Verilog: obiekty

- Obiekty posiadają typy.
- Obiekty deklarowane są wraz z typem. Istnieją typy domyślne.
- Nieformalny podział:
  - SIECI (*NET*) - przypisania ciągłe
  - ZMIENNE (*VARIABLE*) - przypisania blokujące
- Verilog ma jednolitą strukturę (biblioteki i konfiguracja od 2001), pełny dostęp do obiektów jest możliwy poprzez nazwy hierarchiczne.
- Pojedynczy bit ma 4 wartości: 0, 1, x, i z. Wartość domyślna obiektu to z (wysoka impedancja) dla typu wire, x (unknown) lub 0 dla reg.
- **W Verilog istnieje automatyczna konwersja typów i dopasowanie wymiarów.**

Verilog: typy obiektów

- W Verilog wszystkie obiekty są interpretowane jako liczby w kodzie NKB lub U2 (U2 jeśli obiekt zadeklarowany jako signed).
- Rozmiary obiektów po lewej stronie podstawienia są rozszerzane lub skracane do rozmiaru obiektu po prawej stronie.

- Typ kombinacyjny – net:
  - **Podstawowy: wire – domyślny dla portów**
  - Rozwiązywalne (wor, wand, trireg)
  - Na poziomie tranzystorowym: supply0(1), tri0(1) – stałe 0 (1) lub pull down(up)
- Typ rejestrowy
  - **reg – konieczny dla przypisań blokujących**
- integer (32-bitowy reg)
  - zawsze signed
- real
- time (zawsze unsigned)

Verilog: literały

[<rozmiar>] <'podstawa> <wartość>

- Podstawy obiektów: dziesiętne (**domyślne**), binarne x'bx, heksadecymalne (x'hx)
- Ze względu na automatyczną konwersję literały można mieszać
  - 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
```

Verilog: wektory i pamięci

Obiekty typu reg i net mogą występować jako tablice jedno i dwuwymiarowe:

- unpacked np.:

  wire A [7:0] ; reg **B[0:15] [0:7];**
- packed (dostęp do całych słów) np.:

  reg [63:0]M, reg**[7:0] M[63:0];**

Deklarację rozmiaru przed nazwą obiektu packed nazywamy szerokością wektora, a po nazwie obiektu rozmiarem tablicy.
Wektory obiektów typu reg nazywane są pamięciami.

reg [7:0] mema[0:255];	// declares a memory mema of 256 8-bit registers.
(mema[1] = 0;)	//The indices are 0 to 255, 8 bit value
reg arrayb[7:0][0:255];	// declare a two-dimensional array of
(arrayb[1][0] = 0;)	// one bit registers
wire w_array[7:0][5:0];	// declare array of wires
integer inta[1:64];	// an array of 64 integer values (not the numer of bits)
time chng_hist[1:1000]	// an array of 1000 time values
integer t_index;	**//** 32-bit integer

Arytmetyka w Verilog: liczby ze znakiem

- W pierwszym standardzie podstawowe typy **wire** i **reg** były traktowane jako wektory bitów (konwersja do NKB). Tylko typ **integer** był interpretowany jako **liczba** ze znakiem.
- Dopiero modyfikacja standardu z 2001 r. dodała możliwość stosowania wektorów bitowych wire i reg ze znakiem (kodowanie U2) oraz funkcje systemowe **$signed()** i **$unsigned()** umożliwiające konwersje.

```
module add _signed (
input signed [ 2 : 0 ] A,
input signed[ 2 : 0 ] B ,
output signed [3 : 0] Sum ) ;

assign Sum = $signed ( { A[2], A } ) + $signed ( { B[2], B } );
* sklejenie { A[2], A } daje wynik bitowy, konieczność konwersji do signed

endmodule
```

Arytmetyka w Verilog: liczby niecałkowite

- Verilog nie daje prostej możliwości wykorzystywania liczb niecałkowitych – brak jest literałów „z kropką” (i pakietów jak w VHDL np. fixed_pkg).
- Często jedyną możliwością jest samodzielne zaprojektowanie modułów, wykonujących odpowiednie operacje arytmetyczne. Operacje +, -, * na liczbach stałoprzecinkowych można wykonywać używając podkreślnika zamiast kropki i pamiętając o współczynniku skalowania do formatu Qn.m

```
module fixedtest();
reg signed [7:0] a; reg signed [7:0] b; reg signed [8:0] ab;
localparam sf = 2.0**-4.0; // Q4.4 scaling factor
initial
begin
a = 8'b0011_1010; //  58 interpretowane jako 3.6250
b = 8'b0100_0001; // 4.0625
ab = a + b; // 0111.1011 = 7.6875
$display("a = ", $itor(ab)*sf);  //$itor converts integer into real
……..
```

Verilog: operatory

{}	sklejenie
+ - * / %	Arytmetyczne (% - modulo)
> >= < <=	relacji
! && ||	logiczne: negacji, and , or
== !=	logiczne: równości , nierówności (jeśli argument x lub z to wynik x)
=== !==	równości , nierówności (porównuje wartości x i z, wynik zawsze 1 lub 0)
~ & | ^ ^~	bitowe: negacji, and , or , exor, exnor
& , ~& | ~| ^ ~^	redukujące: and, nand, or, nor, exor, exnor (jednoargumentowe)
<< >>	przesunięcia
or	tzw. event or, alternatywa zdarzeń (np. @(trig or enable))
?:	op. warunkowy

<wyrażenie_warunkowe> ? <wyrażenie_T> : <wyrażenie_F>

- Konwencja operatorów jest podobna do języka C.
- Verilog wyróżnia operatory **logiczne** (operujące na wartościach boolowskich), **bitowe** (operujące na wartościach bitowych) i **redukujące** (jeden bit jako wynik operacji na wielu bitach) oraz **arytmetyczne** (operujące na typie integer).
- Operatory bitowe działają na równoległych bitach. Wynik jest bitowy i ma rozmiar identyczny jak rozmiar dłuższego argumentu.
- Wynikiem operacji logicznej jest true lub false.
- nowsze wersje (SystemVerilog) nie są restrykcyjne co do dopasowania operatora do typów argumentów.

Mechanizm symulacji

1. **Elaboracja**: rozwinięcie hierarchii, łączenie komponentów/instancji , utworzenie sterowników.
2. **Inicjacja**: nadanie wartości początkowej sterownikom.
3. **Symulacja**. Procesy uruchamiane są w reakcji na zdarzenia jakimi są zmiany wartości:
   - sieci, zmiennych lub nazwanych zdarzeń, tzw. *active events* . Ze względu na bardziej elastyczne sterowanie i stąd bardziej skomplikowany mechanizm szeregowania transakcji, symulator Verilog może się zachowywać **niedeterministyczne** (to które aktywne zdarzenie jest rozpatrywane jako pierwsze bywa losowe).

Np. poprawne jest wyświetlenie 0 lub 1 dla kodu:

```
assign p = q;
        initial begin
        q = 1;  #1 q = 0;
        $display(p);
        end
```

Sterowanie czasem symulacji

Moment wykonania instrukcji można określić poprzez:

- specyfikację opóźnienia:
  #<liczba> #10
  #<identyfikator> #d
  #<min_typ_max_wyr> #(2:3:4), #((d+e)/2)

- **zdarzenie:**
  @<zdarzenie> np.:
  @(posedge c), @(negedge c)
  @(a, b), @(a or b), @*

- instrukcję wait
  wait <wyrażenie> wait(!enable)

blokuje wykonanie kolejnych instrukcji dopóki nieprawdziwe jest wyrażenie

Sterowanie czasem: uwagi

- Opóźnienie w Verilog ma zawsze charakter inercyjny.
- Każda konstrukcja z podaną wartością czasu (#, after, wait for) jest **niesyntezowalna**.

- W Verilog można zdefiniować własne zdarzenie (tzw. *named event*), np.:

```
event sent_data;                    // deklaracja zdarzenia
always @ (posedge clk) begin
if(i ==8) -> sent_data;             //wyzwolenie zdarzenia, sent_data staje się
    true
end
always@ (sent_data) begin
……..
```

- Niedeterministyczne działanie symulatora Verilog w niektórych przypadkach wynika z innej koncepcji cyklu symulacyjnego niż w VHDL – braku restrykcyjnego podziału na sygnały i zmienne oraz koncepcji opóźnienia delta.

Model układu synchronicznego i kombinacyjnego

- Proces synchroniczny (taktowany) - zawiera konstrukcję wykrywającą zbocze zegara

```
process (rst, clk) --VHDL
 begin
 if rst = '0' then
            count <=0;
 elsif (clk'event and clk='1') then
            count <= dane;
 end if;
end process;
```

```
always @(posedge clk or negedge rst)
   begin
// begin niekonieczne gdy jedna instr.
     if ( !rst)  count =0;
     else count = dane;
   end
// end „zamykający" begin
```

- Proces kombinacyjny: brak reakcji na zbocze. Dla poprawnej syntezy podane muszą być wyrażenia określające wartości wyjściowe dla wszystkich wartości wejściowych.

```
process(a, b)
begin
    c <=  a and b;
end process;
```

```
always  @(a , b)
    c <=  a & b;
always  @(a  or b)
   c =  a & b;
always @*
  c =  a & b;
```

HDLs: poziomy abstrakcji

Verilog, poziomy:

- kluczy
- bramek
- rejestrowy
- funkcjonalny

VHDL, poziomy:

- strukturalny
- RTL
- abstrakcyjny funkcjonalny

Modelowanie na poziomie bramek

- W odróżnieniu od VHDL, Verilog w modelowaniu strukturalnym wykorzystuje bramki wbudowane i inne elementy prymitywne. Nie trzeba opisywać ich działania, ponieważ jest to zdefiniowane w standardzie poprzez odpowiednie tablice prawdy.
- Użycie bramki jest podobne jak powołanie instancji własnego modułu i zwane jest deklaracją bramki. Zawiera:
  - słowo kluczowe (nazwa bramki),
  - listę połączeń
- oraz opcjonalnie
  - nazwę instancji
  - opóźnienie
  - siłę sterowania
  - rozmiar macierzy

```
wire y, a, b, c;

and (y, a, b);

and a1(y, a, b);
nand na3(y, a, b, c);

// opóźnienia

and #(4) (y, a, b);
and #(2, 4, 2) (y, a, b); //rise, fall, off
and #(1:2:3, 2:4:2) (y, a, b); //min, typ, max
```

Wbudowane elementy prymitywne

n_input gates	n_output gates	three-state gates	pull gates	MOS switches	bidirectional switches
and	buf	bufif0	pulldown	cmos	rtran
nand	not	bufif1	pullup	nmos	rtranif0
nor		notif0		pmos	rtranif1
or		notif1		rcmos	tran
xnor				rnmos	tranif0
xor				rpmos	tranif1

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

not	
input	output
0	1
1	0
x	x
z	x

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

Podłączenie do zasilania lub masy (stałe 0 lub 1 logiczne)

nmos rnmos	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	z	z	z

Bramki transmisyjne

Zadania i funkcje systemowe

- Do standardu Verilog należy **bogaty** zestaw zadań (*tasks*) i funkcji (*functions*) systemowych wspomagających sterowanie symulacją i obserwację wyników.
- Zadania systemowe poprzedzone są znakiem $.

Wyróżnia się następujące grupy zadań (funkcji) systemowych:

- wyświetlanie na ekranie (*display*)
- funkcje czasu symulacji, skali czasu i sterowania symulacją **($stop, $finish)**
- funkcje konwersji
- komunikacja z plikami IO
- analiza statystyczna i rozkład prawdopodobieństwa

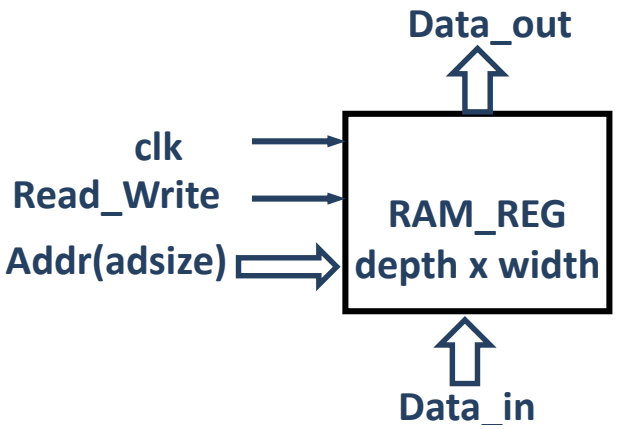
Popularnym zadaniem jest $display
ze składnią podobną jak printf w C:

```
$display("rval = %h hex %d decimal",rval,rval);
$display("rval = %o octal\nrval = %b bin",rval,rval);
$display("rval has %c ascii character value",rval);
$display("pd strength value is %v",pd);
$display("current scope is %m");
$display("%s is ascii value for 101",101);
$display("simulation time is %t", $time);
```

VHDL a Verilog

- Podstawowowa różnica w filozofii twórców VHDL i Verilog polega na tym, na ile za błędy w modelu odpowiedzialny jest projektant a na ile kompilator.
- VHDL zakłada, że większość szczegółów musi być opisana przy użyciu specjalnie do tego przeznaczonych konstrukcji językowych, w związku z czym nieprawidłowości zostają wychwycone na etapie kompilacji.
  - sztandarowym reprezentantem tego podejścia jest sposób operowania typami danych w VHDL
- Verilog zostawia większą swobodę, pozwalając na bardziej zwarty i elastyczny kod. Wymaga to jednak od projektanta większej świadomości w modelowaniu. Podejście to jest utrzymywane, a nawet rozwijane w przypadku SystemVerilog.

case 1: pamięć VHDL




```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-------------------------------------------------------------
entity RAM_REG is
 generic( width: integer:=4; depth: integer:=4;  adsize:
integer:=2);
 port( clk, Read_Write: in std_logic;
        Addr: in std_logic_vector(adsize-1 downto 0);
        Data_in: in std_logic_vector(width-1 downto 0);
        Data_out: out std_logic_vector(width-1 downto 0) );
 end RAM_REG;
```

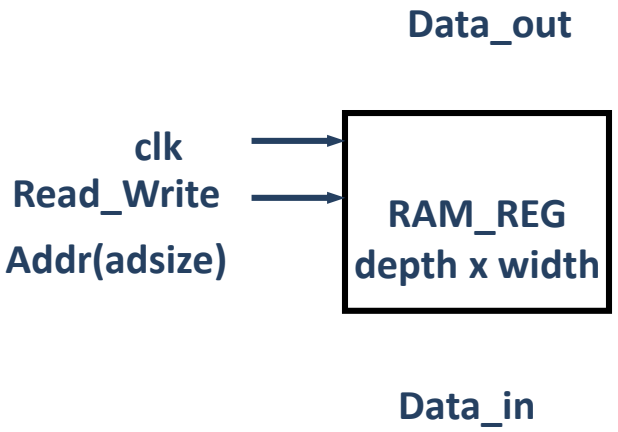
```
architecture df of RAM_REG is
type ram_array is array (0 to depth-1) of
std_logic_vector(width-1 downto 0);
signal ram: ram_array;

begin
process(read_write)
begin
    if read_write ='1' then
    Data_out <=    ram(to_integer(unsigned(Addr)));
    else    Data_out <= (Data_out'range => 'Z');   end if;
end process;

process(clk)
begin
       if (rising_edge(clk)) then
          if read_write = '0' then
                ram(to_integer(unsigned(Addr))) <= Data_in;
end if;
       end if;
end process;
end df;
```

case 1: VHDL komentarz

- biblioteki
- podział na entity i architecture
- konwersja typów

case 1: pamięć Verilog

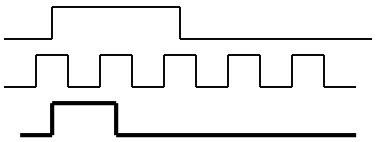
```
module RAM_REG #(parameter width=4, depth = 4, adsize = 2)
(input clk, Read_Write,
  input [adsize -1:0] Addr,
  input [width-1 : 0] Data_in,
  output reg [width-1 : 0] Data_out);

reg [width-1:0] ram [depth-1:0]; // deklaracja pamięci

always @(Read_Write)
   if (Read_Write)
                Data_out =   ram[Addr];
    else
       Data_out ='b Z;


 always @(posedge clk)
   if (!Read_Write)
                ram[Addr] =   Data_in;

endmodule
```

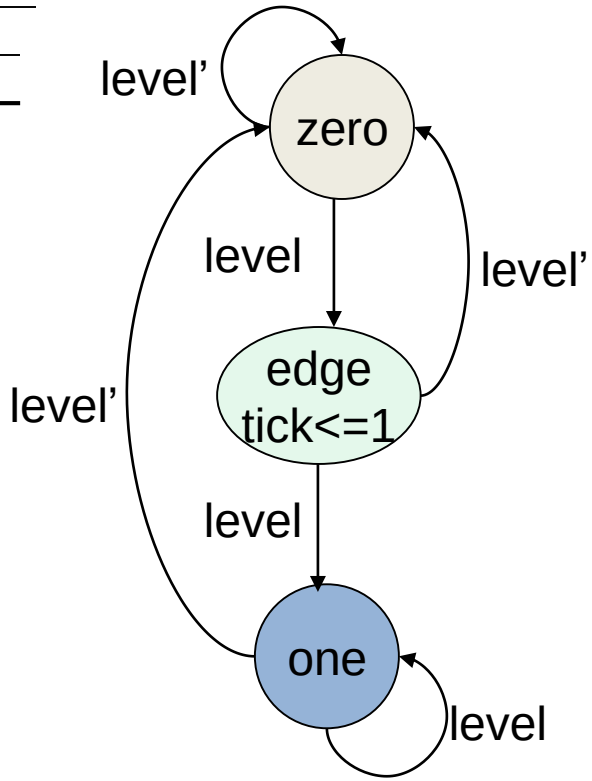


- niepotrzebne biblioteki do konwersji typów

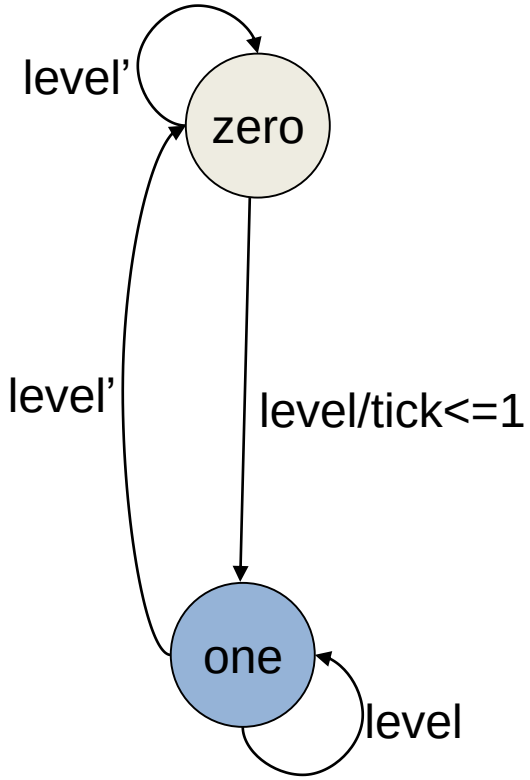
Przykład 2: układ wykrywający zbocze

- Przyjmijmy, że układ jest aktywowany szerokim sygnałem level, dla którego czas trwania ‘1’ jest znacznie dłuższy od okresu zegara w naszym układzie. Zadaniem naszego FSM jest wygenerowanie impulsu nie dłuższego od taktu zegara po wykryciu zmiany sygnału level z ‘0’na ‘1’.





automat Moore’a – 3 stany

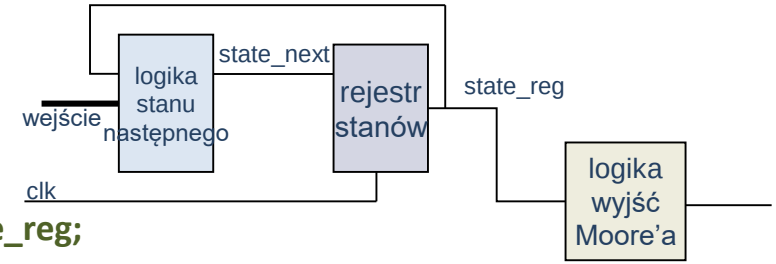


wyjścia Mealy’ego – 2 stany

case 2: implementacja Moore

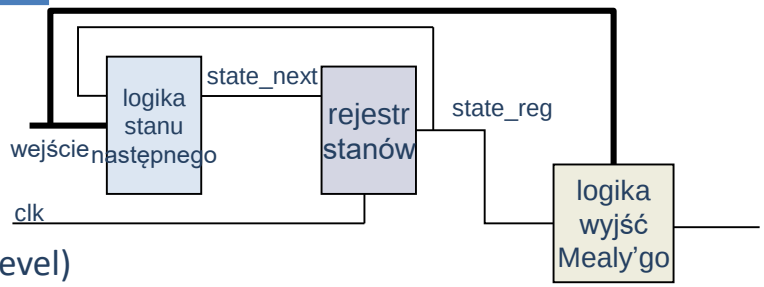
```
module edgeDetector
( input wire clk, reset,
   input wire level,
   output reg Moore_tick );

localparam [1:0] // 3 states are required for Moore
   zeroMoore = 2'b00,
   edgeMoore = 2'b01,
   oneMoore = 2'b10;

reg[1:0] stateMoore_reg, stateMoore_next;

//proces synchroniczny
always @(posedge clk, posedge reset)
begin
   if(reset) // reset asynchroniczny
      begin
      stateMoore_reg <= zeroMoore;
      end
   else // zmiana stanu
      begin
      stateMoore_reg <= stateMoore_next;
      end
end
```

```
//proces asynchroniczny
always @(stateMoore_reg, level)
begin
   // wartości domyślne
   stateMoore_next = stateMoore_reg;
   Moore_tick = 1'b0;

   case(stateMoore_reg)
      zeroMoore:
         if(level)
            stateMoore_next = edgeMoore; // do stanu edge else default value
      edgeMoore:
         begin
            Moore_tick = 1'b1;
            if(level)
               stateMoore_next = oneMoore;
            else
               stateMoore_next = zeroMoore;
         end
      oneMoore:
         if(~level)
            stateMoore_next = zeroMoore; // do stanu zero else default value
   endcase
end
endmodule
```



case 2: implementacja Mealy

```
module edgeDetector
(
   input  clk, reset,
   input level,
   output reg Mealy_tick
);

localparam  // 2 stany wystarczą
   zeroMealy = 1'b0,
   oneMealy =  1'b1;

reg stateMealy_reg, stateMealy_next;

always @(posedge clk, posedge reset)
begin
   if(reset)
      begin
      stateMealy_reg <= zeroMealy;
      end
   else
      begin
      stateMealy_reg <= stateMealy_next;
       end
end
```




```
// proces asynchroniczny
always @(stateMealy_reg, level)
begin
   stateMealy_next = stateMealy_reg;
   Mealy_tick = 1'b0;
   case(stateMealy_reg)
      zeroMealy:
         if(level)
            begin
               stateMealy_next = oneMealy;
               Mealy_tick = 1'b1;
            end
      oneMealy:
         if(~level)
            stateMealy_next = zeroMealy;
   endcase
end

endmodule
```

Ilustracja różnic: rozmiary wektorów

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.all;
 3  USE ieee.std_logic_arith.all;
 4  USE ieee.STD_LOGIC_UNSIGNED.all;
 5
 6  ENTITY adder IS
 7     PORT(
 8        A     : IN     std_logic_vector ( 7 DOWNTO 0 );
 9        B     : IN     std_logic_vector ( 7 DOWNTO 0 );
10        clk   : IN     std_logic;
11        rst_n : IN     std_logic;
12        carry : OUT    std_logic;
13        sum   : OUT    std_logic_vector ( 7 DOWNTO 0 )
14     );
15
16  END adder ;
17
18  ARCHITECTURE rtl OF adder IS
19    signal sum_int : std_logic_vector (8 downto 0);
20    signal A8      : std_logic_vector (8 downto 0);
21    signal B8      : std_logic_vector (8 downto 0);
22
23  BEGIN
24
25    A8 <= '0' & A;
26    B8 <= '0' & B;
27    sum_int <= A8 + B8;
28
29    adder: process (clk, rst_n)
30     begin
31       if rst_n = '0' then
32         carry <= '0';
33         sum   <= "00000000";
34       elsif clk'event and clk = '1' then
35         carry <= sum_int(8);
36         sum   <= sum_int(7 downto 0);
37       end if;
38     end process adder;
39  END ARCHITECTURE rtl;
```

```
 1  module adder
 2     ( input [7:0] A,
 3       input [7:0] B,
 4       input clk,
 5       input rst_n,
 6       output reg [7:0] sum,
 7       output reg carry);
 8
 9    always @(posedge clk or negedge rst_n)
10       if (!rst_n)
11         {carry,sum} <= 0;
12       else
13         {carry,sum} <= A + B;
14  endnodule
```