Symulacja

Symulacja

- **wyścigi**
- **modelowanie opóźnień**
- **cykl Delta-Time**
- ***sensitivity list***
- **arbitraż**

Wyścig – ostateczny wynik zależy od kolejności wykonywania działań:

```
always @ (posedge clock)
  begin
    x = y; // wyścig...
    y = x;
  end
```

```
always @ (posedge clock)
  begin
    x <= y; // brak wyścigu...
    y <= x;
  end
```

Przypisania nieblokujące emulowane przez przypisania blokujące:

```
always @ (posedge clock)
  begin
    temp_x = x; // odczyt...
    temp_y = y;

    x = temp_y; // zapis...
    y = temp_x;
  end
```

Czy to jest rejestr przesuwny ?

```
always @ (posedge clock)
  begin  // blokujące...
    qout[0] = sin;
    qout[1] = qout[0];
    qout[2] = qout[1];
    qout[3] = qout[2];
  end
```

To jest rejestr przesuwny !

```
always @ (posedge clock)
  begin  // nieblokujące...
    qout[0] <= sin;
    qout[1] <= qout[0];
    qout[2] <= qout[1];
    qout[3] <= qout[2];
  end
```

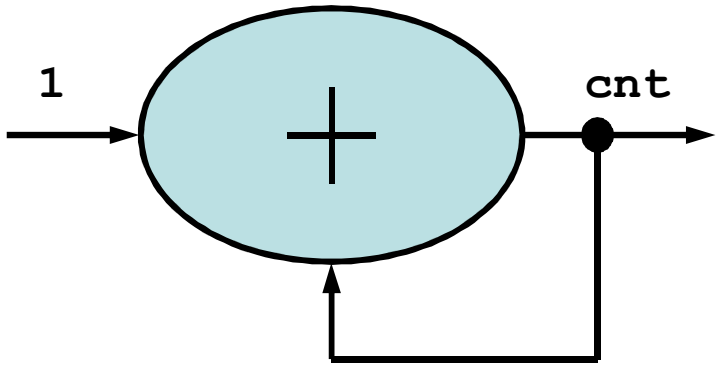
- `qout = {qout[2:0], sin}; // dobrze - dlaczego?`
- **jeszcze inaczej?**

Modelowanie opóźnień

```
`timescale 1 ns / 1 ps

always @ (val)
  #10 inc = val + 1;
```

W powyższym przykładzie `inc` przyjmuje nową wartość dokładnie po 10 ns (nie 9.999 lub 10.001 ns).

```
`timescale 1 ns / 1 ps

always
  #10 cnt = cnt + 1;
```



Deklaracja opóźnień jest ignorowana przez narzędzia do syntezy.

Cykl symulacji – *Delta Time*

- **Pierwsze symulatory:**
  ***One-List Algorithm* (ewaluacja i przypisywanie).**
- **Symulatory HDL:**
  ***Two-List Algorithm* (ewaluacja / przypisywanie).**

Symulowanie zdarzeń o zerowym czasie opóźnienia jest wykonywane podczas fikcyjnej jednostki czasowej zwanej *delta-time*. Stanowi ona całkowity cykl symulacyjny, nie zwiększając jednak licznika czasu:

- **symulator modeluje zdarzenia o zerowym czasie opóźnienia, używając cyklu *delta-time*,**
- **zdarzenia uruchomione w tym samym czasie są symulowane w *delta-time* w określonym porządku,**
- **związana z nimi logika jest resymulowana w celu propagacji zmian dla następnego cyklu,**
- **cykle *delta-time* są powtarzane do momentu odnotowania braku zmian,**
- **wtedy następuje zwiększenie licznika czasu do kolejnego zdarzenia (np. zbocza zegara).**

Cykl symulacji – *Delta Time*

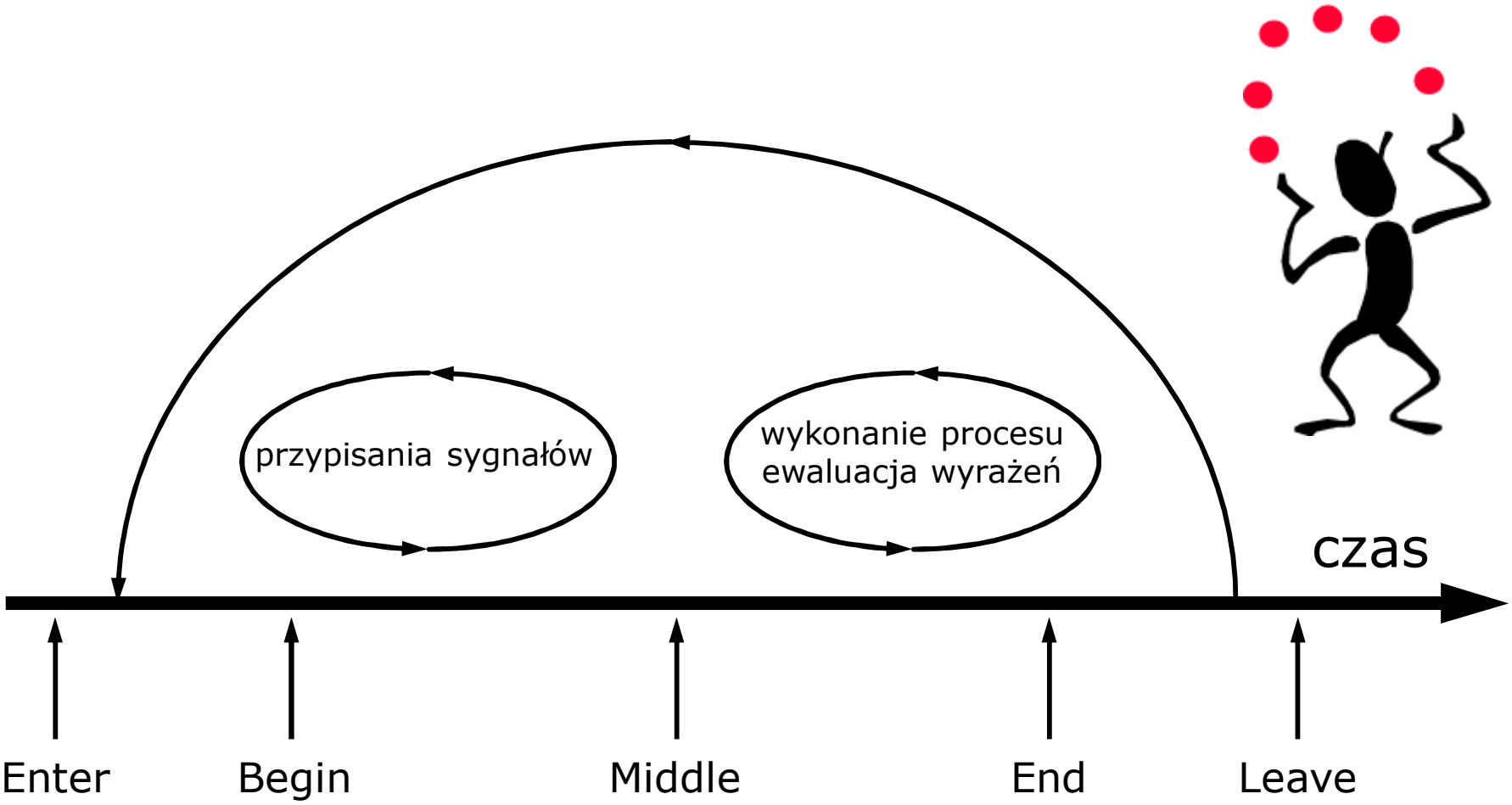
Dla symulacji przypisań nieblokujących symulator wykonuje 3-krokowy algorytm:

1. **<u>Odczyt:</u>**
   Czytanie wszystkich zmiennych po prawej stronie przypisani.

2. **<u>Ewaluacja:</u>**
   Obliczanie wyrażeń z wartości po prawej stronie przypisań i wpisywanie ich ich na listę wartości nowych.

3. **<u>Przypisania</u>**
   Wpisywanie wartości z listy wartości nowych do listy wartości bieżących.

Cykl symulacji – *Delta Time*



Symulacja zerowych opóźnień

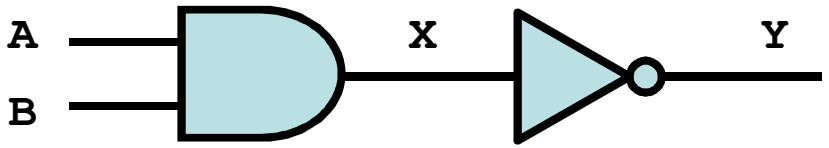
```
always @ (event)
  begin
  ...
  end;

X <= 1; // przypisanie zachodzi po czasie 0 od aktywacji

X <= Y; // zamiana zachodzi po czasie 0 od aktywacji
Y <= X;

A  = 1; // A=3 => ?, przypisanie blokujące
S <= A; // S=5 => ?, przypisanie nieblokujące
V  = S; // V=9 => ?, V otrzymuje …?

X <= 1;
X <= 2; // po czasie 0 od aktywacji X otrzyma wartość 2
```

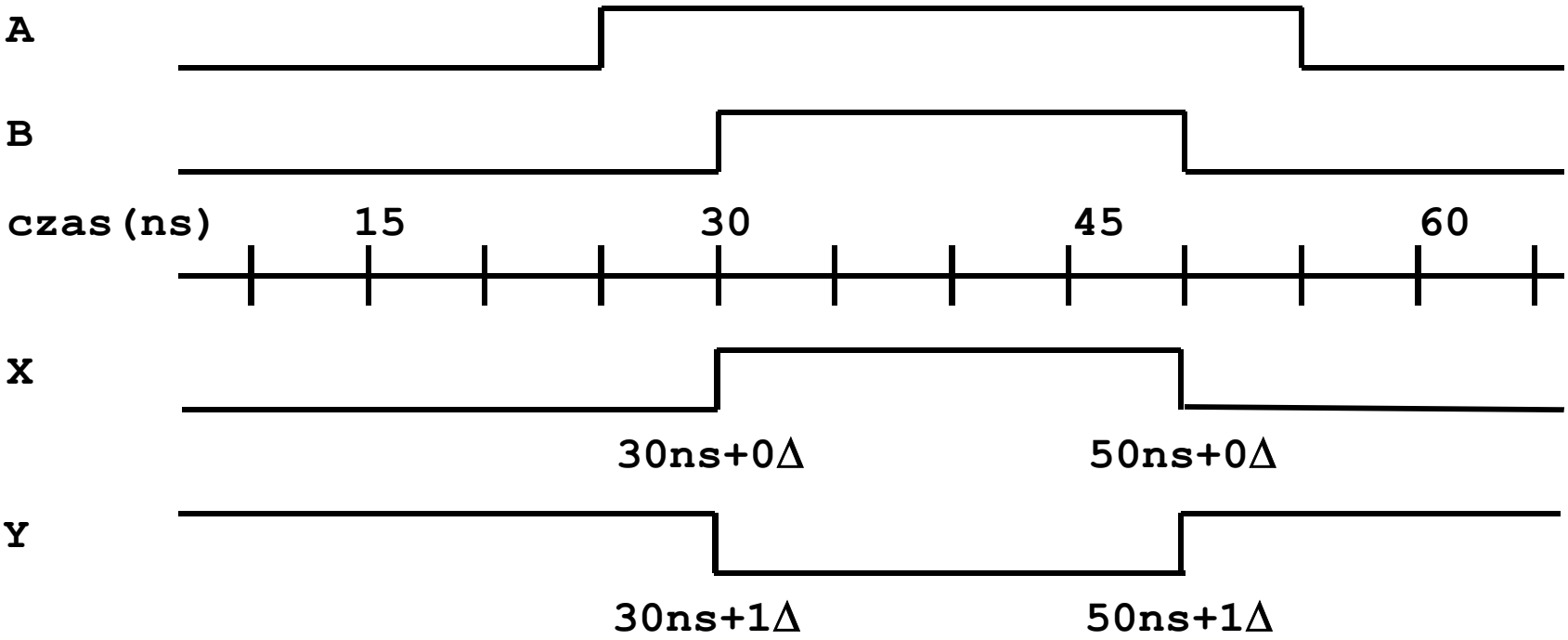
Cykl symulacji – *Delta Time*

Symulacja przypisań ciągłych z użyciem cyklu Δ.

Przykład:

```
assign X = A & B;
assign Y = ~ X;
```





Jakie są wartości X oraz A po jednym cyklu *delta-time* ?

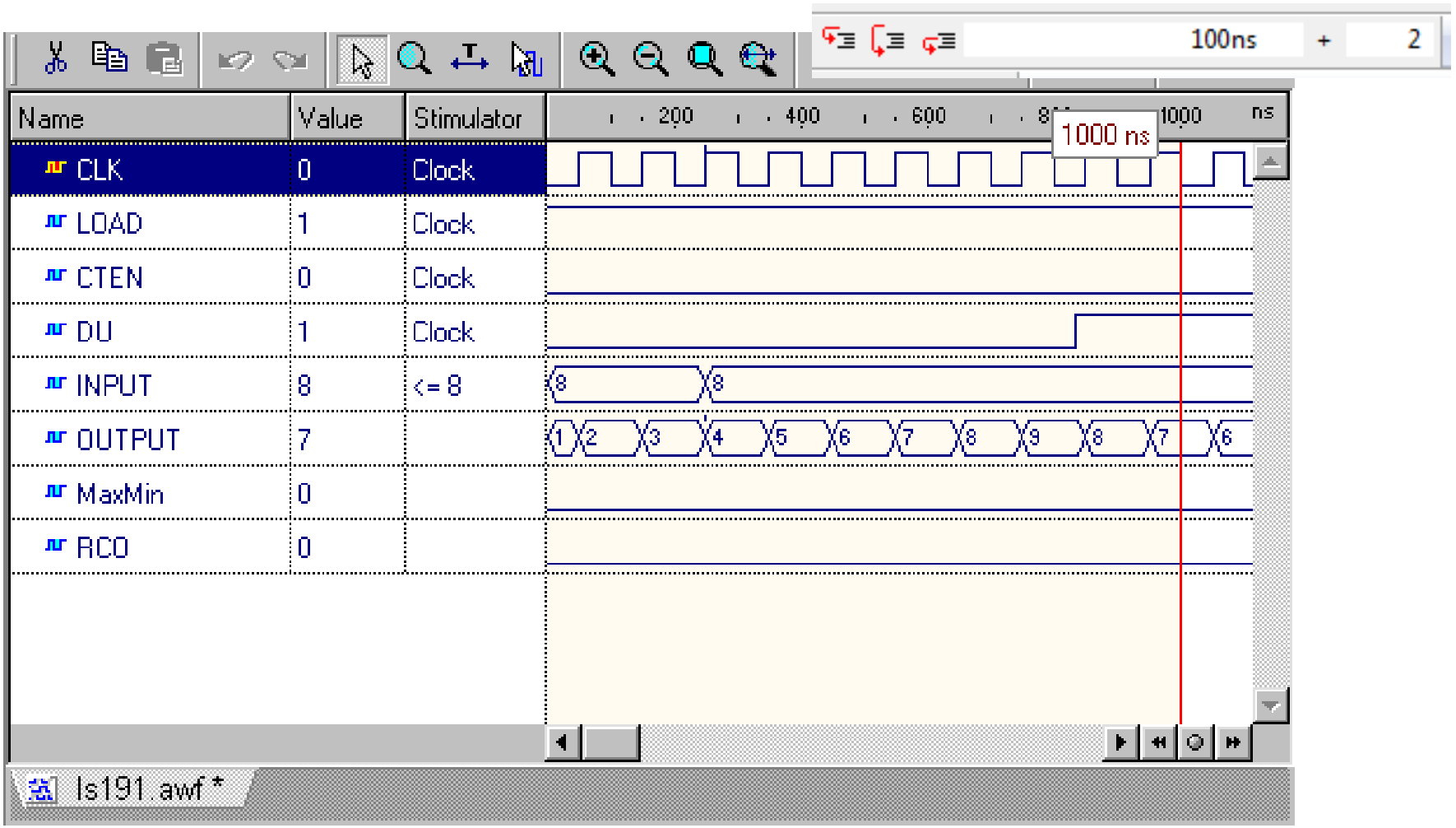
```
always @ (event)
  begin
    X <= 1;
    X <= 2;
    A <= X;
    X <= 3;
  end;
```

Jaka jest różnica w zachowaniu się dwóch poniższych bloków?

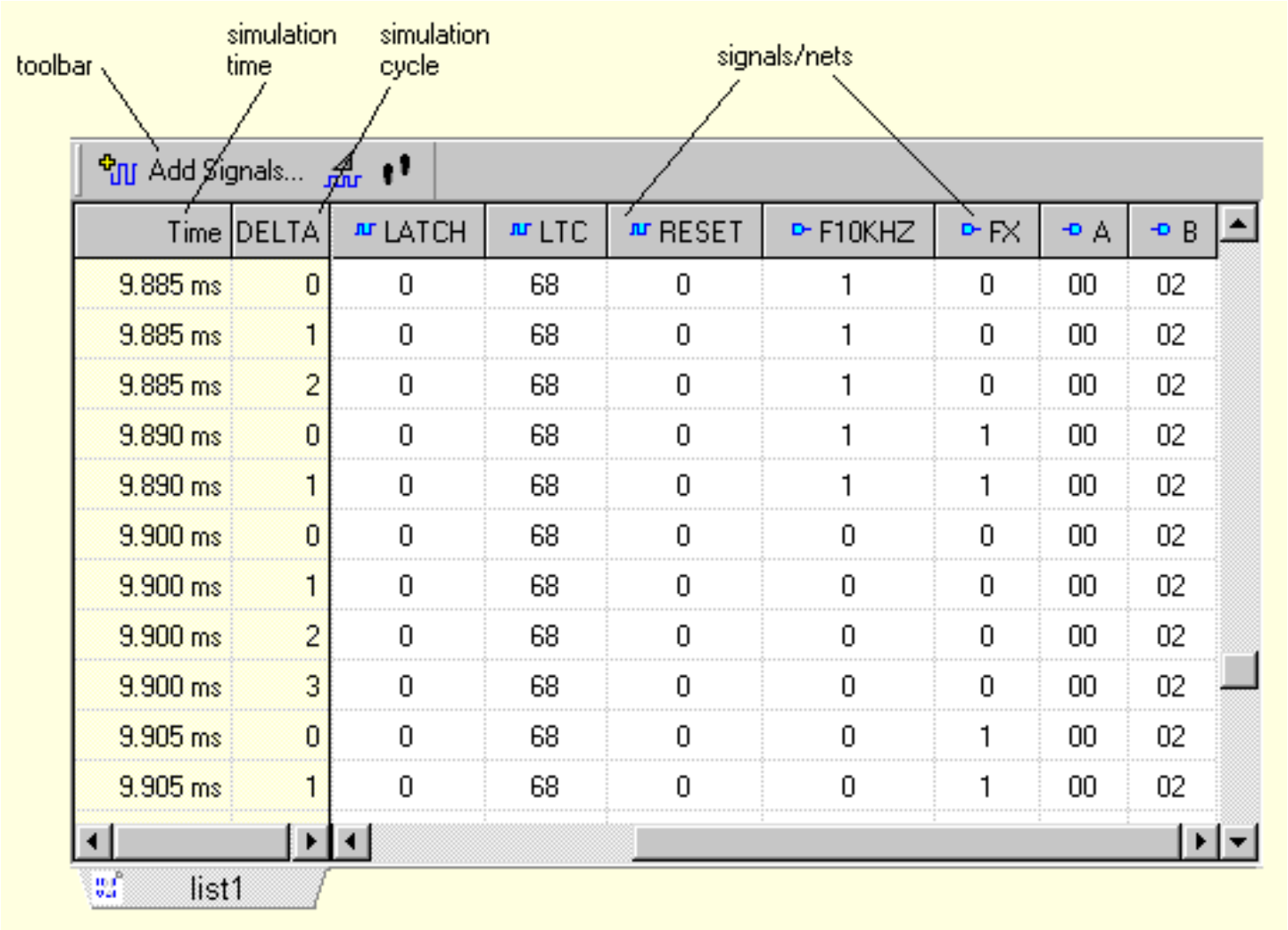
```
always @(A, B)
  begin
    S <= A;
    T <= B;
    V <= S | T;
  end
```

```
always @(A, B, S, T)
  begin
    S <= A;
    T <= B;
    V <= S | T;
  end
```

ActiveHDL – *Waveform Viewer*



ActiveHDL – *List Viewer*



Time	DELTA	LATCH	LTC	RESET	F10KHZ	FX	A	B
9.885 ms	0	0	68	0	1	0	00	02
9.885 ms	1	0	68	0	1	0	00	02
9.885 ms	2	0	68	0	1	0	00	02
9.890 ms	0	0	68	0	1	1	00	02
9.890 ms	1	0	68	0	1	1	00	02
9.900 ms	0	0	68	0	0	0	00	02
9.900 ms	1	0	68	0	0	0	00	02
9.900 ms	2	0	68	0	0	0	00	02
9.900 ms	3	0	68	0	0	0	00	02
9.905 ms	0	0	68	0	0	1	00	02
9.905 ms	1	0	68	0	0	1	00	02

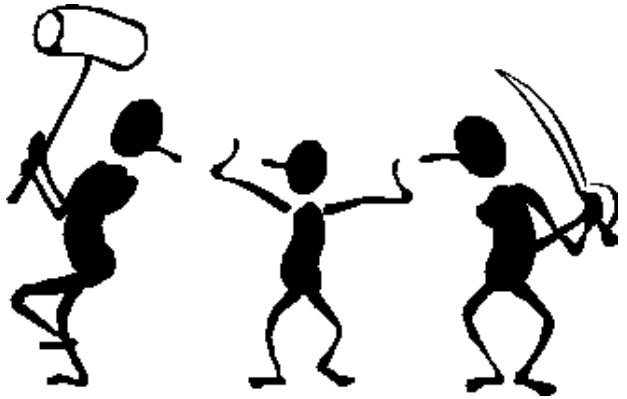
Funkcja arbitrażu

Verilog zezwala na sterowanie węzła (sieci) z wielu źródeł, ale:

- **każde ze źródeł musi być w osobnym procesie albo przypisaniu ciągłym,**
- **dla takich węzłów musi być uruchamiana funkcja arbitrażu (*resolution function*).**

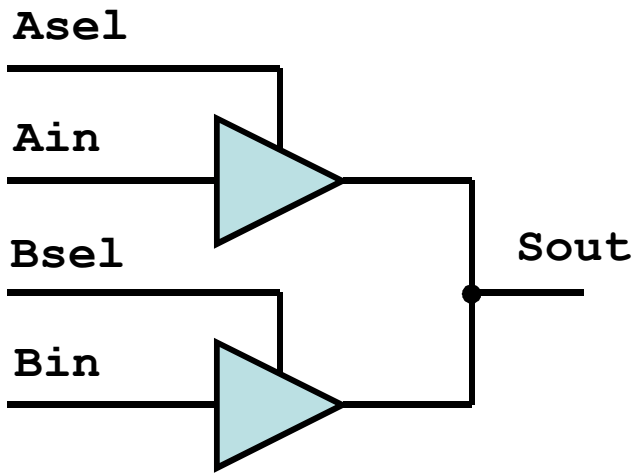
Funkcja arbitrażu:

- **funkcja taka jest predeklarowana dla logiki cztero wartościowej.**
- **jest wywoływana w przypadku zmian w każdym ze źródeł sterujących tym węzłem,**
- **zawiera tablicę koincydencji wartości logiki cztero wartościowej,**
- **otrzymuje wektor *driverów* do arbitrażu,**
- **przeprowadza arbitraż wszystkich *driverów* przy każdej zmianie stanu któregokolwiek z nich.**



Funkcja arbitrażu

```
always @(edge Ain, edge Asel)
  if (Asel == 1'b1)
    Sout = Ain;
  else
    Sout = 1'bZ;

always @(edge Bin, edge Bsel)
  if (Bsel == 1'b1)
    Sout = Bin;
  else
    Sout = 1'bZ;
```




```
assign Sout = Asel ? Ain : 1'bZ;
assign Sout = Bsel ? Bin : 1'bZ;
```

Symulacja

Funkcja arbitrażu VHDL – STD_LOGIC_1164

```
TYPE std_ulogic IS ('U','X','0','1','Z','W','L','H','-');
TYPE std_ulogic_vector IS ARRAY (NATURAL RANGE <>) OF std_ulogic;
FUNCTION resolved (s : std_ulogic_vector) RETURN std_ulogic;
SUBTYPE std_logic IS resolved std_ulogic;
TYPE std_logic_vector IS ARRAY (NATURAL RANGE <>) OF std_logic;

TYPE stdlogic_table IS ARRAY (std_ulogic, std_ulogic) OF std_ulogic;
CONSTANT resolution_table : stdlogic_table := (
------------------------------------------------
--| U   X   0   1   Z   W   L   H   -        |   |
------------------------------------------------
  ('U','U','U','U','U','U','U','U','U'),  -- | U |
  ('U','X','X','X','X','X','X','X','X'),  -- | X |
  ('U','X','0','X','0','0','0','0','X'),  -- | 0 |
  ('U','X','X','1','1','1','1','1','X'),  -- | 1 |
  ('U','X','0','1','Z','W','L','H','X'),  -- | Z |
  ('U','X','0','1','W','W','W','W','X'),  -- | W |
  ('U','X','0','1','L','W','L','W','X'),  -- | L |
  ('U','X','0','1','H','W','W','H','X'),  -- | H |
  ('U','X','X','X','X','X','X','X','X')); -- | - |
```

Funkcja arbitrażu VHDL – STD_LOGIC_1164

```
FUNCTION resolved (s: std_ulogic_vector ) RETURN std_ulogic IS
VARIABLE result : std_ulogic := 'Z';  -- weakest state default
BEGIN
-- The test for a single driver is essential otherwise the
-- loop would return 'X' for a single driver of '-' and that
-- would conflict with the value of a single driver unresolved
-- signal.
  IF (s'LENGTH = 1) THEN RETURN s(s'LOW);
  ELSE
    FOR i IN s'RANGE LOOP
      result := resolution_table(result, s(i));
    END LOOP;
  END IF;
  RETURN result;
END resolved;
```

Ciąg dalszy nastąpi…