i zastosowania języka Verilog

- Umożliwia opisywanie złożonych układów cyfrowych na wysokim poziomie abstrakcji (podobnie jak język VHDL)
- Podobnie jak inne języki HDL dla układów cyfrowych, może być stosowany zarówno do modelowania jak i syntezy
- Poziom abstrakcji jest podobny jak w języku VHDL. Verilog jest bardziej zawansowany na poziomie bramek i umozliwia opis na poziomie kluczy MOSFET (brak w VHDL)
- Możliwy jest zarówno opis behawioralny (funkcjonalny), strukturalny (komponentowy, hierarchiczny) jak i dowolny opis mieszany, będący ich kombinacją
- Oparty częściowo na składni języka C
- Jest językiem przenośnym, obsługiwanym przez rozmaite oprogramowanie:
  - Do budowania aplikacji w oparciu o układy programowalne, np., Active-HDL, Synopsys, Synplify, ispLever, MaxPlus
  - Do projektowania i symulacji cyfrowych układów ASIC, np. Synopsys, Envisia, Verilog-XL, Silicon Ensemble

Sposoby opisu sprzętu w języku Verilog

- Opis behawioralny (funkcjonalny)

  Opis algorytmiczny, wykorzystujący wysokopoziomowe konstrukcje języka Verilog

- Opis strukturalny:
  - Gate-level – opis na poziomie bramek logicznych oraz elementarnych komórek cyfrowych definiowanych przez użytkownika (UDP)
  - Switch-level – opis na tranzystorów jako kluczy oraz pojemności jako elementów pamiętających (nie będzie omawiany w tym materiale)
  - Opis hierarchiczny, wielopoziomowy, RTL

Verilog moduł: podstawowa konstrukcja języka

Najważniejszym elementem każdego projektu w języku Verilog jest moduł. Wszystkie deklaracje, instrukcje, procesy itp. znajdują się wewnątrz modułu. Stanowi on abstrakcyjny model układu, lub danej części układu. Można go sobie wyobrazić, jako czarną skrzynkę z interfejsem wejściowym, wyjściowym oraz pewną funkcjonalnością.

Moduł może opisywać układ o dowolnej złożoności. Mogą to być podstawowe bramki logiczne, liczniki, sterowniki, a także bardziej skomplikowane układy jak mikroprocesory, sprzętowe kodeki wizyjne itp.

Najczęściej złożone układy dzieli się na mniejsze części, z których każdą można opisać w osobnym module. Dostarcza się w ten sposób prostych bloków funkcjonalnych, które później mogą być wykorzystane w wielu miejscach projektu.

1. Budowa modułu

Deklaracja modułu rozpoczyna się słowem kluczowym module po której podawana jest jego nazwa. Koniec modułu wyznacza słowo kluczowe endmodule.

Wygląda to mniej więcej tak:

```
module <nazwa_modułu> (<opcjonalna_lista_portów>);
...
endmodule
```

Nazwa modułu

Nazwa modułu nie może być słowem kluczowym języka Verilog oraz nie może zaczynać się od cyfry. Prawidłowa nazwa może się składać z liter, cyfr oraz znaków _ oraz $. Ważna jest także wielkości liter w nazwie, dlatego przykładowa nazwa Adder nie jest równoważna nazwie adder.

Porty

Po nazwie modułu w nawiasie podajemy nazwy portów w jakie ma być wyposażony moduł. Rolę portu tzn. czy ma on być wejściowy, wyjściowy, czy dwukierunkowy określa się w ciele modułu (można też inaczej, ale o tym później). Kolejność deklarowania portów jest dowolna.

```
<funkcja_portu> <nazwa_portu>;
```

Port może pełnić jedną z trzech funkcji:

- **input –** wejście,
- **output –** wyjście,
- **inout** – port dwukierunkowy

Przykład:

```
module and3 (x1, x2, x3, y);
 input x1, x2, x3;
 output y;
 assign y = x1 & x2 & x3;
endmodule
```

Powyższy przykład przedstawia realizację 3-wejściowej funkcji AND. Słowo kluczowe **assign** oznacza „przypisz”, czyli przypisujemy sygnałowi y kombinację sygnałów x1, x2 oraz x_3.

UWAGA: moduł jest odpowiednikiem **entity** w języku VHDL. **Komentarze:**

W języku Verilog dozwolone są komentarze znane z języka C, czyli: blokowy znajdujący się pomiędzy znacznikami /* oraz */, a także jedno liniowy zaczynający się od //.

2. Instancja

Czym jest instancja? Instancja jest, mówiąc kolokwialnie, powołanym do życia modułem. Podobnie jak w języku obiektowym np. C++, mamy osobne pojęcia stanowiące klasę i obiekt. Klasa stanowi pewien opis, a obiekt jest już istniejącym w programie tworem, stworzonym na podstawie tej klasy.

W przypadku języka Verilog moduł stanowi opis, a instancja jego wywołanie. Zobaczmy na przykładzie. Mamy dwa moduły (moduły jest najlepiej definiować w osobnych plikach, ale nie jest to obowiązek), gdzie drugi z nich wywołuje dwie instancje (a1 oraz a2) pierwszego

```
module add3 (x1, x2, x3, y);
  input x1, x2, x3;
  output y;
  assign y = x1 & x2 & x3;
endmodule

module box(x1, x2, x3, x4, x5, x6, y);
  input x1, x2, x3, x4, x5, x6;
  output y;
  wire y1, y2;
  add3 a1(x1, x2, x3, y1);
  add3 a2(x4, x5, x6, y2);
  assign y = y1 | y2;
endmodule
```

Każda z instancji musi posiadać własną unikalną nazwę. Wywołując instancję nie musimy połączać wszystkich wejść oraz wyjść. Jeśli któreś z nich opuścimy, syntezer odpowiednio zoptymalizuje układ wycinając niepotrzebną część (jeśli niczego nie podłączymy wycięty zostanie cała instancja).

3. Moduł podsumowanie składni

Na nastepnej stronie w ramach podsumowania wiedzy o module jest przedstawiona jego ogólna struktura.

Ogólna struktura modułu

```
module jk_flop_case (j, k, clock, rst, q, qb);

    input j, k, clock, rst;
    output q, qb;
    reg q;

    assign qb = ~q;

    always @ (posedge clock or posedge rst)
        begin
            if (rst == 1'b1)
                q = 1'b0;
            else
                case ({j,k})
                    2'b00: q = q;
                    2'b01: q = 1'b0;
                    2'b10: q = 1'b1;
                    2'b11: q = ~q;
                endcase
        end

endmodule
```

nagłówek modułu

deklaracje sygnałów zewnętrznych i wewnętrznych

część kombinacyjna – przypisania ciągłe

część sekwencyjna – bloki proceduralne

koniec modułu

Stany logiczne i wyrażenia

- Dopuszczalne stany logiczne
  - Stan wysoki (1)

    ```
    assign high = 1'b1;
    ```

  - Stan niski (0)

    ```
    wire low = 1'b0;
    ```

  - Stan nieokreślony (x)

    ```
    bus = 4'bx;
    ```

  - Stan wysokiej impedancji (z)

    ```
    tbus = 16'bz
    ```

- Elementy składowe wyrażeń:
  - Sygnały
  - Zmienne
  - Stałe
  - Operatory

Stałe

- liczby bez określonego rozmiaru w bitach (domyślnie zwykle 32-bit):

```
32767       // liczba w kodzie dziesiętnym (można dodać prefiks ‘d)
'h0fa0      // liczba w kodzie szesnastkowym
'o7460      // liczba w kodzie ósemkowym
'b0101      // liczba w kodzie dwójkowym
```

- liczby z określonym rozmiarem:

```
4'b1001     // 4-bitowa liczba w kodzie dwójkowym
8'd255      // 8-bitowa liczba w kodzie dziesiętnym (d można pominąć)
3'b10x      // 3-bit liczba z najmniej znaczącą cyfrą nieokreśloną
12'bx       // nieokreślona liczba 12-bitowa
16'hz       // stan wysokiej impedancji na 16 liniach
```

- liczby ze znakiem i bez znaku:

```
4'd1        // 4-bitowa liczba 1 bez znaku
-4'd1       // 4-bitowa liczba -1 bez znaku (czyli tak naprawdę 15)
4'sd1       // 4-bitowa liczba 1 ze znakiem
-4'sd1      // 4-bitowa liczba -1 ze znakiem (traktowana jako –1)
```

- liczby zmiennoprzecinkowe:

```
2301.15     // liczba zmiennoprzecinkowa w zwykłym formacie
1.30e-2     // liczba zmiennoprzecinkowa w formacie wykładniczym
```

- łańcuchy tekstowe:

```
"hello"     // łańcuch tekstowy
```

Znaki specjalne w łańcuchach tekstowych

Specifying special characters in strings

Escape String	Character Produced by Escape String
`\n`	new line character
`\t`	tab character
`\\`	slash (`\`) character
`\"`	double quote (`"`) character
`\ddd`	a character specified in 1-3 octal digits (0 <= d <= 7)
`%%`	percent (`%`) character

Komentarze i makrodefinicje

- Ignorowanie kodu od danego miejsca do końca linii:

```
wire w;  // pominięcie kodu od tego miejsca do końca linii
// pominięcie całej linii
```

- Ignorowanie całego fragmentu kodu:

```
/* pominięcie całego
   fragmentu kodu */
```

- Makrodefinicje (podstawienia):

```
`define wordsize 8
`define greeting "hello"
```

Typy sygnałów i zmiennych

- sygnały „kombinacyjne” typu **net** (wire, wor, wand, tri1, tri0, supply1, supply0)

```
wire a1, a2;            // zwykłe sygnały kombinacyjne
wand w1, w2;            // sygnały typu iloczyn galwaniczny
```

- sygnały „rejestrowe” typu **reg**

```
reg q1, q2;             // wyjścia przerzutników
```

- sygnały związane z pamięciami ulotnymi typu **trireg**
- wektory

```
wire [7:0] A1, A2;      // magistrale
reg [0:3] R1, R2;       // rejestry
```

- pamięci

```
reg M1 [0:63];          // pamięć 64x1
```

- łańcuchy tekstu

```
reg [8*14:1] stringvar; // tekst o dł. 13 znaków + '\0'
```

- zmienne całkowite typu **integer**

```
integer i;              // 32-bit liczba całkowita ze znakiem
```

- zmienne zmiennoprzecinkowe typu **real**

```
real r;                 // liczba zmiennoprzecinkowa
```

- zmienne czasowe typu **time** (czas dyskretny)

```
time t;                 // 64-bit liczba całkowita bez znaku
```

Zmienne typu „net” (z funkcjami galwanicznymi)

wire/tri (sprzeczne stany dają **x**)

Truth table for wire and tri nets

wire/tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

wand/triand – WIRED AND

Truth table for wand/triand nets

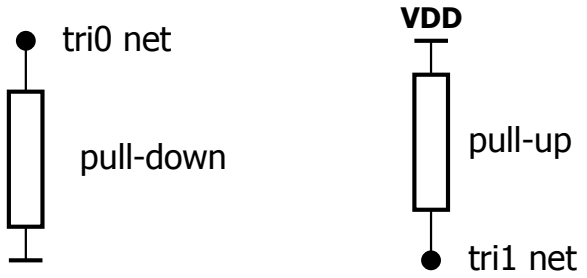
wand/triand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

wor/trior – WIRED OR

Truth table for wor/trior nets

wor/trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

tri0/tri1 – słabe zero/jeden (pull0/pull1)



supply0/supply1 – zawsze zero/jeden

Wektory i pamięci

- **Wektory**
  - Umożliwiają połączenie pojedynczych sygnałów w jeden ciąg:

```
wire [11:0] busA, busB;      // 12-bitowe magistrale
reg [7:0] A, B;              // 8-bitowe rejestry
```

  - Domyślnie, wektory są równoważne liczbom całkowitym bez znaku (unsigned)
  - Wektory mogą również odpowiadać liczbom całkowitym ze znakiem (signed):

```
reg signed [7:0] A, B;       // 8-bitowe rejestry ze znakiem
```

  - Konwersje wektorów na liczby całkowite i na odwrót są wykonywane automatycznie, bez potrzeby korzystania z żadnych bibliotek dodatkowych
  - Możliwe jest wyłuskanie pojedynczych sygnałów lub całych fragmentów wektorów:

```
wire [3:0] AH = A[7:4];      // wyłuskanie bardziej znaczącej połowy
wire A7 = A[7];              // wyłuskanie najbardziej znaczącego bitu
```

  - Możliwe jest scalanie (konkatenacja) wektorów ze sobą i z pojedynczymi sygnałami:

```
reg [3:0] AH, AL;            // deklaracja 4-bit rejestrów AH i AL
reg E;                       // deklaracja przerzutnika E
wire [8:0] EA = {E,AH,AL};   // połączenie w jeden 9-bit wektor EA
```

- **Pamięci**
  - deklaracja pamięci:

```
reg mem1 [0:63];             // pamięć 63x1
reg [7:0] mem2 [0:255];      // pamięć 255x8
```

Deklaracje portów (sygnałów zewnętrznych)

- **Rodzaje portów**
  - porty wejściowe (input)
  - porty wyjściowe (output)
  - porty dwukierunkowe (inout)

końcówki modułu

```
module jkff (j, k, clock, rst, q, qb);
    input j, k, clock, rst; // sygnały wejściowe
    output q, qb;           // sygnały wyjściowe
    reg q;                  // bo sygnał q jest wyjściem przerzutnika

    ...
endmodule
```

magistrale zewnętrzne

```
module mux21_4 (SEL, A, B, Y);
    input [1:0] SEL;        // magistrala sterująca
    input [3:0] A, B;       // magistrale wejściowe
    output [3:0] Y;         // magistrala wyjściowa

    ...
endmodule
```

Deklaracje sygnałów wewnętrznych

- **Rodzaje sygnałów wewnętrznych:**
  - sygnały kombinacyjne (nets)
  - sygnały rejestrowe (registers)
  - magistrale wewnętrzne
  - rejestry
  - pamięci

```
module datapath (clock, load, ctrl, din, dout);
    input clock;
    input [1:0] ctrl;
    input [15:0] din;
    output [15:0] dout;
    reg [15:0] dout;
    wire lda, ldb, add, nand;      // sygnały wewnętrzne
    reg [15:0] A, B;               // rejestry wewnętrzne
    wire [15:0] busA, busB, busC;  // magistrale wewnętrzne
    ...
endmodule
```

Operatory i ich priorytety

Table 4-1 Operators

Operators	Type	Reals
{}	concatenation	N
+ - * /	arithmetic	Y
%	modulus	N
> >= < <=	relational	Y
!	logical negation	Y
&&	logical and	Y
||	logical or	Y
==	logical equality	Y
!=	logical inequality	Y
===	case equality	N
!==	case inequality	N
~	bit-wise negation	N
&	bit-wise and	N
|	bit-wise inclusive or	N
^	bit-wise exclusive or	N
^~ or ~^	bit-wise equivalence	N
&	reduction and	N
~&	reduction nand	N

Table 4-1 Operators, *continued*

Operators	Type	Reals
|	reduction or	N
~|	reduction nor	N
^	reduction xor	N
~^ or ^~	reduction xnor	N
<<	shift left	N
>>	shift right	N
<<<	arithmetic shift left	N
>>>	arithmetic shift right	N
? :	conditional	Y

Precedence rules for operators

```
!     ~                                   highest precedence
*     /     %
+     -
<<     >>     <<<     >>>
<     <=     >     >=
==     !=     ===     !==
&
^     ^~
|
&&
||
?:    (ternary operator)                  lowest precedence
```

Operatory bitowe: funkcje prawdy

Bit-wise operators logic tables

bit-wise unary negation

~	
0	1
1	0
x	x

bit-wise binary AND operator

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bit-wise binary inclusive OR operator

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bit-wise binary exclusive OR operator

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise binary exclusive NOR operator

^~	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Operatory redukcyjne

Reduction Operators logic tables

reduction unary AND operator				reduction unary inclusive OR operator				reduction unary exclusive OR operator			
&	0	1	x	|	0	1	x	^	0	1	x
0	0	0	0	0	0	1	x	0	0	1	x
1	0	1	x	1	1	1	1	1	1	0	x
x	0	x	x	x	x	1	x	x	x	x	x

Note that the reduction unary NAND and reduction unary NOR operators operate the same as the reduction unary AND and OR operators, respectively, but with their outputs negated. The effective results produced by the unary reduction operators are listed in the following two tables.

Results of AND, OR, NAND, and NOR unary reduction operations

Operand	**&**	**|**	**~&**	**~|**
no bits set	0	0	1	1
all bits set	1	1	0	0
some bits set but not all	0	1	1	0

Results of Exclusive OR and exclusive NOR unary reduction operations

Operand	^	~^
odd number of bits set	1	0
even number of bits set (or none)	0	1

Przypisania ciągłe (kombinacyjne)

Składnia:

```
assign <drive_strength> <delay> net1 = ... <,net2 = ..., net3 = ...>
```

lub razem z deklaracją sygnału wewnętrznego:

```
net_type <drive_strength> <delay> net1 = ... <,net 2 = ..., net3 = ...>
trireg <charge_strength> <delay> net1 = ... <,net 2 = ..., net3 = ...>
```

```
module andorinv(z, a, b, c, d);
    input a, b, c, d;
    output z;
    assign z = ~(a & b | c & d);   // przypisanie ciągłe
endmodule

module andorinv(z, a, b, c, d);
    input a, b, c, d;
    output z;
    wire ab = a & b, cd = c & d;   // deklaracje z przypisaniami ciągłymi
    assign z = ~(ab | cd);         // przypisanie ciągłe
endmodule
```

Uwaga: Po lewej stronie przypisania kombinacyjnego może występować tylko sygnał typu net (wire, wand, wor, tri0, tri1) albo trireg!

Przykłady zastosowań przypisań ciągłych

- Sumator 4-bitowy

```
module adder (sum_out, carry_out, carry_in, ina, inb);
    output [3:0] sum_out;
    input [3:0] ina, inb;
    output carry_out;
    input carry_in;
    wire carry_out, carry_in;
    wire[3:0] sum_out, ina, inb;
    assign {carry_out, sum_out} = ina + inb + carry_in;
endmodule
```

- Dekoder 2 na 4

```
module decoder24 (out, in);
    output [3:0] out;
    input [1:0] in;
    assign out[0] = ~in[1] & ~in[0];
    assign out[1] = ~in[1] & in[0];
    assign out[2] = in[1] & ~in[0];
    assign out[3] = in[1] & in[0];
endmodule
```

Siły wymuszania (drive strength)

Strength levels for scalar net signal values

Strength Name	Strength Level	Abbreviation	
supply0	7	Su0	
strong0	6	St0	← domyślny dla stanu niskiego
pull0	5	Pu0	
large0	4	La0	
weak0	3	We0	
medium0	2	Me0	
small0	1	Sm0	
highz	0	HiZ0	
highz1	0	HiZ1	
small1	1	Sm1	
medium1	2	Me1	
weak1	3	We1	
large1	4	La1	
pull1	5	Pu1	
strong1	6	St1	← domyślny dla stanu wysokiego
supply1	7	Su1	

Przypisania ciągłe z siłą wymuszania i opóźnieniem

- Przypisania ciągłe z podanymi siłami wymuszania w stanie niskim i wysokim

  składnia: (siła_dla_stanu_wys, siła_dla_stanu_nis) **lub** (siła_dla_stanu_nis, siła_dla_stanu_wys)

```
module open_drain(out, in1, in2);          // z wyjściem typu otwarty dren
    input in1, in2;
    output out;
    assign (pull1, strong0) out = in1;     // przypisania silne w stanie niskim
    assign (pull1, strong0) out = in2;     // i słabe w stanie wysokim
endmodule
```

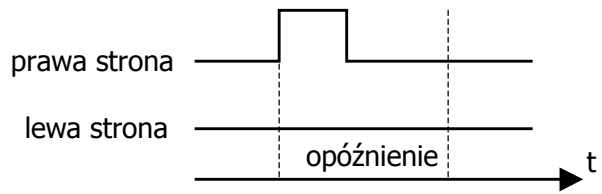
- Przypisania ciągłe z opóźnieniami

  składnia: #(opóźn_narastania <,opóźń_opadania> <,opóźń_do_st_wys_impedancji>)

```
module delay_inv_and(out, in1, in2);
    input in1, in2;
    output out;
    wire #5 w1 = ~in1;                 // opóźnienie 5 ns dla wszystkich zboczy
    wire #(5,3) w2 = ~in2;             // 5 ns dla zb. dod. i 3 ns dla zb. ujemn.
    assign #(2,3,5) out = in1 & in2;   // 5 ns przy przejściu w stan wys. imped.
endmodule
```

Uwagi do opóźnień

- Wszelkie konstrukcje z opóźnieniami są niesyntezowalne i mogą być wykorzystywane jedynie do modelowania i symulacji opóźnień!
- Reguły stosowania opóźnień są następujące:
  - Opóźnienia są wyrażone w domyślnych jednostkach danego symulatora
  - Jeżeli podano tylko jeden parametr, to będzie on stosowany do określenia opóźnienia na wszystkich możliwych zboczach, włącznie ze stanami przejścia ze stanów **x** i **z** lub do stanów **x** i **z**;
  - Jeżeli podano dwa parametry (pierwszy dla zbocza narastającego 0-1 i drugi dla zbocza opadającego 1-0), to czas przejścia w stan wys. impedancji (**z**) będzie równy dłuższemu z tych czasów;
  - Czas przejścia ze stanu w stan nieokreślony (**x**) jest równy najkrótszemu z podanych dwóch (trzech) czasów;
  - Opóźnienia mają charakter **inercyjny**, czyli jeżeli prawa strona wyrażenia po zmianie powróci do swojej poprzedniej wartości przed upływem podanego opóźnienia, to wartość lewej strony wyrażenia nie ulegnie zmianie. Oznacza to, że impulsy o szerokości mniejszej od podanego opóźnienia zostaną odfiltrowane.



Przypisania ciągłe z operatorem warunkowym

Składnia:

```
assign net = condition ? value_if_true : value_if_false;
```

konstrukcję z operatorem warunkowym można zagnieżdżać w celu zbudowania konstrukcji warunkowej wielowariantowej:

```
assign net =  condition1 ?  value_if_true :
              condition2 ?  value_if_true :
                            value_if_else;
```

```
module tristate_buffer(out, in, en);  // bufor trójstanowy

    input in, en;
    output out;

    assign out = en ? in : 1'bz;      // przypisanie z operatorem warunkowym

endmodule

module mux41(out, din, sel);          // multiplekser 4 na 1

    input [1:0] sel;
    input [3:0] din;
    output out;

    assign out = (sel == 0) ? in[0] :
                 (sel == 1) ? in[1] :
                 (sel == 2) ? in[2] :
                 (sel == 3) ? in[3] : 1'bx;

endmodule
```

Bloki proceduralne

- Bloki realizujące część sekwencyjną modelu – odpowiedniki procesów w języku VHDL
- Bloki proceduralne dzielą się na:
  - Sekwencyjne **begin-end** i równoległe **fork-join**
  - Wykonywane jednokrotnie **initial** (niesyntezowalne) i wielokrotnie **always**
  - Nazwane (z etykietą) i nie nazwane (bez etykiety)
- Bloki proceduralne składają się z przypisań proceduralnych, realizujących operacje sekwencyjne
- Jeśli w jednym modelu jest wiele bloków proceduralnych, to będą one wykonywane współbieżnie, tak jak procesy języka VHDL
- Wykonanie bloku jest może zostać wyzwolone:
  - Po upłynięciu danego czasu (konstrukcja niesyntezowalna)
  - Po wystąpieniu jakiegoś zdarzenia, np. zbocza sygnału lub tzw. zdarzenia jawnego (nie wszystkie przypadki są syntezowalne)
- Bloki proceduralne mogą zawierać opóźnienia, jednakże wówczas nie są one syntezowalne i mogą być wykorzystywane jedynie do modelowania i symulacji

Struktura bloku proceduralnego

Współbieżne bloki proceduralne

```
module behave;
    reg a,b;
    initial
        begin
            a = 1'b1;
            b = 1'b0;
        end
    always
        begin
            #50 a = ~a;
        end
    always
        begin
            #100 b = ~b;
        end
endmodule
```

przypisania sekwencyjne bez opóźnień → `a = 1'b1; b = 1'b0;`

blok sekwencyjny initial → `initial begin ... end`

przypisanie sekwencyjne z opóźnieniem → `#50 a = ~a;`

blok sekwencyjny always → `always begin ... end`

przypisanie sekwencyjne z opóźnieniem → `#100 b = ~b;`

blok sekwencyjny always → `always begin ... end`

Uwaga: Słowa kluczowe **begin** i **end** można pominąć, jeśli blok proceduralny zawiera tylko jedno przypisanie proceduralne

Blok proceduralny initial

- Wykonuje się tylko raz w momencie rozpoczęcia symulacji
- Jest niesyntezowalny i wykorzystywany głównie w modułach testujących zwanych test fixtures (odpowiednik testbench w języku VHDL)
- Może zawierać opóźnienia (i zwykle zawiera), ale teoretycznie nie musi
- Opóźnienia są wyrażone w domyślnych jednostkach symulatora, zwykle ps lub ns. Domyślną jednostkę można zmienić dyrektywą `*timescale* :

`timescale <time_unit> / <time_precision>

```
`timescale 1 ns / 10 ps

initial
    begin
        inputs = 'b000000;          // wymuszenie w chwili 0 ns
        #10 inputs = 'b011001;      // wymuszenie w chwili 10 ns
        #10 inputs = 'b011011;      // wymuszenie w chwili 20 ns
        #10 inputs = 'b011000;      // wymuszenie w chwili 30 ns
        #10 inputs = 'b001000;      // wymuszenie w chwili 40 ns
    end
```

Blok proceduralny always

- Wykonuje się „na okrągło”, tak więc musi zawierać przynajmniej jedno opóźnienie lub instrukcję oczekiwania na zdarzenie:
  - Wyzwalany opóźnieniem:

```
always #5
    begin
        clk = ~clk;
    end
```

lub

```
always
    begin
        #5 clk = ~clk;
    end
```

lub po prostu:

```
always #5 clk = ~clk;
```

ale nigdy:

```
always clk = ~clk;
```

  - Wyzwalany zboczem narastającym, opadającym lub dowolnym:

```
always @ a
    begin
        b = a;
    end

always @(posedge clk)
    begin
        q = d;
    end
```

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

lub

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

Blok proceduralny always (c.d.)

- Wyzwalany zboczami więcej niż jednego sygnału:

```
always @(posedge rst or posedge clk)
    begin
        if (rst)
            q = 1'b0;
        else
            q = d;
    end
```

lub

```
always
    begin
        @(posedge rst or posedge clk)
        if (rst)
            q = 1'b0;
        else
            q = d;
    end
```

- Z instrukcją oczekiwania:

```
always
    begin
        wait (ce);
        @(posedge clk) q = d;
    end
```

lub

```
always
    begin
        wait (!ce);
        @(posedge clk) q = d;
    end
```

ale nigdy:

```
always
    begin
        wait (!latch) q = d;
    end
```

Bloki instrukcji sekwencyjne i równoległe

- Blok sekwencyjny **begin-end**
  - Przypisania są wykonywane po kolei
  - Opóźnienie jest liczone od zakończenia poprzedniego przypisania

```
begin
        r = 4'h0;   // t = 0
    #10 r = 4'h2;   // t = 10 ns
    #10 r = 4'h4;   // t = 20 ns
    #10 r = 4'h8;   // t = 30 ns
end
```

- Blok równoległy **fork-join**
  - Przypisania są wykonywane współbieżnie
  - Wszystkie opóźnienia są liczone względem początku bloku

```
fork
        r = 4'h0;   // t = 0
    #10 r = 4'h2;   // t = 10 ns
    #20 r = 4'h4;   // t = 20 ns
    #30 r = 4'h8;   // t = 30 ns
join
```

Intra-assignments

Intra-assignment timing control with intra-assignment construct	Intra-assignment timing control without intra-assignment construct

`a = #5 b;`	`begin`
	`  temp = b;`
	`  #5 a = temp;`
	`end`

`a = @(posedge clk) b;`	`begin`
	`  temp = b;`
	`  @(posedge clk) a =`
	`temp;`

`a = repeat(3)@(posedge clk) b;`	`begin`
	`  temp = b;`
	`  @(posedge clk;`
	`  @(posedge clk;`
	`  @(posedge clk) a =`
	`temp;`

Zastosowanie do zapobiegania wyścigom:

```
fork
    #5 a = b;   // rezultat będzie
    #5 b = a;   // przypadkowy
join
```

```
fork
    a = #5 b;   // nastąpi zamiana
    b = #5 a;   // wartości a i b
join
```

Przypisania nieblokujące

Wykonywanie tego przypisania nie wstrzymuje wykonywania następnych instrukcji

```
module evaluates2(out);

output out;
reg a, b, c;

initial
   begin
   a = 0;
   b = 1;
   c = 0;
   end

always          c = #5 ~c;

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

Evaluates, schedules, and executes in two steps.

Step 1:

The simulator evaluates the right-hand side of the non-blocking assignments and schedules the assignments of the new values at posedge c.

Non-blocking assignment scheduled changes at time 5

a = 0

b = 1

Step 2:

At posedge c, the simulator updates the left-hand side of each non-blocking assignment statement.

Assignment values are:

a = 1

b = 0

Przypisania nieblokujące (c.d.)

```
//non_block1.v
module non_block1(out);
//input
output out;
reg a, b, c, d, e, f;
//blocking assignments
initial begin
   a = #10 1;
   b = #2 0;
   c = #4 1;
   end

//non-blocking assignments
   initial begin
   d <= #10 1;
   e <= #2 0;
   f <= #4 1;
   end
initial begin
$monitor ($time, ,"a = %b b =
 d = %b e = %b f = %b", a,b,
#100 $finish;
   end
endmodule // non_block1
```

The simulator assigns 1 to register a at simulation time 10, assigns 0 to register b at simulation time 12, and assigns 1 to register c at simulation time 16.

The simulator assigns 1 to register d at simulation time 10, assigns 0 to register e at simulation time 2, and assigns 1 to register f at simulation time 4.

non-blocking assignment lists

Scheduled changes at time 2

e = 0

Scheduled changes at time 4

f = 1

Scheduled changes at time 10

d = 1

Mieszanie przypisań blokujących i nieblokujących

```
//non_block1.v
module non_block1(out,);
output out;
reg a, b;
initial begin
   a = 0;
   b = 1;
   a <= b;
   b <= a;
end
      initial begin
$monitor ($time, ,"a = %b b = %b", a,b);
#100 $finish;
   end
endmodule
```

Evaluates, schedules, and executes in two steps.

Step 1:

The simulator evaluates the right-hand side of the non-blocking assignments and schedules the assignments for the end of the current time step.

Step 2:

At the end of the current time step, the simulator updates the left-hand side of each non-blocking assignment statement.

Assignment values are:

a = 1

b = 0

Przypisania warunkowe

- Instrukcja warunkowa **if-else**

```
if (sel)
    C = A + B;
else
    C = A - B;
```

```
if (state == PASS1)
    begin
        A = A1 + A2;
        state = PASS2;
    end
else
    begin
        A = A + A1 + A2;
        state = PASS1;
    end
```

- Konstrukcja wielowariantowa **if-else-if**

```
if (state == RED)
    begin
        RYG = 3'b100;
        state = RED_YELLOW;
    end
else if (state == RED_YELLOW)
    begin
        RYG = 3'b110;
        state = GREEN;
    end
else if (state == GREEN)
    begin
        RYG = 3'b001;
        state = YELLOW;
    end
else
    begin
        RYG = 3'b010;
        state = RED;
    end
```

Konstrukcje wielodecyzyjne case/casez/casex

- Instrukcja **case**

```
reg [15:0] rega;
reg [9:0] result;
    ...
case (rega)
    16'd0: result = 10'b0111111111;
    16'd1: result = 10'b1011111111;
    16'd2: result = 10'b1101111111;
    16'd3: result = 10'b1110111111;
    16'd4: result = 10'b1111011111;
    16'd5: result = 10'b1111101111;
    16'd6: result = 10'b1111110111;
    16'd7: result = 10'b1111111011;
    16'd8: result = 10'b1111111101;
    16'd9: result = 10'b1111111110;
    default result = 'bx;
endcase
```

UWAGA: Instrukcja **case** traktuje stany *z* i *x* tak samo jak stany 0 i 1

- Instrukcja **casez**

przy porównaniu ignoruje bity w stanie z

```
reg [7:0] ir;
    ...
casez (ir)
    8'b1???????: instruction1(ir);
    8'b01??????: instruction2(ir);
    8'b00010???: instruction3(ir);
    8'b000001??: instruction4(ir);
endcase
```

- Instrukcja **casex**

przy porównaniu ignoruje bity w stanach z i x

```
reg [7:0] r, mask;
    ...
mask = 8'bx0x0x0x0;

casex (r ^ mask)
    8'b001100xx: stat1;
    8'b1100xx00: stat2;
    8'b00xx0011: stat3;
    8'bxx001100: stat4;
endcase
```

Przykłady zastosowania bloków proceduralnych

- Przerzutnik D z kasowaniem asynchronicznym

```
module dffar (Q, nQ, rst, clk, D);
   output Q, nQ;
   input rst, clk, D;
   reg Q;

   assign nQ = ~Q;

   always @(posedge rst or posedge clk)
      begin
         if (rst)
            Q = 1'b0;
         else
            Q = D;
      end
endmodule
```

- Przerzutnik JK z kasowaniem asynchronicznym

```
module jkffar (Q, nQ, rst, clk, J, K);
   output Q, nQ;
   input rst, clk, J, K;
   reg Q;

   assign nQ = ~Q;

   always @(posedge rst or posedge clk)
      begin
         if (rst)
            Q = 1'b0;
         else
            case ({J,K})
               2'b01: Q = 1'b0;
               2'b10: Q = 1'b1;
               2'b11: Q = ~Q;
               default Q = Q;
            endcase
      end
endmodule
```

Pętle

- Pętla **repeat**

```
initial
  begin
    repeat (10)
      begin
        $display("i= %d", i);
        i = i + 1;
      end
  end
```

- Pętla **while**

```
always #10
  begin
    i = 0;
    while (i < 10)
      begin
        $display("i= %d", i);
        i = i + 1;
      end
  end
```

- Pętla **forever**

```
initial
  begin
    counter = 0;
    forever #10 counter = counter + 1;
  end
```

- Pętla **for**

```
always #10
  begin
    for (i = 1; i < 16; i = i+1)
      begin
        $display("i= %d", i);
      end
  end
```

Syntezowalny kod zawierający pętlę

Komparator z pętlą **for**, przerywaną instrukcją **disable**:

```
module comparator_with_loop (a, b, a_gt_b, a_lt_b, a_eq_b);
  parameter size = 2;
  input [size: 1] a, b;
  output a_gt_b, a_lt_b, a_eq_b;
  reg a_gt_b, a_lt_b, a_eq_b;
  integer k;

    always @ (a or b)
      begin: compareLoop

        for (k = size; k > 0; k = k-1) begin
          if (a[k] != b[k]) begin
            a_gt_b = a[k];
            a_lt_b = ~a[k];
            a_eq_b = 0;
            disable compareLoop;
          end    // if
        end      // for loop

        a_gt_b = 0;
        a_lt_b = 0;
        a_eq_b =1;
      end         // compare loop

endmodule
```

etykieta bloku jest wymagana

warunkowe przerwanie działania bloku compareLoop

Proceduralne przypisania ciągłe

- Konstrukcja **assign-deassign**
  - Stosowana do modelowania sprzętu, np. układów z asynchronicznym kasowaniem

```
module dff(q,d,clear,preset,clock);
  output q;
  input d,clear,preset,clock;
  reg q;
  always @(clear or preset)
    begin
      if (!clear)
        assign q = 0;      // asynchroniczne kasowanie
      else if (!preset)
        assign q = 1;      // asynchroniczne ustawianie
      else
        deassign q;  // odblokowanie przerzutnika po skasowaniu/ustawieniu
    end
  always @(posedge clock)
    q = d;           // klasyczne przypisanie proceduralne
endmodule
```

- Konstrukcja **force-release**
  - Stosowana w symulacji do uruchamiania (debuggowania) projektów
  - Ma wyższy priorytet niż assign-deassign

Zadania słowo kluczowe **task**

- Są to procedury składające się z instrukcji sekwencyjnych, takich jakie występują w blokach proceduralnych
- Mogą przyjmować zero lub więcej argumentów dowolnego typu: input, output lub inout
- Przy wywołaniu zadania, jeśli argument jest typu input, może zostać do niego podstawiony dowolny rodzaj danej, wyrażenie lub stała
- Jeśli argument jest typu inout lub output, podstawiona dana jest ograniczona do tylko tych typów, które mogą znajdować się po lewej stronie przypisania proceduralnego, a więc rejestrów, odwołań do pamięci oraz ich dowolnych fragmentów i połączeń (konkatenacji)

Definicja zadania:

```
module this_task;
   task my_task;
      input a, b;
      inout c;
      output d, e;
      reg foo1, foo2, foo3;
      begin
         <statements>        // the set of statements that performs the work of the task
         c = foo1;           // the assignments that initialize
         d = foo2;           // the results variables
         e = foo3;
      end
   endtask
endmodule
```

Wywołanie zadania:

```
my_task (v, w, x, y, z);
```

Zadania - przykład

```
module SHREG4(CLK, S, IN, IL, IR, Q);
   input [3:0] IN;
   input [1:0] S;
   input CLK, IL, IR;
   output [3:0] Q;
   reg [3:0] Q;

   parameter left = 0, right = 1;

   task load;
     Q = IN;                                  // load contents into the register
   endtask

   task shift;
     input direction;                         // 0 - shift left, 1 - shift right

     begin
       if (direction)
         Q = {IL,Q[3:1]};                     // shift right register contents
       else
         Q = {Q[2:0],IR};                     // shift left register contents
     end
   endtask

 always @(posedge CLK)                        // main procedural block
     begin
       case (S)
         2'b00: ;                             // do nothing

         2'b11: load;                         // load contents
         2'b01: shift(right);                 // shift right
         2'b10: shift(left);                  // shift left

         default: Q = 4'bx;                   // unhandled states
       endcase
     end
 endmodule
```

Funkcje – przykład 1

- Funkcja obliczająca liczbę jedynek w 4-bitowym słowie – przykład syntezowalny

```
module test_ones (in, out);
  input [3:0] in;
  output [2:0] out;

  function [2:0] get_ones;
    input [3:0] operand;

    get_ones = operand[0] + operand[1] + operand[2] + operand[3];
  endfunction

  assign out = get_ones(in);
endmodule
```

Funkcje – przykład 2

- Funkcja obliczająca 2^n – przykład niesyntezowalny

```
module test_power2;
  reg [4:0] value;
  reg [5:0] n;

  function [31:0] power2;
    input [4:0] exponent;
    reg [4:0] index;

    begin
      power2 = 1;
      for(index = 0; index < exponent; index = index + 1)
        power2 = power2 * 2;
    end

  endfunction

  initial
   begin
    for(n = 0; n <= 31; n = n + 1)
      $display("2^%0d = %0d", n, power2(n));
   end
endmodule
```

Ograniczenia funkcji

- Funkcje nie mogą zawierać żadnych instrukcji sterowanych przez czas, czyli żadnych instrukcji poprzedzonych przez #, @, lub wait.
- Funkcje nie mogą uruchamiać zadań.
- Funkcje muszą posiadać przynajmniej jeden argument typu input.
- Lista argumentów funkcji nie może zawierać argumentów typu inout ani output.
- Funkcja musi zawierać przypisanie wyniku funkcji do wewnętrznej (lokalnej) zmiennej o takiej samej nazwie co nazwa funkcji (przypisanie to jest zwykle na końcu).

Zadania a funkcje:

- Funkcja musi się wykonać w jednej chwili czasowej symulacji, natomiast zadanie może zawierać instrukcje czasowe (#, @, lub wait).
- Podczas gdy funkcja nie może uruchamiać zadania, zadanie może uruchamiać funkcje i inne zadania.
- Funkcja musi mieć przynajmniej jeden argument typu input, natomiast zadanie może mieć zero lub więcej argumentów dowolnego typu.
- Funkcja zwraca wartość, natomiast zadanie nie zwraca żadnej wartości.

Zdarzenia jawne

Wykorzystywane jako semafory synchronizujące współbieżne bloki proceduralne:

```
module test_ones (dl, din, dout);
  input dl;
  input [7:0] din;
  output [7:0] dout;
  event latch;

  always @(posedge dl)
    begin
      ->latch;

    end

  always @(posedge clk)
    begin
      @latch dout = din;
    end

endmodule
```

deklaracja zdarzenia jawnego

wywołanie zdarzenia jawnego

instrukcja oczekująca na zdarzenie jawne

Opis strukturalny

Rodzaje opisu strukturalnego:

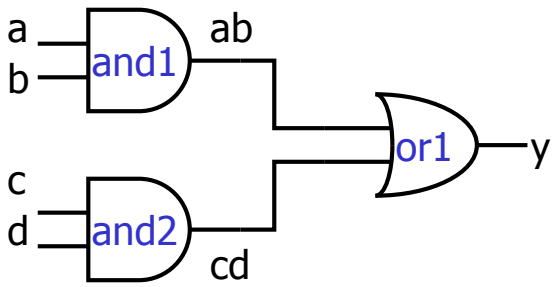
- Opis na poziomie bramek logicznych i kluczy (gate- and switch-level description)
- Modelowanie z wykorzystaniem prostych układów kombinacyjnych lub sekwencyjnych definiowanych przez użytkownika (UDP)
- Opis hierarchiczny (wielopoziomowy)
- Makromoduły

Wbudowane elementy podstawowe:

- Bramki AND, OR, NAND, NOR, XOR, XNOR
- Inwertery i bufory dwustanowe i trójstanowe
- Klucze MOS i CMOS
- Dwukierunkowe bramki transmisyjne
- Rezystory pull-up i pull-down

Składnia opisu na poziomie bramek i kluczy

- Z etykietami i jawną deklaracją sygnałów wewnętrznych




```
module and_or (y, a, b, c, d);
  output y;
  input a, b, c, d;

  wire ab, cd;

  and and1 (ab, a, b);
  and and2 (cd, c, d);
  or or1 (y, ab, cd);

endmodule
```

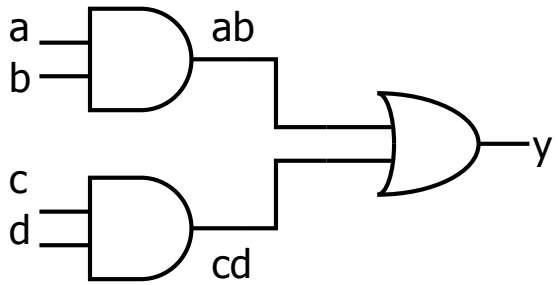
deklaracje sygnałów wew.

etykieta elementu (opcjonalna)

sygnały wejściowe

sygnał wyjściowy

- Bez etykiet i niejawną deklaracją sygnałów wewnętrznych



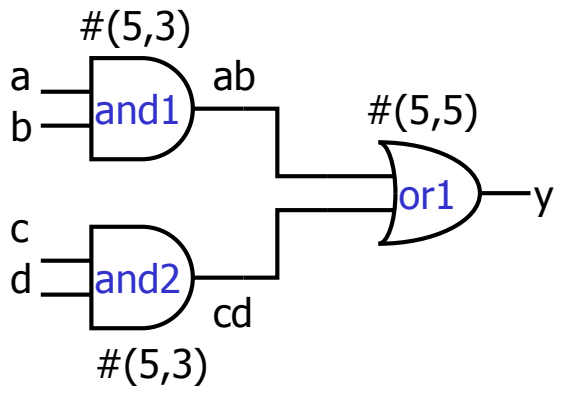

```
module and_or (y, a, b, c, d);
  output y;
  input a, b, c, d;

  and (ab, a, b);
  and (cd, c, d);
  or (y, ab, cd);

endmodule
```

Składnia opisu na poziomie bramek i kluczy

- Komponenty z opóźnieniami i różnymi siłami wymuszania




```
module and_or (y, a, b, c, d);
  output y;
  input a, b, c, d;
  wire ab, cd;

  and (strong0,strong1) #(5,3) and1 (ab, a, b);
  and (strong0,strong1) #(5,3) and2 (cd, c, d);
  or (strong0,strong1) #5 or1 (y, ab, cd);

endmodule
```

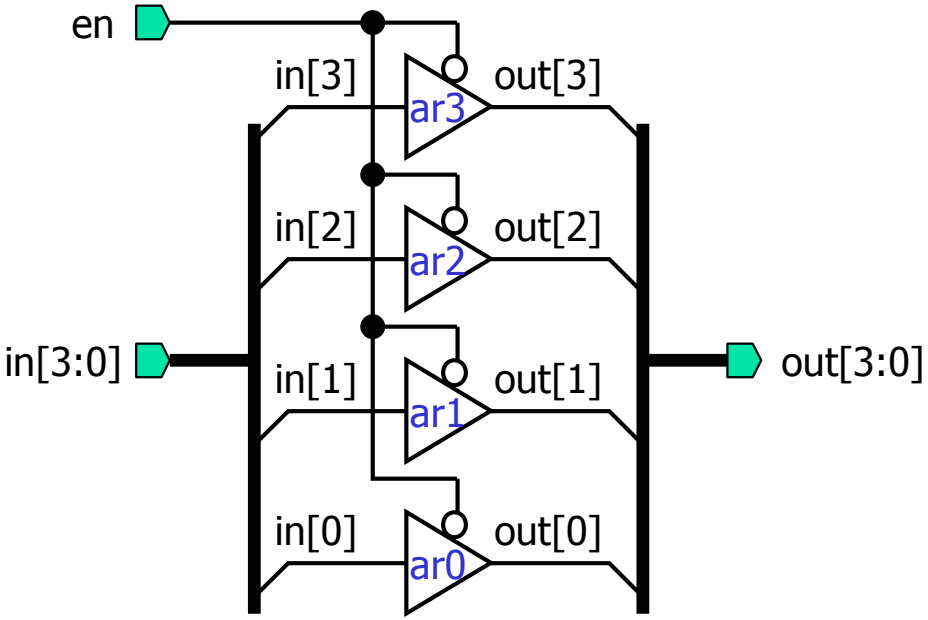
opóźnienie przy przejściu w stan wysoki i niski

siła wymuszenia przy przejściu z w stan wysoki

siła wymuszenia przy przejściu z w stan niski

Ciągi elementów

- Bez ciągów elementów

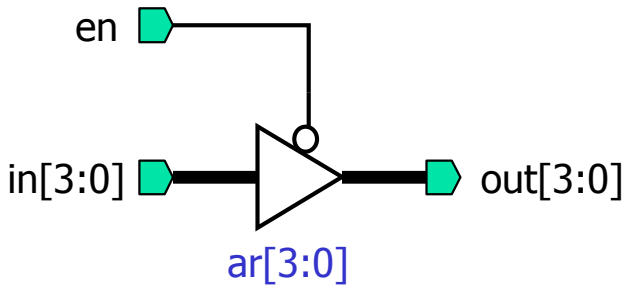



```
module driver (in, out, en);
  input [3:0] in;
  output [3:0] out;
  input en;

  bufif0 ar3 (out[3], in[3], en);
  bufif0 ar2 (out[2], in[2], en);
  bufif0 ar1 (out[1], in[1], en);
  bufif0 ar0 (out[0], in[0], en);

endmodule
```

- Z ciągiem elementów




```
module driver (in, out, en);
  input [3:0] in;
  output [3:0] out;
  input en;

  bufif0 ar[3:0] (out, in, en);

endmodule
```

Bramki logiczne (1)

- Bramki AND, OR, NAND, NOR, XOR

Logic tables for and, or, and xor gates

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

Logic tables for nand, nor, and xnor gates

nand	0	1	x	z
0	1	1	1	1
1	1	0	x	x
x	1	x	x	x
z	1	x	x	x

nor	0	1	x	z
0	1	0	x	x
1	0	0	0	0
x	x	0	x	x
z	x	0	x	x

xnor	0	1	x	z
0	1	0	x	x
1	0	1	x	x
x	x	x	x	x
z	x	x	x	x

```
and AND1 (z, a, b);          // 2-wejściowa bramka AND z etykietą AND1
nand NA1 (z, a, b, c);       // 3-wejściowa bramka NAND z etykietą NA1
xnor (y, v, w);              // 2-wejściowa bramka XNOR bez etykiety
nor #(1,2) (z, a, b);        // 2-wejściowa bramka NOR z opóźnieniami
or (strong0,pull1) (z, a, b) // 2-wejściowa bramka OR z siłami wymuszania
```

Bramki logiczne (2)

- Inwertery i bufory

Logic tables for buf and not gates

buf			not	
input	output		input	output
0	0		0	1
1	1		1	0
x	x		x	x
z	x		z	x

```
not NEG1 (out, in);          // pojedynczy inwerter z etykietą NEG1
buf (out1,out2,in);          // pojedynczy bufor bez etykiety
not NEG[3:0] (C, A[3:0]);    // 4 równoległe inwertery
buf (o1, o2, o3, o4, i);     // bufor z czterema wyjściami i jednym wyjściem
```

Bramki logiczne (3)

- Bufory i inwertery trójstanowe

Logic tables for bufif0 and bufif1 gates

bufif0	CONTROL				
		0	1	x	z
DATA	0	0	z	L	L
	1	1	z	H	H
	x	x	z	x	x
	z	x	z	x	x

bufif1	CONTROL				
		0	1	x	z
DATA	0	z	0	L	L
	1	z	1	H	H
	x	z	x	x	x
	z	z	x	x	x

Logic tables for notif0 and notif1 gates

notif0	CONTROL				
		0	1	x	z
DATA	0	1	z	H	H
	1	0	z	L	L
	x	x	z	x	x
	z	x	z	x	x

notif1	CONTROL				
		0	1	x	z
DATA	0	z	1	H	H
	1	z	0	L	L
	x	z	x	x	x
	z	z	x	x	x

```
bufif0 (weak1,pull0) #(4,5,3) (data_out, data_in, ctrl);
```

sygnał wyjściowy

sygnał sterujący

sygnał wejściowy