



# Verilog HDL

---

część I i II

# 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 prawie tak wysoki jak w języku VHDL, ale nieco niższy – mały krok w kierunku języków typu ABEL
- 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
  - Opis hierarchiczny, wielopoziomowy, RTL

# 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  
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 wyrażen:

- 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
<code>\n</code>	new line character
<code>\t</code>	tab character
<code>\\</code>	slash ( <code>\</code> ) character
<code>\"</code>	double quote ( <code>"</code> ) character
<code>\ddd</code>	a character specified in 1-3 octal digits ( $0 \leq d \leq 7$ )
<code>%%</code>	percent ( <code>%</code> ) 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 „rejestrów” 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”

**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

**wor/trior** – WIRED OR

Truth table for wor/trior nets

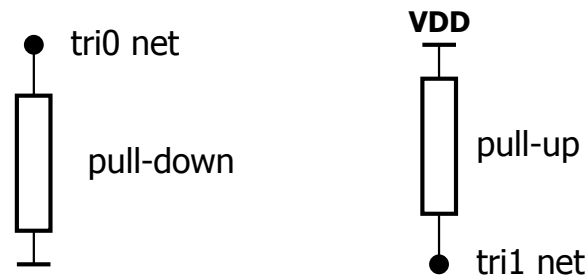
wor/trior	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

**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

**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)

```
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
```

końcówki modułu

```
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
```

magistrale zewnętrzne

# 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
```

# Operator i ich priority

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



# Operatorzy 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 = ...>
```

```
triereg <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 triereg!

# 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
```

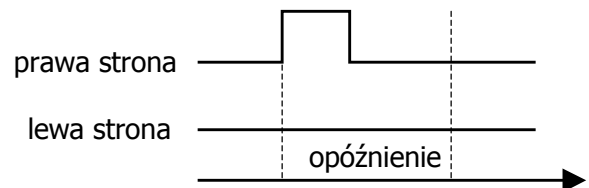
- Przypisania ciągłe z opóźnieniami

**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 niesyntezywalne 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 ciągle 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); // multiplexer 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
```

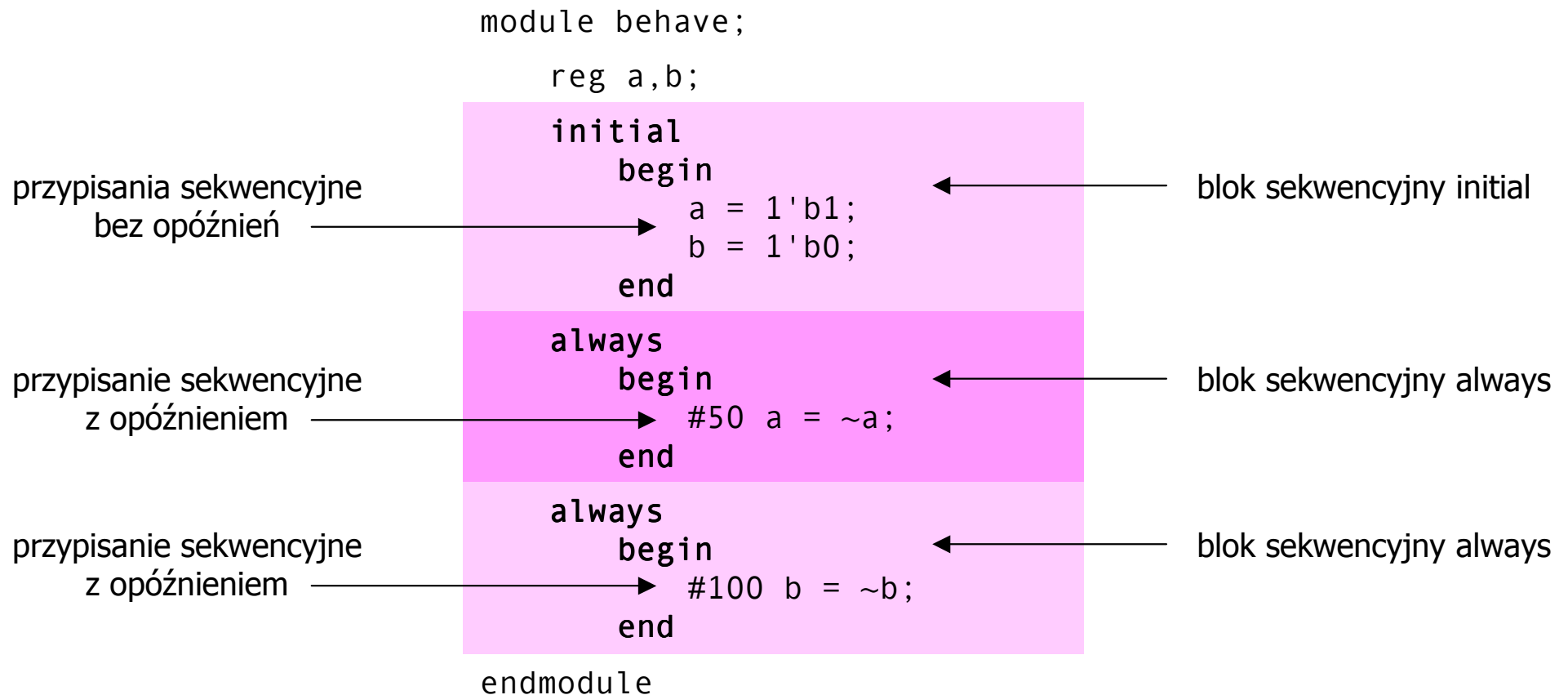
# Bloki proceduralne

---

- 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** (niesynteżowalne) 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 upływie danego czasu (konstrukcja niesynteżowalna)
  - Po wystąpieniu jakiegoś zdarzenia, np. zbocza sygnału lub tzw. zdarzenia jawnego (nie wszystkie przypadki są synteżowalne)
- Bloki proceduralne mogą zawierać opóźnienia, jednakże wówczas nie są one synteżowalne i mogą być wykorzystywane jedynie do modelowania i symulacji

# Struktura bloku proceduralnego

## Współbieżne bloki proceduralne



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

lub

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

```
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
```

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

lub

- 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

---

```
a = #5 b;
```

---

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

---

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

---

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

---

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

---

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

---

```
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
```

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.

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

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
```

*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*



# Pytanie: Czy obydwa przykłady są równoważne?

---

```
reg a, b;  
  
begin  
    a <= b;  
    b <= a;  
end
```



```
reg a, b;  
  
fork  
    a = b;  
    b = a;  
join
```

# 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

## ■ Pęta **repeat**

```
initial  
begin
```

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

```
end
```

## ■ Pęta **forever**

```
initial  
begin
```

```
    counter = 0;
```

```
    forever #10 counter = counter + 1;
```

```
end
```

## ■ Pęta **while**

```
always #10
```

```
begin  
    i = 0;
```

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

```
    end
```

```
end
```

## ■ Pęta **for**

```
always #10
```

```
begin
```

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

```
    end
```

```
end
```

# Synteżowalny 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; etykieta bloku jest wymagana  
    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
```

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
```

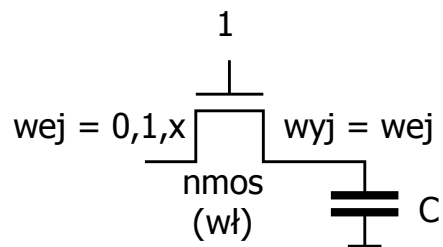
## ■ Konstrukcja **force-release**

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

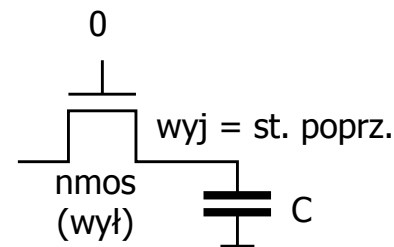
# Modelowanie pamięci dynamicznych

## Węzeł typu **trireg**:

- Może znajdować się w jednym z dwóch stanów:
  - Stan wymuszony (driven state), jeżeli przynajmniej jedno z wymuszeń jest w stanie 0, 1 lub X. Wówczas węzeł propaguje stan wynikający z wymuszeń.
  - Stan pojemnościowy (capacitive state), kiedy wszystkie wymuszenia są w stanie wys. impedancji (Z). Wówczas węzeł pamięta poprzedni stan.
- Typ **trireg** stoi po lewej stronie **przypisań ciągłych** i dlatego jest często klasyfikowany jako podtyp typu **net**
- W stanie wymuszonym, siły wymuszeń mogą być ustawione na **strong**, **pull** albo **weak**
- W stanie pojemnościowym, węzeł typu **trireg** może stanowić wymuszenie o różnej sile (charge strength): **small**, **medium** lub **large**, co jest bezpośrednio związane z wartością pojemności komórki pamięci dynamicznej
- W stanie pojemnościowym, stan węzła może utrzymywać się w nieskończoność (pojemność idealna) lub po pewnym czasie może ulec zniszczeniu (charge decay), czyli przejściu w stan nieokreślony (X) (pojemność z upływem)



węzeł trireg w stanie wymuszonym

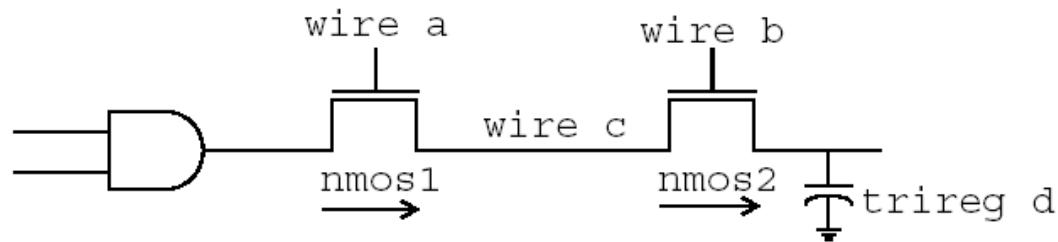


węzeł trireg w stanie pojemnościowym



# Symulacja działania węzła typu trireg

## Simulation values of a trireg and its driver



simulation time	wire a	wire b	wire c	trireg d
0	1	1	strong 1	strong 1
10	0	1	HiZ	medium 1

Simulation of the design in this figure reports the following results:

1. At simulation time 0, wire a and wire b have a value of 1. A value of 1 with a strong strength propagates from the AND gate through the NMOS switches connected to each other by wire c, into trireg d.
2. At simulation time 10, wire a changes value to 0, disconnecting wire c from the AND gate. When wire c is no longer connected to the AND gate, its value changes to HiZ. The value of wire b remains 1 so wire c remains connected to trireg d through the NMOS2 switch. The HiZ value does not propagate from wire c into trireg d. Instead, trireg d enters the capacitive state, storing its last driven value of 1 with a medium strength.

# Przykłady zastosowania węzła typu trireg

```
reg data1, data2, select;
trireg c1; // deklaracja węzła c1 o pojemności medium
           // i o nieskończonym czasie podtrzymania
trireg (small) #(0,0,1000) c2; // deklaracja węzła c2 o pojemności small
           // i z czasem podtrzymania 1000 ns

assign c1 = select ? data1 : 1'bz; // sygnał select przelacza c1 i c2 pomiędzy
assign c2 = select ? data2 : 1'bz; // stanem wymuszonym a pojemnościowym

initial
begin
    data = 1'b1 // wymuszenie stanu 1 na węzłach c1 i c2
    #0 select = 1'b1; // ustawienie c1 i c2 w stan wymuszony
    #50 select = 1'b0; // przestawienie c1 i c2 w stan pojemnościowy
                       // (zapamiętany stan 1) po 50 ns
end
```

siła wymuszenia w stanie poj. (wielkość ładunku)

czas podtrzymania informacji w c2

# Modelowanie pamięci dynamicznych (przykład)

---

## Komórka pamięci dynamicznej

```
`timescale 1ns / 10 ps

module dram_cell(data, wl, rw);
    inout data;                                // wejście/wyjście danych
    input wl;                                  // selektor komórki (word line)
    input rw;                                  // 1 - zapis, 0 - odczyt
    wire bit;

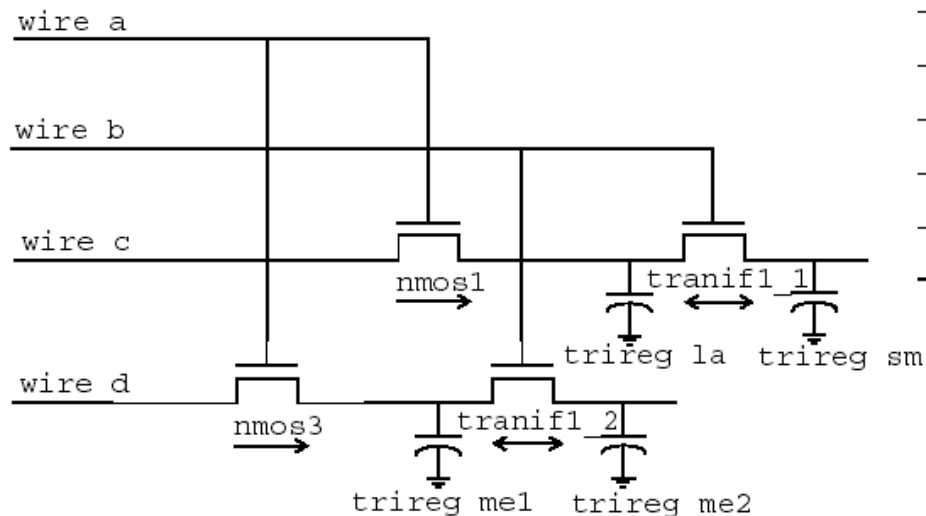
    trireg #(0,0,8000) mem;                    // deklaracja komórki pamięci

    assign mem = wl ? bit : 1'bz;             // zapis do komórki
    assign (weak0,weak1) bit = wl ? mem : 1'bz; // odczyt z komórki
    assign bit = rw ? data : 1'bz;
    assign data = rw ? 1'bz : bit;

endmodule
```

# Symulacja działania sieci pojemnościowej

Simulation results of a capacitive network

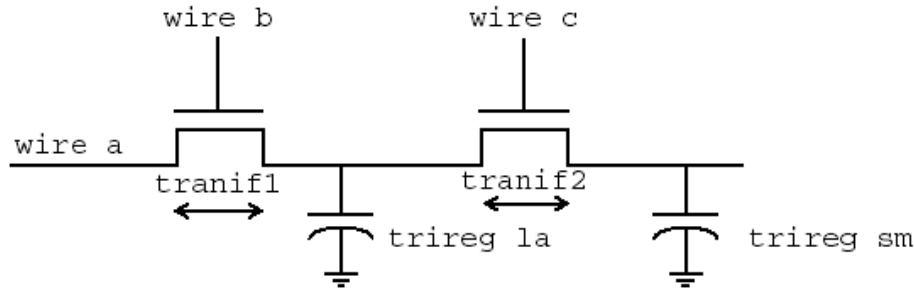


simulation time	wire a	wire b	wire c	wire d	trireg la	trireg sm	trireg me1	trireg me2
0	1	1	1	1	1	1	1	1
10	1	0	1	1	1	1	1	1
20	1	0	0	1	0	1	1	1
30	1	0	0	0	0	1	0	1
40	0	0	0	0	0	1	0	1
50	0	1	0	0	0	0	x	x

1. At simulation time 0, wire a and wire b have a value of 1. The wire c drives a value of 1 into triregs la and sm, wire d drives a value of 1 into triregs me1 and me2.
2. At simulation time 10, the value of wire b changes to 0, disconnecting trireg sm and me2 from their drivers. These triregs enter the capacitive state and store the value 1; their last driven value.
3. At simulation time 20, wire c drives a value of 0 into trireg la.
4. At simulation time 30, wire d drives a value of 0 into trireg me1.
5. At simulation time 40, the value of wire a changes to 0, disconnecting trireg la and me1 from their drivers. These triregs enter the capacitive state and store the value 0.
6. At simulation time 50, the value of wire b changes to 1. This change of value in wire b connects trireg sm to trireg la; these triregs have different sizes and stored different values. This connection causes the smaller trireg to store the larger trireg value and trireg sm now stores a value of 0. This change of value in wire b also connects trireg me1 to trireg me2; these triregs have the same size and stored different values. The connection causes both trireg me1 and me2 to change value to x.

# Symulacja współdzielenia ładunku

Simulation results of charge sharing



simulation time	wire a	wire b	wire c	trireg la	trireg sm
0	strong 1	1	1	strong 1	strong 1
10	strong 1	0	1	large 1	large 1
20	strong 1	0	0	large 1	small 1
30	strong 1	0	1	large 1	large 1
40	strong 1	0	0	large 1	small 1

1. At simulation time 0, the value of wire a, b, and c is 1 and wire a drives a strong 1 into trireg la and sm.
2. At simulation time 10, the value of wire b changes to 0, disconnecting trireg la and sm from wire a. The triregs la and sm enter the capacitive state. Both triregs share the large charge of trireg la because they remain connected through tranif2.
3. At simulation time 20, the value of wire c changes to 0, disconnecting trireg sm from trireg la. The trireg sm no longer shares the large charge of trireg la and now stores a small charge.
4. At simulation time 30, the value of wire c changes to 1, connecting the two triregs. These triregs now share the same charge.
5. At simulation time 40, the value of wire c changes again to 0, disconnecting trireg sm from trireg la. Once again, trireg sm no longer shares the large value of trireg la and now stores a small charge.

# Zadania

- 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 niesyntezywalny

```
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 semaforey synchronizujące współbieżne bloki proceduralne:

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

← deklaracja zdarzenia jawnego

```
  always @(posedge dl)  
  begin
```

```
    ->latch;
```

← wywołanie zdarzenia jawnego

```
  end
```

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

← instrukcja oczekująca na zdarzenie jawne

```
endmodule
```

# Opis strukturalny

---

## **Rodzaje opisu strukturalnego:**

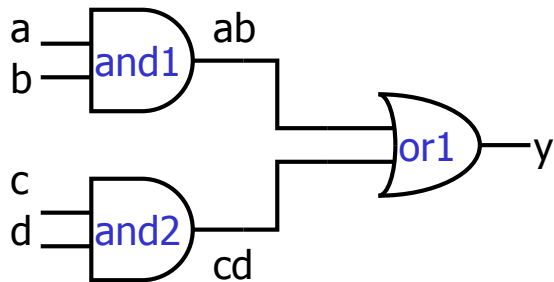
- Opis na poziomie bramek logicznych i kluczy (gate- and switch-level description)
- Modelowanie z wykorzystaniem prostych układów kombinacyjnych lub sekwencyjnych definiowanych przez użytkownika (UDP)
- Opis hierarchiczny (wielopoziomowy)
- Makromoduły

## **Wbudowane elementy podstawowe:**

- Bramki AND, OR, NAND, NOR, XOR, XNOR
- Inwertery i bufony 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; ← deklaracje sygnałów wew.
```

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

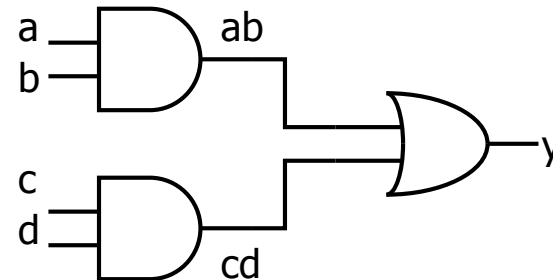
```
endmodule
```

etykieta elementu (opcjonalna)

sygnały wejściowe

sygnał wyjściowy

- Bez etykiet i niejawną deklaracją sygnałów wewnętrznych



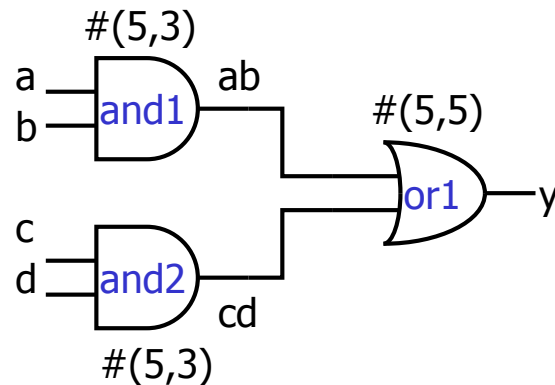
```
module and_or (y, a, b, c, d);  
  output y;  
  input a, b, c, d;
```

```
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;
```

opóźnienie przy przejściu w stan wysoki i niski

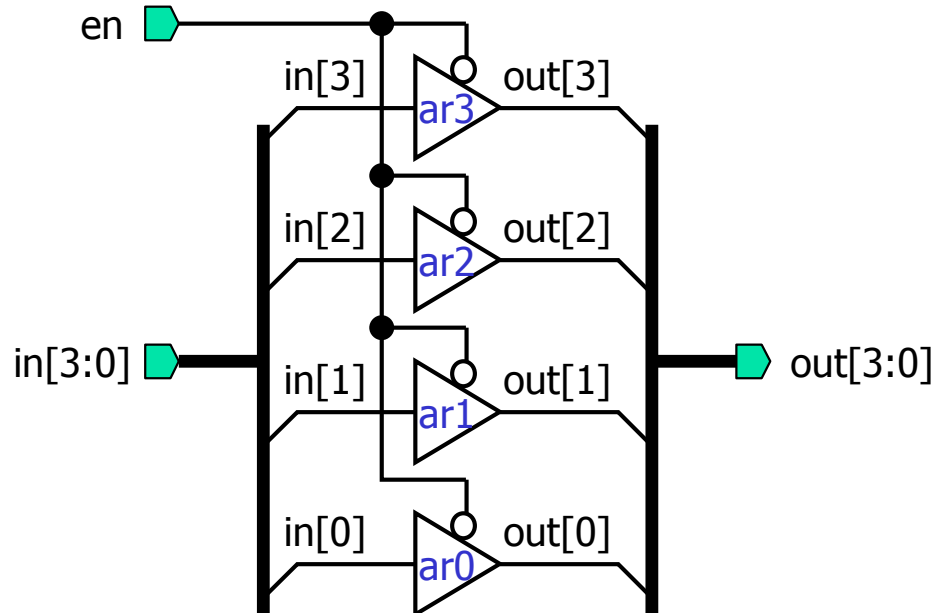
```
  and (strong0,strong1) #(5,3) and1 (ab, a, b);  
  and (strong0,strong1) #(5,3) and2 (cd, c, d);  
  or (strong0,strong1) #5 or1 (y, ab, cd);
```

endmodule

siła wymuszenia przy przejściu z w stan wysoki  
siła wymuszenia przy przejściu z w stan niski

# Ciągi elementów

## ■ Bez ciągów elementów

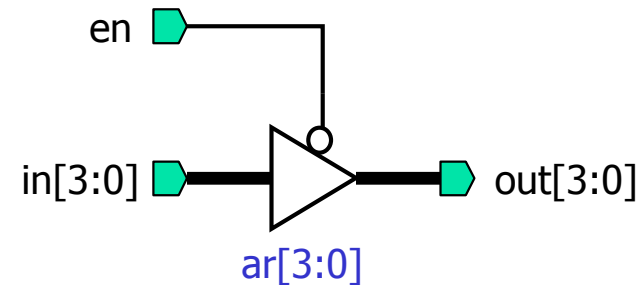


```
module driver (in, out, en);  
  input [3:0] in;  
  output [3:0] out;  
  input en;
```

```
  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		or	0	1	x	z		xor	0	1	x	z
0	0	0	0	0		0	0	1	x	x		0	0	1	x	x
1	0	1	x	x		1	1	1	1	1		1	1	0	x	x
x	0	x	x	x		x	x	1	x	x		x	x	x	x	x
z	0	x	x	x		z	x	1	x	x		z	x	x	x	x

Logic tables for nand, nor, and xnor gates

nand	0	1	x	z		nor	0	1	x	z		xnor	0	1	x	z
0	1	1	1	1		0	1	0	x	x		0	1	0	x	x
1	1	0	x	x		1	0	0	0	0		1	0	1	x	x
x	1	x	x	x		x	x	0	x	x		x	x	x	x	x
z	1	x	x	x		z	x	0	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 wejściem
```

# Bramki logiczne (3)

- Bufory i inwertery trójstanowe

Logic tables for bufif0 and bufif1 gates

bufif0	CONTROL				
		0	1	x	z
D	0	0	z	L	L
A	1	1	z	H	H
T	x	x	z	x	x
A	z	x	z	x	x

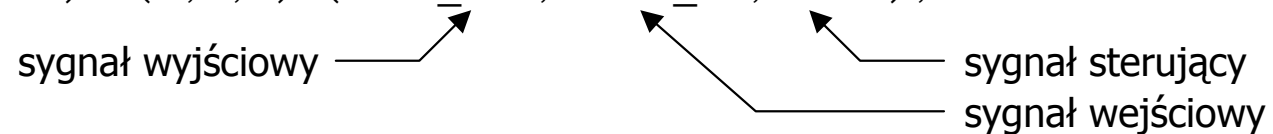
bufif1	CONTROL				
		0	1	x	z
D	0	z	0	L	L
A	1	z	1	H	H
T	x	z	x	x	x
A	z	z	x	x	x

Logic tables for notif0 and notif1 gates

notif0	CONTROL				
		0	1	x	z
D	0	1	z	H	H
A	1	0	z	L	L
T	x	x	z	x	x
A	z	x	z	x	x

notif1	CONTROL				
		0	1	x	z
D	0	z	1	H	H
A	1	z	0	L	L
T	x	z	x	x	x
A	z	z	x	x	x

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





# Przykład zastosowania kluczy MOS

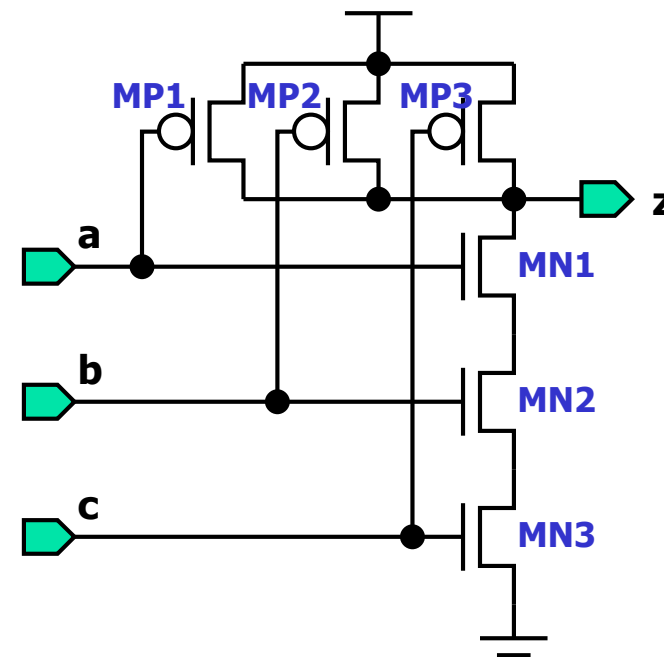
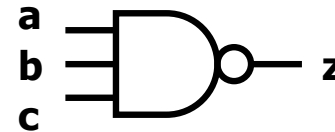
## 3-wejściowa bramka NAND

```
module nand3 (z, a, b, c);  
  output z;           // wyjście  
  input a, b, c;     // wejścia  
  
  supply0 gnd;       // masa  
  supply1 pwr;       // zasilanie
```

```
nmos MN1 (z, i12, a); // sekcja  
nmos MN2 (i12, i23, b); // pull-down  
nmos MN3 (i23, gnd, c);
```

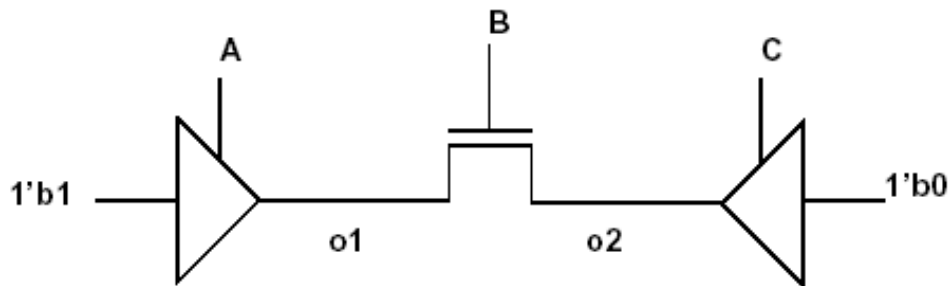
```
pmos MP1 (z, pwr, a); // sekcja  
pmos MP2 (z, pwr, b); // pull-up  
pmos MP3 (z, pwr, c);
```

```
endmodule
```



# Dwukierunkowe bramki transmisyjne

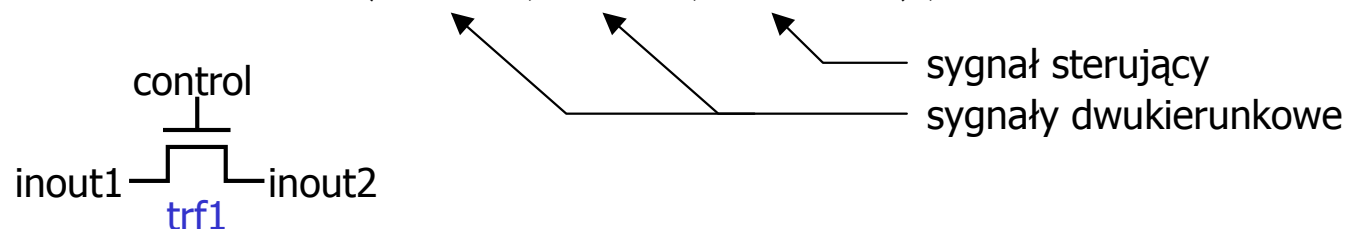
- Nie przyjmują specyfikacji siły wymuszania
- Rodzaje dwukierunkowych bramek transmisyjnych:
  - Dwukierunkowe bramki przewodzące przez cały czas (nie przyjmują opóźnień):
    - **tran** – redukuje siłę **supply** na **strong**, a pozostałe pozostawia bez zmian
    - **rtran** – redukuje siłę **supply** na **pull**, **pull** na **weak**, itd.
  - Klucze aktywne stanem niskim (tranif0, rtranif0)
  - Klucze aktywne stanem wysokim (tranif1, rtranif1)



A	B	C	o1	o2
0	0	0	z	z
0	0	1	z	0
0	1	0	z	z
0	1	1	0	0
1	0	0	1	z
1	0	1	1	0
1	1	0	1	1
1	1	1	x	x

```
tran tr1 (inout1, inout2);
```

```
tranif1 #100 trf1 (inout1, inout2, control);
```

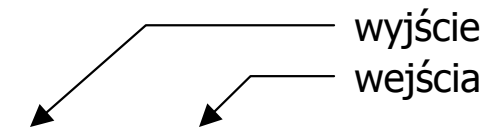


# Elementy definiowane przez użytkownika (UDP)

- UDP są prostymi układami kombinacyjnymi lub sekwencyjnymi definiowanymi przez użytkownika, których używa się w taki sam sposób jak elementów wbudowanych
- Mogą posiadać tylko jedno wyjście (pojedynczy sygnał), a wejścia mogą być jedynie pojedynczymi sygnałami (nie mogą być magistralami)
- Dla sekwencyjnego UDP, jego stan odpowiada stanowi pojedynczego przerzutnika (0, 1 lub x) i jest jednocześnie sygnałem wyjściowym
- Działanie kombinacyjnego UDP określa się tablicą prawdy, a sekwencyjnego UDP – tablicą przejść

```
// Description of an AND-OR gate.  
// out = (a1 & a2 & a3) | (b1 & b2).
```

```
primitive and_or(out, a1, a2, a3, b1, b2);  
output out;  
input a1, a2, a3, b1, b2;
```



```
table  
    // a b : out ;  
    111 ?? : 1 ;  
    ??? 11 : 1 ;  
    0?? 0? : 0 ;  
    0?? ?0 : 0 ;  
    ?0? 0? : 0 ;  
    ?0? ?0 : 0 ;  
    ??0 0? : 0 ;  
    ??0 ?0 : 0 ;  
endtable
```

```
endprimitive
```

# Oznaczenia występujące w tablicach UDP

---

Symbol	Interpretation	Notes
0	Logic 0	
1	Logic 1	
x	Unknown	
?	Iteration of 0, 1, and x	Cannot be used in output field
b	Iteration of 0 and 1	Like ?, except x is excluded Cannot be used in output field
-	No change	Can only be used in output field of a sequential UDP
(vw)	Value change from v to w	v and w can be any one of: 0, 1, x, ?, or b
*	Same as ??	Any value change on input
r	Same as 01	Rising edge on input
f	Same as 10	Falling edge on input
p	Iteration of (01), (0x), and (x1)	Positive edge on input
n	Iteration of (10), (1x), and (x0)	Negative edge on input

# Kombinacyjne UDP

## Multiplexer 2 na 1

```
primitive multiplexer(mux, control, dataA, dataB );  
  output mux;  
  input control, dataA, dataB;
```

```
table  
  // control dataA dataB mux  
  0 1 0 : 1 ;  
  0 1 1 : 1 ;  
  0 1 x : 1 ;  
  0 0 0 : 0 ;  
  0 0 1 : 0 ;  
  0 0 x : 0 ;  
  1 0 1 : 1 ;  
  1 1 1 : 1 ;  
  1 x 1 : 1 ;  
  1 0 0 : 0 ;  
  1 1 0 : 0 ;  
  1 x 0 : 0 ;  
  x 0 0 : 0 ;  
  x 1 1 : 1 ;  
endtable
```

```
endprimitive
```

← tablica prawdy



# Kombinacyjny UDP ze skróconym opisem

---

Ten sam multiplexer 2 na 1

```
primitive multiplexer(mux,control,dataA,dataB );
  output mux;
  input control, dataA, dataB;

  table
    // control dataA dataB mux
    0 1 ? : 1 ; // ? = 0,1,x
    0 0 ? : 0 ;
    1 ? 1 : 1 ;
    1 ? 0 : 0 ;
    x 0 0 : 0 ;
    x 1 1 : 1 ;
  endtable
endprimitive
```

? = dowolny stan: 0, 1 lub x

# Sekwencyjne UDP wrażliwe na poziomy

Zatrząsk (przerzutnik D wyzwalany poziomem)

```
primitive latch(q, clock, data);  
  output q;  
  reg q; ← deklaracja stanu  
  input clock, data;  
  
  table  
    // clock data q q+  
    0 1 : ? : 1 ; ← tablica przejść  
    0 0 : ? : 0 ;  
    1 ? : ? : - ; // - = no change  
  endtable  
endprimitive
```

# Sekwencyjne UDP wrażliwe na zbocza

---

Przerzutnik D wyzwalany zboczem narastającym

```
primitive d_edge_ff(q, clock, data);
  output q;
  reg q;
  input clock, data;

  table
    // obtain output on rising edge of clock
    // clock data q q+
    (01) 0 : ? : 0 ;
    (01) 1 : ? : 1 ;
    (0?) 1 : 1 : 1 ;
    (0?) 0 : 0 : 0 ;
    // ignore negative edge of clock
    (?0) ? : ? : - ;
    // ignore data changes on steady clock
    ? (??) : ? : - ;
  endtable
endprimitive
```

# UDP wrażliwe zarówno na poziomy jak i zbocza

Przerzutnik D wyzwalany zboczem narastającym i ustawianiem i kasowaniem

```
primitive jk_edge_ff(q, clock, j, k, preset, clear);
  output q;
  reg q;
  input clock, j, k, preset, clear;

  table
    //clock jk pc state output/next state
    ? ?? 01 : ? : 1 ; //preset logic
    ? ?? *1 : 1 : 1 ;
    ? ?? 10 : ? : 0 ; //clear logic
    ? ?? 1* : 0 : 0 ;
    r 00 00 : 0 : 1 ; //normal clocking cases
    r 00 11 : ? : - ;
    r 01 11 : ? : 0 ;
    r 10 11 : ? : 1 ;
    r 11 11 : 0 : 1 ;
    r 11 11 : 1 : 0 ;
    f ?? ?? : ? : - ;
    b *? ?? : ? : - ; //j and k transition cases
    b ?* ?? : ? : - ;
  endtable
endprimitive
```

# Zmniejszanie pesymizmu w UDP

## ■ Zatrząsk typu D

```
primitive latch(q, clock, data);
  output q; reg q ;
  input clock, data ;
  table
    // clock data state output/next state
    0 1 : ? : 1 ;
    0 0 : ? : 0 ;
    1 ? : ? : - ; // - = no change

    // ignore x on clock when data equals state
    x 0 : 0 : - ;
    x 1 : 1 : - ;

  endtable
endprimitive
```

## ■ Przerzutnik JK

```
primitive jk_edge_ff(q, clock, j, k, preset, clear);
  output q; reg q;
  input clock, j, k, preset, clear;
  table
    // clock jk pc state output/next state
    // preset logic
    ? ?? 01 : ? : 1 ;
    ? ?? *1 : 1 : 1 ;
    // clear logic
    ? ?? 10 : ? : 0 ;
    ? ?? 1* : 0 : 0 ;
    // normal clocking cases
    r 00 00 : 0 : 1 ;
    r 00 11 : ? : - ;
    r 01 11 : ? : 0 ;
    r 10 11 : ? : 1 ;
    r 11 11 : 0 : 1 ;
    r 11 11 : 1 : 0 ;
    f ?? ?? : ? : - ;
    // j and k cases
    b *? ?? : ? : - ;
    b ?* ?? : ? : - ;

    // cases reducing pessimism
    p 00 11 : ? : - ;
    p 0? 1? : 0 : - ;
    p ?0 ?1 : 1 : - ;
    (?0)?? ?? : ? : - ;
    (1x)00 11 : ? : - ;
    (1x)0? 1? : 0 : - ;
    (1x)?0 ?1 : 1 : - ;
    x *0 ?1 : 1 : - ;
    x 0* 1? : 0 : - ;

  endtable
endprimitive
```

# Struktury hierarchiczne (wielopoziomowe)

- Moduły na niższym poziomie hierarchii:

```
module AND (out, in1, in2);  
    output out;  
    input in1, in2;  
  
    assign out = in1 & in2;  
endmodule
```

```
module OR (out, in1, in2);  
    output out;  
    input in1, in2;  
  
    assign out = in1 | in2;  
endmodule
```

- Moduł na wyższym poziomie hierarchii:

- Niejawne mapowanie portów (przez zachowanie tej samej kolejności portów i odpowiadających im sygnałów)

```
module AND_OR (y, x1,x2, x3, x4);  
    output y;  
    input x1, x2, x3, x4;
```

```
    AND A1 (x12, x1, x2);  
    AND A2 (x34, x3, x4);  
    OR O1 (y, x12, x34);
```

```
endmodule
```

- Jawne mapowanie portów (przez określenie które porty mają być podłączone do których sygnałów)

```
module AND_OR (y, x1,x2, x3, x4);  
    output y;  
    input x1, x2, x3, x4;
```

```
    AND A1 (.in1(x1), .in2(x2), .out(x12));  
    AND A2 (.in1(x3), .in2(x4), .out(x34));  
    OR O1 (.in1(x12), .in2(x34), .out(y));
```

```
endmodule
```

# Moduły parametryzowane

- Moduł na niższym poziomie hierarchii z parametrami `size` i `delay`:

```
module vdff (out, in, clk);  
    parameter size = 1, delay = 1;  
    input [0:size-1] in;  
    input clk;  
    output [0:size-1] out;  
    reg [0:size-1] out;  
  
    always @(posedge clk)  
        # delay out = in;  
  
endmodule
```

- Moduł na wyższym poziomie hierarchii deklarujący komponenty z parametrami:

```
module m;  
    reg clk;  
    wire[1:10] out_a, in_a;  
    wire[1:5] out_b, in_b;  
  
    // create an instance and set  
    parameters  
    vdff #(10,15) mod_a(out_a, in_a, clk);  
    // create an instance leaving default  
    values  
    vdff mod_b(out_b, in_b, clk);  
  
endmodule
```

# Makromoduły

- Są „spłaszczane” w celu przyspieszenia wykonywania symulacji, czyli nie otrzymują nazwy instancji, nie ma podłączenia portów do sygnałów i makro znajduje się na tym samym poziomie hierarchii co moduł nadrzędny
- Podlegają pewnym ograniczeniom
  - Nie można w nich deklarować rejestrów ani stosować przypisań proceduralnych
  - W listach sygnałów wbudowanych bramek i elementów UDP nie mogą znajdować się wyrażenia, lecz jedynie pojedyncze sygnały
  - Jedynym dopuszczalnym wykorzystaniem parametrów jest użycie ich do określania opóźnień w makromodule

```
module topmod;  
  wire [4:0] v;  
  wire a,b,c,w;
```

```
  modB b1 (v[0], v[3], w, v[4]);
```

```
  initial $list;
```

```
endmodule
```

```
macromodule modB(wa, wb, c, d);  
  inout wa, wb;  
  input c, d;  
  parameter d1=6, d2=5;  
  parameter d3=2, d4=6;
```

```
  tranif1 g1(wa, wb, cinvert);  
  not #(d3, d4) (cinvert, int);  
  and #(d1, d2) g2(int, c, d);
```

```
endmodule
```