Programowanie układów FPGA

Wykład 2: programowanie proceduralne w Verilogu

Marek Materzok

15 października 2019

Wektory bitowe i tablice

Zapis liczb – prosty

Liczba całkowita ze znakiem: `0`, `12`, `+5`, `-8`

- Liczby ujemne są zapisywane w kodzie dopełnieniowym do dwóch
- Implementacja minimum 32-bitowa

Zapis liczb – z podstawą

Format: `[rozmiar] ' [znak] podstawa wartosc`

- `rozmiar` – liczba bitów
- `znak` – `s` lub brak
- `podstawa` – `o` (ósemkowa), `b` (dwójkowa), `d` (dziesiętna), `h` (szesnastkowa)
- `wartosc` – ciąg cyfr, od `0` do `f`, albo `x`, `z`, `_`
- Za mało bitów – dopełnianie `0`, lub `x` albo `z`, jeśli takim bitem kończy się wartość

Zapis liczb – przykłady

```
5'o37  // 31, 5'b11111
4'D2   // 2,  4'b0010
4'bx1z //    4'bxx1z
8'h1z  //    8'b0001zzzz
'b101  // 5
'sb101 // -3
32'hDEAD_BEEF
```

Deklaracja

Druty i rejestry mogą mieć wiele bitów:

```
wire [msb:lsb] druty;
```

Na przykład:

```
reg [7:0] bajt; // lsb 0, konwencja
reg [0:7] odwrotny_bajt; // msb 0
reg [-5:2] dziwny_bajt;
reg [0:0] jeden_bit;
```

Dostęp do bitów

Składnia: `wektor[nr_bitu]`

Wybór bitu zależy od deklaracji!

```
bajt = 1; odwrotny_bajt = 1;
dziwny_bajt = 1;
jeden_bit = bajt[0];           // daje 1
jeden_bit = odwrotny_bajt[0]; // daje 0
jeden_bit = odwrotny_bajt[7]; // daje 1
jeden_bit = dziwny_bajt[7];   // daje x
```

Podwektory od-do

Składnia: `wektor[msb:lsb]`

```
reg [3:0] polbajt;
polbajt = bajt[3:0];              // młodszy, 1
polbajt = bajt[0:3];              // błąd kompilacji
polbajt = odwrotny_bajt[0:3]; // 0
polbajt = odwrotny_bajt[4:7]; // 1
polbajt = dziwny_bajt[4:7];   // x
```

Podwektory od-ile

Składnia: `wektor[mniejszy_idx+:ile]`, `wektor[wiekszy_idx-:ile]`

Liczą się indeksy – `mniejszy_idx` to nie to samo, co lsb! Liczba `ile` nieujemna.

```
polbajt = bajt[0+:4];           // == [3:0]
polbajt = odwrotny_bajt[0+:4]; // == [0:3]
polbajt = bajt[3-:4];           // == [3:0]
```

Konkatenacja

Składnia: `{element_1, element_2, ...}`

- Elementy ze znanym rozmiarem
- Indeksy nie mają znaczenia (lsb skrajnie prawy)

```
{bajt, odwrotny_bajt} // 16'h0101
{1'b1, 4'h2}          // 5'b10010
```

Replikacja

Składnia: `{ile{co}}`

```
{4{1'b1}} // 4'b1111
{2{bajt}} // 16'h0101
```

Lvalue w przypisaniach

```
bajt[7] = 1'b1;    // przypisuje msb
bajt[7:4] = 4'hf; // przypisuje starszy półbajt
{bajt, odwrotny_bajt} = 16'h1337; // dwa bajty
```

Tablice

- Mogą być wielowymiarowe
- Mogą zawierać wektory bitów
- Nie są typem, nie mogą być używane jako argumenty modułów.
- (ale w SystemVerilogu mogą)

```
reg bity[15:0];        // 16 bitów
reg [7:0] bajty[3:0]; // 4 bajty
```

Wyrażenia

Operatory arytmetyczne

```
+, - // unarne
+, - ,*, /, %, ** // binarne
```

- Którykolwiek bit argumentu `z` lub `x` – cały wynik `x`
- Domyślnie arytmetyka bez znaku
  - `wire signed`, `reg signed`
  - `$signed()`, `$unsigned()`
  - obydwa argumenty operacji binarnych muszą być ze znakiem, aby operacja była ze znakiem

Operatory bitowe

```
!, ~, &, ~&, |, ~|, ^, ~^    // unarne
&&, ||, &, ~&, |, ~|, ^, ~^ // binarne
```

- `~&` to NAND
- unarny `&` daje koniunkcję wszystkich bitów wektora

Przesunięcia bitowe

```
<<,  >>  // logiczne
<<<, >>> // arytmetyczne
```

- Przesunięcia zerowe uzupełniają zerami
- `<<<` jest równoważny `<<`
- `>>>` uzupełnia bitem znaku liczby ze znakiem, liczby bez znaku uzupełnia zerami

Operatory równości

```
==,  !=  // logiczne
===, !== // dokładne
```

- Jeśli operand zawiera `x` lub `z`, równość logiczna zwraca `x`
- Równość dokładna rozróżnia `x` i `z` (ale jest niesyntezowalna!)

Operatory porównania

`<`, `>`, `<=`, `>=`

- Jeśli operand zawiera `x` lub `z`, zwracają `x`
- Porównanie ze znakiem tylko gdy oba operandy ze znakiem

Dodatkowe typy danych

Typy używane na metapoziomie, nie przeznaczone do syntezy:

- `integer` – liczby całkowite (min. 32 bit)
- `time` – czas symulacji, liczby naturalne (min. 64 bit) (`$time`)
- `real` – liczba zmiennoprzecinkowa
- `realtime` – czas symulacji, zmiennoprzecinkowy (`$realtime`)

Instrukcje warunkowe i pętle

Instrukcja warunkowa

```
if (expr) stmt1 [else stmt2]
```

- Fałsz – 0, `x` lub `z`

Case

```
case (expr)
  expr{,expr}: stmt
  default: stmt
endcase
```

- `default` opcjonalny
- gałęzie niezależne (brak breaków)
- dowolne wyrażenia dla gałęzi
- sprawdzana dokładna równość (`x` i `z` rozróżniane!)

Case – nietypowy przykład

Rozróżnianie sygnałów one-hot:

```
reg [2:0] val; // deklaracja

case (1'b1)
  val[0]: stmt1
  val[1]: stmt2
  val[2]: stmt3
endcase
```

Casex, casez

- `casez` traktuje z jako don't-care
- `casex` traktuje x i z jako don't-care
- w liczbach można użyć ? zamiast z

```
casez(val)
  4'b1???: stmt1
  4'b01??: stmt2
  4'b00??: stmt3
endcase
```

Pętla forever

```
forever stmt
```

- Pętla nieskończona
- Przy braku opóźnienia może zablokować symulator!

```
forever #1 clk = ~clk;
```

Pętla repeat

```
repeat (expr) stmt
```

- Powtórz n razy
- x lub z równoważne z 0

```
repeat(8) begin
  val[0] = in;
  #1 val = val << 1;
end
```

Pętle for i while

```
while (expr) stmt
for (assign; expr; assign) stmt
```

- Wzorowane na pętlach z C
- Ale w pętli for muszą być przypisania (nie dowolne wyrażenia)

Abstrakcja proceduralna

Zadania i funkcje

Zadania

Mogą stosować opóźnienia

Może wywoływać inne zadania

Dowolne argumenty

Nie zwraca wartości

Funkcje

Bez opóźnień

Nie może wywoływać zadań

Tylko wejściowe argumenty

Zwraca jedną wartość

Funkcje

```
function typ_zwr funkcja(input i1, input i2...);
  deklaracje
  instrukcja
endfunction
```

Na przykład:

```
function [7:0] dodaj(input [7:0] a, input [7:0] b);
  dodaj = a + b;
endfunction
```

Zmienne lokalne

Domyślnie zmienne są statyczne:

```
function [7:0] sumuj(input [7:0] a)
reg [7:0] s;
begin
  s = s + a; sumuj = s
end
endfunction

sumuj.s = 0;
$display(sumuj(2)); // 2
$display(sumuj(2)); // 4
```

Funkcje automatyczne

Funkcje automatyczne mają „ludzkie” zmienne lokalne oraz rekursję:

```
function automatic integer fact(input integer a);
  if (a > 1) fact = a * fact(a-1);
  else fact = 1;
endfunction
```

Zadania

```
task zadanie(input p1, output p2...);
  deklaracje
  instrukcja
endtask
```

- Może zwracać wiele wartości przez parametry
- Może mieć opóźnienia
- Może być automatyczne

Zadania wbudowane

display, write, strobe, monitor

```
$display(r);
$display("%d %d", r, s); // wyświetl dwie
$write("%b %h", r, s);   // nie kończy newlinem
$write(r, "\n"); // jak $display(r)
$strobe(r);  // wypisz na końcu kroku symulacji
$monitor(r); // wypisz na końcu każdego kroku,
             // gdy r ulegnie zmianie
$monitoroff; // wyłącz monitoring
$monitoron;  // włącz monitoring
```

Wyjście plikowe

```
integer plik;
plik = $fopen("plik");
plik2 = $fopen("plik2");
$fdisplay(plik, r);
$fwrite(plik | plik2, "%d", r);
$fstrobe(plik2, "%h", r);
$fclose(plik); $fclose(plik2);
```

Wejście plikowe

```
plik = $fopen("plik", "r");
c = $fgetc(plik);
reg [1023:0] bufor;
reg [7:0] mem[0:255];
kod = $fgets(bufor, plik);
kod = $fscanf(plik, "%d", r);
kod = $fread(bufor, plik);
kod = $fread(mem, plik, 5, 10); // do mem[5:14]
$fclose(plik);
```

Wczytywanie danych

```
reg [7:0] mem[0:255];
$readmemh("hexplik.data", mem);
$readmemb("binplik.data", mem);
$readmemh("hexplik.data", mem, 32); // do mem[32:255]
$readmemh("hexplik.data", mem, 32, 47); // do mem[32:47]
```

Format danych

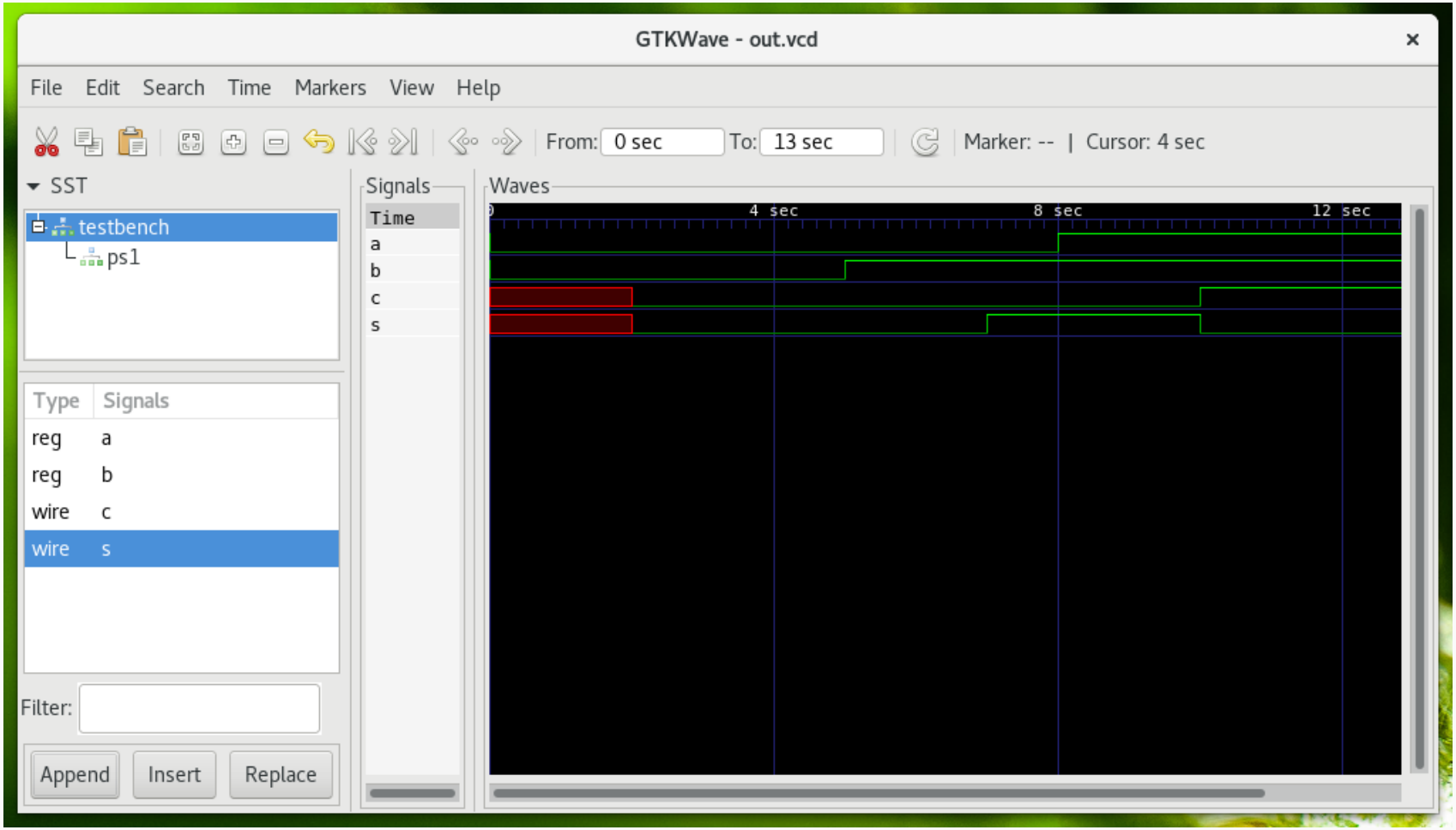
- Liczby (binarne lub szesnastkowe)
- Komentarze (w stylu C)
- Białe znaki
- Informacje o adresowaniu

```
BA DD FO 0D // magic number
@10 /* address 16 */
12 34 56 78
```

Zapisywanie przebiegów

```
$dumpfile("plik.vcd");
$dumpvars(0, mod); // rekurencyjnie wszystko w module mod
$dumpvars(1, mod); // tylko zmienne zadeklarowane w mod
$dumpoff;   // przerwij zapis
$dumpon;    // wznów zapis
$dumpall;   // zapisz stan wszystkich zmiennych w pliku
$dumpflush; // zapisz bufor zapisu
```

Podgląd przebiegów – gtkwave



Dyrektywy

define

Definicja makra, podobnie jak w C:

```
`define STALA 123
`define max(a,b) ((a) > (b) ? (a) : (b))
```

Wywołanie makra:

```
x = `STALA
y = `max(y, x)
```

undef

Usuwa makro:

```
`undef STALA
```

include

Wstawia zawartość innego pliku:

```
`include "stale.v"
```

ifdef, ifndef, else, elsif, endif

Sprawdzają istnienie zdefiniowanego makra:

```
`ifdef gates
  and a1 (a,b,c);
`else
  wire a = b & c;
`endif
```

timescale

Definiuje sposób liczenia czasu oraz przeliczania między czasem fizycznym a symulacyjnym:

```
`timescale jednostka / precyzja
// na przykład:
`timescale 1 ns / 10 ps
```

- Jednostka – fizyczny odpowiednik jednostki czasu symulacji
- Precyzja – najmniejsza symulowana jednostka czasu

timescale – przykład

```
`timescale 10 ns / 1 ns
module test;
  reg set;
  parameter d = 1.55;
  initial begin
    #d set = 0;
    #d set = 1;
  end
endmodule
```

Opóźnienie wyniesie 16 ns.

Parametryzacja i generowanie

Parametry

```
parameter [typ] przypisania;
// na przykład:
parameter msb = 7;
parameter real pi = 3.14;
parameter [31:0] magic = 32'hDEADBEEF;
```

- Lokalne stałe w modułach
- Redefiniowalne w zewnętrznych modułach

Sparametryzowany moduł

```
module polsumator(output s, c, input a, b);
  parameter d_xor = 2;
  parameter d_and = 1;

  xor #(d_xor) xor1(s, a, b);
  and #(d_and) and1(c, a, b);

endmodule
```

Albo:

```
module polsumator #(d_xor = 2, d_and = 1)
    (output s, c, input a, b);
```

Użycie sparametryzowanego modułu

```
polsumator ps1(s,c,a,b); // wartości domyslne
polsumator #(1) ps2(s,c,a,b);   // d_xor=1
polsumator #(2,2) ps3(s,c,a,b); // d_xor=2, d_and=2
polsumator #(.d_and(2)) ps4(s,c,a,b);
polsumator ps5(s,c,a,b);
defparam ps5.d_and = 3;
```

Parametry lokalne

```
localparam par1 = 42;
localparam par2 = par1 + 5;
```

- Stałe wewnątrz modułu (nie redefiniowalne)
- Mogą zależeć od innych parametrów

Generowanie – warunek

```
module polsumator #(with_nand = 0)
    (output s, c, input a, b);
  if (with_nand == 0) begin
    xor xor1(s, a, b);
    and xor1(c, a, b);
  end else begin
    xor_nand xor1(s, a, b);
    and_nand xor1(c, a, b);
  end
endmodule
```

To nie jest taki `if`, co w blokach `initial` itd.

Instrukcje na poziomie modułu tworzą **metaprogram**, wykonywany przed uruchomieniem właściwego programu.

Generowanie – case

```
module polsumator #(with_nand = 0)
    (output s, c, input a, b);
  case (with_nand)
    0: begin
      xor xor1(s, a, b);
      and xor1(c, a, b);
    end
    default: begin
      xor_nand xor1(s, a, b);
      and_nand xor1(c, a, b);
    end
  endcase
endmodule
```

Generowanie – pętla

```
module big_and #(SIZE = 8)(
    output [SIZE-1:0] c,
    input [SIZE-1:0] a,
    input [SIZE-1:0] b);
  genvar i;
  for (i = 0; i < SIZE; i = i + 1)
    and a(c[i], a[i], b[i]);
endmodule
```

To jest metapętla – wartości muszą być znane w czasie kompilacji.

`genvar` oznacza metazmienną.

Generowanie – przykład

```
module sumator_szereg #(parameter SIZE = 8)(
    output [SIZE-1:0] s, output c_out,
    input [SIZE-1:0] a, input [SIZE-1:0] b, input c_in);
  genvar i;
  wire [SIZE-1:-1] c;
  for (i = 0; i < SIZE; i = i + 1)
    sumator sum(s[i], c[i], a[i], b[i], c[i-1]);
  assign c_out = c[SIZE-1],
    c[-1] = c_in;
endmodule
```

Zdarzenia

Zdarzenia

Zdarzenia opóźniają wykonanie instrukcji, aż coś się stanie.

Deklarowanie zdarzeń:

```
event zdarz1, zdarz2;
```

Wywoływanie zdarzeń:

```
-> zdarz1;
```

Czekanie na zdarzenie:

```
@zdarz1 instrukcja;
```

Zdarzenia – przykład

```
module eventtest;
  reg [7:0] s, v;
  event zdarz;
  initial s = 0;
  initial v = 0;
  initial repeat(8) #1 begin
    v = v + 1; -> zdarz;
  end
  initial begin
    repeat(8) @zdarz s = s + v;
    $finish;
  end
  initial $monitor($time, " s=%d v=%d", s, v);
endmodule
```

Zdarzenia – przykład

```
$ ./a.out
0 s=  0 v=  0
1 s=  1 v=  1
2 s=  3 v=  2
3 s=  6 v=  3
4 s= 10 v=  4
5 s= 15 v=  5
6 s= 21 v=  6
7 s= 28 v=  7
8 s= 36 v=  8
```

Zdarzenia zmiany wartości

```
reg r;
@r instrukcja; // czekaj na zmianę r
@(posedge r) instrukcja;  // czekaj na zbocze narastające
@(negedge r) instrukcja;  // czekaj na zbocze opadające
@(r or zdarz) instrukcja; // czekaj na jedno ze zdarzeń
@* instrukcja; // czekaj na zmianę dowolnej zmiennej
               // czytanej przez instrukcję
```

Czekanie na warunek

```
wait (i > 0) r = i; // czeka z przypisaniem, aż i dodatnie
```

Funkcjonalnie równoważny program:

```
while (!(i > 0)) @i;
r = i;
```

Modelowanie układów kombinacyjnych

assign – przypisanie ciągłe

```
assign lhs1 = expr1, lhs2 = expr2;
```

- Deklaracja top-level w module
- Przypisanie do drutów
- Można przypisywać więcej niż raz ten sam drut
- Można przypisywać jednocześnie kilka drutów

Przykład – półsumator, sumator

```
module polsumator(output s, c, input a, b);
  assign {c, s} = a + b;
endmodule

module sumator(output s, c, input a, b, c_in);
  assign {c, s} = a + b + c_in;
endmodule
```

Przykład – multiplekser

```
module mux4(output o, input i1, i2, i3, i4, input [1:0] s);
  assign
    o = (s == 0) ? i1 : 1'bz,
    o = (s == 1) ? i2 : 1'bz,
    o = (s == 2) ? i3 : 1'bz,
    o = (s == 3) ? i4 : 1'bz;
endmodule
```

Przykład – multiplekser, inaczej

```
module mux4(output o, input i1, i2, i3, i4, input [1:0] s);
  assign
    o = (s == 0) ? i1 :
        (s == 1) ? i2 :
        (s == 2) ? i3 :
        (s == 3) ? i4 : 1'bx;
endmodule
```

Przykład – opóźnienia

```
module polsumator(output s, c, input a, b);
  assign #2 {c, s} = a + b;
endmodule
```

Uwaga – zmiana wejścia przed czasem propagacji **anuluje** poprzednią zmianę! Stany pośrednie nie są modelowane.

Opóźnienia – testbench

```
module testbench;
  reg a, b; wire s, c;
  polsumator ps1(s, c, a, b);
  initial begin
    a = 0; b = 0;
    #5 b = 1;
    #2 a = 1;
    #1 b = 0;
    #1 a = 0;
    #5 $finish;
  end
  initial $monitor($time, " a=%d b=%d s=%d c=%d",
    a, b, s, c);
endmodule
```

Opóźnienia – wynik

```
$ ./testbench
                0 a=0 b=0 s=x c=x
                2 a=0 b=0 s=0 c=0
                5 a=0 b=1 s=0 c=0
                7 a=1 b=1 s=1 c=0
                8 a=1 b=0 s=1 c=0
                9 a=0 b=0 s=1 c=0
               11 a=0 b=0 s=0 c=0
```

- Nigdzie nie pojawiło się c=1!
- To nie problem – w ogólności naruszenia timingu nie dają żadnych gwarancji!

Blok always

```
always instrukcja;
// równoważne z:
initial forever instrukcja;
```

- Blok top-level w module
- Instrukcja musi mieć opóźnienie, inaczej zablokuje symulację!
- SystemVerilog wprowadza warianty `always_comb`, `always_latch` i `always_ff`, użyteczne do syntazy

Multiplekser – blokiem always

```
module mux4(output reg o, input i1, i2, i3, i4,
            input [1:0] s);
  always @* case (s)
    2'd0: o = i1;
    2'd1: o = i2;
    2'd2: o = i3;
    2'd3: o = i4;
    default: o = 'bx;
  endcase
endmodule
```

W SystemVerilogu zamiast `always @*` używamy `always_comb` do opisywania układów kombinacyjnych

Opóźnienia w bloku always

```
// wersja 1
module polsumator(output reg s, c, input a, b);
  always @* #2 {c, s} = a + b;
endmodule
// wersja 2
module polsumator(output reg s, c, input a, b);
  always @* {c, s} = #2 a + b;
endmodule
```

Przypisanie typu `lhs = #k expr` równoważne:

```
temp = expr;
#k lhs = temp;
```

Wersja 1 – wykonanie

```
$ ./testbench
 0 a=0 b=0 s=x c=x
 2 a=0 b=0 s=0 c=0
 5 a=0 b=1 s=0 c=0
 7 a=1 b=1 s=0 c=1
 8 a=1 b=0 s=0 c=1
 9 a=0 b=0 s=0 c=1
10 a=0 b=0 s=0 c=0
```

- Źle – w kroku 7 powinno być s=1 c=0! Zbyt szybka propagacja!
- Nie należy tak modelować opóźnień

Wersja 2 – wykonanie

```
$ ./testbench
                0 a=0 b=0 s=x c=x
                2 a=0 b=0 s=0 c=0
                5 a=0 b=1 s=0 c=0
                7 a=1 b=1 s=1 c=0
                8 a=1 b=0 s=1 c=0
                9 a=0 b=0 s=1 c=0
```

- Źle – na końcu powinno być s=0 c=0! Pośredni wynik wygrał z końcowym!
- Nie należy tak modelować opóźnień

Opóźnienia – wersja 3

```
module polsumator(output s, c, input a, b);
  reg [1:0] tmp;
  always @* tmp = a + b;
  assign #2 {c, s} = tmp;
endmodule
```

- Fuj ☹ smrodek
- Ale to jest najlepsza wersja (jeśli nie można się obyć bez `always`)